Introduction to Combinatorial Optimization

Ding-Zhu Du **Panos M Pardalos** **Xiaodong Hu** **Weili Wu**

Ding-Zhu Du
Department of Computer Science
University of Texas at Dallas, Richardson TX 75080, USA
E-mail: dzdu@dallas.edu

Panos M. Pardalos
Department of Industrial and System Engineering
University of Florida, Gainesville FL 32611-6595, USA
E-mail: pardalos@ufl.edu

Xiaodong Hu
Institute of Applied Mathematics
Chinese Academy of Sciences
Beijing, China
E-mail: xdhu@amss.ac.cn

Weili Wu
Department of Computer Science
University of Texas at Dallas, Richardson TX 75080, USA
E-mail: weiliwu@dallas.edu

"Since the fabric of the world is the most perfect and was established by the wisest Creator, nothing happens in this world in which some reason of maximum or minimum would not come to light."

- Euler

"When you say it, its marketing. When they say it, its social proof."

- Andy Crestodina

"God used beautiful mathematics in creating the world."

- Paul Dirac

Contents

1 Introduction **1**
- 1.1 What is Combinatorial Optimization? . . . 1
- 1.2 Optimal and Approximation Solution . . . 2
- 1.3 Preprocessing . . . 6
- 1.4 Running Time . . . 7
- 1.5 Data Structure . . . 8
- Exercises . . . 10
- Historical Notes . . . 11

2 Sorting and Divide-and-Conquer **13**
- 2.1 Algorithms with Self-Reducibility . . . 13
- 2.2 Heap . . . 19
- 2.3 Counting Sort . . . 25
- 2.4 Examples . . . 29
- Exercises . . . 34
- Historical Notes . . . 36

3 Dynamic Programming and Shortest Path **37**
- 3.1 Dynamic Programming . . . 37
- 3.2 Shortest Path . . . 44
- 3.3 Dijkstra Algorithm . . . 51
- 3.4 Priority Queue . . . 54
- 3.5 Bellman-Ford Algorithm . . . 58
- 3.6 All Pairs Shortest Paths . . . 58
- Exercises . . . 65
- Historical Notes . . . 67

4 Greedy Algorithm and Spanning Tree **69**
- 4.1 Greedy Algorithms . . . 69

4.2 Matroid 75
4.3 Minimum Spanning Tree 81
4.4 Local Ratio Method 84
Exercises 89
Historical Notes 92

5 Incremental Method and Network Flow 95
5.1 Maximum Flow 95
5.2 Edmonds-Karp Algorithm 102
5.3 Bipartite Matching 105
5.4 Dinitz Algorithm for Maximum Flow 110
5.5 Minimum Cost Maximum Flow 111
5.6 Chinese Postman and Graph Matching 114
Exercises 117
Historical Notes 121

8 NP-hard Problems and Approximation Algorithms 123
8.1 What is the class NP? 123
8.2 What is NP-completeness? 131
8.3 Hamiltonian Cycle 137
8.4 Vertex Cover 147
8.5 Three-Dimensional Matching 149
8.6 Partition 154
8.7 Planar 3SAT 165
8.8 Complexity of Approximation 170
Exercises 184
Historical Notes 189

Chapter 1

Introduction

> "True optimization is the revolutionary contribution of modern research to decision processes."
> - George Dantzig

Let us start this textbook from a fundamental question and tell you what will constitute this book.

1.1 What is Combinatorial Optimization?

The aim of combinatorial optimization is to find an optimal object from a finite set of objects. Those candidate objects are called *feasible solutions* while the optimal one is called an *optimal solution*. For example, consider following problem.

Problem 1.1.1 (Minimum Spanning Tree)**.** *Given a connected graph $G = (V, E)$ with nonnegative edge weight $c : E \to R_+$, find a spanning tree with minimum total weight, where "spanning" means that all nodes are involved and a spanning tree interconnects all nodes in V.*

Clearly, the set of all spanning trees is finite and the aim of this problem is to find one with minimum total weight from this set. Each spanning tree is a feasible solution and the optimal solution is the spanning tree with minimum total weight, which is also called the *minimum spanning tree*. Therefore, this is a combinatorial optimization problem.

The combinatorial optimization is a proper subfield of discrete optimization. In fact, there exists problem in discrete optimization, which does not belong to combinatorial optimization. For example, consider the integer

programming. It always belongs to discrete optimization. However, when feasible domain is infinite, it does not belong to combinatorial optimization. But, such a difference is not recognized very well in the literature. Actually, if a paper on lattice-point optimization is submitted to *Journal of Combinatorial Optimization*, then usually, it will not be rejected due to out of scope.

In view of methodologies, combinatorial optimization and discrete optimization have very close relationship. For example, to prove NP-hardness of integer programming, we need to cut its infinitely large feasible domain into a finite subset containing optimal solution (see Chapter 8 for detail), i.e., transform it into a combinatorial optimization problem.

Geometric optimization is another example. Consider following problem.

Problem 1.1.2 (Minimum Length Guillotine Partition)**.** *Given a rectangle with point-holes inside, partition it into smaller rectangles without hole inside by a sequence of guillotine cuts to minimize the total length of cuts.*

There exist infinitely many number of partitions. Therefore, it is not a combinatorial optimization problem. However, we can prove that optimal partition can be found from a finite set of partitions of a special type (for detail, see Chapter 3). Therefore, to solve the problem, we need only to study partitions of this special type, i.e., a combinatorial optimization problem.

Due to above, we do not make a clear cut to exclude other parts of discrete optimization. Actually, this book is methodology-oriented. Problems are selected to illustrate methodology. Especially, for each method, we may select a typical problem as companion to explore the method, such as its requirements and applications. For example, we use sorting problem to explain divide-and-conquer technique, employ the shortest path problem to illustrate dynamic programming, etc..

1.2 Optimal and Approximation Solution

Let us show an optimality condition for the minimum spanning tree.

Theorem 1.2.1 (Path Optimality)**.** *A spanning tree T^* is a minimum spanning tree if and only if it satisfies following condition:*

Path Optimality Condition *For every edge (u, v) not in T^*, there exists a path p in T^*, connecting u and v, and moreover, $c(u, v) \geq c(x, y)$ for every edge (x, y) in path p.*

Proof. Suppose, for contradiction, that $c(u,v) < c(x,y)$ for some edge (x,y) in the path p. Then $T' = (T^* \setminus (x,y)) \cup (u,v)$ is a spanning tree with cost less than $c(T^*)$, contradicting the minimality of T^*.

Conversely, suppose that T^* satisfies the path optimality condition. Let T' be a minimum spanning tree such that among all minimum spanning tree, T' is the one with the most edges in common with T^*. Suppose, for contradiction, that $T' \neq T^*$. We claim that there exists an edge $(u,v) \in T^*$ such that the path in T' between u and v contains an edge (x,y) with length $c(x,y) \geq c(u,v)$. If this claim is true, then $(T' \setminus (x,y)) \cup (u,v)$ is still a minimum spanning tree, contradicting the definition of T'.

Now, we show the claim by contradiction. Suppose the claim is not true. Consider an edge $(u_1,v_1) \in T^* \setminus T'$. the path in T' connecting u_1 and v_1 must contain an edge (x_1,y_1) not in T^*. Since the claim is not true, we have $c(u_1,v_1) < c(x_1,y_1)$. Next, consider the path in T^* connecting x_1 and y_1, which must contain an edge $(u_2,v_2) \notin T'$. Since T^* satisfies the path optimality condition, we have $c(x_1,y_1) \leq c(u_2,v_2)$. Hence, $c(u_1,v_1) < c(u_2,v_2)$. As this argument continues, we will find a sequence of edges in T^* such that $c(u_1,v_2) < c(u_2,v_2) < c(u_3,v_3) < \cdots$, contradicting the finiteness of T^*. □

An algorithm can be designed based on path optimality condition.

Kruskal Algorithm
input: A connected graph $G + (V,E)$ with nonnegative edge weight $c : E \to R_+$.
output: A minimum spanning tree T.

```
Sort all edges e_1, e_2, ..., e_m in nondecreasing order of weight,
i.e., c(e_1) ≤ c(e_2) ≤ ··· ≤ c(e_m);
T ← ∅;
for i ← 1 to m do
    if T ∪ e_i does not contain a cycle
        then T ← T ∪ e_i;
return T.
```

From this algorithm, we see that it is not hard to find the optimal solution for the minimum spanning tree problem. If every combinatorial optimization problem likes the minimum spanning tree, then we would be very happy to final optimal solution for it. Unfortunately, there exist a large number of problems that it is unlikely to be able to compute their optimal solution efficiently. For example, consider following problem.

Problem 1.2.2 (Minimum Length Rectangular Partition)**.** *Given a rectangle with point-holes inside, partition it into smaller rectangles without hole to minimize the total length of cuts.*

Problems 1.1.2 and 1.2.2 are quite different. Problem 1.2.2 is intractable while there exists an efficient algorithm to compute an optimal solution for Problem 1.1.2. Actually, in theory of combinatorial optimization, we need to study not only how to design and analysis of algorithms to find optimal solutions, but also how to design and analysis of algorithms to compute approximation solutions. When do we put our efforts on optimal solution? When should we pay attention to approximation solutions? Ability for making such a judgement has to be growth from study computational complexity.

The book consists of three building blocks, design and analysis of computer algorithm for exact optimal solution, design and analysis of approximation algorithms, and nonlinear combinatorial optimization.

The first block contains six Chapters 2-7, which can be divide into two parts (Fig.1.1). The first part is on algorithms with self-reducibility, includ-

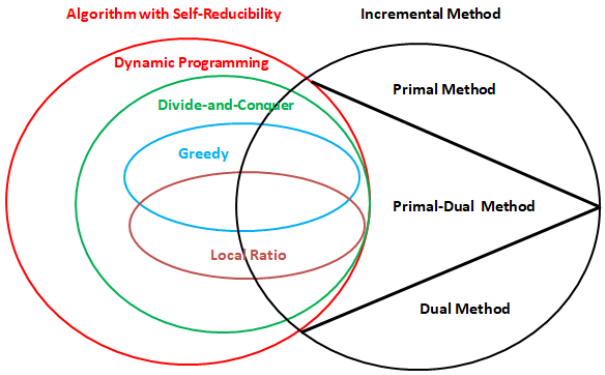


Figure 1.1: Design and Analysis of Computer Algorithms.

ing the divide-and-conquer, the dynamic program, the greedy algorithm, the local search, the local ratio, etc., which are organized into three chapters 2-4. The second part is on incremental method, including the primal algorithm, the dual algorithm, and the primal-dual algorithm, which are organized also into three chapters 5-7. There is an intersection between algorithms with self-reducibility and primal-dual algorithms. In fact, in computation

process of the former, an optimal feasible solution is built up step by step based on certain techniques, and the latter also has a process to build up an optimal primal solution by using information from dual side. Therefore, some algorithm can be illustrated as an algorithm with self-reducibility, and meanwhile it can also be explained as a prima-dual algorithm.

The second block contains four Chapters 8-11, covering the fundamental knowledge on computational complexity, including theory on NP-hardness and inapproximability, and basic techniques for design of approximation including the restriction, the greedy approximation, and the relaxation with rounding.

The third block contains three Chapters 10, 11, 12. Since Chapters 10-11 serve both the second and the third blocks, selected examples are mainly coming from the submodular optimization. Then, Chapter 12 is contributed to the noncubmodular optimization. Nonsubmodular optimization is an active research area currently. There are a lot of recent publications in the literature. Probably, Chapter 12 can be seen an introduction to this area. For a complete coverage, we may need a new book.

Now, we put above structure of this book into Fig.1.2 for a clear overview.

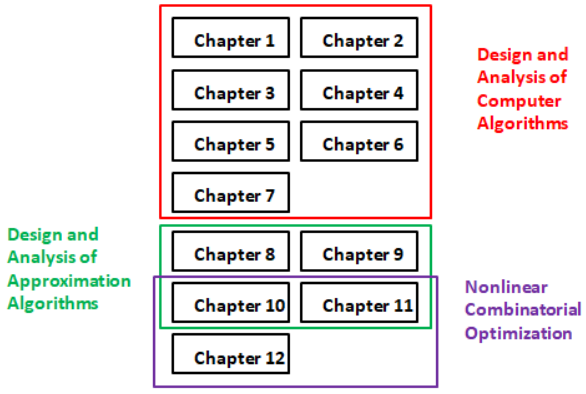


Figure 1.2: Structure of This book.

1.3 Preprocessing

In Kruskal algorithm, the first line is to sort all edges into a nondecreasing order of cost. This requires a preprocessing procedure for solving the sorting problem as follows.

Problem 1.3.1 (Sorting)**.** *Given a sequence of positive integers, sort them into nondecreasing order.*

Following is a simple algorithm to do sorting job.

Insertion Sort
input: An array A with a sequence of positive integers.
output: An array A with a sequence of positive integers in nondecreasing order.

```
for  j ← 2 to  length[A]
     do  key ← A[j]
         i ← j − 1
         while  i > 0 and A[i] > key
                do  A[i + 1] ← A[i]
                    i ← i − 1
         A[i + 1] ← key.
```

An example for using Insertion Sort is as shown in Fig.1.3.

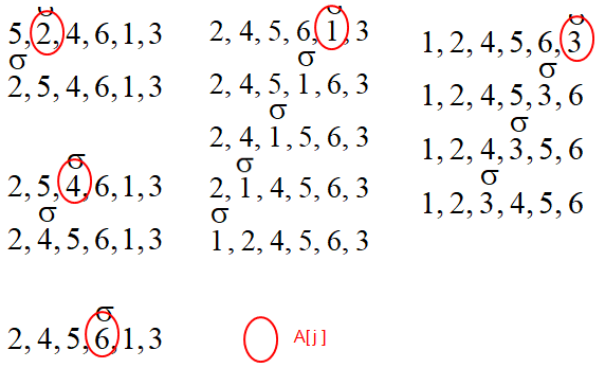


Figure 1.3: An example for Insertion Sort. σ is the *key* lying outside of array A.

Although Insertion Sort is simple, it runs a little slow. Since sorting appears very often in algorithm design for combinatorial optimization problems, we have to spend some space in Chapter 1 to introduce faster algorithms.

1.4 Running Time

The most important measure of quality for algorithms is the running time. However, for the same algorithm, it may take different times when we run it in different computers. To give a uniform standard, we have to get an agreement that run algorithms in a theoretical computer model. This model is the multi-tape Turing machine which has been accepted by a very large of population. Based on Turing machine, theory of computational complexity has been built up. We will touch this part of theory in Chapter 8.

But, we will use RAM model to evaluate the running time for algorithms throughout this book except Chapter 8. In RAM model, assume that each line of pseudocode requires a constant time. For example, the running time of Insertion Sort is calculated in Fig.1.4.

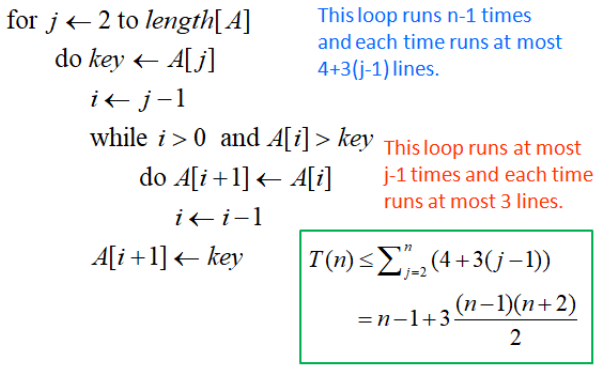


Figure 1.4: Running time calculation.

Actually, RAM model and Turing machine model are closely related. The running time estimated based on these two models is considered to be close enough. However, they are sometimes different in estimation of running time. For example, following is a piece of pseudocode.

for $i = 1$ to n
  do assign First$(i) \leftarrow i$
end-for

According to RAM model, the running time of this piece is $O(n)$. However, based on Turing machine, the running time of this piece is $O(n \log n)$ because the assigned value has to be represented by a string with $O(\log n)$ symbols.

Theoretically, a constant factor is often ignored. For example, we usually say that the running time of Insertion Sort is $O(n^2)$ instead of giving the specific quadratic function with respect to n. Here $f(n) = O(g(n))$ means that there exist constants $c > 0$ and $n_0 > 0$ such that

$$f(n) \leq c \cdot g(n) \text{ for } n \geq n_0$$

There are two more notations which appear very often in representation of running time. $f(n) = \Omega(g(n))$ means that there exist constant $c > 0$ and $n_0 > 0$ such that

$$0 \leq c \cdot g(n) \leq f(n) \text{ for } n \geq n_0.$$

$f(n) = \Theta(g(n))$ means that there exist constants $c_1 > 0$, $c_2 > 0$ and $n_0 > 0$ such that

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for } n \geq n_0.$$

1.5 Data Structure

A data structure is a data storage format which is organized and managed to have efficient access and modification. Each data structure has several standard operations. They are building bricks to construct algorithms. The data structure plays an important role in improving efficiency of algorithms. For example, we may introduce a simple data structure "Disjoint Sets" to improve Kruskal algorithm.

Consider a collection of disjoint sets. For each set S, let First(S) denote the first node in set S. Foe each element x in set S, denote First(x)=First(S). Define three operations as follows.

Make-Set(x) creates a new set containing only x.

Union(x, y) unions sets S_x and S_y containing x and y, respectively, into $S_x \cup S_y$, Moreover, set

$$\text{First}(S_x \cup S_y) = \begin{cases} \text{First}(S_x) & \text{if } |S_x| \geq |S_y|, \\ \text{First}(S_y) & \text{otherwise.} \end{cases}$$

Find-Set(x) returns First(S_x) where S_x is the set containing element x.

With this data structure, Kruskal algorithm can be modified as follows.

Kruskal Algorithm
input: A connected graph $G = (V, E)$ with nonnegative edge weight $c : E \to R_+$.
output: A minimum spanning tree T.

Sort all edges $e_1, e_2, ..., e_m$ in nondecreasing order of weight,
i.e., $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_m)$;
$T \leftarrow \emptyset$;
for each node $v \in V$ **do**
    Make-Set(v);
for $i \leftarrow 1$ **to** m **do**
    if Find-Set(x) $\neq$ Find-Set(y) where $e_i = (x, y)$
    then $T \leftarrow T \cup e_i$
        and Union(x, y);
return T.

An example for running this algorithm is as shown in Fig.1.5.

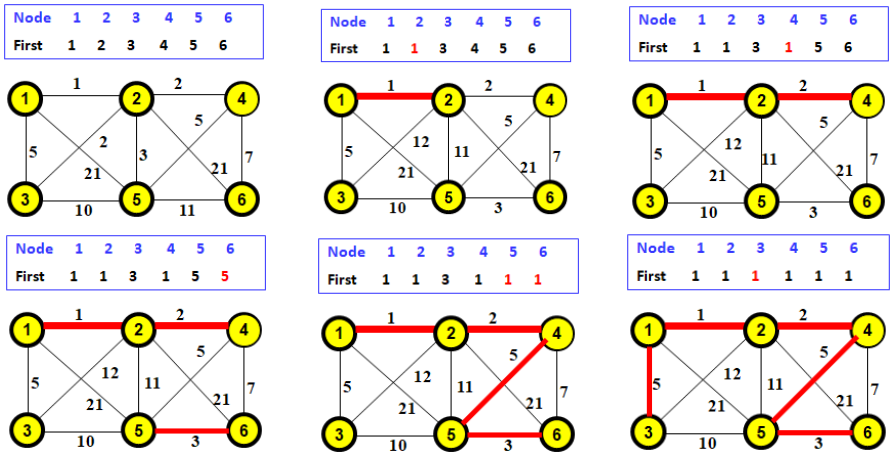


Figure 1.5: An example for Kruskal algorithm.

Denote $m = |E|$ and $n = |V|$. Let us estimate the running time of Kruskal algorithm.

- Sorting on all edges takes $O(m \log n)$ time.
- Assigning First(v) for all $v \in V$ takes $O(n)$ time.

- For each node v, the value of First(v) can be changed at most $O(\log n)$ time. This is because the value of First(v) is changed only if v is involved in Union operation and after the operation, the set containing v has size doubled.

- Thus, the second "for loop takes $O(n \log n)$ time.

- Put total together, The running time is $O(m \log n) = O(m \log n + n \log n)$.

Exercises

1. In a city there are N houses, each of which is in need of a water supply. It costs W_i dollars to build a well at house i, and it costs C_{ij} to build a pipe in between houses i and j. A house can receive water if either there is a well built there or there is some path of pipes to a house with a well. Give an algorithm to find the minimum amount of money needed to supply every house with water.

2. Consider a connected graph G with all distinct edge weights. Show that the minimum spanning tree of G is unique.

3. Consider a connected graph $G = (V, E)$ with nonnegative edge weight $c : E \to R_+$. Suppose $e_1^*, e_2^*, ..., e_k^*$ are edges generated by Kruskal algorithm, and $e_1, e_2, ..., e_k$ are edges of a spanning tree in ordering $c(e_1) \leq c(e_2) \leq \cdots \leq c(e_k)$. Show that $c(e_i^*) \leq c(e_i)$ for all $1 \leq i \leq k$.

4. Let V be a fixed set of n vertices. Consider a sequence of m undirected edges e_1, e_2, ..., e_m. For $1 \leq i \leq m$, let G_i denote the graph with vertex set V and edge set $E_i = \{e_1, ..., e_i\}$. Let c_i denote the number of connected components of G_i. Design an algorithm to compute c_i for all i. Your algorithm should be asymptotically as fast as possible. What is the running time of your algorithm?

5. There are n points lying in the Euclidean plane. Show that there exists a minimum spanning tree on these n points such that every node has degree at most five.

6. Can you modify Kruskal algorithm to compute a maximum weight spanning tree?

7. Consider a connected graph $G = (V, E)$ with edge weight $c : E \to R$, i.e., the weight is possibly negative. Does Kruskal algorithm work for computing a minimum weight spanning tree.

8. Consider a connected graph $G = (V, E)$ with nonnegative edge weight $c : E \to R_+$. Suppose edge e is unique longest edge in a cycle. Show that e cannot be include in any minimum spanning tree.

9. Consider a connected graph $G = (V, E)$ with nonnegative edge weight $c : E \to R_+$. While a cycle exists, delete a longest edge from the cycle. Show that this computation ends at a minimum spanning tree.

Historical Notes

There are many books, which have been written for combinatorial optimization [1, 2, 3, 4, 5, 15, 6, 7]. There are also many books published in design and analysis of computer algorithms [9, 10], which cover a large portion on combinatorial optimization problems. However, those books mainly on computing exact optimal solutions and possibly a small part on approximation solutions. For approximation solutions, a large part of materials are usually covered in separated books [11, 12, 16].

In recent developments of technology, combinatorial optimization gets a lot of new applications [8, 13, 14]. This book tries to meet requests from various areas for teaching, research, and reference, to put together three components, the classic part of combinatorial optimization, approximation theory developed in recent year, and newly appeared nonlinear combinatorial optimization.

Chapter 2

Sorting and Divide-and-Conquer

> "Defeat Them in Detail: The Divide and Conquer Strategy. Look at the parts and determine how to control the individual parts, create dissension and leverage it."
> - `Robert Greene`

Sorting is not a combinatorial optimization problem. However, it appears in algorithms very often as a procedure, especially in algorithms for solving combinatorial optimization problems. Therefore, we would like to start with sorting for introducing an important technique for design of algorithms, divide-and-conquer.

2.1 Algorithms with Self-Reducibility

There exist a large number of algorithms in which the problem is reduced to several subproblems each of which is the same problem on a smaller-size input. Such a problem is said to have the self-reducibility and the algorithm is said to be with self-reducibility.

For example, consider sorting problem again. Suppose input contains n numbers. We may divide these n numbers into two subproblems. One subproblem is the sorting problem on $\lfloor n/2 \rfloor$ numbers and the other subproblem is the sorting problem on $\lceil n/2 \rceil$ numbers. After complete sorting in each subproblem, combine two sorted sequences into one. This idea will result in a sorting algorithm, called *Merge Sort*. The pseudocode of this algorithm is shown in Algorithms 1.

Algorithm 1 Merge Sort.

Input: n numbers $a_1, a_2, ..., a_n$ in array $A[1...n]$.
Output: n numbers $a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}$ in array A.

```
1: Sort(A, 1, n)
2: return A[1...n]
```

Procedure Sort(A, p, r)**.**
% Sort $r - p + 1$ numbers $a_p, a_{p+1}, ..., a_r$ in array $A[p...r]$. %

```
1: if p < r then
2:    q ← ⌊(p + r)/2⌋
3:    Sort(A, p, q)
4:    Sort(A, q + 1, r)
5:    Merge(A, p, q, r)
6: end if
7: return A[p...r]
```

Procedure Merge(A, p, q, r)**.**
% Merge sorted two arrays $A[p...q]$ and $A[p + 1...r]$ into one. %

```
1: for i ← 1 to q − p + 1 do
2:    B[i] ← A[p + i − 1]
3: end for
4: i ← 1
5: j ← p + 1
6: B[q − p + 2] ← +∞
7: A[r + 1] ← +∞
8: for k ← p to r do
9:    if B[i] ≤ A[j] then
10:      A[k] ← B[i]
11:      i ← i + 1
12:   else
13:      A[k] ← A[j]
14:      j ← j + 1
15:   end if
16: end for
17: return A[p...r]
```

The main body calls a procedure. This procedure contains two self calls, which means that the merge sort is a recursive algorithm, that is, the divide will continue until each subproblem has input of single number. Then this procedure employs another procedure (Merge) to combine solutions of subproblems with smaller inputs into subproblems with larger inputs. This computation process on input $\{5, 2, 7, 4, 6, 8, 1, 3\}$ is shown in Fig. 2.1.

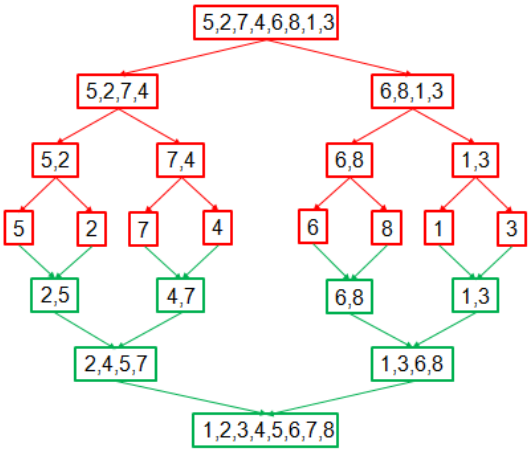


Figure 2.1: Computation process of Merge Sort.

It is easy to estimate that the running time of procedure Merge at each level is $O(n)$. Let $t(n)$ be the running time of merge sort on input of size n. By the recursive structure, we can obtain that $t(1) = 0$ and

$$t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + O(n).$$

Suppose

$$t(n) \leq t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + c \cdot n$$

for some positive constant c. Define $T(1) = 0$ and

$$T(n) = 2 \cdot T(\lceil n/2 \rceil) + c \cdot n.$$

By induction, it is easy to prove that

$$t(n) \leq T(n) \text{ for all } n \geq 1.$$

Now, let us discuss how to solve recursive equation about $T(n)$. Usually, we use two stages. In the first stage, we consider special numbers $n = 2^k$ and employ the recursive tree to find $T(2^k)$ (Fig. 2.2), that is,

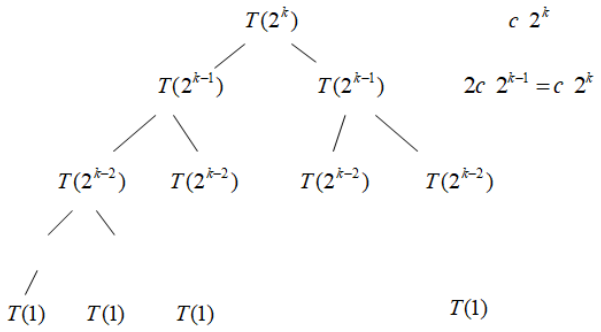


Figure 2.2: Recursive tree.

$$\begin{aligned} T(2^k) &= 2 \cdot T(2^{k-1}) + c \cdot 2^k \\ &= 2 \cdot (2 \cdot T(2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\ &= \ldots \\ &= 2^k T(1) + kc \cdot 2^k \\ &= c \cdot k 2^k. \end{aligned}$$

In general, we may guess that $T(n) \leq c' \cdot n \log n$ for some constant $c' > 0$. Let us show it by mathematical induction.

First, we choose c' to satisfy $T(n) \leq c'$ for $n \leq n_0$ where n_0 will be determined later. This choice will make $T(n) \leq c' n \log n$ for $n \leq n_0$, which meets the requirement for basis step of mathematical induction.

For induction step, consider $n \geq n_0 + 1$. Then we have

$$\begin{aligned} T(n) &= 2 \cdot T(\lceil n/2 \rceil) + c \cdot n \\ &\leq 2 \cdot c' \lceil n/2 \rceil \log \lceil n/2 \rceil) + c \cdot n \\ &\leq 2 \cdot c'((n+1)/2)(\log(n+1) - 1)) + c \cdot n \\ &= c' \cdot (n+1) \log(n+1) - c'(n+1) + c \cdot n \\ &\leq c'(n+1)(\log n + 1/n) - (c' - c)n - c' \\ &= c' n \log n + c' \log n - (c' - c)n + c'/n. \end{aligned}$$

Now, we choose n_0 sufficiently large such that $n/2 > \log n + 1/n$ and $c' > \max(2c, T(1), \ldots, T(n_0))$. Then above mathematical induction proof will be passed. Therefore, we obtained following.

Theorem 2.1.1. *Merge Sort runs in $O(n \log n)$ time.*

By the mathematical induction, we can also prove following result.

Theorem 2.1.2. *Let $T(n) = aT(n/b) + f(n)$ where constants $a > 1$, $b > 1$, and n/b means $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$. Then we have following.*

1. *If $f(n) = O(n^{\log_b a - \varepsilon})$ for some positive constant ε, then $T(n) = \Theta(n^{\log_b a})$.*

2. *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.*

3. *If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some positive constant ε and moreover, $af(n/b) \leq cf(n)$ for sufficiently large n and some constant $c < 1$, then $T(n) = \Theta(f(n))$.*

In Fig. 2.1, we see a tree structure between problem and subproblems. In general, for any algorithm with delf-reducibility, its computational process will produce a set of subproblems on which we can also construct a graph to describe relationship between them by adding an edge from subproblem A to subproblem B if at an iteration, subproblem A is reduced to several problems, including subproblem B. This graph is called the *self-reducibility structure* of the algorithm.

All algorithms with tree self-reducibility structure form a class, called *divide-and-conquer*, that is, **an algorithm is in class of divide-and-conquer if and only if its self-reducibility structure is a tree.** Thus, the merge sort is a divide-and-conquer algorithm.

In a divide-and-conquer algorithm, it is not necessary to divide a problem evenly or almost evenly. For example, we consider another sorting algorithm, called *Quick Sort*. The idea is as follows.

In Merge Sort, the procedure Merge takes $O(n)$ time, which is the main consumption of time. However, if $A[i] \leq A[q]$ for $p \leq i < q$ and $A[q] \leq A[j]$ for $q < j \leq r$, then this procedure can be skipped and after sort $A[p...q-1]$ and $A[q+1...r]$, we can simply put them together to obtain sorted $A[p...r]$.

In order to have above property satisfied, Quick Sort uses $A[r]$ to select all elements $A[p...r-1]$ into two subsequences such that one contains elements less than $A[r]$ and the other one contains elements at least $A[r]$. A pseudocode of Quick Sort is as shown in Algorithm 2.

The division is not balanced in Quick Sort. In the worst case, one part contains nothing and the other contains $r - p$ elements. This will result in running time $O(n^2)$. However, Quick Sort has expected running time

Algorithm 2 Quick Sort.

Input: n numbers $a_1, a_2, ..., a_n$ in array $A[1...n]$.
Output: sorted numbers $a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}$ in array A.

1: Quicksort$(A, 1, n)$
2: **return** $A[1...n]$

Procedure Quicksort(A, p, r)**.**
% Sort $r - p + 1$ numbers $a_p, a_{p+1}, ..., a_r$ in array $A[p...r]$. %

1: **if** $p < r$ **then**
2: $q \leftarrow$ Partition(A, p, r)
3: Quicksort$(A, p, q - 1)$
4: Quicksort$(A, q + 1, r)$
5: **end if**
6: **return** $A[p...r]$

Procedure Partition(A, p, r)**.**
% Find q such that there are $q - p + 1$ elements less that $A[r]$ and others bigger than or equal to $A[r]$ %

1: $x \leftarrow A[r]$
2: $i \leftarrow p - 1$
3: **for** $j \leftarrow p - 1$ to $r - 1$ **do**
4: **if** $A[j] < x$ **then**
5: $i \leftarrow i + 1$ and exchange $A[i] \leftrightarrow A[j]$
6: **end if**
7: exchange $A[i + 1] \leftrightarrow A[r]$
8: **end for**
9: **return** $i + 1$

$O(n \log n)$. To see it, let $T(n)$ denote the running time for n numbers. Note that the procedure Partition runs in linear time. Then, we have

$$\begin{aligned} E[T(n)] &\leq \frac{1}{n}(\qquad\qquad E[T(n-1)] + c_1 n) \\ &\quad + \frac{1}{n}(E[T(1)] + E[T(n-2)] + c_1 n) \\ &\quad + \cdots \\ &\quad + \frac{1}{n}(E[T(n-1)] \qquad\qquad + c_1 n) \\ &= c_1 n + \frac{2}{n}\sum_{i=1}^{n-1} E[T(i)]. \end{aligned}$$

Now, we prove by induction on n that

$$E[T(n)] \leq cn \log n$$

for some constant c. For $n = 1$, it is trivial. Next, consider $n \geq 2$. By induction hypothesis,

$$\begin{aligned} E[T(n)] &\leq c_1 n + \frac{2c}{n}\sum_{i=1}^{n-1} i \log i \\ &= c_1 n + c(n - 1 \log \left(\Pi_{i=1}^{n-1} i^i\right)^{2/(n(n-1))} \\ &\leq c_1 n + c(n-1) \log \frac{1^2 + 2^2 + \cdots + (n-1)^2}{n(n-1)/2} \\ &= c_1 n + c(n-1) \log \frac{2n-1}{3} \\ &\leq c_1 n + cn \log \frac{2n}{3} \\ &= cn \log n + (c_1 - c \log \frac{3}{2})n. \end{aligned}$$

Choose $c \geq c_1 / \log \frac{3}{2}$. We obtain $E[T(n)] \leq cn \log n$.

Theorem 2.1.3. *Expected running time of Quick Sort is $O(n \log n)$.*

2.2 Heap

Heap is a quite useful data structure. Let us introduce it here and by the way, give another sorting algorithm, Heap Sort.

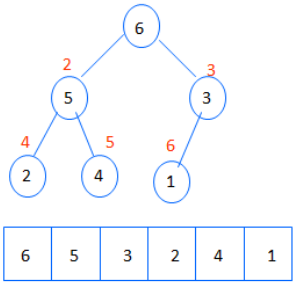


Figure 2.3: A heap.

A heap is a nearly complete binary tree, stored in an array (Fig.2.3). What is *nearly complete binary tree*? It is a binary tree satisfying following conditions.

- Every level other than bottom is complete.
- On the bottom, nodes are placed as left as possible.

For example, binary trees in Fig.2.4 are not nearly complete. An advantage

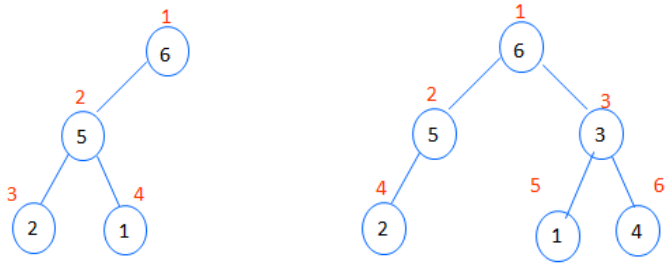


Figure 2.4: They are not nearly complete.

of nearly complete binary tree is easy to operate on it. For example, for node i (i.e., a node with address i), its parent, left-child, and right-child can be easily figured out as follows:

Parent(i)
    return $\lfloor i/2 \rfloor$.

Left(i)
    return $2i$.

Right(i)
return $2i + 1$.

Thee are two types of heaps with special properties, respectively.

Max-heap: For every node i other than root, $A[\text{Parent}(i)] \geq A[i]$.

Min-heap: For every node i other than root, $A[\text{Parent}(i)] \leq A[i]$.

Max-heap has two special operations: Max-Heapify and Build-Max-Heap. We describe them as follows.

When operation Max-Heapify(A, i) is called, two subtree rooted at Left(i) and Right(i) are max-heaps, but $A[i]$ may not satisfy the max-heap property. Max-Heapify(A, i) makes the subtree rooted at $A[i]$ becomes a max-heap by moving $A[i]$ downside. An example is as shown in Fig.2.5.

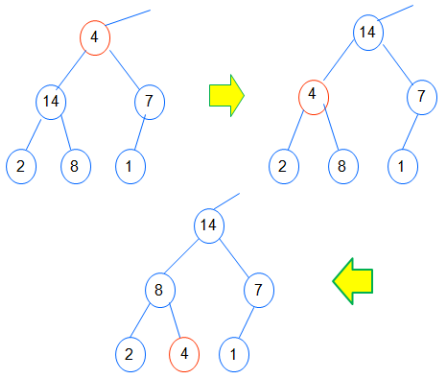


Figure 2.5: An example for Max-Heapify(A, i).

Following is algorithmic description for this operation.

Max-Heapify(A, i)
if Left(i) $\geq$Right(i) and Left$(i) > A[i]$
then Exchange $A[i]$ and Left(i)
Max-Heapify$(A, \text{Left}(i))$
if Left(i) $<$Right(i) and Right$(i) > A[i]$
then Exchange $A[i]$ and Right(i)
Max-Heapify$(A, \text{Right}(i))$;

Operation Build-Max-Heap applies to a heap and make it become a max-heap, which can be described as follows. (Note that Parent$(size[A]) = \lfloor size[A]/2 \rfloor$.)

Build-Max-Heap(A)
  for $i \leftarrow \lfloor size[A]/2 \rfloor$ down to 1
    do Max-Heapify(A, i);

An example is as shown in Fig.2.6.

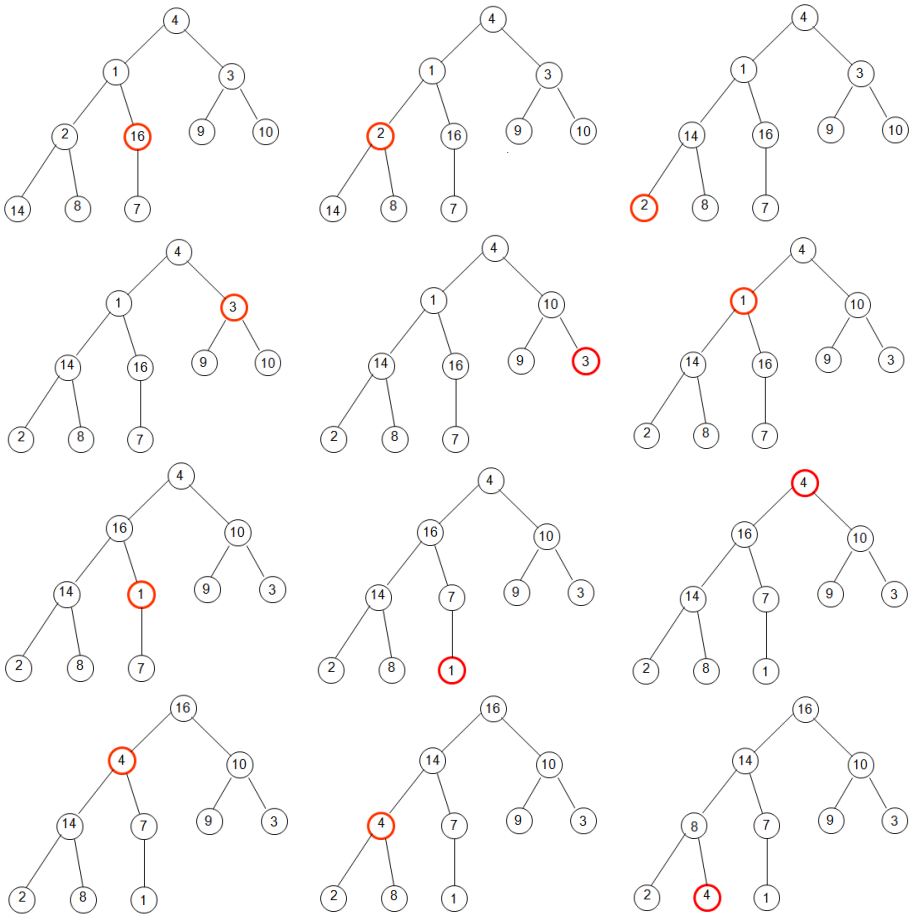


Figure 2.6: An example for Build-Max-Heap(A).

It is interesting to estimate the running time of this operation. Let h be the height of heap A. Then $h = \lfloor \log_2 n \rfloor$. At level i, A has 2^i nodes, at each of which Max-Heapify spends at most $h - i$ steps to float down. Therefore,

the running time of Build-Max-Heap(A) is

$$\begin{aligned} O(\sum_{i=0}^{h} 2^i(h-i)) &= O(2^h \sum_{i=0}^{h} \frac{h-i}{2^{h-i}}) \\ &= O(2^h \sum_{i=0}^{h} \frac{i}{2^i}) \\ &= O(n). \end{aligned}$$

Algorithm 3 Heap Sort.

Input: n numbers $a_1, a_2, ..., a_n$ in array $A[1...n]$.
Output: n numbers $a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}$ in array A.

1: Build-Max-Heap(A)
2: **for** $i \leftarrow n$ down to 2 **do**
3: exchange $A[1] \leftrightarrow A[i]$
4: $heap - size[A] \leftarrow i - 1$
5: Max-Heapify$(A, 1)$
6: **end for**
7: **return** $A[1...n]$

Nowas shown in Algorithm 3, a sorting algorithm can be designed with max-heap. Initially, build a max-heap A. In each subsequential steps, the algorithm first exchange $A[1]$ and $A[heap-size(A)]$, and then reduce $heap-size(A)$ by 1, meanwhile with Max-Heapify$(A, 1)$ to recover the max-heap. An example is as shown in Fig.2.7.

Since the number of steps is $O(n)$ and Max-Heapify$(A, 1)$ takes $O(\log n)$ time, the running time of Heap Sort is $O(n \log n)$.

Theorem 2.2.1. *Heap Sort runs in $O(n \log n)$ time.*

We already have two sorting algorithms with $O(n \log n)$ running time and one sorting algorithm with expected $O(n \log n)$ running time. But, there is no sorting algorithm with running time faster than $O(n \log n)$. Is $O(n \log n)$ a barrier of running time for sorting algorithm? In some sense, the answer is yes. All sorting algorithms presented previously belong to a class, called *comparison sort*.

In comparison sort, order information about input sequence can be obtained only by comparison between elements in the input sequence. Suppose input sequence contains n positive integers. Then there are $n!$ possible permutations. The aim of sorting algorithm is to determine a permutation

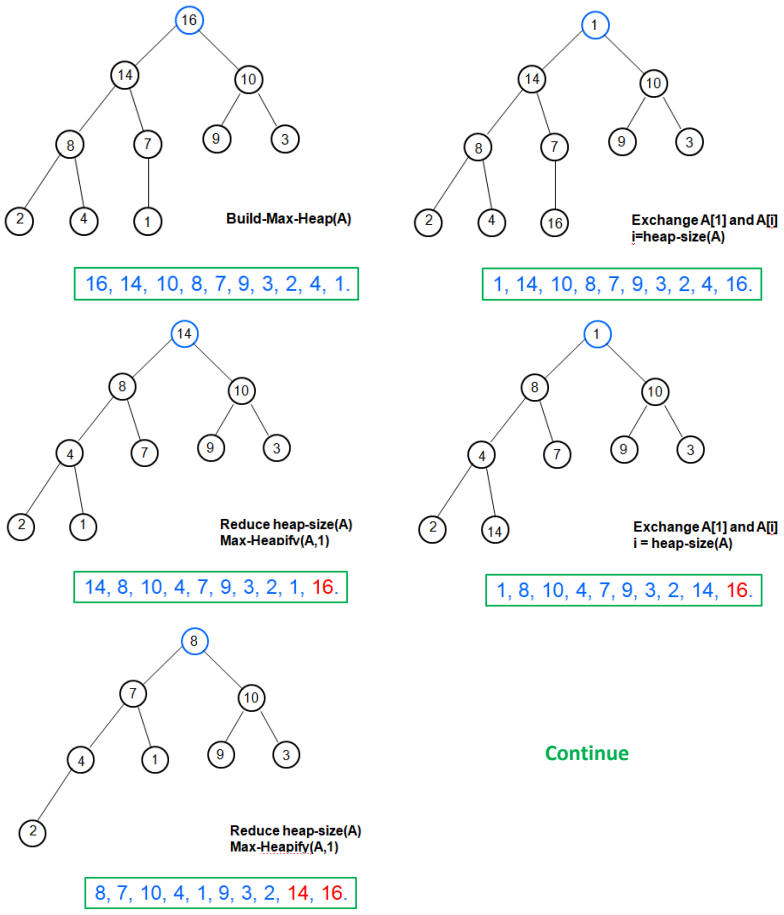


Figure 2.7: An example for Heap Sort.

which gives nondecreasing order. Each comparison divides the set of possible permutations into two subsets. The comparison result tells which subset contains a nondecreasing order. Therefore, every comparison sort algorithm can be represented by a binary decision tree (Fig.2.8). The (worst case) running time of the algorithm is the height (or depth) of the decision tree.

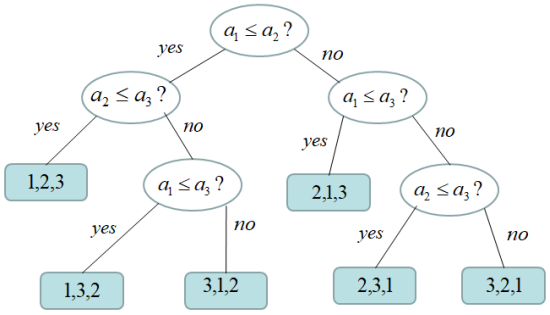


Figure 2.8: Decision tree.

Since the binary decision has $n!$ leaves, its height $T(n)$ satisfies

$$1 + 2 + \cdots + 2^{T(n)} \geq n!$$

that is,

$$2^{T(n)+1} - 1 \geq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

Thus,

$$T(n) = \Omega(n \log n).$$

Therefore, no comparison sort can do better than $O(n \log n)$.

Theorem 2.2.2. *The running time of any comparison sort is* $\Omega(n \log n)$.

2.3 Counting Sort

To break the barrier of running time $O(n \log n)$, one has to design a sorting algorithm without using comparison. Counting sort is such an algorithm.

Algorithm 4 Counting Sort.

Input: n numbers $a_1, a_2, ..., a_n$ in array $A[1...n]$.
Output: n numbers $a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}$ in array B.

```
1: for i ← 1 to k do
2:    C[i] ← 0
3: end for
4: for j ← 1 to n do
5:    C[A[j]] ← C[A[j]] + 1
6: end for
7: for i ← 2 to k do
8:    C[i] ← C[i] + C[i − 1]
9: end for
10: for j ← n down to 1 do
11:    B[C[A[j]]] ← A[j]
12:    C[A[j]] ← C[A[j]] − 1
13: end for
14: return B[1...n]
```

Let us use an example to illustrate Counting Sort as shown in Algorithm 4. This algorithm involves three arrays A, B, and C. Array A contains input sequence of positive integers. Suppose $A = \{4, 6, 5, 1, 4, 5, 2, 5\}$. Let k be the largest integer in input sequence. Initially, the algorithm makes preprocessing on array C in three stages.

1. Clean up array C.

2. For $1 \leq i \leq k$, assign $C[i]$ with the number of i's appearing in array A. (In the example, $C = \{1, 1, 0, 2, 3, 1\}$ at this stage.)

3. Update $C[i]$ such that $C[i]$ is equal to the number of integers with value at most i appearing in A. (In the example, $C = \{1, 2, 2, 4, 7, 8\}$ at this stage.)

With help of array C, the algorithm move element $A[j]$ to array B for $j = n$ down to 1, by

$$B[C[A[j]]] \leftarrow A[j]$$

and then update array C by

$$C[A[j]] \leftarrow C[A[j]] - 1.$$

This part of computation about the example is as follows.

C	1	2	2	4	7	8		
A	4	6	5	1	4	5	2	$\hat{5}$
B							5	

C	1	2	2	4	6	8		
A	4	6	5	1	4	5	$\hat{2}$	5
B		2					5	

C	1	1	2	4	6	8		
A	4	6	5	1	4	$\hat{5}$	2	5
B		2				5	5	

C	1	1	2	4	5	8		
A	4	6	5	1	$\hat{4}$	5	2	5
B		2		4		5	5	

C	1	1	2	3	5	8		
A	4	6	5	$\hat{1}$	4	5	2	5
B	1	2		4		5	5	

C	0	1	2	3	5	8		
A	4	6	$\hat{5}$	1	4	5	2	5
B	1	2		4	5	5	5	

C	0	1	2	3	4	8		
A	4	$\hat{6}$	5	1	4	5	2	5
B	1	2		4	5	5	5	6

C	0	1	2	3	4	7		
A	$\hat{4}$	6	5	1	4	5	2	5
B	1	2	4	4	5	5	5	6

Now, let us estimate the running time of Counting Sort.

Theorem 2.3.1. *Counting Sort runs in $O(n+k)$ time.*

Proof. The loop at line 1 takes $O(k)$ time. The loop at line 4 takes $O(n)$ time. The loop at line 7 takes $O(k)$ time. The loop at line 10 takes $O(n)$ time. Putting all together, the running tine is $O(n+k)$. □

A student found a simple way to improve Counting Sort. Let consider the same example. At the 2nd stage, $C = \{1, 1, 0, 2, 3, 1\}$ where $C[i]$ is equal to the number of i's appearing in array A. The student found that with this array C, array B can be put in integers immediately without array A.

C	1	1	0	2	3	1		
B	1							
B	1	2						
B	1	2	4	4				
B	1	2	4	4	5	5	5	
B	1	2	4	4	5	5	5	6

Is this method acceptable? The answer is no. Why not? Let us explain.

First, we should note that those numbers in input sequence may come from labels of objects. The same numbers may come from different objects. For example, consider a sequence of objects $\{329, 457, 657, 839, 436, 720, 355\}$. If we use their the first digits from left as labels, then we will obtain a sequence $\{9, 7, 7, 9, 6, 0, 5\}$. When apply Counting Sort on this sequence, we will obtain a sequence $\{720, 355, 436, 457, 657, 329, 839\}$. This is because label gets moved together with its object in Counting Sort.

Moreover, consider two objects 329 and 839 with the same label 9. In input sequence, 329 lies on the left-side of 839. After Counting Sort, 329 lies still on the left-side of 839.

A sorting algorithm is *stable* if for different objects with the same label, after labels are sorted, the ordering of objects in output sequence is the same as their ordering in input sequence. Following can be proved easily.

Lemma 2.3.2. *Counting Sort is stable.*

The student's method cannot keep stable property.

With stable property, we can use Counting Sort in following way. Remember, after sort the leftmost digit, we obtain sequence

$$\{720, 355, 436, 457, 657, 329, 839\}.$$

Now, we continue to sort this sequence based on the second leftmost digit. Then we will obtain sequence

$$\{720, 329, 436, 839, 355, 457, 657\}.$$

Continue to sort based on the rightmost digit, we will obtain sequence

$$\{329, 355, 436, 457, 657, 720, 839\}.$$

Now, let us use this technique to solve a problem.

Example 2.3.3. *There are n integers between 0 and $n^2 - 1$. Design a algorithm to sort them. The algorithm is required to run in $O(n)$ time.*

Each integer between 0 and $n^2 - 1$ can be represented as

$$an + b \text{ for } 0 \leq a \leq n-1, 0 \leq b \leq n-1.$$

Apply Counting Sort first to b and then to a. Each application takes $O(n) = O(n+k)$ time since $k = n$. Therefore, total time is still $O(n)$.

In general, suppose there are n integers, each of which can be represented in form

$$a_d k^d + a_{d-1} 1 k^{d-1} + \cdots + a_0$$

where $0 \leq a_i \leq k-1$ for $0 \leq i \leq d$. Then we can sort these n integers by using Counting Sort first on a_0, second on a_1, ..., finally on a_d. This method is called *Radix Sort*.

Theorem 2.3.4. *Radix Sort takes $O(d(n+k))$ time.*

2.4 Examples

Let us study some examples with divide-and-conquer technique and sorting algorithms.

Example 2.4.1 (Maximum Consecutive Subsequence Sum)**.** *Given a sequence of n integers, find a consecutive subsequence with maximum sum.*

Divide input sequence S into two subsequence S_1 and S_2 such that $|S_1| = \lfloor n/2 \rfloor$ and $|S_2| = \lceil n/2 \rceil$. Let $Max - Sub(S)$ denote the consecutive subsequence of S with maximum sum. Then there are two cases.

Case 1. $Max - Sub(S)$ is contained in either S_1 or S_2. In this case, $Max - Sub(s) = Max - Sub(S_1)$ or $Max - Sub(s) = Max - Sub(S_2)$.

Case 2. $Max - Sub(S) \cap S_1 \neq \emptyset$ and $Max - Sub(S) \cap S_2 \neq \emptyset$. In this case, $Max - Sub(S) \cap S_1$ is the tail subsequence with maximum sum. That is, suppose $S_1 = \{a_1, a_2, ..., a_p\}$. Then among subsequences $\{a_p\}$, $\{a_{p-1}, a_p\}$, ..., $\{a_1, ..., a_p\}$, $Max - Sub(S) \cap S_1$ is the one with maximum sum. Therefore, it can be find in $O(n)$ time. Similarly, $Max - Sub(S) \cap S_2$ is the head subsequence with maximum sum, which can be computed in $O(n)$ time.

Suppose $Max - Sub(S)$ can be computed in $T(n)$ time. Summarized from above two cases, we obtain

$$T(n) = 2T(\lceil n/2 \rceil) + O(n).$$

Therefore, $T(n) = O(n \log n)$.

Example 2.4.2 (Closest Pair of Points)**.** *Given n points in the Euclidean plane, find a pair of points to minimize the distance between them.*

Initially, we may assume that all n points have distinct x-coordinates since, if not, we may rotate the coordinate system a little.

Now, divide all points into two half parts based on x-coordinates. Find the closest pair of points in each part. Suppose δ_1 and δ_2 are distances of closest pairs in two parts, respectively, Let $\delta = \min(\delta_1, \delta_2)$. We next study if there is a pair of points lying in both parts, respectively and with distance less than δ.

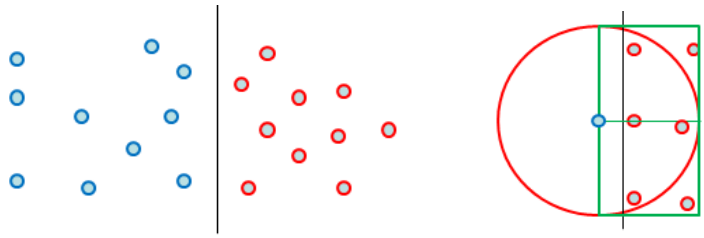


Figure 2.9: Closest pair of points.

For each point $u = (x_u, y_u)$ in the left part (Fig.2.10), consider the rectangle $R_u = \{(x, y) \mid x_u \leq x \leq x_u + \delta, y_u - \delta \leq y \leq y_u + \delta\}$. It has following properties.

- Every point in the right part and within distance δ from u lies in this rectangle.
- This rectangle contains at most six points in the right part because every two points have distance at least δ.

For each u in the left part, check every point v lying in R_u, if distance $d(u, v) < \delta$. If yes, then we keep the record and choose the closest pair of points from them, which should be the solution. If not, then the solution should either the closest pair of points in the left part or the closest pair of points in the right part.

Let $T(n)$ be the time for finding the closest pair of points from n points. Above method give a recursive relation

$$T(n) = 2T(\lceil n/2 \rceil) + O(n).$$

Therefore, $T(n) = O(n \log n)$.

Example 2.4.3 (The ith Smallest Number)**.** *Given a sequence of n distinct numbers and a positive integer i, find ith smallest number in $O(n)$ time.*

This algorithm consists of five steps. Let us name this algorithm as $A(n, i)$ for convenience of recursive call.

Step 1. Divide n numbers into $\lceil n/5 \rceil$ groups of 5 elements, possibly except the last one of less than 5 elements (Fig.2.10).

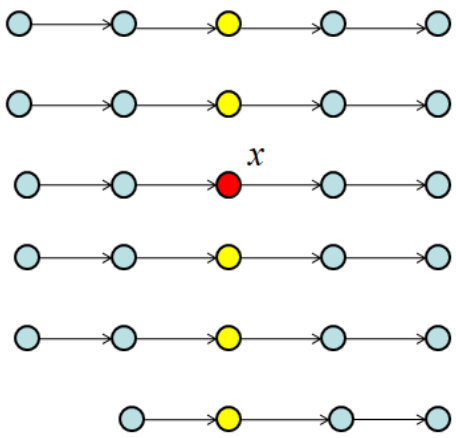


Figure 2.10: x is selected through first three steps.

Step 2. Find the median of each group by Merge Sort. Possibly, for last group, there are two median; in such a case, take the smaller one (Fig.2.10).

Step 3. Make a recursive call $A(\lceil n/5 \rceil, \lceil \lceil n/5 \rceil /2 \rceil)$. This call will find the median x of $\lceil n/5 \rceil$ group median, and moreover, will select the smaller one in case that two candidates of x exist (Fig 2.10).

Step 4. Exchange x with the last element in input array and partition all numbers into two parts by using Partition procedure in Quick Sort. One part (on the left) contains numbers less than x and the other part (on the right) contains numbers larger than x (Fig.2.11).

Step 5. Let k be the number of elements in the left part (Fig.2.11). If $k = i - 1$, then x is the ith smallest number. If $k \geq i$, then the ith smallest number lies on the left of x and hence make a recursive call $A(k, i)$. If $k \leq i-2$, then the ith smallest number lies in the right of x and hence make a recursive call $A(n - k - 1, i - k - 1)$.

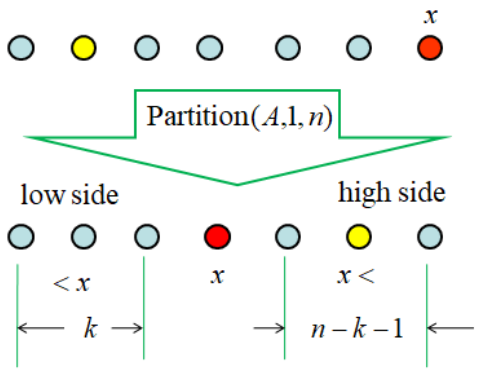


Figure 2.11: x is selected through first three steps.

Now, let us analyze this algorithm. Let $T(n)$ be the running time of $A(n, i)$.

- Steps 1 and 2 take $O(n)$ time.
- Step 3 takes $T(\lceil n/5 \rceil)$ time.
- Step 4 takes $O(n)$ time.
- Step 5 takes $T(\max(k, n-k-1))$ time.

Therefore,

$$T(n) = T(\lceil n/5 \rceil) + T(\max(k, n-k-1)) + O(n).$$

We claim that

$$\max(k, n-k-1) \leq n - (3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2).$$

In fact, as shown in Fig.2.12,

$$k+1 = 3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil$$

and

$$n-k \geq 3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2.$$

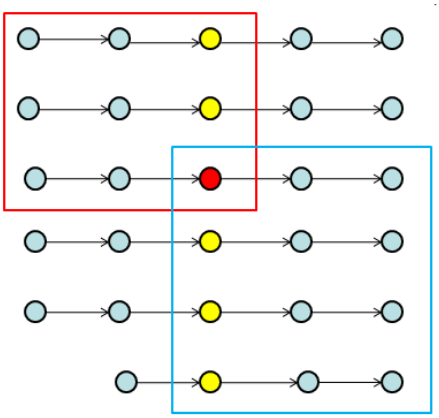


Figure 2.12: Estimation of $k+1$ and $n-k$.

Therefore,

$$n-k-1 \leq n-3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil$$

and

$$k \leq n-(3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil -2).$$

Note that

$$n-(3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil -2) \leq n-(\frac{3n}{10}-2) \leq \frac{7n}{10}+2.$$

By the claim,

$$T(n) \leq T(\lceil n/5\rceil)+T(\frac{7n}{10}+2)+c'n$$

for some constant $c'>0$. Next, we show that

$$T(n) \leq cn \tag{2.1}$$

for some constant $c>0$. Choose

$$c=\max(20c', T(1), T(2)/2, ..., T(59)/59).$$

Therefore, (2.1) holds for $n \leq 59$. Next, consider $n \geq 60$. By induction hypothesis, we have

$$\begin{aligned} T(n) &\leq c(n/5+1)+c(7n/10+2)+c'n \\ &\leq cn-(cn/10-3c-c'n) \\ &\leq cn \end{aligned}$$

since

$$c(n/10 - 3) \geq n/20 \geq c'n.$$

The fist inequality is due to $n \geq 60$ and the second one is due to $c \geq 20c'$. This ends the proof of $T(n) = O(n)$.

Exercises

1. Use a recursion tree to estimate a good upper bound on the recurrence $T(n) = 3T(\lfloor n/2 \rfloor) + n$ and $T(1) = 0$. Use the mathematical induction to prove correctness of your estimation.

2. Draw the recursion tree for $T(n) = 3T(\lfloor n/2 \rfloor) + cn$, where c is a positive constant, and guest an asymptotic upper bound on its solution. Prove your bound by mathematical induction.

3. Show that for input sequence in decreasing order, the running time of Quick Sort is $\Theta(n^2)$.

4. Show that Counting Sort is stable.

5. Find an algorithm to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

6. Let $A[1 : n]$ be an array of n distinct integers sorted in increasing order. (Assume, for simplicity, that n is a power of 2.) Give an $O(\log n)$-time algorithm to decide if there is an integer i, $1 \leq i \leq n$, such that $A[i] = i$.

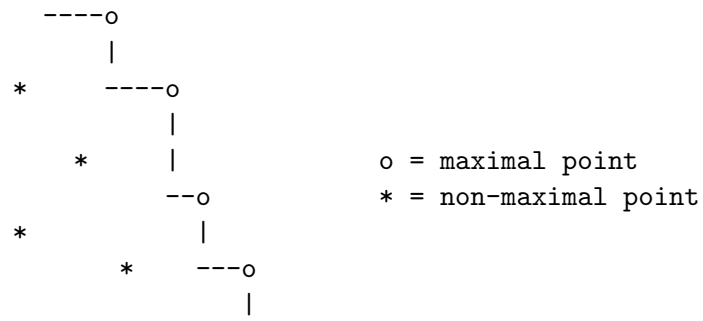
7. Given an array A of integers, please return an array B such that $B[i] = |\{A[k] \mid k > i \text{ and } A[k] < A[i]\}|$.

8. Given a string S and an integer $k > 0$, find a longest substring of s such that each symbol appears at least k times if it appears in the substring.

9. Given an integer array A, please compute the number of pairs $\{i, j\}$ with $A[i] > 2 \cdot A[j]$.

10. Given a sorted sequence of distinct nonnegative integers, find the smallest missing number.

11. Let S be a set of n points, $p_i = (x_i, y_i), 1 \leq i \leq n$, in the plane. A point $p_j \in S$ is a *maximal point of* S if there is no other point $p_k \in S$ such that $x_k \geq x_j$ and $y_k \geq y_j$. The figure below illustrates the maximal points of a point-set S. Note that the maximal points form a "staircase" which descends rightwards.



    Give an efficient divide–and–conquer algorithm to determine the maximal points of S.

12. Given two sorted sequences with m, n elements, respectively, design and analyze an efficient divide-and-conquer algorithm to find the kth element in the merge of the two sequences. The best algorithm runs in time $O(\log(\max(m, n)))$.

13. Design a divide-and-conquer algorithm for the following longest ascending subsequence problem: Given an array $A[1..n]$ of natural numbers, find the length of the longest ascending subsequence. (A subsequence is a list $A[i_1]$, $A[i_2]$, ..., $A[i_m]$ where m is the length.)

14. Show that in a max-heap of length n, the number of nodes rooted at which the subtree has height h is at most $\lceil \frac{n}{2^{h+1}} \rceil$.

15. Let A be an $n \times n$ matrix of integers such that each row is strictly increasing from left to right and each column is strictly increasing from top to bottom. Given an $O(n)$-time algorithm for finding whether a given number x is an element of A, i.e., whether $x = A(i, j)$ for some i, j.

16. The maximum subsequence sum problem is defined as follows: Given an array $A[1..n]$ of integer numbers, find values of i and j with $1 \leq i \leq j \leq n$ such that $\sum_{k=i}^{j} A[i]$ is maximized. Design a divide-and-conquer algorithm for solving the maximum subsequence sum problem in time $O(n \log n)$.

17. Design a divide-and-conquer algorithm for multiplying n complex numbers using only $3(n-1)$ real multiplications.

Historical Notes

Divide-and-conquer is a popular technique for algorithm design. It has a special case, decrease-and-conquer. In decrease and conquer, the problem is reduced to single subproblem. Both divide-and-conquer and decrease-and-conquer have a long history. Their stamps can be found in many earlier works, such as Gauss's work Fourier transform in 1850 [17], John von Neumann's work on Merge Sort in 1945 [18], and John Mauchly's work in 1946 [18]. Quick Sort was developed by Tony Hoare in 1959[19] (publishedin1962 [20]). Counting Sort and its applications to Radix Sort were found by Harold H. Seward in 1954 [9, 18, 21].

Chapter 3

Dynamic Programming and Shortest Path

"The art of programming is the art of organizing complexity."
- Edsger Dijkstra

A divide-and-conquer algorithm consists of many itrations. Usually, each iteration contains three steps. In the first step (called the divide step), divide the problem into smaller subproblems. In the second step (called conquer step), solve those subproblems. In the third step (called the combination step), combine solutions for subproblems into a solution for the original problem. **Is it true that every algorithm with each iteration consisting of above three steps belongs to the class of divide-and-conquer?** The answer is No. In this chapter, we would like to introduce a class of algorithms, called *dynamic programming*. Every algorithm in this class consists of discrete iterations each of which contains the divide step, the conquer step and the combination step. However, they may not be the divide-and-conquer algorithms. Actually, their self-reducibility structure may not be a tree.

3.1 Dynamic Programming

Let us first study several examples and start from simpler one.

Example 3.1.1 (Fibonacci Number)**.** *Fibonacci number F_i for $i = 0, 1, ...$ is defined by*

$$F_0 = 0, F_1 = 1, \text{ and } F_i = F_{i-1} + F_{i-2}.$$

The computational process can be considered as a dynamic programming with self-reducibility structure as shown in Fig. 3.1.

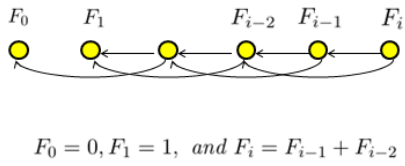


Figure 3.1: Fibonacci numbers.

Example 3.1.2 (Labeled Tree)**.** *Let $a_1, a_2, ..., a_n$ be a sequence of positive integers. A* labeled tree *for this sequence is a binary tree T of n leaves named $v_1, v_2, ..., v_n$ from left to right. We label v_i by a_i for all i, $1 \leq i \leq n$. Let D_i be the length of the path from v_i to the root of T. The cost of T is defined by*

$$cost(T) = \sum_{i=1}^{n} a_i D_i.$$

The problem is to construct a labeled tree T to minimize the cost $cost(T)$ for given sequence of positive integers $a_1, a_2, ..., a_n$.

Let $T(i, j)$ be the optimal labeled tree for subsequence $\{a_i, a_{i+1}, ..., a_j\}$ and $sum(i, j) = a_i + a_{i+1} + \cdots + a_j$. Then

$$cost(T(i, j)) = \min i \leq k < j\{cost(T(i, k)) + cost(T(k + 1, j))\} + sum(i, j)$$

where

$$sum(i, j) = \begin{cases} a_i & \text{if } i = j \\ a_i + sum(i + 1, j) & \text{if } i < j. \end{cases}$$

As shown in Fig.3.2, there are $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$ subproblems $T(i, j)$ in the table. From recursive formula, it can be seen that solution of each subproblem $T(i, j)$ can be computed in $O(n)$ time. Therefore, this dynamic programming runs totally in $O(n^3)$ time.

Actually, the running time of a dynamic programming is often estimated by following formula:

running time = (number of subproblems) $\times$ (computing time of recursion).

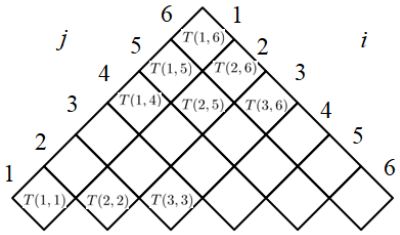


Figure 3.2: The table of subproblems $T(i, j)$.

There are two remarks on this formula: (1) There are some exceptional cases. We will see one in next section. (2) The divide-and conquer can be considered as a special case of the dynamic programming. Therefore, its running time can also be estimated with this formula. However, the outcome is usually too rough.

It is similar to the divide-and-conquer that there are two ways to write software codes for the dynamic programming. The first way is to employ recursive call as shown in Algorithm 5. The second way is as shown in Algorithm 6 which saves the recursive calls and hence in practice, it runs faster with smaller space requirement.

Before study next example, let us first introduce a concept, guillotine cut. Consider a rectangle P, a cut on P is called a *guillotine cut* if it cuts P into two parts. A guillotine partition is a sequence of guillotine cuts.

Example 3.1.3 (Minimum Length Guillotine Partition)**.** *Given a rectangle with point-holes inside (Fig. 3.3), partition it into smaller rectangles without hole inside by a sequence of guillotine cuts to minimize the total length of cuts.*

Example 3.1.3 is a geometric optimization problem. It has infinitely many feasible solutions. Therefore, strictly speaking, it is not a combinatorial optimization problem. However, it can be reduced to a combinatorial optimization problem.

Lemma 3.1.4 (Canonical Partition)**.** *There exists a minimum length guillotine partition such that every guillotine cut passes through a point-hole.*

Proof. Suppose there exists a guillotine cut AB not passing through any point-hole. Without loss of generality, assume that AB is a vertical cut.

Algorithm 5 Algorithm for Labeled Tree.

Input: A sequence of positive integers $a_1, a_2, ..., a_n$.
Output: Minimum cost of a labeled tree.

return $cost(T(1,n))$.

function $cost(T(i,j))$ $(i \leq j)$

if $i = j$ **then**
  $tempt \leftarrow a_i$
else
  $temp \leftarrow +\infty$
  for $k = i$ to $j-1$ **do**
    $temp \leftarrow \min(temp, cost(T(i,k)) + cost(T(k+1,j)) + sum(i,j))$
  end for
end if
return $cost(T(i,j)) \leftarrow temp$;

function $sum(i,j)$ $(i \leq j)$

if $i = j$ **then**
  return $sum(i,j) \leftarrow a_i$
else
  return $sum(i,j) \leftarrow a_i + sum(i+1,j)$
end if

Algorithm 6 Algorithm for Labeled Tree.

Input: A sequence of positive integers $a_1, a_2, ..., a_n$.
Output: Minimum cost of a labeled tree.

for $i = 1$ to n **do**
  $cost(T(i,i)) \leftarrow a_i$; $sum(i,i) \leftarrow a_i$
end for
for $l = 2$ to n **do**
  for $i = 1$ to $n - l + 1$ **do**
    $j \leftarrow i + l - 1$
    $cost(T(i,j)) \leftarrow +\infty$; $sum(i,j) \leftarrow sum(i,j-1) + a_j$
    for $k = i$ to $j-1$ **do**
      $q \leftarrow cost(T(i,k)) + cost(T(k+1,j)) + sum(i,j)$
      $cost(T(i,j)) \leftarrow \min(cost(T(i,j)), q)$
    end for
  end for
end for
return $cost(T(1,n))$

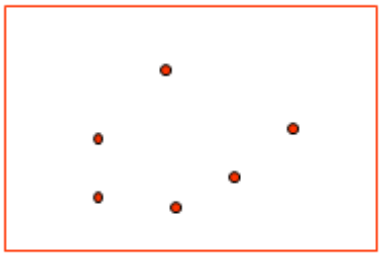


Figure 3.3: A rectangle with point-holes inside.

Let n_1 be the number of guillotine cuts touching AB on the left and n_2 the number of guillotine cuts touching AB on the right. Without loss of generality, assume $n_1 \geq n_2$. Then we can move AB to the right without increasing the total length of rectangular guillotine partition, until meet a point-hole. If this moving cannot meet a point-hole, then AB can be moved to meet another vertical cut or vertical boundary and in either case, AB can be deleted, contradicting the optimality of the partition. □

By Lemma 3.1.4, we may consider only canonical guillotine partitions. During the canonical guillotine partition, each subproblem can be determined by a rectangle in which each boundary edge is obtained by a guillotine cut or a boundary edge of given rectangle and hence there are $O(n)$ possibility. This implies that the number of subproblems is $O(n^4)$.

To find an optimal one, let us study a guillotine cut on a rectangle P. Let n be the number of point-holes. Since the guillotine cut passes a point-hole, there are at most $2n$ possible positions. Suppose P_1 and P_2 are two rectangles obtained from P by the guillotine cut. Let $opt(P)$ denote the minimum total length of guillotine partition on P. Then we have

$$opt(P) = \min_{\text{candidate cuts}} [opt(P_1) + opt(P_2) + (\text{cut length})],$$

The computation time for this recurrence is $O(n)$. Therefore, the optimal rectangular guillotine partition can be computed by a dynamic programming in $O(n^5)$ time.

One of important technique for design of dynamic programming for a given problem is to replace the original problem by a proper one which can be easily found to have a self-reducibility. Following is such an example.

Example 3.1.5. *Consider a horizontal strip. There are n target points lying inside and m unit disks with centers lying outside of the strip where each unit disk d_i has radius one and a positive weight $w(d_i)$. Each target point is covered by at least one unit disk. The problem is to find a subset of unit disks, with minimum total weight, to cover all target points.*

First, without loss of generality, assume all target points have distinct x-coordinates; otherwise, we may rotate the strip together with coordinate system a little to reach such a property. Line up all target points $p_1, p_2, ..., o_n$ in the increasing ordering of x-coordinate. Let $\mathcal{D}_a$ be the set of all unit disks with centers lying above the strip and $\mathcal{D}_b$ the set of all unit disks with centers lying below the strip. Let ℓ_1, ℓ_2, ..., ℓ_n be vertical lines passing through p_1, p_2, ..., p_n, respectively. For any two disks $d, d' \in \mathcal{D}_a$, define $d \prec_i d'$ if the lowest intersection between the boundary of disk d and ℓ_i is not lower than the lowest intersection between the boundary of disk d' and ℓ_i. Similarly, for ny two sensors $d, d' \in \mathcal{D}_b$, define $d \prec_i d'$ if the highest intersection between the boundary of disk d and ℓ_i is not higher than the highest intersection between the boundary of disk d' and ℓ_i.

For any two disks $d_a \in \mathcal{D}_a$ and $d_b \in \mathcal{D}_b$ with p_i covered by d_a or d_b, let $D_i(d_a, d_b)$ be an optimal solution of following problem.

$$\begin{aligned} \min \quad & w(D) = \sum_{d \in D} w(d) \\ \text{subject to} \quad & d_a, d_b \in D, \\ & \forall d \in D \cap \mathcal{D}_a : d \prec_i d_a, \\ & \forall d \in D \cap \mathcal{D}_b : d \prec_i d_b, \\ & D \text{ covers targets points } p_1, p_2, ..., p_i. \end{aligned} \tag{3.1}$$

Then, we have following recursive formula.

Lemma 3.1.6.

$$\begin{aligned} w(D_i(d_a, d_b)) \quad = \quad & \min\{w(S_{i-1}(d'_a, d'_b)) + [d_a \neq d'_a]w(d_a) + [d_b \neq d'_b]w(d_b) \\ & \mid d'_a \prec_i d_a,\ d'_b \prec_i d_b,\ \textit{and } p_{i-1} \textit{ is covered by } d'_a \textit{ or } d'_b\} \end{aligned}$$

where

$$[d \neq d'] = \begin{cases} 1 & \textit{if } d \neq d', \\ 0 & \textit{otherwise}. \end{cases}$$

Proof. . Let d'_a be the disk in $D_i(d_a, d_b) \cap \mathcal{D}_a$ whose boundary has the lowest intersection with ℓ_{i-1} and d'_b the disk in $D_i(d_a, d_b) \cap \mathcal{D}_b$ whose boundary has

the highest intersection with ℓ_{i-1}. We claim that

$$w(D_i(d_a, d_b)) = w(D_{i-1}(d'_a, d'_b)) + [d_a \neq d'_a]w(d_a) + [d_b \neq d'_b]w(d_b). \quad (3.2)$$

To prove it, we first show that if $d_a \neq d'_a$, then $d_a \notin D_{i-1}(d'_a, d'_b)$ for $w(d_a) > 0$. In fact, for otherwise, there exists $i' < i-1$ such that $p_{i'}$ is covered by d_a, but not covered by d'_a. This is impossible (Fig. 3.4). To see this, let A

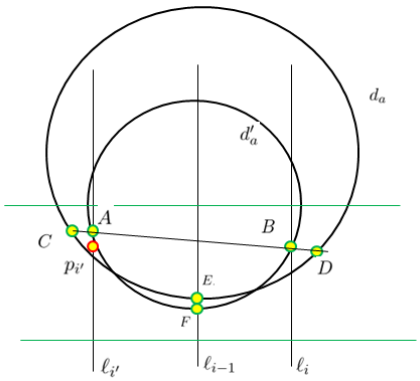


Figure 3.4: Proof of Lemma 3.1.6.

be the lowest intersection between the boundary of disk d'_a and $\ell_{i'}$ and B the lowest intersection between the boundary of disk d'_a and ℓ_i. Then A and B lie inside the disk d_a. Let C and D be intersection points between line AB and the boundary of disk d_a. Let E be the lowest intersection between the boundary of disk d_a and ℓ_{i-1} and F the lowest intersection between the boundary of disk d'_a and ℓ_{i-1}. Note that d_a and d'_a lies above the strip. We have $\angle CED > \angle AFB > \pi/2$ and hence $\sin \angle CED < \sin \angle AFB$. Moreover, we have $|AB| < |CD|$. Thus,

$$\text{radius } (d_a) = \frac{|CD|}{2 \sin \angle CED} > \frac{|AB|}{2 \sin \angle AFB} = \text{radius}(d'_a),$$

contradicting the homogeneous assumption of disks. Therefore, our claim is true. Similarly, if $d_b \neq d'_b$, then $d_b \notin S_{i-1}(d'_a, d'_b)$ for $w(s_b) > 0$. Therefore, (3.2) holds. This means that for equation in Lemma 3.1.6, the left-side $\geq$ the right-side.

To see the left-side $\leq$ the right-side for the equation in Lemma 3.1.6, we note that in the right-side, $S_{i-1}(d'_a, d'_b) \cup \{d_a, d_b\}$ is always a feasible solution of the problem (3.1). $\square$

Let us employ the recursive formula in Lemma 3.1.6 to compute all $S_i(d_a, d_b)$. There are totally $O(m^2 n)$ problems. With the recursive formula, each $S_i(d_a, d_b, k)$ can be computed in time $O(m^2)$. Therefor, all $S_i(d_a, d_b, k)$ can be computed by dynamic programming in time $O(m^4 n)$. The solution of Example 3.1.5 can be computed by

$$S = \text{argmin}_{S_n(d_a,d_b)} w(S_n(d_a, d_b))$$

where $d_a \in \mathcal{D}_a$, $d_b \in \mathcal{D}_b$, and p_n is covered by d_a or d_b. This requires additional computation within time $O(m^2)$. Therefore, putting all computations together, the time is $O(m^4 n)$.

3.2 Shortest Path

Often, the running time of a dynamic programming algorithm can be estimated by the product of the table size (the number of subproblems) and the computation time of the recursive formula (i.e., the time for recursively computing the solution of each subproblem). **Does this estimation hold for every dynamic programming algorithm?** The answer is No. In this section, we would like to provide a counterexample, the shortest path problem. For this problem, we must consider something else in order to estimate the running time of a dynamic programming algorithm.

Problem 3.2.1 (Shortest Path)**.** *Given a directed graph $G = (V, E)$ with arc cost $c : E \to Z$, and a source node s and a sink node t in V, where Z is the set of integers, find a path from s to t with minimum total arc cost.*

In study of shortest path, arcs coming to s and arc going out from t are useless. Therefore, we assume that those arcs do not exist in G, which may simplify some statements later.

For any node $u \in V$, let $d^*(s, u)$ denote the total cost of the shortest path from node s to node u and $N^-(u)$, the in-neighbor set of u, i.e., the set of nodes each with an arc coming to u. Then it is easy to obtain the following recursive formula (Fig. 3.5).

$$\begin{aligned} d^*(s) &= 0, \\ d^*(u) &= \min_{v \in N^-(v)} \{d^*(v) + c(v, u)\}. \end{aligned}$$

Based on this recursive formula, we may write down an algorithm as follows:

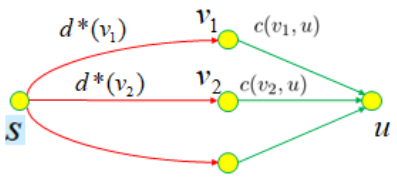


Figure 3.5: Recursive relation of $d^*(u)$.

DP1 for the Shortest Path
$S \leftarrow \{s\}$;
$T \leftarrow V - S$;
while $T \neq \emptyset$ **do begin**
find $u \in T$ such that $N^-(u) \subseteq S$;
compute $d^*(u) = \min_{v \in N^-(u)}\{d^*(v) + c(v,u)\}$;
$S \leftarrow S \cup \{u\}$;
$T \leftarrow T - \{u\}$;
end-while
output $d^*(t)$.

This is a dynamic programming algorithm which works correctly for all acyclic digraphs due to following.

Theorem 3.2.2. *Consider an acyclic network $G = (V, E)$ with a source node s and a sink node t. Assume that for any $v \in V - \{s\}$, $N^-(v) \neq \emptyset$. Let (S, T) be a partition of V such that $s \in S$ and $t \in T$. Then there exists $u \in T$ such that $N^-(u) \subseteq S$.*

Proof. Note that for any $u \in T$, $N^-(u) \neq \emptyset$. If $N^-(u) \not\subseteq S$, then there exists $v \in N^-(u)$ such that $v \in T$. If $N^-(v) \not\subseteq S$, then there exists $w \in N^-(v)$ such that $w \in T$. This process cannot go forever. Finally, we would find $z \in T$ such that $N^-(z) \subseteq S$. □

In this theorem, the acyclic condition cannot be dropped. In Fig. 3.6, a counterexample is shown that a simple cycle may make no node u in T satisfy $N^-(u) \subseteq S$.

To estimate the running time of algorithm DP1, we note that $d^*(u)$ needs to be computed for u over all nodes, that is, the size of table for holding

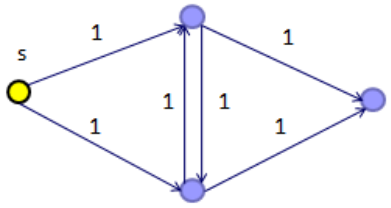


Figure 3.6: When $S = \{s\}$, there is no node u such that $N^-(u) \subseteq S$.

all subproblems is $O(n)$ where n is the number of nodes. In the recursive formula for computing each $d^*(u)$, the "min" operation is over all nodes in $N^-(u)$ which may contains $O(n)$ nodes. Thus, the product of the table size and the computation time of recursive formula is $O(n^2)$. However, this estimation for the running time of algorithm DP1 is not correct. In fact, we need also to consider the time for finding $u \in T$ such that $N^-(u) \subseteq S$. This requires to check if a set is a subset of another set. What is the running time of this computation? Roughly speaking, this may takes $O(n \log n)$ time and hence, totally the running time of algorithm DP1 is $O(n(n+n\log n)) = O(n^2 \log n)$.

Can we improve this running time by a smarter implementation? The answer is yes. Let us do this in two steps.

First, we introduce a new number $d(u) = \min_{v \in N^-(u) \cap S}(d^*(v) + c(v,u))$ and rewrite the algorithm DP1 as follows.

DP2 for the Shortest Path
$S \leftarrow \emptyset$;
$T \leftarrow V$;
while $T \neq \emptyset$ **do begin**
    find $u \in T$ such that $N^-(u) \subseteq S$;
    $S \leftarrow S \cup \{u\}$;
    $T \leftarrow T - \{u\}$;
    $d^*(u) = d(u)$;
    for every $w \in N^+(u)$ update $d(w) \leftarrow \min(d(w), d^*(u) + c(u,w))$;
end-while
output $d^*(t)$.

In this algorithm, updating value of $d(u)$ would be performed on all edges

and for each edge, update once. Therefore, the total time is $O(m)$ where m is the number of edges, i.e., $m = |E|$.

Secondly, we introduce the topological sort. The *topological sort* of nodes in a digraph $G = (V, E)$ is a ordering such that for any arc $(u, v) \in E$, node u has position before node v. There is an algorithm with running time $O(m)$ for topological sort as shown in Algorithm 7. Actually, in Algorithm 7,

Algorithm 7 Topological Sort.

Input: A directed graph $G = (V, E)$.
Output: A topologically sorted sequence of nodes.

```
1: L ← ∅
2: S ← {s}
3: while S ≠ ∅ do
4:    remove a node u from S
5:    put u at tail of L
6:    for each node v ∈ N+(u) do
7:       remove arc (u, v) from graph G
8:       if v has no other incoming arc then
9:          insert v into S
10:      end if
11:   end for
12: end while
13: if graph G has an arc then
14:   return error (G contains at least one cycle)
15: else
16:   return L
17: end if
```

line 3 takes $O(n)$ time and line 7 takes $O(m)$ time. Hence, it runs totally in $O(m+n)$ time. However, for the shortest path problem, input directed graph is connected if ignore the arc direction and hence $n = O(m)$. Therefore, $O(m + n) = O(m)$.

An example for topological sort is shown in Fig. 3.7. In each iteration, yellow node is the one selected from S to initiate the iteration. During the iteration, the yellow node will be moved from S to end of L and all arcs from the yellow node will be deleted, meanwhile new nodes will be added to S.

Now, we can first do topological sort and them carry out dynamic programming, which will result in a dynamic programming (Algorithm 8 for the shortest path problem, running in $O(m)$ time.

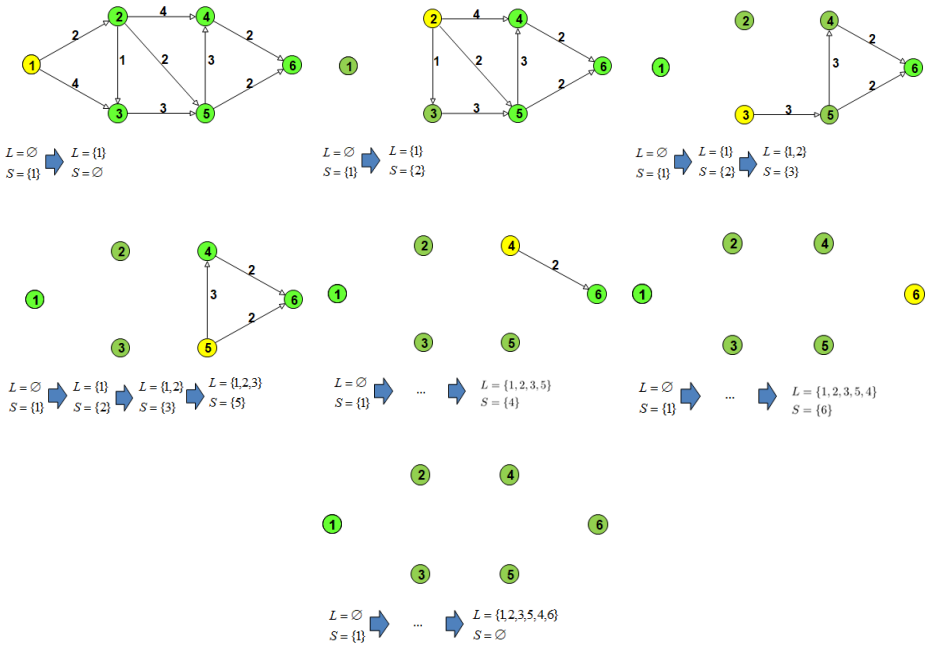


Figure 3.7: An example of topological sort.

Algorithm 8 Dynamic Programming for Shortest Path.

Input: A directed graph $G = (V, E)$ with arc weight $c : E \to Z$, and two nodes s and t in V.

Output: The length of shortest path from s to t.

```
1: S ← ∅
2: T ← V
3: do topological sort on T
4: d(s) ← 0
5: for every u ∈ V \ {s} do
6:    d(u) ← ∞
7: end for
8: while T ≠ ∅ do
9:    remove the first node u from T
10:   S ← S ∪ {u}
11:   d*(u) ← d(u)
12:   for every (u, v) ∈ E do
13:      d(v) ← min(d(v), d*(u) + c(u, v))
14:   end for
15: end while
16: return d*(t)
```

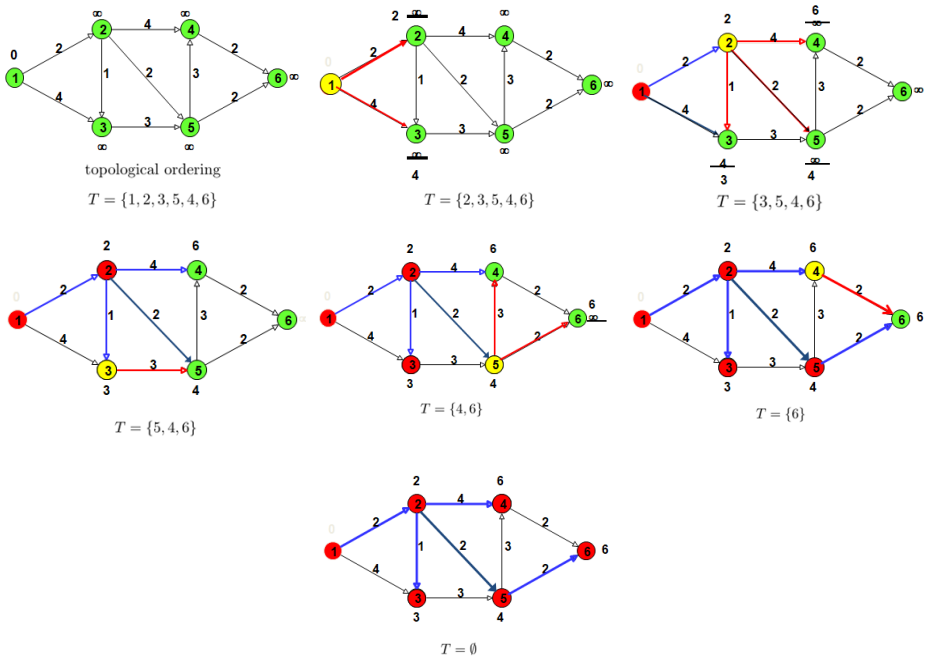


Figure 3.8: An example of dynamic programming for shortest path.

An example is shown in Fig. 3.8. At beginning, the topological sort is done in previous example as shown in Fig. 3.7. In Fig. 3.8, the yellow node represents the one removed from the front of T to initiate an iteration. During the iteration, all red arcs from the yellow node are used for updating the value of $d(\cdot)$ and meanwhile the yellow node is added to S whose $d^*(\cdot)$'s value equals to $d(\cdot)$'s value.

It may worth mentioning that Algorithm 8 works for acyclic directed graph without restriction on arc weight, i.e., arc weight can be negative. This implies that the longest path problem can be solved in $O(m)$ time if input graph is acyclic. For definition of the longest path problem, please find it in Chapter 8. The longest path problem is NP-hard and hence unlikely to have a polynomial-time solution. This means that for the shortest path problem, if input directed graph is not acyclic and arc weights can be negative, then solution may not polynomial-time computable. What is about the case that input directed graph is not acyclic and all arc weights are nonnegative? In next section, we present a polynomial-time solution.

3.3 Dijkstra Algorithm

Dijkstra algorithm is able to find the shortest path in any directed graph with nonnegative arc weights. Its design is based on following important discovery.

Theorem 3.3.1. *Consider a directed network $G = (V, E)$ with a source node s and a sink node t, and every arc (u, v) has a nonnegative weight $c(u, v)$. Suppose (S, T) is a partition of V such that $s \in S$ and $t \in T$. If $d(u) = \min_{v \in T} d(v)$, then $d^*(u) = d(u)$.*

Proof. For contradiction, suppose $d(u) = \min_{v \in T} d(v) > d^*(u)$. Then there exists a path p (Fig. 3.9)from s to u such that

$$length(p) = d^*(u) < d(u).$$

Let w be the first node in T on path p. Then $d(w) = length(p(s, w))$ where $p(s, w)$ is the piece of path p from s to w. Since all arc weights are nonnegative, we have

$$length(p) \geq length(p(s, w)) = d(w) \geq d(u) > d^*(u) = length(p)$$

a contradiction. □

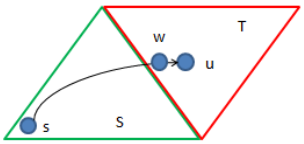


Figure 3.9: In proof of Theorem 3.3.1.

By Theorem 3.3.1, in dynamic programming for shortest path, we may replace $N^-(u) \subseteq S$ by $d(u) = \min_{v \in T} d(v)$ when all arc weights are nonnegative. This replacement results in Dijkstra algorithm.

Dijkstra Algorithm
$S \leftarrow \emptyset$;
$T \leftarrow V$;
while $T \neq \emptyset$ **do begin**
find $u \leftarrow \text{argmin}_{v \in T} d(v)$;
$S \leftarrow S \cup \{u\}$;
$T \leftarrow T - \{u\}$;
$d^*(u) = d(u)$;
for every $w \in N^+(u)$, update $d(w) \leftarrow \min(d(w), d^*(u) + c(u, w))$;
end-while
output $d^*(t)$.

With different data structures, Dijkstra algorithm can be implemented with different running times.

With min-priority queue, Dijkstra algorithm can be implemented in time $O((m + n) \log n)$.

With Fibonacci heap, Dijkstra algorithm can be implemented in time $O(m + n \log n)$.

With Radix heap, Dijkstra algorithm can be implemented in time $O(m + n \log c)$ where c is the maximum arc weight.

We will pick up one of them to introduce in next section. Before doing it, let us first implement Dijkstra algorithm with simple buckets (also known as Dial algorithm). This simple implementation can achieve running time $O(m+nc)$. When c is small, e.g., $c = 1$, it could be a good choice. (This case occurs in study of Admonds-Karp algorithm for maximum flow in Chapter 5.)

In this implementation, $(n-1)c+2$ buckets are prepared with labels $0, 1, ..., (n-1)c, \infty$. They are used for store nodes in T such that every node u is stored in bucket $d(u)$. Therefore, initially, s is in bucket 0 and other nodes are in bucket ∞. As $d(u)$'s value is updated, node u will be moved from a bucket to another bucket with smaller label. Note that if $d(u) < \infty$, then there must exist a simple path from s to u such that $d(u)$ is equal to the total weight of this path. Therefore, $d(u) \leq c(n-1)$, i.e., buckets set up as above are enough for our purpose. In Fig.3.10, an example is computed by Dihkstra algorithm with simple buckets.

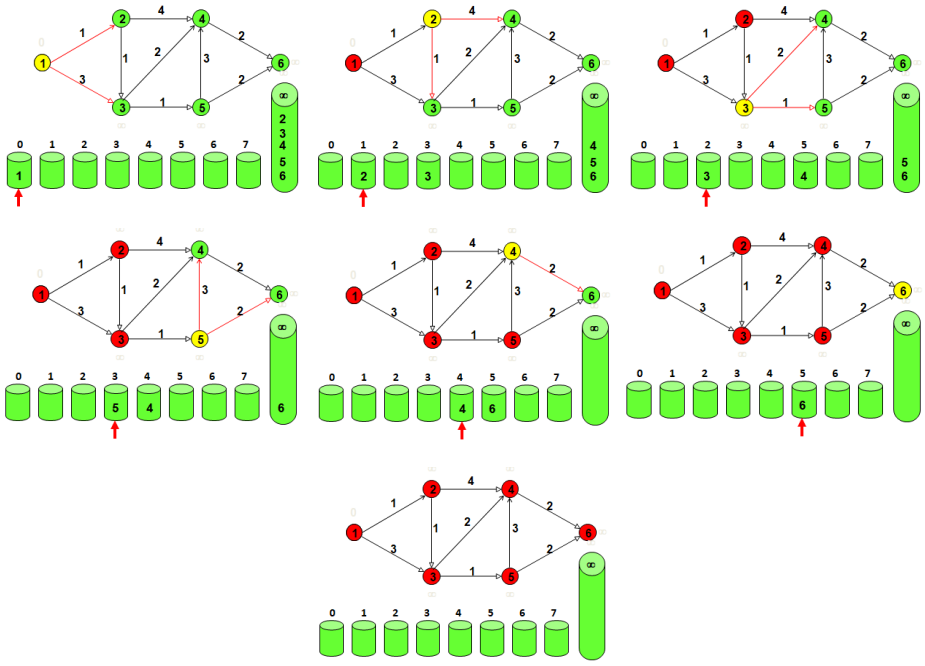


Figure 3.10: An example for Dijstra algorithm with simple buckets.

Now let estimate the running time of Dijstra algorithm with simple buckets.

- Time to create buckets is $O(nc)$.
- Time for finding u to satisfy $d(u) = \min_{v \in T} d(v)$ is $O(nc)$. In fact, u can be chosen arbitrarily from the nonempty bucket with smallest label. Such a bucket in Fig.3.10 is pointed by a red arrow, which is

traveling from left to right without going backward. This is because, after update $d(w)$ for $w \in T$, we always have $d^*(u) \leq d(w)$ for $w \in T$.

- Time to update $d(w)$ for $w \in T$ and update buckets is $O(m)$.
- Therefore, total time is $O(m + nc)$.

3.4 Priority Queue

Although Dijkstra algorithm with simple buckets runs faster for small c, it cannot be counted as a polynomial-time solution. In fact, the input size of c is $\log c$. Therefore, we would like to select a data structure with which, implement Dijkstra algorithm in polynomial-time. This data structure is priority queue.

A priority queue is a data structure for maintaining a set S f elements, each with an associated value, called a key. All keys are stored in an array A such that an element belongs to set S if and only if its key is in array A. There are two types of priority queues, the min-priority queue and the max-priority queue. Since they are similar, we introduce one of them, the min-priority queue.

A min-priority queue supports following operations: Minimum(S), Extract-Min(S), Increase-Key(S, x, k), Insert(S, x).

The min-heap can be employed in implementation of those operations.

Minimum(S) returns the element of S with the smallest key, which can be implemented as follows.

Heap-Minimum(A)
    return $A[1]$.

Extract-Min(S) removes and returns the element of S with the smallest key, which can be implemented by using min-heap as follows.

Heap-Extract-Min(A)
    min $\leftarrow A[1]$;
    $A[1] \leftarrow A$[heap-size[A]];
    heap-size[A] $\leftarrow$ heap-size[A]-1;
    Min-Heapify($A, 1$);
    return min.

Decrease-Key(S, x, k) decreases the value of element x's key to the new value k, which is assumed to be no more than x's current key value. Suppose that $A[i]$ contains x's key. Then, Decrease-Key(S, x, k) can be implemented as an operation of min-heap as follows.

Heap-Decrease-Key(A, i, key)
  if $key > A[i]$
    then error "new key is larger than current key";
  $A[i] \leftarrow key$;
  while $i > 1$ and $A[\text{Parent}(i)] > A[i]$
    do exchange $A[i] \leftrightarrow A[\text{Parent}(i)]$
      and $i \leftarrow \text{Parent}(i)$.

Insert$(S, x.key)$ inserts the element x into S, which is implemented in following.

Insert(A, key)
  array-size$[A] \leftarrow$ array-size$[A] + 1$;
  $A[\text{array-size}[A]] \leftarrow +\infty$;
  Decrease-Key$(A, \text{array-size}[A], key)$.

Now, we analyze these four operations. Minimum(S) runs clearly in $O(1)$ time. Each of other three operations runs in $O(\log n)$ time. Actually, since Min-Heapify$(A, 1)$ runs in $O(\log n)$ time, so does Extract-Min(S). For Decrease-Key(S, x, k), as shown in Fig.3.11, computation is along a path from a node approaching to the root of the heap and hence runs in $O(\log n)$ time. This also implies that Insert$(S, x.key)$ can be implemented in $O(\log n)$ time.

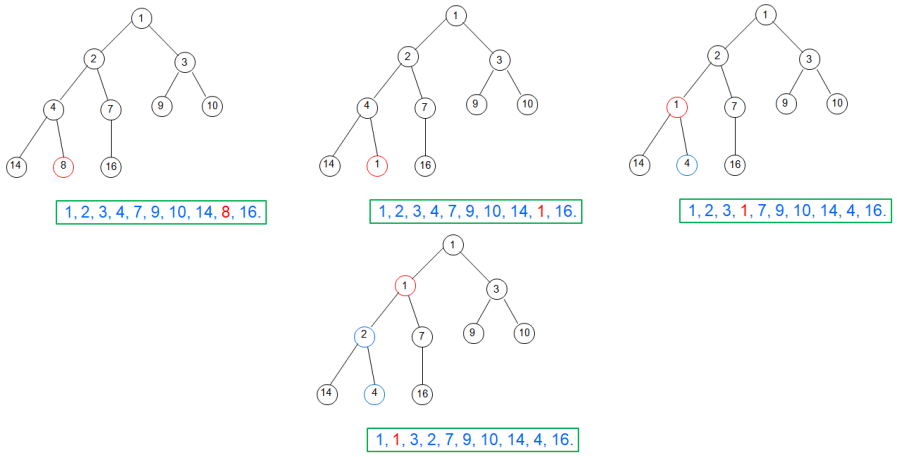


Figure 3.11: Heap-Decrease-Key$(A, 9, 1)$.

In Algorithm 9, Dijkstra algorithm is implemented with priority queue as follows.

- Use min-priority queue to keep set T and for every node $u \in T$, use $d(u)$ for the key of u.
- Use operation Extract-Min(T) to obtain u satisfying $d(u) = \min_{v \in T} d(v)$. This operation at line 9 will be used for $O(n)$ times.
- Use operation Decrease-Key(T, v, key) on each edge (u, v) to update $d(v)$ and the min-heap. This operation on line 14 will be used for $O(m)$ times.
- Therefore, the total running time is $O((m + n) \log n)$.

Algorithm 9 Dijkstra Algorithm with Priority Queue.

Input: A directed graph $G = (V, E)$ with arc weight $c : E \to Z$, and source node s and sink node t in V.

Output: The length of shortest path from s to t.

```
1: S ← ∅
2: T ← V
3: d(s) ← 0
4: for every u ∈ V \ {s} do
5:    d(u) ← ∞
6: end for
7: build a min-priority queue on T with key d(u) for each node u ∈ T, i.e.,
   use keys to build a min-heap.
8: while T ≠ ∅ do
9:    u ← Extract-Min(T)
10:   S ← S ∪ {u}
11:   d*(u) ← d(u)
12:   for every (u, v) ∈ E do
13:      if d(v) > d*(u) + c(u, v)) then
14:         Decrease-Key(T, v, d*(u) + c(u, v))
15:      end if
16:   end for
17: end while
18: return d*(t)
```

An example is as shown in Fig.3.12.

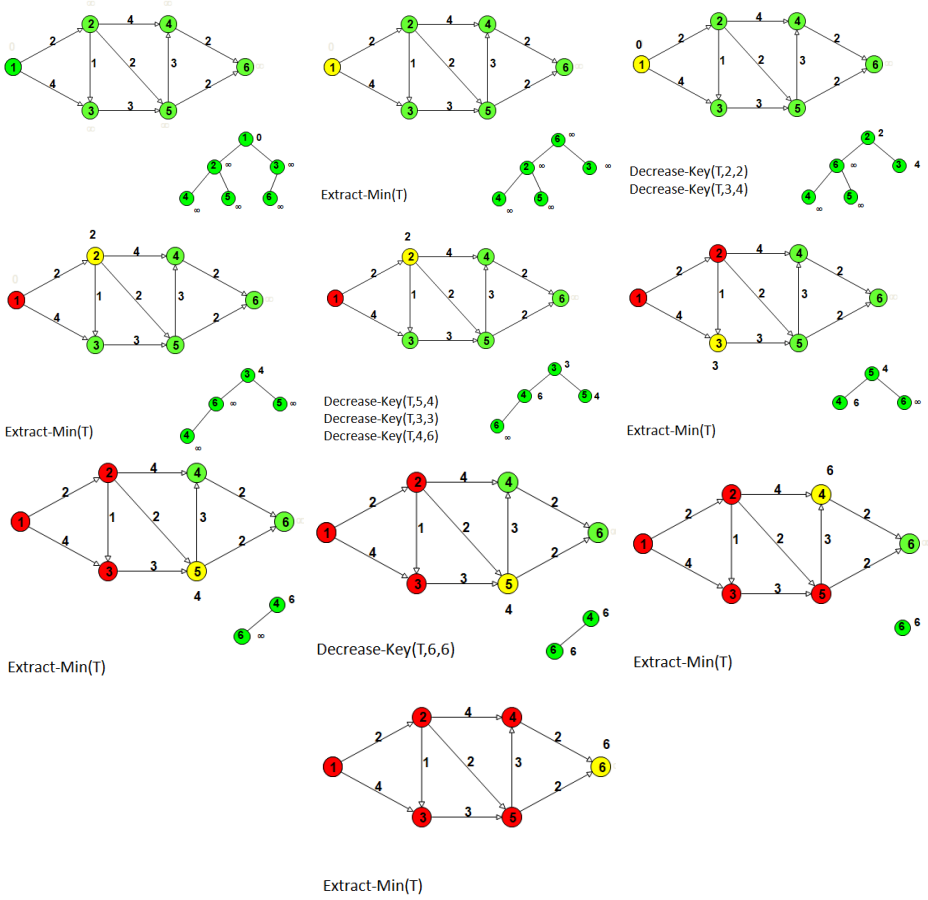


Figure 3.12: An example for Dijkstra algorithm with priority queue.

3.5 Bellman-Ford Algorithm

Bellman-Ford algorithm allows negative arc cost, only restriction is no negative weight cycle. The disadvantage of this algorithm is that the running time is a little slow. The idea behind this algorithm is very simple.

Initially, assign $d(s) = 0$ and $d(u) = \infty$ for every $u \in V \setminus \{s\}$. Then, algorithm updates $d(u)$ such that in iteration i, $d(u)$ is equal to the shortest distance from s to u passing through at most i arcs. If there is no negative weight cycle, then the shortest path contains at most $n-1$ arcs. Therefore, after $n-1$ iterations, $d(t)$ is the minimum weight of the path from s to t. If in the nth iteration, $d(u)$ is still updated for some u, then it means that a negative weight cycle exists.

Bellman-Ford Algorithm
input:A directed graph $G = (V, E)$ with weight $c : E \to R_+$,
a source node s and a sink node t.
output: The minimum weight of path from s to t,
or a message "G contains a negative weight cycle".

$d(s) \leftarrow 0$;
for $u \in V \setminus \{s\}$ **do**
$d(u) \leftarrow \infty$;
for $i \leftarrow 1$ **t** $n-1$ **do**
for each arc $(u, v) \in E$ **do**
if $d(u) + c(u, v) < d(v)$
$d(v) \leftarrow d(u) + c(u, v)$;
for each arc $(u, v) \in E$ **do**
if $d(u) + c(u, v) < d(v)$
then return "G contains a negative weight cycle".
else return $d(t)$.

Its running time is easily estimated.

Theorem 3.5.1. *Bellman-Ford algorithm computes a shortest path from s to t within $O(mn)$ time where n is the number of nodes and m is the number of arcs in input directed graph.*

3.6 All Pairs Shortest Paths

In this section, we study following problem.

Problem 3.6.1 (All-Pairs-Shortest-Paths)**.** *Given a directed graph $G = (V, E)$, find the shortest path from s to t for all pairs $\{s, t\}$ of nodes.*

If apply Bellman-Ford algorithm for single pair of nodes for each of $O(n^2)$ pairs, then the total time for computing a solution of the all-pairs-shortest-paths problem is $O(n^3m)$. In the following, we will present two faster algorithms, with running time $O(n^3 \log n)$ and $O(n^3)$, respectively with only restriction that no negative weight cycle exists. Before doing so, let us consider an example on which, we introduce an approach which can be used for the all-pairs-shortest-paths problem.

Example 3.6.2 (Path Counting)**.** *Given a directed graph $G = (V, E)$ and a positive integer k, count the number of paths with exactly k arcs from s to t for all pairs $\{s, t\}$ of nodes*

Let $a_{st}^{(k)}$ denote the number of paths with exactly k arcs from s to t. Then, we have

$$a_{st}^{(1)} = \begin{cases} 1 & \text{if } (s,t) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

This means that $(a_{st}^{(1)})$ is the adjacency matrix of graph G. Denote

$$A(G) = (a_{st}^{(1)}).$$

We claim that

$$A(G)^k = (a_{st}^{(k)}).$$

Let us prove this claim by induction on k. Suppose it is true for k. Consider a path from s to t with exactly $k+1$ arcs. Decompose the path at a node h such that the subpath from s to h contains exactly k arcs and (h, t) is an arc. Then the subpath from s to h has $a_{sh}^{(k)}$ choices and (h, t) has $a_{ht}^{(1)}$ choices. Therefore,

$$a_{st}^{(k+1)} = \sum_{h \in V} a_{sh}^{(k)} a_{ht}^{(1)}.$$

It follows that

$$(a_{st}^{(k+1)}) = (a_{ht}^{(k)})(a_{ht}^{(1)}) = A(G)^k \cdot A(G) = A(G)^{k+1}.$$

Now, we come back to the all-pairs shortest paths problem. First, we assume that G has no loop. In fact, a loop with nonnegative weight does not play any role in a shortest path and a loop with negative weight means that the problem is meaning less.

Let $\ell_{st}^{(k)}$ denote the weight of the shortest path with at most k arcs from s to t. For $k = 1$, we have

$$\ell_{st}^{(1)} = \begin{cases} c(s,t) & \text{if } (s,t) \in E, \\ \infty & \text{if } (s,t) \notin E \text{ and } s \neq t, \\ 0 & \text{if } s = t. \end{cases}$$

Denote

$$L(G) = (\ell_{st}^{(1)}).$$

This is called the *weighted adjacency matrix*. For example, the graph in Fig.3.13 has weighted adjacency matrix

$$\begin{pmatrix} 0 & 4 & \infty \\ \infty & 0 & 6 \\ 5 & \infty & 0 \end{pmatrix}.$$

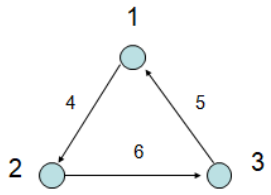


Figure 3.13: A weighted directed graph.

We next establish a recursive formula for $\ell_{st}^{(k)}$.

Lemma 3.6.3.

$$\ell_{st}^{k+1)} = \min_{h \in V}(\ell_{sh}^{(k)} + \ell_{st}^{(1)}).$$

Proof. Since the shortest path from s to h with at most k arcs and the shortest path from h to t with at most one arc form a path from s to t with at most $k+1$ arcs, we have

$$\ell_{st}^{k+1)} \leq \min_{h \in V}(\ell_{sh}^{(k)} + \ell_{ht}^{(1)}).$$

Next, we show

$$\ell_{st}^{k+1)} \geq \min_{h \in V}(\ell_{sh}^{(k)} + \ell_{ht}^{(1)}).$$

To do so, consider two cases.

Case 1. There is a path with weight $\ell_{st}^{(k+1)}$ from s to t containing at most k arcs. In this case, we have

$$\begin{aligned}\ell_{st}^{(k+1)} &= \ell_{st}^{(k)} \\ &= \ell_{st}^{(k)} + \ell_{ht}^{(1)} \\ &\geq \min_{h \in V}(\ell_{sh}^{(k)} + \ell_{ht}^{(1)}).\end{aligned}$$

Case 2. Every path with weight $\ell_{st}^{(k+1)}$ from s to t contains exactly $k+1$ arcs. In this case, we can find a node h' on the path such that the piece from s to h' contains exactly k arcs and $(h', t) \in E$. Their weights should be $\ell_{sh'}^{(k)}$ and $\ell_{h't}^{(1)}$, respectively. Therefore,

$$\begin{aligned}\ell_{st}^{(k+1)} &= \ell_{sh'}^{(k)} + \ell_{h't}^{(1)} \\ &\geq \min_{h \in V}(\ell_{sh}^{(k)} + \ell_{ht}^{(1)}).\end{aligned}$$

□

If G does not contain negative weight cycle, then each shortest path does not need to contain a cycle. Therefore, we have

Theorem 3.6.4. *If G does not have a negative weight cycle, then $\ell_{st}^{(n-1)}$ is the weight of shortest path from s to t where $n = |V|$.*

This suggests a dynamic programming to solve the all-pairs-shortest-paths problem by using recursive formula in Lemma 3.6.3. Since each $\ell_{st}^{(k)}$ is computed in $O(n)$ time, this algorithm will run in $O(n^4)$ time to compute $\ell_{st}^{(n-1)}$ for all pairs $\{s, t\}$.

Next, we give a method to speed up this algorithm. To do so, let us define a new operation for matrixes. Consider two $n \times n$ square matrixes $A = (a_{ij})$ and $B = (b_{ij})$. Define

$$A \circ B = (\min_{1 \leq h \leq n}(a_{ih} + b_{hj})).$$

An example is as shown in Fig.3.14.

This operation satisfies associative law.

Lemma 3.6.5.

$$(A \circ B) \circ C = A \circ (B \circ C).$$

$$(a_{ih})_{m\times n} \circ (b_{hj})_{n\times p} = (\min_{1\le h\le n}\{a_{ih} + b_{hj}\})_{m\times p}$$

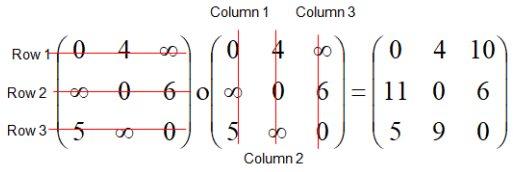


Figure 3.14: A new matrix multiplication.

We leave proof of this lemma as an exercise.
By this lemma, following is well-defined.

$$A^{(k)} = \underbrace{A \circ \cdots \circ A}_{k}.$$

Note that if G has no negative weight cycle, then for $m \ge n-1$, $L(G)^{(m)} = L(G)^{(n-1)}$. This observation suggests following algorithm to compute $L(G)^{(n-1)}$.

$n \leftarrow |V|$;
$m \leftarrow 1$;
$L^{(1)} \leftarrow L(G)$;
while $m < n-1$
  do $L^{2m} \leftarrow L^{(m)} \circ L^{(m)}$ and
    $m \leftarrow 2m$;
return $L^{(m)}$.

With this improvement, the dynamic programming with recursive formula in Lemma 3.6.3 is called the *faster-all-pairs-shortest-paths* al algorithm, which runs in $O(n^3 \log n)$ time.

Above result is derived under assumption that G does not have a negative weight cycle. Suppose G is unknown to have a negative weight cycle or not. Can we modify the faster-all-pairs-shortest-paths algorithm to find a negative weight cycle if G has one? The answer is yes. However, we need to compute $L(G)^{(m)}$ for $m \ge n$.

Theorem 3.6.6. *G contains a negative weight cycle if and only if $L(G)^{(n)}$ contains a negative diagonal entry. Moreover, if $L(G)^{(n)}$ contains a negative diagonal entry then such an entry keeps negative sign in every $L(G)^{(m)}$ for $m \geq n$.*

Proof. It follows immediately from fact that a simple cycle contains at most n arcs. □

.

Next, let us study another algorithm for the all-pairs-shortest-paths problem. First, we show a lemma.

Lemma 3.6.7. *Assume $V = \{1, 2, ..., n\}$. Let $d_{ij}^{(k)}$ denote the weight of shortest path from i to j with internal nodes in $\{1, 2, ..., k\}$. Then for $i \neq j$,*

$$d_{ij}^{(k)} = \begin{cases} c(i,j) & \text{if } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} & \text{if } 1 \leq k \leq n, \end{cases}$$

and $d_{ij}^{(k)} = 0$ for $i = j$ and $k \geq 0$.

Proof. We need only to consider $i \neq j$. Let p be the shortest path from i to j with internal nodes in $\{1, 2, ..., k\}$. For $k = 0$, p does not contain any internal node. Hence, its weight is $c(i, j)$. For $k \geq 1$, there are two cases (Fig.3.15).

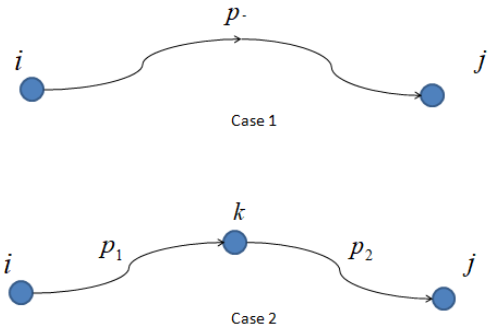


Figure 3.15: Proof of Lemma 3.6.7.

Case 1. p does not contain internal node k. In this case,

$$d_{ij}^{(k)} = d_{ij}^{(k-1)}.$$

Case 2. p contains an internal node k. Since p does not contain a cycle, node k appears exactly once. Suppose that node k decomposes path p into two pieces p_1 and p_2, from i to k and from k to j, respectively. Then the weight of p_1 should be $d_{ik}^{(k-1)}$ and the weight of p_2 should be $d_{ij}^{(k-1)}$. Therefore, in this case, we have

$$d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}.$$

□

Denote $D^{(k)} = (d_{ij}^{(k)}$. Based on recursive formula in Lemma 3.6.7, we obtain a dynamic programming as shown in Algorithm 10, which is called *Floyd-Warshall algorithm.*

Algorithm 10 Floyd-Warshall Algorithm.

Input: A directed graph $G = (V, E)$ with arc weight $c : E \to Z$.
Output: The weight of shortest path from s to t for all pairs of nodes s and t.

```
1: n ← |V|
2: D^(0) ← L(G)
3: for k ← 1 to n do
4:    for i ← 1 to n do
5:       for j ← 1 to n do
6:          d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7:       end for
8:    end for
9: end for
10: return D^(n)
```

From algorithm description, it is easy to see following.

Theorem 3.6.8. *If G does not contain a negative weight cycle, then Floyd-Warshall algorithm computes all-pairs shortest paths in $O(n^3)$ time.*

If G contains a negative weight cycle, could Floyd-Warshall algorithm tells us this fact? The answer is yes. Actually, we also have

Theorem 3.6.9. *G contains a negative weight cycle if and only if $D^{(n)}$ contains negative diagonal element.*

Exercises

1. Please construct a directed graph $G = (V, E)$ with arc weight and source vertex s such that for every arc $(u, v) \in E$, there is a shortest-paths tree rooted at s that contains (u, v) and there is another shortest-paths tree rooted at s that does not contain (u, v).

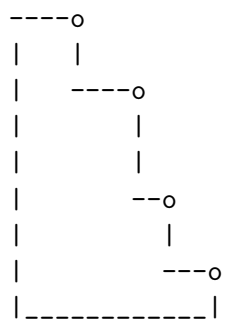
2. Show that the graph G contains a negative weight cycle if and only if $A(G)^{(n-1)} \neq A(G)^{(2n-1)}$.

3. Please give a simple example of a directed graph with negative-weight arces to show that Dijkstra's algorithm produces incorrect answers in this case. Please also explain why the proof of correct of Dijkstra's algorithm cannot go through when negative-weight edges are allowed?

4. Please design an $O(n^2)$-time algorithm to compute the longest monotonically increasing subsequence for a given sequence of n numbers.

5. How can we use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle?

6. A stair is a rectilinear polygon as shown in the following figure:



   Show that the minimum length rectangular partition for given stair can be computed by a dynamic programming in time $O(n^2 \log n)$.

7. Given a rectilinear polygon containing some rectilinear holes inside, guillotine-partition it into small rectangles without hole inside with minimum total length of guillotine cuts. Design a dynamic programming to solve this problem in polynomial-time with respect to n where n is the number of concave boundary points in input rectangle (a boundary point is concave if it faces inside with an angle of more than 180^o.)

8. Consider a horizontal line. There are n points lying below the line and m unit disks with centers above the line. Everyone of the n points is covered by some unit disk. Each unit disk has a weight. Design a dynamic programming to find a subset of unit disks covering all n points, with the minimum total weight. The dynamic programming should runs in polynomial time with respect to m and n.

9. Given a convex polygon in the Euclidean plane, partition it into triangles with minimum total length of cuts. Design a dynamic programming to solve this problem in time $O(n^3)$ where n is the number of vertices of input polygon.

10. Does Dijkstra's algorithm for shortest path work for input with negative weight and without negative weight cycle? If yes, please give a proof. If not, please give a counterexample and a way to modify the algorithm to work for input with negative weight and without negative weight cycle.

11. Given a directed graph $G = (V, E)$ and a positive integer k, count the number of paths with at most k arcs from s to t for all pairs of nodes s and t.

12. Given a graph $G = (V, E)$ and a positive integer k, count the number of paths with at most k edges from s to t for all pairs of nodes s and t.

13. Given a directed graph $G = (V, E)$ without loop, and a positive integer k, count the number of paths with at most k arcs from s to t for all pairs of nodes s and t.

14. Show that $A \circ (B \circ C) = (A \circ B) \circ C$.

15. Does Faster-All-Pair-Shortest-Path algorithm work for input with negative weight and without negative weight cycle? If yes, please give a proof. If not, please give a counterexample.

16. Does Floyd-Warshall algorithm work for input with negative weight and without negative weight cycle? If yes, please give a proof. If not, please give a counterexample.

17. Given a sequence $x_1, x_2, ..., x_n$ of (not necessary positive) integers, find a subsequence $x_i, x_{i+1}, ..., x_{i+j}$ of consecutive elements to maximize the sum $x_i + x_{i+1} + \cdots + x_{i+j}$. Can your algorithm run in linear time?

18. Assume that you have an unlimited supply of coins in each of the integer denominations $d_1, d_2, \ldots, d_n$, where each $d_i > 0$. Given an integer amount $m \geq 0$, we wish to make change for m using the minimum number of coins drawn from the above denominations. Design a dynamic programming algorithm for this problem.

19. Recall that $C(n, k)$—the binomial coefficient—is the number of ways of choosing an unordered subset of k objects from a set of n objects. (Here $n \geq 0$ and $0 \leq k \leq n$.) Give a dynamic programming algorithm to compute the value of $C(n, k)$ in $O(nk)$ time and $O(k)$ space.

20. Given a directed graph $G = (V, E)$, with nonnegative weight on its edges, and in addition, each edge is colored red or blue. A path from u to v in G is characterized by its total length, and the number of times it switches colors. Let $\delta(u, k)$ be the length of a shortest path from a source node s to u that is allowed to change color at most k times. Design a dynamic program to compute $\delta(u, k)$ for all $u \in V$. Explain why your algorithm is correct and analyze its running time.

21. Given a rectilinear polygon without hole, partition this polygon into rectangles with minimum total length of cut-lines. Design a dynamic programming to solve this problem.

Historical Notes

Dynamic programming was proposed by Richard Bellman in 1953 [22] and later became a popular method in optimization and control theory. The basic idea is stemmed from self-reducibility. In design of computer algorithms, it is a powerful and elegant technique to find an efficient solution for many optimization problems, which attracts a lot of researchers' efforts in the literature, especially in the direction of speed-up dynamic programming. For example, Yao [23] and Borchers and Gupta [24] speed up dynamic programming with the quadrangle inequality, including a construction for the rectilinear Steiner arborescence [25] from $O(n^3)$ time to $O(n^2)$ time.

The shortest path problem became a classical graph problem as early as in 1873 [30]. A. Schrijver [29] provides a quite detail historical note with a large list of references. There are many algorithms in the literature. Those closely related to dynamic programming algorithms algorithms can be found in Bellman [26], Dijkstra [27], Dial [31], and Fredman and Tarjan [28].

All-pair-shortest-paths problem was first studied by Alfonso Shimbel in 1953 [32], who gave a $O(n^4)$-time solution. Floyd [33] and Marshall [34] found a $O(n^3)$-time solution independently in the same year.

Chapter 4

Greedy Algorithm and Spanning Tree

"Greed, in the end, fails even the greedy."
- Cathryn Louis

Self-reducibility is the backbone of each greedy algorithm in which self-reducibility structure is a tree of special kind, i.e., its internal nodes lie on a path. In this chapter, we study algorithms with such a self-reducibility structure and related combinatorial theory supporting greedy algorithms.

4.1 Greedy Algorithms

A problem that the greedy algorithm works for computing optimal solutions often has the self-reducibility and a simple exchange property. Let us use two examples to explain this point.

Example 4.1.1 (Activity Selection)**.** *Consider n activities with starting times $s_1, s_2, ..., s_n$ and ending times $f_1, f_2, ..., f_n$, respectively. They may be represented by intervals $[s_1, f_1)$, $[s_2, f_2]$, ..., $[s_n, f_n)$. The problem is to find a maximum subset of nonoverlapping activities, i.e., nonoverlapping intervals.*

This problem has following exchange property.

Lemma 4.1.2 (Exchange Property)**.** *Suppose $f_1 \leq f_2 \leq \cdots \leq f_n$. In a maximum solution without interval $[s_1, f_1)$, we can always exchange $[s_1, f_1)$ with the first activity in the maximum solution preserving the maximality.*

Proof. Let $[s_i, f_i)$ be the first activity in the maximum solution mentioned in the lemma. Since $f_1 \leq f_i$, replacing $[s_i, f_i)$ by $[s_1, f_1)$ will not cost any overlapping. □

Following lemma states a self-reducibility.

Lemma 4.1.3 (self-reducibility)**.** *Suppose* $\{I_1^*, I_2^*, ..., I_k^*\}$ *is an optimal solution. Then*$\{I_2^*, ..., I_k^*\}$ *is an optimal solution for the activity problem on input* $\{I_i \mid I_i \cap I_1^*\}$ *where* $I_i = [s_i, f_i)$*.*

Proof. For contradiction, Suppose that $\{I_2^*, ..., I_k^*\}$ is not an optimal solution for the activity problem on input $\{I_i \mid I_i \cap I_1^*\}$. Then, $\{I_i \mid I_i \cap I_1^*\}$ contains k nonoverlapping acitivities, whicl all are not overlapping with I_1^*. Putting I_1^* in these k activities, we will obtain a feasible solution containing $k+1$ activities, contradicting the assumption that $\{I_1^*, I_2^*, ..., I_k^*\}$ is an optimal solution. □

Based on Lemmas 4.1.2 and 4.1.3, we can design a greedy algorithm in Algorithm 11 and obtain following result.

Algorithm 11 Greedy Algorithm for Activity Selection.

Input: A sequence of n activities $[s_1, f_1), [s_2, f_2)$, ..., $[s_n, f_n)$.
Output: A maximum subset of nonoverlapping activities.

```
1: sort all activities into ordering f_1 ≤ f_2 ≤ ... ≤ f_n
2: S ← ∅
3: for i ← 1 to n do
4:    if [s_i, f_i) does not overlap any activity in S then
5:       S ← S ∪ {[s_i, f_i)}
6:    end if
7: end for
8: return S
```

Theorem 4.1.4. *Algorithm 11 produces an optimal solution for the activity selection problem.*

Proof. Let us prove it by induction on n. For $n = 1$, it is trivial.

Consider $n \geq 2$. Suppose $\{I_1^*, I_2^*, ..., I_k^*\}$ is an optimal solution. By Lemma 4.1.2, we may assume that $I_1^* = [s_1, f_1)$. By Lemma 4.1.3, $\{I_2^*, ..., I_k^*\}$ is an optimal solution for the activity selection problem on input $\{I_i \mid I_i \cap I_1^* = \emptyset\}$.

Note that after select $[s_1, f_1)$, if we ignore all iterations i with $[s_i, f_i) \cap [s_1, f_1) \neq \emptyset$, then remaining part is the same as greedy algorithm running on input $\{I_i \mid I_i \cap I_1^* = \emptyset\}$. By induction hypothesis, it will produces an optimal solution for the activity selection problem on input $\{I_i \mid I_i \cap I_1^* = \emptyset\}$, which must contain $k-1$ activities. Together with $[s_1, f_1)$, they form a subset of k nonoverlappng activities, which should be optimal. □

Next, we study another example.

Example 4.1.5 (Huffman Tree)**.** *Given n characters $a_1 a_2, ..., a_n$ with weights $f_1, f_2, ..., f_n$, respectively, find a binary tree with n leaves labeled by $a_1, a_2, ..., a_n$, respectively, to minimize*

$$d(a_1) \cdot f_1 + d(a_2) \cdot f_2 + \cdots + d(a_n) \cdot f_n$$

where $d(a_i)$ is the depth of leaf a_i, i.e., the number of edges on the path from the root to a_i.

First, we show a property of optimal solutions.

Lemma 4.1.6. *In any optimal solution, every internal node has two children.*

Proof. If an internal node has only one child, then this internal node can be removed to reduce the objective function value. □

We can also show an exchange property and a self-reducibility.

Lemma 4.1.7 (Exchange Property)**.** *If $f_i > f_j$ and $d(a_i) > d(a_j)$, then exchanging a_i and a_j would make the objective function value decrease.*

Proof. Let $d'(a_i)$ and $d(a_j)$ be the depths of a_i and a_j, respectively after exchange a_i and a_j. Then $d'(a_i) = d(a_j)$ and $d'(a_j) = d(a_i)$. Therefore, the difference of objective function values before and after exchange is

$$\begin{aligned} & (d(a_i) \cdot f_i + d(a_j) \cdot f_j) - (d'(a_i) \cdot f_i + d'(a_j) \cdot f_j) \\ = \; & (d(a_i) \cdot f_i + d(a_j) \cdot f_j) - (d(a_j) \cdot f_i + d(a_i) \cdot f_j) \\ = \; & (d(a_i) - d(a_j))(f_i - f_j) \\ > \; & 0 \end{aligned}$$

□

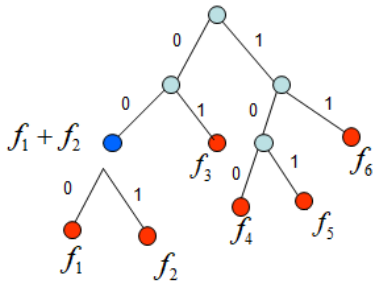


Figure 4.1: A self-reducibility.

Lemma 4.1.8 (Self-Reducibility)**.** *In any optimal tree T^*, if we assign the weight of an internal node u with the total weight w_u of its descendant leaves, then removal the subtree T_u at the internal node results in an optimal tree T'_u for weights at remainder's leaves (Fig.4.1).*

Proof. Let $c(T)$ denote the objective function value of tree T, i.e.,

$$c(T) = \sum_{a \text{over leaves of } T} d(a) \cdot f(a)$$

where $d(a)$ is the depth of leaf a and $f(a)$ is the weight of leaf a. Then we have

$$c(T^*) = c(T_u) + c(T'_u).$$

If T'_u is not optimal for weights at leaves of T'_u, then we have a binary tree T''_u for those weights with $c(T''_u) < c(T'_u)$. Therefore $c(T_u \cup T''_u) < c(T^*)$, contradicting optimality of T^*. □

By Lemmas 4.1.7 and 4.1.8, we can construct an optimal Huffman tree in following.

- Sort $f_1 \leq f_2 \leq \cdots \leq f_n$.
- By exchange property, there must exist an optimal tree in which a_1 and a_2 and sibling at bottom level.
- By self-reducibility, the problem can be reduced to construct optimal tree for leaves weights $\{f_1 + f_2, f_3, ..., f_n\}$.
- Go back to initial sorting step. This process continue until only two weights exist.

In Fig.4.2, an example is presented to explain this construction. This construction can be implemented with min-priority queue (Algorithm 12

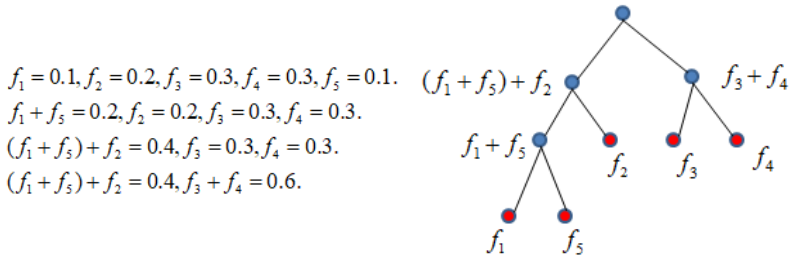


Figure 4.2: An example for construction of Huffman Tree.

Algorithm 12 Greedy Algorithm for Huffman Tree.

Input: A sequence of leaf weights $\{f_1, f_2, ..., f_n\}$.
Output: A binary tree.

1: Put $f_1, f_2, ..., f_n$ into a min-priority queue Q
2: **for** $i \leftarrow 1$ to $n-1$ **do**
3: allocate a new node z
4: $left[z] \leftarrow x \leftarrow$ Extract-Min(Q)
5: $right[z] \leftarrow y \leftarrow$ Extract-Min(Q)
6: $f[z] \leftarrow f[x] + f[y]$
7: Insert(Q, z)
8: **end for**
9: **return** Extract-Min(Q)

The Huffman tree problem is raised from study of Huffman codes as follows.

Problem 4.1.9 (Huffman Codes)**.** *Given n characters $a_1 a_2, ..., a_n$ with frequencies $f_1, f_2, ..., f_n$, respectively, find prefix binary codes $c_1, c_2, ..., c_n$ to minimize*

$$|c_1| \cdot f_1 + |c_2| \cdot f_2 + \cdots + |c_n| \cdot f_n,$$

where $|c_i|$ is the length of code c_i, i.e., the number of symbols in c_i.

Actually, $c_1, c_2, ..., c_n$ are called *prefix* binary codes if no one is a prefix of another one. Therefore, they have a binary tree representation.

- Each edge is labeled with 0 or 1.
- Each code is represented by a path from the root to a leaf.
- Each leaf is labeled with a character.
- The length of a code is the length of corresponding path.

An example is as shown in Fig4.3. With this representation, the Huffman cods problem can be transformed exactly the Huffman tree problem.

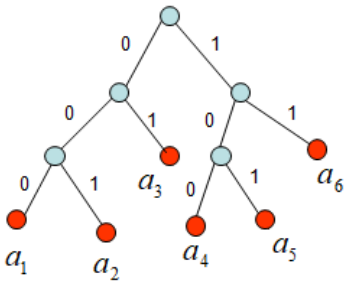


Figure 4.3: Huffman codes.

In Chapter 1, we see that Kruskal greedy algorithm can compute the minimum spanning tree. Thus, we may have a question: Does the minimum spanning tree problem have an exchange property and self-reducibility? The answer is yeas and they are given in the following.

Lemma 4.1.10 (Exchange Property)**.** *For an edge e with the smallest weight in a graph G and a minimum spanning tree T without e, there must exist an edge e' in T such that $(T \setminus e') \cup e$ is still a minimum spanning tree.*

Proof. Suppose u and v are two endpoints of edge e. Then T contains a path p connecting u and v. On path p, every edge e' must have weight $c(e') = c(e)$. Otherwise, $(T \setminus e') \cup e$ will be a spanning tree with total weight smaller that $c(T)$, contradicting minimality of $c(T)$.

Now, select any edge e' in path p. Then $(T \setminus e') \cup e$ is a minimum spanning tree. □

Lemma 4.1.11 (Self-Reducibility)**.** *Suppose T is a minimum spanning tree of a graph G and edge e in T has the smallest weight. Let G' and T' be obtained from G and T, respectively by shrinking e into a node (Fig.4.4). Then T' is a minimum spanning tree of G'.*

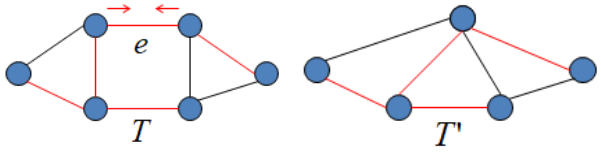


Figure 4.4: Lemma 4.1.11.

Proof. It is easy to see that T is a minimum spanning tree of G if and only if T' is a minimum spanning tree of G'. □

With above two lemma, we are able to give an alternative proof for correctness of Kruskal algorithm. We leave it as an exercise for reader.

4.2 Matroid

There is a combinatorial structure has a close relationship with greedy algorithms. This is the matroid. To introduce matroid, let us first study independent systems.

Consider a finite set S and a collection $\mathcal{C}$ of subsets of S. $(S, \mathcal{C})$ is called an *independent system* if

$$A \subset B, B \in \mathcal{C} \Rightarrow A \in \mathcal{C},$$

i.e., it is *hereditary*. In the independent system $(S, \mathcal{C})$, each subset in $\mathcal{C}$ is called an independent set.

Consider a maximization as follows.

Problem 4.2.1 (Independent Set Maximization)**.** *Let c be a nonnegative cost function on S. Denote $c(A) = \sum_{x \in A} c(x)$ for any $A \subseteq S$. The problem is to maximize $c(A)$ subject to $A \in \mathcal{C}$.*

Also, consider the greedy algorithm in Algorithm 13.

For any $F \subseteq E$, a subset I of F is called a *maximal* independent subset if no independent subset of E contains F as a proper subset. Define

$$\begin{aligned} u(F) &= \max\{|I| \mid I \text{ is an independent subset of } F\} \\ v(F) &= \min\{|I| \mid I \text{ is a maximal independent subset of } F\} \end{aligned}$$

Algorithm 13 Greedy Algorithm for Independent Set Maximization.

Input: An independent system $(S, \mathcal{C})$ with a nonnegative cost function c on S.

Output: An independent set.

```
1: Sort all elements in S into ordering c(x_1) ≥ c(x_2) ≥ ··· ≥ c(x_n)
2: A ← ∅
3: for i ← 1 to n do
4:    if A ∪ {x_i} ∈ C then
5:       A ← A ∪ {x_i}
6:    end if
7: end for
8: return A
```

where $|I|$ is the number of elements in I. Then we have following theorem to estimate the performance of Algorithm 13.

Theorem 4.2.2. *Let A_G be a solution obtained by Algorithm 13. Let A^* be an optimal solution for the independent set maximization. Then*

$$1 \leq \frac{c(A^*)}{c(A_G)} \leq \max_{F \subseteq S} \frac{u(F)}{v(F)}.$$

Proof. Note that $S = \{x_1, x_2, ..., x_n\}$ and $c(x_1) \geq c(x_2) \geq \cdots \geq c(x_n)$. Denote $S_i = \{x_1, \ldots, x_i\}$. Then

$$\begin{aligned} c(A_G) &= c(x_1)|S_1 \cap A_G| + \sum_{i=2}^{n} c(x_i)(|S_i \cap A_G| - |A_{i-1} \cap A_G|) \\ &= \sum_{i=1}^{n-1} |S_i \cap A_G|(c(x_i) - c(x_{i+1})) + |A_n \cap A_G| c(x_n). \end{aligned}$$

Similarly,

$$c(A^*) = \sum_{i=1}^{n-1} |S_i \cap A^*|(c(x_i) - c(x_{i+1})) + |S_n \cap AI^*| c(x_n).$$

Thus,

$$\frac{c(A^*)}{c(A_G)} \leq \max_{1 \leq i \leq n} \frac{|A^* \cap S_i|}{|A_G \cap S_i|}.$$

We claim that $A_i \cap A_G$ is a maximal independent subset of S_i. In fact, for contradiction, suppose that $S_i \cap A_G$ is not a maximal independent subset of

S_i. Then there exists an element $x_j \in S_i \setminus A_G$ such that $(S_i \cap A_G) \cup \{x_j\}$ is independent. Thus, in the computation of Algorithm 2.1, $I \cup \{e_j\}$ as a subset of $(S_i \cap A_G)\{x_j\}$ should be independent. This implies that x_j should be in A_G, a contradiction.

Now, from our claim, we see that

$$|S_i \cap A_G| \geq v(S_i).$$

Moreover, since $S_i \cap A^*$ is independent, we have

$$|S_i \cap A^*| \leq u(S_i).$$

Therefore,

$$\frac{c(A^*)}{c(A_G)} \leq \max_{F \subseteq S} \frac{u(F)}{v(F)}.$$

□

The matroid is an independent system satisfying an additional property, called *Augmentation Property*:

$$\begin{aligned} & A, B \in \mathcal{C} \text{ and } |A| > |B| \\ \Rightarrow \quad & \exists x \in A \setminus B : B \cup \{x\} \in \mathcal{C}. \end{aligned}$$

This property is equivalent to some others.

Theorem 4.2.3. *An independent system $(S, \mathcal{C})$ is a matroid if and only if for any $F \subseteq S$, $u(F) = v(F)$.*

Proof. For forward direction, consider two maximal independent sets A and B. If $|A| > |B|$, then there exists $x \in A \setminus B$ such that $B \cup \{x\} \in \mathcal{C}$, contradicting maximality of B.

For backward direction, consider two independent sets with $|A| > |B|$. Set $F = A \cup B$. Then every maximal independent set of F has size at least $|A|$ $(> |B|)$. Hence, B cannot be a maximal independent set of F. Thus, there exists an element $x \in F \setminus B = A \setminus B$ such that $B \cup \{x\} \in \mathcal{C}$. □

Theorem 4.2.4. *An independent system $(S, \mathcal{C})$ is a matroid if and only if for any cost function $c(\cdot)$, Algorithm 13 gives a maximum solution.*

Proof. . For necessity, we note that when $(S, \mathcal{C})$ is matroid, we have $u(F) = v(F)$ for any $F \subseteq S$. Therefore, Algorithm 13 gives an optimal solution.

For sufficiency, we give a contradiction argument. To this end, suppose independent system $(S, \mathcal{C})$ is not a matroid. Then, there exists $F \subseteq S$ such that F has two maximal independent sets I and J with $|I| < |J|$. Define

$$c(e) = \begin{cases} 1+\varepsilon & \text{if } e \in I \\ 1 & \text{if } e \in J \setminus I \\ 0 & \text{otherwise} \end{cases}$$

where ε is a sufficient small positive number to satisfy $c(I) < c(J)$. The greedy algorithm will produce I, which is not optimal. □

This theorem gives tight relationship between matroids and greedy algorithms, which is built up on all nonnegative objective function. It may be worth mentioning that the greedy algorithm reaches optimal for certain class of objective functions may not provide any additional information to the independent system. Following is a counterexample.

Example 4.2.5. *Consider a complete bipartite graph $G = (V_1, V_2, E)$ with $|V_1| = |V_2|$. Let $\mathcal{I}$ be the family of all matchings. Clearly, $(E, \mathcal{I})$ is an independent system. However, it is not a matroid. An interesting fact is that maximal matchings may have different cardinalities for some subgraph of G although all maximal matchings for G have the same cardinality.*

Furthermore, consider the problem $\max\{c(\cdot) \mid I \in \mathcal{I}\}$*, called the* maximum assignment *problem.*

If $c(\cdot)$ is a nonnegative function such that for any $u, u' \in V_1$ and $v, v' \in V_2$,

$$c(u,v) \geq \max(c(u,v'), c(u',v)) \Longrightarrow c(u,v) + c(u',v') \geq c(u,v') + c(u',v),$$

This means that replacing edges (u_1, v') and (u', v_1) in M^* by (u_1, v_1) and (u', v') will not decrease the total cost of the matching. Similarly, we can put all (u_i, v_i) into an optimal solution, that is, they form an optimal solution. This gives an exchange property. Actually, we can design a greedy algorithm to solve the maximum assignment problem. (We leave this as an exercise.)

Next, let us present some examples of the matroid.

Example 4.2.6 (Linear Vector Space)**.** *Let S be a finite set of vectors and $\mathcal{I}$ the family of linearly independent subsets of S. Then $(S, \mathcal{I})$ is a matroid.*

Example 4.2.7 (Graph Matroid)**.** *Given a graph $G = (V, E)$ where V and E are its vertex set and edge set, respectively. Let $\mathcal{I}$ be the family of edge sets of acyclic subgraphs of G. Then $(E, \mathcal{I})$ is a matroid.*

Proof. . Clearly, $(E, \mathcal{I})$ is an independent system. Consider a subset F of E. Suppose that the subgraph (V, F) has m connected components. Note that in each connected components, every maximal acyclic subgraph must be a spanning tree which has the number of edges one less than the number of vertices. Thus, every maximal acyclic subgraph of (V, E) has exactly $|V| - m$ edges. By Theorem 4.2.3, $(E, \mathcal{I})$ is a matroid. □

In a matroid, all maximal independent subsets have the same cardinality. They are also called *bases*. In a graph matroid obtained from a connected graph, every base is a spanning tree.

Let $\mathcal{B}$ be the family of all bases of a matroid $(S, \mathcal{C})$. Consider the following problem:

Problem 4.2.8 (Base Cost Minimization)**.** *Consider a matroid $(S, \mathcal{C})$ with base family $\mathcal{B}$ and a nonnegative cost function on S. Te problem is to minimize $c(B)$ subject to $B \in \mathcal{B}$.*

Algorithm 14 Greedy Algorithm for Base Cost Minimization.

Input: A matroid $(S, \mathcal{C})$ with a nonnegative cost function c on S.
Output: A base.

```
1: Sort all elements in S into ordering c(x_1) ≤ c(x_2) ≤ ··· ≤ c(x_n)
2: A ← ∅
3: for i ← 1 to n do
4:    if A ∪ {x_i} ∈ C then
5:       A ← A ∪ {x_i}
6:    end if
7: end for
8: return A
```

Theorem 4.2.9. *An optimal solution of the base cost minimization can be computed by Algorithm 14, a variation of Algorithm 13.*

Proof. Suppose that every base has the cardinality m. Let M be a positive number such that for any $e \in S$, $c(e) < M$. Define $c'(e) = M - c(e)$ for all $e \in E$. Then $c'(\cdot)$ is a positive function on S and the non-decreasing ordering with respect to $c(\cdot)$ is the non-increasing ordering with respect to $c'(\cdot)$. Note that $c'(B) = mM - c(B)$ for any $B \in \mathcal{B}$. Since Algorithm 13 produces a base with maximum value of c', Algorithm 14 produces a base with minimum value of function c. □

The correctness of greedy algorithm for the minimum spanning tree can also be obtained from this theorem.

Next, consider following problem.

Problem 4.2.10 (Unit-Time Task Scheduling)**.** *Consider a set of n unit-time tasks, $S = \{1, 2, ..., n\}$. Each task i can be processed during a unit-time and has to be completed before an integer deadline d_i and if not completed, will receive a penalty w_i. The problem is to find a schedule for S on a machine within time n to minimize total penalty.*

A set of tasks is independent if there exists a schedule for these tasks without penalty. Then we have following.

Lemma 4.2.11. *A set A of tasks is independent if and only if for any $t = 1, 2, ..., n$, $N_t(A) \leq t$ where $N_t(A) = |\{i \in A \mid d_i \leq t\}|$.*

Proof. It is trivial for "only if" part. For "if" part, note that if the condition holds, then tasks in A can be scheduled in order of nondecreasing deadlines without penalty. □

Example 4.2.12. *Let S be a set of unit-time tasks with deadlines and penalties and $\mathcal{C}$ the collection of all independent subsets of S. Then, $(S, \mathcal{C})$ is a matroid. Therefore, an optimal solution for the unit-time task scheduling problem can be computed by a greedy algorithm (i.e., Algorithm 13).*

Proof. (Hereditary) Trivial.

(Augmentation) Consider two independent sets A and B with $|A| < |B|$. Let k the largest k such that $N_t(A) \geq N_t(B)$. (A few examples are presented in Fig.4.5 to explain the definition of k.) Then $k < n$ and $N_t(A) < N_t(B)$ for $k + 1 \leq t \leq n$. Choose $x \in \{i \in B \setminus A \mid d_i = k + 1\}$. Then

$$N_t(A \cup \{x\}) = N_t(A) \leq t \text{ for } 1 \leq t \leq k$$

and

$$N_t(A \cup \{x\}) \leq N_t(A) + 1 \leq N_t(B) \leq t \text{ for } k + 1 \leq t \leq n.$$

□

Example 4.2.13. *Consider an independent system $(S, \mathcal{C})$. For any fixed $A \subseteq S$, define*

$$\mathcal{C}_A = \{B \subseteq S \mid A \not\subseteq B\}.$$

Then, $(S, \mathcal{C}_A)$ is a matroid.

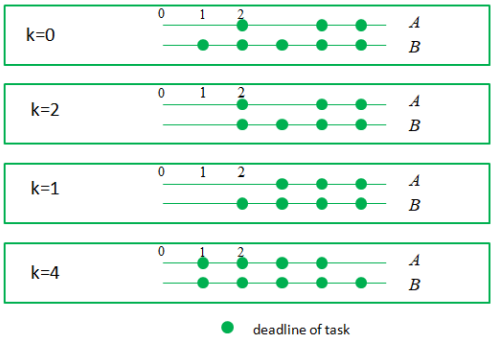


Figure 4.5: In proof of Example 4.2.12.

Proof. Consider any $F \subseteq S$. If $A \nsubseteq F$, then F has unique maximal independent set, which is F. Hence, $u(F) = v(F)$.

If $A \subseteq F$, then every maximal independent subset of F is in form $F \setminus \{x\}$ for some $x \in A$. Hence, $u(F) = v(F) = |F| - 1$. □

4.3 Minimum Spanning Tree

Let us revisit the minimum spanning tree problem.

Consider a graph $G = (V, E)$ with nonnegative edge weight $c : E \to R_+$, and a spanning tree T. Let (u, v) be an edge in T. Removal (u, v) would break T into two connected components. Let U and W be vertex sets of these two components, respectively. The edges between U and V constitute a *cut*, denoted by (U, W). The cut (U, W) is said to be induced by deleting (u, v). For example, in Fig.4.6, deleting $(3, 4)$ induces a cut $(\{1, 2, 3\}, \{4, 5, 6, 7, 8\})$.

Theorem 4.3.1 (Cut Optimality)**.** *A spanning tree T^* is a minimum spanning tree if and only if it satisfies the cut optimality condition as follows.*

Cut Optimality Condition *For every edge (u, v) in T^*, $c(u, v) \leq c(x, y)$ for every edge (x, y) contained in the cut induced by deleting (u, v).*

Proof. Suppose, for contradiction, that $c(u, v) > c(x, y)$ for some edge (x, y) in the cut induced by deleting (u, v) from T^*. Then $T' = (T^* \setminus (u, v)) \cup (x, y)$ is a spanning tree with cost less than $c(T^*)$, contradicting the minimality of T^*.

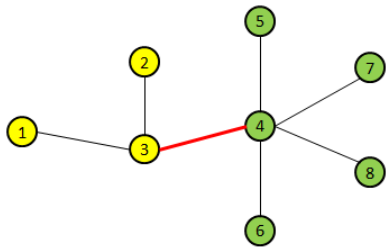


Figure 4.6: A cut induced by deleting an edge from a spanning tree.

Conversely, suppose that T^* satisfies the cut optimality condition. Let T' be a minimum spanning tree such that among all minimum spanning tree, T' is the one with the most edges in common with T^*. Suppose, for contradiction, that $T' \neq T^*$. Consider an edge (u, v) in $T^* \setminus T'$. Let p be the path from u to v in T'. Then p has at least one edge (x, y) in the cut induced by deleting (u, v) from T^*. Thus, $c(u, v) \leq c(x, y)$ by the cut optimality condition. Hence, $T'' = (T' \setminus (x, y)) \cup (u, v)$ is also a minimum spanning tree, contradicting the assumption on T'. □

The following algorithm is designed based on cut optimality condition.

Prim Algorithm
input: A graph $G = (V, E)$ with nonnegative edge weight $c :\to R_+$.
output: A spanning tree T.
$U \leftarrow \{s\}$ for some $s \in V$;
$T \leftarrow \emptyset$;
while $U \neq V$ **do**
find the minimum weight edge (u, v) from cut $(U, V \setminus U)$
and $T \leftarrow T \cup (u, v)$;
return T.

An example for using Prim algorithm is shown in Fig.4.7. The construction starts at node 1 and guarantees that the cut optimality conditions are satisfied at the end.

The min-priority queue can be used for implementing Prim algorithm to obtain following result.

Theorem 4.3.2. *Prim algorithm can construct a minimum spanning tree in $O(m \log m)$ time where m is the number of edges in input graph.*

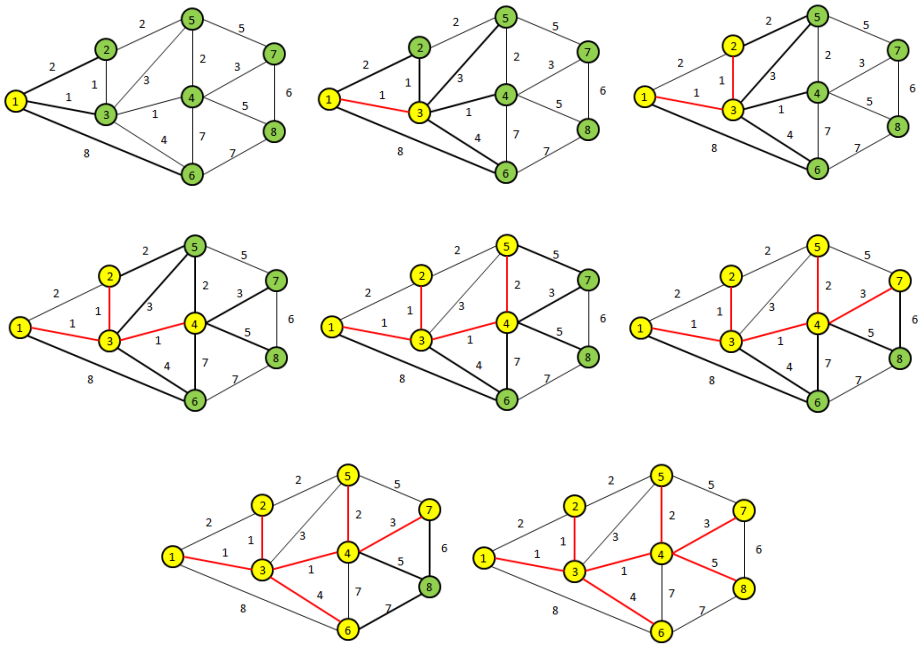


Figure 4.7: An example with Prim algorithm.

Proof. Prim algorithm can be implemented by using min-priority queue in following way.

- Keep to store all edges in a cut (U, W) in the min-priority queue S.
- At each iteration, choose the minimum weight edge (u, v) in the cut (U, W) by using operation Extract-Min(S) where $u \in U$ and $v \in W$.
- For every edge (x, v) with $x \in U$, delete (c, v) from S. This needs a new operation on min-priority queue, which runs $O(m)$ time.
- Add v to U.
- For every edge (v, y) with $y \in V \setminus U$, insert (v, y) into priority queue. This also requires $O(\log m)$ time.

In this implementation, Prim algorithm runs in $O(m \log m)$ time. □

Prim algorithm can be considered as a special type of greedy algorithm. Actually, its correctness can also be established by an exchange property and a self-reducibility as follows.

Lemma 4.3.3 (Exchange Property)**.** *Consider a cut (U, W) in a graph $G = (V, E)$. Suppose edge e has the smallest weight in cut (U, W). If a minimum spanning tree T does not contain e, then there must exist an edge e' in T such that $(T \setminus e') \cup e$ is still a minimum spanning tree.*

Lemma 4.3.4 (Self-Reducibility)**.** *Suppose T is a minimum spanning tree of a graph G and edge e in T has the smallest weight in the cut induced by deleting e from T. Let G' and T' be obtained from G and T, respectively by shrinking e into a node. Then T' is a minimum spanning tree of G'.*

We leave proofs of them as exercises.

4.4 Local Ratio Method

The local ratio method is also a type of algorithm with self-reducibility. Its basic idea is as follows.

Lemma 4.4.1. *Let $c(x) = c_1(x) + c_2(x)$. Suppose x^* is an optimal solution of $\min_{x \in Omega} c_1(x)$ and $\min_{x \in Omega} c_2(x)$. Then x^* is an optimal solution of $\min_{x \in Omega} c(x)$. The similar statememt holds for the maximization problem.*

Proof. For any $x \in \Omega$, $c_1(x) \geq c_1(x^*)$, $c_2(x) \geq c_2(x^*)$, and hence $c(x) \geq c(x^*)$. □

Usually, the objective function $c(x)$ is decomposed into $c_1(x)$ and $c_2(x)$ such that optimal solutions of $\min_{x \in Omega} c_1(x)$ constitute a big pool so that the problem is reduced to find an optimal solution of $\min_{x \in Omega} c_2(x)$ in the pool. In this section, we present two examples to explain this idea.

First, we study following problem

Problem 4.4.2 (Weighted Activity Selection)**.** *Given n activities each with a time period $[s_i, f_i)$ and a positive weight w_i, find a nonoverlapping subset of activities to maximize the total weight.*

Suppose, without loss of generality, $f_1 \leq f_2 \leq \cdots \leq f_n$. First, we consider a special case that for every activity $[s_i, f_i)$, if $s_i < f_1$, i.e., activity $[s_i, f_i)$ overlaps with activity $[s_1, f_1)$, then $w_i = w_1 > 0$, and if $s_i \geq f_1$, then $w_i = 0$. In this case, every feasible solution containing an activity overlapping with $[s_1, f_1)$ is an optimal solution. Motivated from this special case, we may decompose the problem into two subproblems. The first one is in the special case and the second one has weight as follows

$$w_i' = \begin{cases} w_i - w_1 & \text{if } s_i < f_1, \\ w_i & \text{otherwise.} \end{cases}$$

In the second subproblem obtained from the decomposition, some activity may have non-positive weight. Such an activity can be removed from our consideration because putting it in any feasible solution would not increase the total weight. This operation would simplify the problem by removing at least one activity. Repeat the decomposition and simplification until no activity is left.

To explain how to obtain an optimal solution, let A' be the set of remaining activities after the first decomposition and simplification and Opt' is an optimal solution for the weighted activity selection problem on A'. Since simplification does not effect the objective function value of optimal solution, Opt' is an optimal solution of the second subproblem in the decomposition. If Opt' contains an activity overlapping with activity $[s_1, f_1)$, then Opt' is also an optimal solution of the first subproblem and hence by Lemma 4.4.1, Opt' is an optimal solution for the weighted activity selection problem on original input A. If Opt' does not contain an activity overlapping with $[s_1, f_1)$, then $Opt' \cup \{[s_1, f_1)\}$ is an optimal solution for the first subproblem problem and the second subproblem and hence also an optimal solution for the original problem.

Based on the above analysis, we may construct the following algorithm.

input $A = \{[s_1, f_1), [s_2, f_2), ..., [s_n, f_n)\}$ with $f_1 \leq f_2 \leq \cdots \leq f_n$.
$B \leftarrow \emptyset$;
while $A \neq \emptyset$ **do begin**
    $[s_j, f_j) \leftarrow \text{argmin}_{[s_i,f_i)\in A} f_i$;
    $B \leftarrow B \cup \{[s_j, f_j)\}$;
    for every $[s_i, f_i) \in A$ **do**
        if $s_i < f_j$ **then** $w_i \leftarrow w_i - w_j$;
    for every $[s_i, f_i) \in A$ **do**
        if $w_i \leq 0$ **then** $A \leftarrow A - \{[s_i, f_i)\}$;
end-while;
$[s_k, f_k) \leftarrow \text{argmax}_{[s_i,f_i)\in B} f_i$;
$Opt \leftarrow \{[s_k, f_k)\}$;
$B \leftarrow B - \{[s_k, f_k)\}$;
while $B \neq \emptyset$ **do**
    $[s_h, f_h) \leftarrow \text{argmax}_{[s_i,f_i)\in B} f_i$;
    if $s_k \geq f_h$,
        then $Opt \leftarrow Opt \cup \{[s_h, f_h)\}$
            and $[s_k, f_k) \leftarrow [s_h, f_h)$;
    $B \leftarrow B - \{[s_h, f_h)\}$;
end-while;
output Opt.

Now, we run this algorithm on an example as shown in Fig. 4.8.

Next, we study the second example.

Consider a directed graph $G = (V, E)$. A subgraph T is called an *arborescence* rooted at a vertex r if T satisfies the following two conditions:

(a) If ignore direction on every arc, then T is a tree.

(b) For any vertex $v \in V$, T contains a directed path from r to v.

Let T be an arborescence with root r. Then for any vertex $v \in V - \{r\}$, there is exactly one arc coming to v. This property is quite important.

Lemma 4.4.3. *Suppose T is obtained by choosing one incoming arc at each vertex $v \in V - \{r\}$. Then T is an arborescence if and only if T does not contain a directed cycle.*

Proof. Note that the number of arcs in T is equal to $|V| - 1$. Thus, condition (b) implies the connectivity of T when ignore direction, which implies condition (a). Therefore, if T is not an arborescence, then condition (b) does not hold, i.e., there exists $v \in V - \{r\}$ such that there does not exist

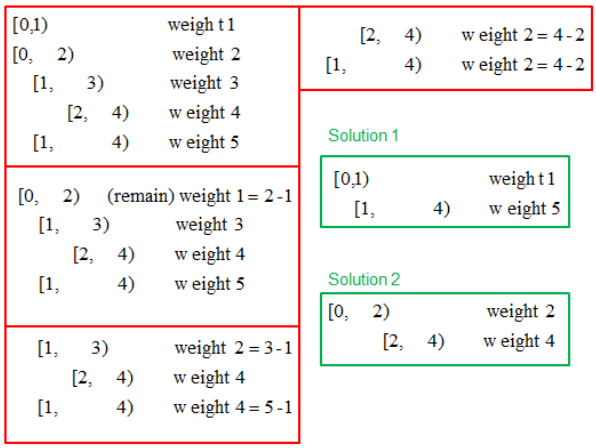


Figure 4.8: An example for weighted activity selection.

a directed path from r to v. Now, T contains an arc (v_1, v) coming to v with $v_1 \neq r$, an arc (v_2, v_1) coming to v_1 with $v_2 \neq v$, and so on. Since the directed graph G is finite. The sequence $(v, v_1, v_2, ...)$ must contains a cycle.

Conversely, if T contains a cycle, then T is not an arborescence by the definition. This completes the proof of the lemma. □

Now, we consider the minimum arborescence problem.

Problem 4.4.4 (Minimum Arborescence)**.** *Given a directed graph $G = (V, E)$ with positive arc weight $w : E \to R^+$, and a vertex $r \in V$, compute an arborescence with root r to minimize total arc weight.*

The following special case gives a basic idea for a local ratio method.

Lemma 4.4.5. *Suppose for each vertex $v \in V - \{r\}$, all arcs coming to v have the same weight. Then every arborescence with root r is optimal for the* Min Arborescence *problem.*

Proof. It follows immediately from the fact that each arborescence contains exactly one arc coming to v for each vertex $v \in V - \{r\}$. □

Since arcs coming to r are useless in construction of an arborescence with root r, we remove them at beginning. For each $v \in V - \{r\}$, let w_v denote the minimum weight of an arc coming to v. By Lemma 4.4.5, we may decompose the minimum arborescence problem into two subproblems.

In the first one, every arc coming to a vertex v has weight w_v. In the second one, every arc e coming to a vertex v has weight $w(e) - w_v$, so that every vertex $v \in V - \{r\}$ has a coming arc with weight 0. If all 0-weight arcs contain an arborescence T, then T must be an optimal solution for the second subproblem and hence also an optimal solution for the original problem. If not, then by Lemma 4.4.3, there exists a directed cycle with weight 0. Contract this cycle into one vertex. Repeat the decomposition and the contraction until an arborescence with weight 0 is found. Then in backward direction, we may find a minimum arborescence for the original weight. An example is shown in Fig. 4.9.

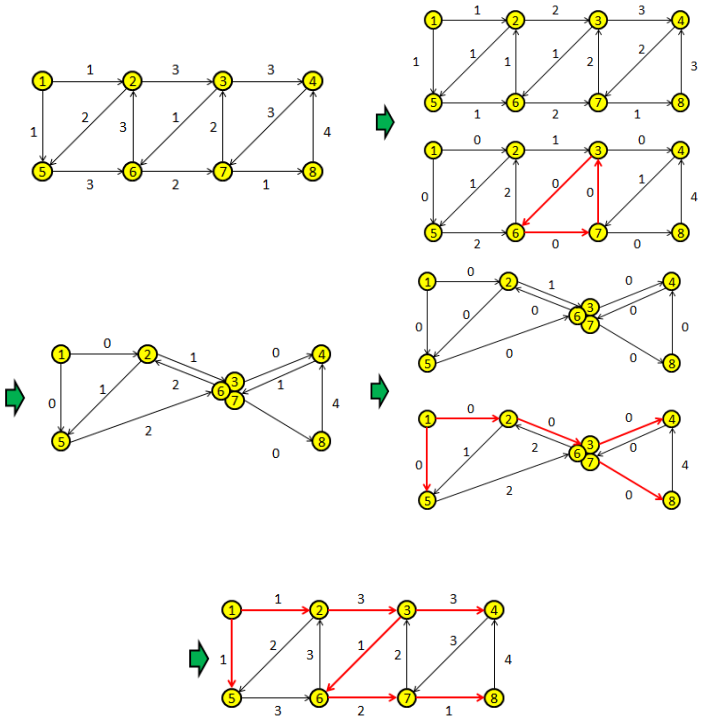


Figure 4.9: An example for computing a minimum arborescence.

According to above analysis, we may construct the following algorithm.

Local Ratio Algorithm for Minimum Arborescence

input a directed graph $G = (V, E)$ with arc weight $w : E \to R^+$, and a root $r \in V$.
outpu An arborescence T with root r.

```
C ← ∅;
repeat
    for every v ∈ V \ {r} do
        let e_v be the one with minimum weight among arcs coming to v and
        T ← T ∪ {e_v};
        for every edge e = (u, v) coming to v do
            w(e) ← w(e) − w_v;
    if T contains a cycle C
      thenC ← C ∪ {C} and
            contract cycle C into one vertex in G and T;
until T does not contain a cycle;
for every C ∈ C do
    add C into T and properly delete an arc of C.
return T.
```

Exercises

1. Suppose that for every cut of the graph, there is a unique light edge crossing the cut. Show that the graph has a unique minimum spanning tree. Does the converse hold? If not, please give a counterexample.

2. Consider a finite set S. Let $\mathcal{I}_k$ be the collection of all subsets of S with size at most k. Show that $(S, \mathcal{I}_k)$ is a matroid.

3. Solve following instance of the unit-time task scheduling problem.

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

   Please solve the problem again when each penalty w_i is replaced by $80 - w_i$.

4. Suppose that the characters in an alphabet is ordered so that their frequencies are monotonically decreasing. Prove that there exists an optimal prefix code whose codeword length are monotonically increasing.

5. Show that if $(S, \mathcal{I})$ is a matroid, then $(S, \mathcal{I}')$ is a matroid, where

$$\mathcal{I}' = \{A' \mid S - A' \text{ contains some maximal } A \in \mathcal{I}\}.$$

That is, the maximal independent sets of $(S, \mathcal{I}')$ are just complements of the maximal independent sets of $(S, \mathcal{I})$.

6. Suppose that a set of activities are required to schedule in a large number of lecture halls. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

7. Consider a set of n files, $f_1, f_2, \ldots, f_n$, of distinct sizes $m_1, m_2, \ldots, m_n$, respectively. They are required to be recorded sequentially on a single tape, in some order, and retrieve each file exactly once, in the reverse order. The retrieval of a file involves rewinding the tape to the beginning and then scanning the files sequentially until the desired file is reached. The *cost* of retrieving a file is the sum of the sizes of the files scanned plus the size of the file retrieved. (Ignore the cost of rewinding the tape.) The *total cost* of retrieving all the files is the sum of the individual costs.

   (a) Suppose that the files are stored in some order $f_{i_1}, f_{i_2}, \ldots, f_{i_n}$. Derive a formula for the total cost of retrieving the files, as a function of n and the m_{i_k}'s.

   (b) Describe a greedy strategy to order the files on the tape so that the total cost is minimized, and prove that this strategy is indeed optimal.

8. We describe here one simple way to merge two sorted lists: Compare the smallest numbers of the two lists, remove the smaller one of the two from the list it is in, and place it somewhere else as the first number of merged list. We now compare the smallest numbers of the two lists of remaining numbers and place the smaller one of the two as the second number of the merged list. This step can be repeated until the merged list is completely built up. Clearly, in the worst case it takes $n_1 + n_2 - 1$ comparisons to merge two sorted lists which have n_1 and n_2 numbers, respectively. Given m sorted lists, we can select two of them and merge these two lists into one. We can then select two lists from the $m - 1$ sorted lists and merge them into one. Repeating this step, we shall eventually end up with one merged list. Describe a general algorithm for determining an order in which m sorted lists A_1, A_2, ..., A_m are to be merged so that the total number of comparisons is minimum. Prove that your algorithm is correct.

9. Let $G = (V, E)$ be a connected undirected graph. The distance between two vertices x and y, denoted by $d(x, y)$, is the number of edges on the shortest path between x and y. The *diameter* of G is the maximum of $d(x, y)$ over all pairs (x, y) in $V \times V$. In the remainder of this problem, assume that G has at least two vertices.

   Consider the following algorithm on G: Initially, choose arbitrarily $x_0 \in V$. Repeatedly, choose x_{i+1} such that $d(x_{i+1}, x_i) = \max_{v \in V} d(v, x_i)$ until $d(x_{i+1}, x_i) = d(x_i, x_{i-1})$.

   Can this algorithm always terminates? When it terminates, is $d(x_{i+1}, x_i)$ guaranteed to equal the diameter of G? (Prove or disprove your answer.)

10. Consider a graph $G = (V, E)$ with positive edge weight $c : E \to R^+$. Show that for any spanning tree T and the minimum spanning tree T^*, there exists a one-to-one onto mapping $\rho : E(T) \to E(T^*)$ such that $c(\rho(e)) \leq c(e)$ for every $e \in E(T)$ where $E(T)$ denotes the edge set of T.

11. Given a strongly connected directed graph $G = (V, E)$ with nonnegative edge weight $w : E \to R_+$ and a node $r \in V$, design a polynomial-time algorithm to compute the minimum weight arborescence rooted at r. (An arborescence rooted at r is a directed tree that, for every $x \in V$, contains a directed path from r to x.)

12. Consider a point set P in the Euclidean plane. Let R be a fixed positive number. A steinerized spanning tree on P is a tree obtained from a spanning tree on P by putting some Steiner points on its edges to break them into pieces each of length at most R. Show that the steinerized spanning with minimum number of Steiner points is obtained from the minimum spanning tree.

13. Consider a graph $G = (V, E)$ with edge weight $w : E \to R^+$. Show that the spanning tree T which minimizes $\sum_{e \in E(T)} \|e\|^\alpha$ for any fixed $1 < \alpha$ is the minimum spanning tree, i.e., the one which minimizes $\sum_{e \in E(T)} \|e\|$.

14. Let $\mathcal{B}$ be the family of all maximal independent subsets of an independent system $(E, \mathcal{I})$. Then $(E, \mathcal{I})$ is a matroid if and only if for any nonnegative function $c(\cdot)$, Algorithm 14 produces an optimal solution for the problem $\min\{c(I) \mid I \in \mathcal{B}\}$.

15. Consider a complete bipartite graph $G = (U, V, E)$ with $|U| = |V|$. Let $c(\cdot)$ be a nonnegative function on E such that for any $u, u' \in V_1$ and $v, v' \in V_2$,

$$c(u,v) \geq \max(c(u,v'), c(u',v)) \Longrightarrow c(u,v)+c(u',v') \geq c(u,v')+c(u',v).$$

    (a) Design a greedy algorithm for problem $\max\{c(\cdot) \mid I \in \mathcal{I}\}$.

    (b) Design a greedy algorithm for problem $\min\{c(\cdot) \mid I \in \mathcal{I}\}$.

16. Given n intervals $[s_i, f_i)$ each with weight $w_i \geq 0$, design an algorithm to compute the maximum-weight subset of disjoint intervals.

17. Give a counterexample to show that an independent system with all maximal independent sets of the same size may not be a matroid.

18. Consider the following scheduling problem. There are n jobs, $i = 1, 2, ..., n$, and there is one super-computer and n identical PCs. Each jobs needs to be pre-processed first on the supercomputer and then finished by one of the PCs. The time required by job i on the supercomputer is p_i; $i = 1, 2, ..., n$; the time required on a PC for job i is f_i; $i = 1, 2, ..., n$. Finishing several jobs can be done in parallel since we have as many PCs as there are jobs. But the supercomputer processes only one job at a time. The input to the problem are the vectors $p = [p_1, p_2, ..., p_n]$ and $f = [f_1, f_2, ..., f_n]$. The objective of the problem is to minimize the completion time of last job (i.e., minimize the maximum completion time of any job). Describe a greedy algorithm that solves the problem in $O(n \log n)$ time. Prove that your algorithm is correct.

19. Design a local ratio algorithm to compute a minimum spanning tree.

Historical Notes

The greedy algorithm is an important class of computer algorithms with self-reducibility, for solving combinatorial optimization problems. It uses the greedy strategy in construction of an optimal solution. There are several variations of greedy algorithms, e.g., Prime algorithm for minimum spanning tree in which greedy principal applies not globally, but a subset of edges.

Could Prim algorithm be considered as a local search method? The answer is no. Actually, in local search method, a solution is improved by finding a better one within a local area. Therefore, the greedy strategy

applies to search for the best moving from a solution to another better solution. This can also be called as incremental method, which will be introduced in next chapter.

The minimum spanning tree has been studied since 1926 [38]. Its history can be found a remarkable article [35]. The best known theoretical algorithm is due to Bernard Chazelle [36, 37]. The algorithm runs almost in $O(m)$ time. However, it is too complicated to implement and hence may not be practical.

Chapter 5

Incremental Method and Network Flow

"Change is incremental. Change is small."
- Theodore Melfi

In this chapter, we study the incremental method which is very different from those methods in previous chapters. This method does not use the self-reducibility. It starts from a feasible solution and in each iteration, computation moves from a feasible solution to another feasible solution by improving the objective function value. The incremental method has been used in study of many problems, especially in study of network flow.

5.1 Maximum Flow

Consider a *flow network* $G = (V, E)$, i.e., a directed graph with a non-negative capacity $c(u, v)$ on each arc (u, v), and two given nodes, *source* s and *sink* t. An example of the flow network is shown in Fig. 5.1. For simplicity of description for flow, we may extend capacity $c(u, v)$ to every pair of nodes u and v by defining $c(u, v) = 0$ if $(u, v) \notin E$.

A *flow* in flow network G is a real function f on $V \times V$ satisfying following three conditions:

1. (Capacity Constraint) $f(u, v) \leq c(u, v)$ for every $u, v \in V$.
2. (Skew Symmetry) $f(u, v) = -f(v, u)$ for all $u, v \in V$.
3. (Flow Conservation) $\sum_{v \in V \setminus \{u\}} f(u, v) = 0$ for every $u \in V \setminus \{s, t\}$.

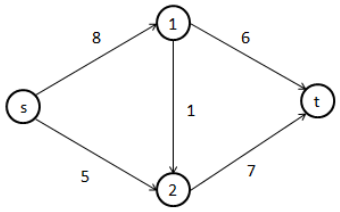


Figure 5.1: A flow network.

The flow has following properties.

Lemma 5.1.1. *Let f be a flow of network $G = (V, E)$. Then following holds.*

(a) If $(u, v) \notin E$ and $(v, u) \notin E$, then $f(u, v) = 0$.
(b) For any $x \in V \setminus \{s, t\}$, $\sum_{f(u,x)>0} f(u, x) = \sum_{f(x,v)>0} f(x, v)$.
(c) $\sum_{f(s,v)>0} f(s, v) - \sum_{f(u,s)>0} f(u, s) = \sum_{f(u,t)>0} f(u, t) - \sum_{f(t,v)>0} f(t, v)$.

Proof. (a) By capacity constraint, $f(u, v) \leq c(u, v) = 0$ and $f(v, u) \leq c(v, u) = 0$. By skew symmetric, $f(u, v) = -f(v, u) \geq 0$. Hence, $f(u, v) = 0$.

(b) By flow conservation, for any $x \in V \setminus \{s, t\}$,

$$\sum_{f(x,u)<0} f(x, u) + \sum_{f(x,v)>0} f(x, v) = \sum_{v \in V} f(x, v) = 0.$$

By skew symmetry,

$$\sum_{f(u,x)>0} f(u, x) = -\sum_{f(x,u)<0} f(x, u) = \sum_{f(x,v)>0} f(x, v).$$

(c) By (b), we have

$$\sum_{x \in V \setminus \{s,t\}} \sum_{f(u,x)>0} f(u, x) = \sum_{x \in V \setminus \{s,t\}} \sum_{f(x,v)>0} f(x, v).$$

For $(y, z) \in E$ with $y, z \in V \setminus \{s, t\}$, if $f(y, z) > 0$, then $f(y, z)$ appears in both the left-hand and the right-hand sides and hence it will be cancelled. After cancellation, we obtain

$$\sum_{f(s,v)>0} f(s, v) + \sum_{f(t,v)>0} = \sum_{f(u,s)>0} f(u, s) + \sum_{f(u,t)>0} f(u, t).$$

□

Now, the *flow value* of f is defined to be

$$|f| = \sum_{f(s,v)>0} f(s,v) - \sum_{f(u,s)>0} f(u,s) = \sum_{f(u,t)>0} f(u,t) - \sum_{f(t,v)>0} f(t,v).$$

In case that the source s does not have arc coming in, we have

$$|f| = \sum_{f(s,v)>0} f(s,v).$$

In general, we can also represent $|f|$ as

$$|f| = \sum_{v \in V \setminus \{s\}} f(s,v) = \sum_{u \in V \setminus \{t\}} f(u,t).$$

In Fig. 5.2, arc labels with underline give a flow. This flow has value 11.

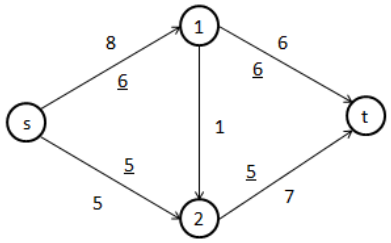


Figure 5.2: A flow in network.

The maximum flow problem is as follows.

Problem 5.1.2 (Maximum Flow)**.** *Given a flow network $G = (V, E)$ with arc capacity $c : V \times V \to R_+$, a source s and a sink t, find a flow f with maximum flow value. Usually, assume that s does not have in-coming arc and t does not have out-going arc.*

An important tool for study of the maximum flow problem is the residual network. The *residual network* for a flow f in a network $G = (V, E)$ with capacity c is the flow network with $G_f(V, E')$ with capacity $c'(u, v) = c(u, v) - f(u, v)$ for any $u, v \in V$ where $E' = \{(u.v) \in V \times V \mid c'(u, v) > 0\}$. For example, the flow in Fig.5.2 has its residual network as shown in Fig.5.3. Two important properties of the residual network are included in following lemmas.

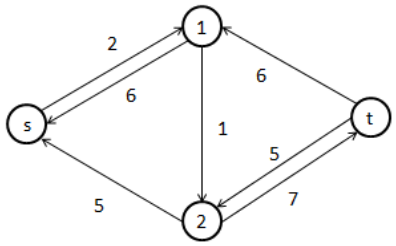


Figure 5.3: The residual network G_f of the flow f in Fig. 5.2.

Lemma 5.1.3. *Suppose f' is a flow in the residual network G_f. Then $f+f'$ is a flow in network G and $|f+f'| = |f| + |f'|$.*

Proof. For any $u, v \in V$, since $f'(u,v) \le c'(u,v) = c(u,v) - f(u,v)$, we have $f(u,v) + f'(u,v) \le c(u,v)$, that is, $f + f'$ satisfies the capacity constraint. Moreover, $f(u,v) + f'(u,v) = -f(v,u) - f'(v,u) = -(f(v,u) + f'(v,u))$ and for every $u \in V \setminus \{s,t\}$,

$$\sum_{v \in V\setminus\{u\}} (f+f')(u,v) = \sum_{v \in V\setminus\{u\}} f(u,v) + \sum_{v \in V\setminus\{u\}} f'(u,v) = 0.$$

This means that $f+f'$ satisfies the skew symmetry and the flow conservation conditions. Therefore, $f + f'$ is a flow. Finally,

$$|f+f'| = \sum_{v \in V\setminus\{s\}} (f+f')(s,v) = \sum_{v \in V\setminus\{s\}} f(s,v) + \sum_{v \in V\setminus\{s\}} f'(s,v) = |f| + |f'|.$$

□

Lemma 5.1.4. *Suppose f' is a flow in the residual network G_f. Then $(G_f)_{f'} = G_{f+f'}$, i.e., the residual network of f' in network G_f is the residual network of $f + f'$ in network G.*

Proof. The arc capacity of $(G_f)_{f'}$ is

$$c'(u,v) - f'(u,v) = c(u,v) - f(u,v) - f'(u,v) = c(u,v) - (f+f')(u,v)$$

which is the same as that in $G_{f+f'}$. □

In order to get a flow with larger value, Lemmas 5.1.3 and 5.1.4 suggest us to find a flow f' in G_f with $|f'| > 0$. A simple way is to find a path P from s to t and define f' by

$$f'(u,v) = \begin{cases} \min_{(x,y)\in P} c'(x,y) & \text{if } (u,v) \in P, \\ 0 & \text{otherwise.} \end{cases}$$

The following algorithm is motivated from this idea.

Algorithm 15 Ford-Fulkerson Algorithm for Maximum Flow

Input: A flow network $G = (V, E)$ with capacity function c, a source s and a sink t.

Output: A flow f.

1: $G \leftarrow G$;
2: $f \leftarrow 0$; (i.e., $\forall u, v \in V, f(u,v) = 0$)
3: **while** there exists a path P from s to t in G **do**
4: $\quad \delta \leftarrow \min\{c(u,v) \mid (u,v) \in P\}$ and
5: $\quad$ send a flow f' with value δ from s to t along path P;
6: $\quad G \leftarrow G_{f'}$;
7: $\quad f \leftarrow f + f'$;
8: **end while**
9: **return** f.

Using this algorithm, an example is shown in Fig.5.4. The s-t path of the residual network is called an *augmenting path* and hence Ford-Fulkerson algorithm is an augmenting path algorithm.

Now, we may have two questions: Can Ford-Fulkerson algorithm stop within finitely many steps? When Ford-Fulkerson algorithm stops, does output reach the maximum?

The answer for the first question is negative, that is, Ford-Fulkerson algorithm may run infinitely many steps. A counterexample can be obtained from the one as shown in Fig.5.5 by setting $m = \infty$. However, with certain condition, Ford-Fulkerson algorithm will run within finitely many steps.

Theorem 5.1.5. *If every arc capacity is a finite integer, then Ford-Fulkerson algorithm runs within finitely many steps.*

Proof. The flow value has upper bound $\sum_{(s,v)\in E} c(s,v)$. Since every arc capacity is integer, in each step, the flow value will be increased by at least one. Therefore, the algorithm will run within at most $\sum_{(s,v)\in E} c(s,v)$ steps. □

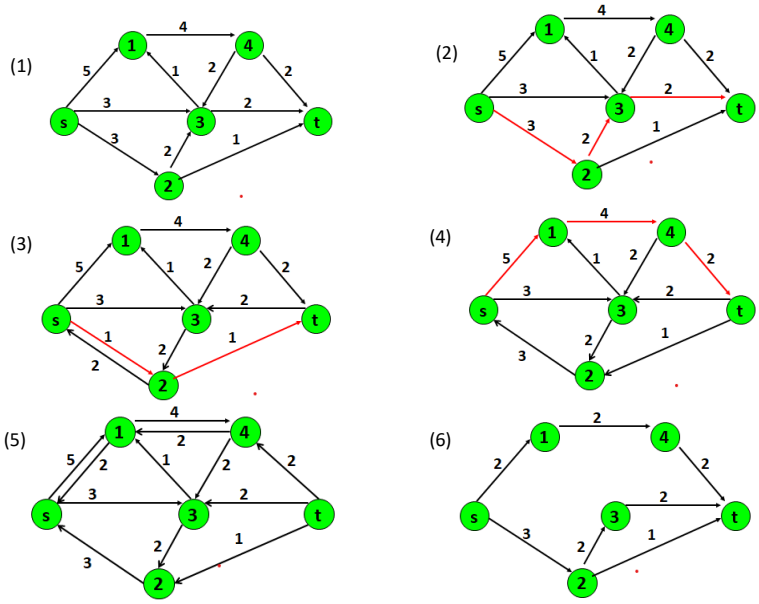


Figure 5.4: An example for using Ford-Fulkerson Algorithm.

The answer for the second question is positive. Actually, we have following.

Theorem 5.1.6. *A flow f is maximum if and only if its residual network G_f does not contain a path from source s to sink t.*

To prove this theorem, let us first show a lemma.

A partition (S,T) of V is called an *s-t cut* if $s \in S$ and $t \in T$. The capacity of an s-t cut is defined by

$$\mathrm{CAP}(S,T) = \sum_{u \in S, v \in T} c(u,v).$$

Lemma 5.1.7. *Let (S,T) be a s-t cut. Then for any flow f,*

$$|f| = \sum_{f(u,v)>0, u \in S, v \in T} f(u,v) - \sum_{f(v,u)>0, u \in S, v \in T} f(v,u) \leq CAP(S,T).$$

Proof. By Lemma 5.1.1(b),

$$\sum_{x \in S \setminus \{s\}} \sum_{f(u,x)>0} f(u,x) = \sum_{x \in S \setminus \{s\}} \sum_{f(x,v)>0} f(x,v).$$

Simplifying this equation, we will obtain

$$\begin{aligned} & \sum_{f(s,x)>0, x\in S\setminus\{s\}} f(s,x) + \sum_{u\in T, x\in S\setminus\{s\}, f(u,x)>0} f(u,x) \\ = & \sum_{f(x,s)>0, x\in S\setminus\{s\}} f(x,s) + \sum_{v\in T, x\in S\setminus\{s\}, f(x,v)>0} f(x,v). \end{aligned}$$

Thus,

$$\begin{aligned} & \sum_{f(s,x)>0} f(s,x) + \sum_{u\in T, x\in S, f(u,x)>0} f(u,x) \\ = & \sum_{f(x,s)>0} f(x,s) + \sum_{v\in T, x\in S, f(x,v)>0} f(x,v), \end{aligned}$$

that is,

$$\begin{aligned} |f| &= \sum_{f(s,x)>0} f(s,x) - \sum_{f(x,s)>0} f(x,s) \\ &= \sum_{v\in T, x\in S, f(x,v)>0} f(x,v) - \sum_{u\in T, x\in S, f(u,x)>0} f(u,x) \\ &\leq \sum_{v\in T, x\in S, f(x,v)>0} f(x,v) \\ &\leq \sum_{x\in S, v\in T} c(x,v). \end{aligned}$$

□

Now, we prove Theorem 5.1.6.

Proof of Theorem 5.1.6. If residual network G_f contains a path from source s to sink t, then a positive flow can be added to f and hence f is not maximum. Next, we assume that G_f does not contain a path from s to t.

Let S be the set of all nodes each of which can be reached by a path from s. Set $T = V \setminus S$. Then (S,T) is a partition of V such that $s \in S$ and $t \in T$. Moreover, G_f has no arc from S to T. This fact implies two important facts:

(a) For any arc (u,v) with $u \in S$ and $v \in T$, $f(u,v) = c(u,v)$.

(b) For any arc (v,u) with $u \in S$ and $v \in T$, $f(v,u) = 0$.

Based on these two facts, by Lemma 5.1.7, we obtain that

$$|f| = \sum_{u\in S, v\in T} c(u,v).$$

Hence, f is a maximum flow. □

Corollary 5.1.8. *The maximum flow is equal to minimum s-t cut capacity.*

Finally, we remark that Ford-Fulkerson algorithm is not polynomial-time. An couterexample is given in Fig. 5.5. On this counterexample, the algorithm runs in $2m$ steps. However, the input size is $O(\log m)$. Clearly, $2m$ is not a polynomial with respect to $O(\log m)$.

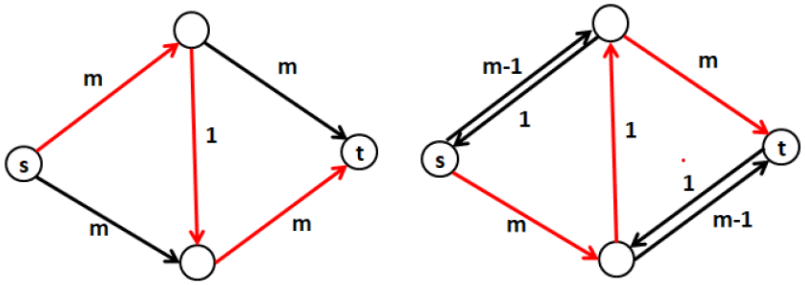


Figure 5.5: Ford-Fulkerson Algorithm runs not in polynomial time.

5.2 Edmonds-Karp Algorithm

To improve the running time of Ford-Fulkerson algorithm, a simple modification is found which works very well, that is, at each iteration, find a shortest augmenting path instead of an arbitrary augmenting. By the shortest, we mean the path contains the minimum number of arcs. This algorithm is called Admonds-Karp algorithm (Algorithm 16).

An example for using Admonds-Karp algorithm is shown in Fig.5.6. Compared with Fig.5.4, we may find that input flow network is same, but obtained maximum flows are different. Thus, for this input flow network, there are two different maximum flow. Actually, in this case, there are infinitely many maximum flows. The reader may prove it as an exercise.

To estimate the running time, let us study some properties of Admonds-Karp algorithm.

Let $\delta_f(x)$ denote the shortest path distance from source s to node x in the residual network G_f of flow f where each arc is considered to have unit distance.

Lemma 5.2.1. *When Edmonds-Karp algorithm runs, $\delta_f(x)$ increases monotonically with each flow augmentation.*

Algorithm 16 Admonds-Karp Algorithm for Maximum Flow

Input: A flow network $G = (V, E)$ with capacity function c, a source s and a sink t.

Output: A flow f.

1: $G \leftarrow G$;
2: $f \leftarrow 0$; (i.e., $\forall u, v \in V, f(u, v) = 0$)
3: **while** there exists a path from s to t in G **do**
4: find a shortest path P from s to t;
5: set $\delta \leftarrow \min\{c(u, v) \mid (u, v) \in P\}$ and
6: send a flow f' with value δ from s to t along path P;
7: $G \leftarrow G_{f'}$;
8: $f \leftarrow f + f'$;
9: **end while**
10: **return** f.

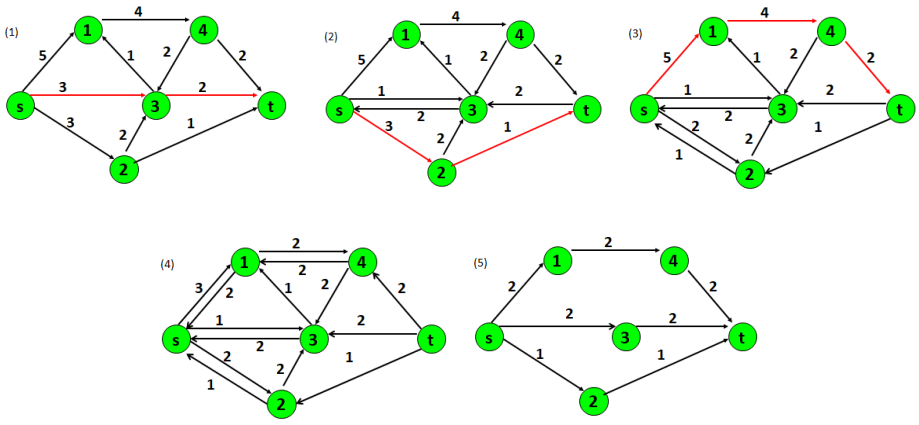


Figure 5.6: An example for using Admonds-Karp Algorithm.

Proof. For contradiction, suppose flow f' is obtained from flow f through an augmentation with path P and $\delta_{f'}(v) < \delta_f(v)$ for some node v. Without loss of generality, assume $\delta_{f'}(v)$ reaches the smallest value among such v, i.e.,

$$\delta_{f'}(u) < \delta_{f'}(v) \Rightarrow \delta_{f'}(u) \geq \delta_f(u).$$

Suppose arc (u, v) is on the shortest path from s to v in $G_{f'}$. Then $\delta_{f'}(u) = \delta_{f'}(v) - 1$ and hence $\delta_{f'}(u) \geq \delta_f(u)$. Next, let us consider two cases.

Case 1. $(u, v) \in G_f$. In this case, we have

$$\delta_f(v) \leq \delta_f(u) + 1 \leq \delta_{f'}(u) + 1 = \delta_{f'}(v),$$

a contradiction.

Case 2. $(u, v) \notin G_f$. Then arc (v, u) must lie on the augmenting path P in G_f (Fig.5.7). Therefore,

$$\delta_f(v) = \delta_f(u) - 1 \leq \delta_{f'}(u) - 1 = \delta_{f'}(v) - 2 < \delta_{f'}(v),$$

a contradiction. □

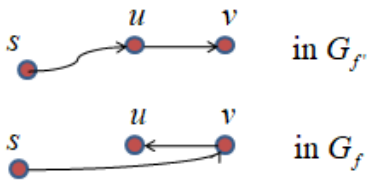


Figure 5.7: Proof of Lemma 5.2.1.

An arc (u, v) is critical in residual network G_f if (u, v) has the smallest capacity in the shortest augmenting path in G_f.

Lemma 5.2.2. *Each arc (u, v) can be critical at most $(|V| + 1)/2$ times.*

Proof. Suppose arc (u, v) is critical in G_f. Then (u, v) will disappear in next residual network. Before (u, v) appears again, $v, u)$ has to appear in augmenting path of a residual network $G_{f'}$. Thus, we have

$$\delta_{f'}(u) = \delta_{f'}(v) + 1.$$

Since $\delta_f(v) \leq \delta_{f'}(v)$, we have

$$\delta_{f'}(u) = \delta_{f'}(v) + 1 \geq \delta_f(v) + 1 = \delta_f(u) + 2.$$

By Lemma 5.2.1, the shortest path distance from s to u will increase by $2(k-1)$ when arc (u, v) can be critical k times. Since this distance is at most $|V| - 1$, we have $2(k-1) \leq |V| - 1$ and hence $k \leq (|V| + 1)/2$. □

Now, we establish a theorem on running time.

Theorem 5.2.3. *Edmonds-Karp algorithm runs in time* $O(|V| \cdot |E|^2)$.

Proof. In each augmentation, there exists a critical arc. Since each arc can be critical $(|V| + 1)/2$ times, there are at most $O(|V| \cdot |E|)$ augmentations. In each augmentation, finding the shortest path takes $O(|E|)$ time and operations on the augmenting path take also $O(|E|)$ time. Putting all together, Edmonds-Karp algorithm runs in time $O(|V| \cdot |E|^2)$. □

Note that above theorem does not require that all arc capacities are integer. Therefore, the modification of Admonds and Karp has two folds: (1) Make the algorithm halt within finitely many iterations and (2) the number of iterations is bounded by a polynomial.

5.3 Bipartite Matching

The maximum flow has many applications. One of them is to deal with the maximum bipartite matching.

Consider a graph $G = (V, E)$. A subset of edges is called a *matching* if edges in the subset are not adjacent each other. In other words, a matching is an independent edge subset. A *bipartite* matching is a matching in a bipartite graph.

Problem 5.3.1 (Maximum Bipartite Matching)**.** *Given a bipartite graph* (U, V, E)*, find a matching with maximum cardinality.*

This problem can be transformed into a maximum flow problem as follows. Add a source node s and a sink node t. Connect s to every node u in U by adding an arc (s, u). Connect every node v in V to t by adding an arc (v, t). Add to every edge in E the direction from U to V. Finally, assign every arc with unit capacity. An example is shown in Fig. 5.8.

Motivated from observation on the example in Fig. 5.8, we may have questions:

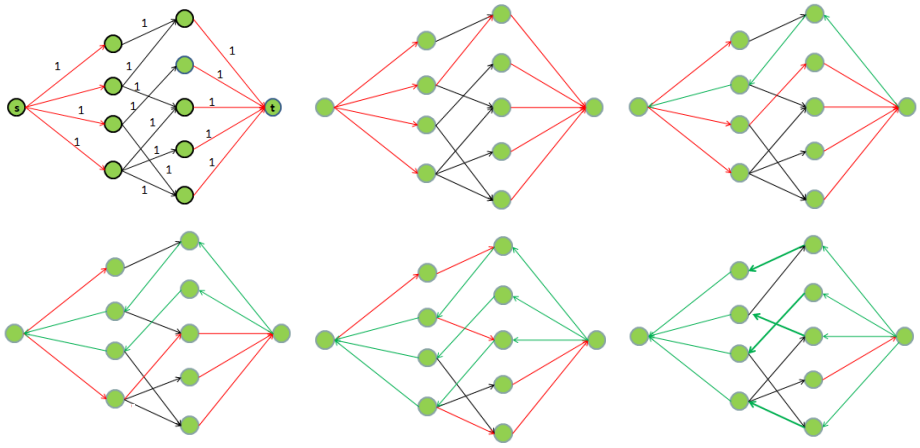


Figure 5.8: Maximum bipartite matching is transformed to maximum flow.

(1) Can we do augmentation directly in bipartite graph without putting it in a flow network?

(2) Can we perform the first three augmentations in the same time?

For both questions, the answer is yes. Let us explain the answer one by one. To give yes-answer for the first question, we need to define the augmenting path as follows.

Consider a matching M in a bipartite graph $G = (U, V, E)$. Let us call every edge in M as *matched* edge and every edge not in M as *unmatched* edge. A node v is called a *free node* if v is not an ending point of a matched edge. The *augmenting path* is now defined to be a path satisfying following two conditions:

(a) It is an *alternating path*, that is, edges on the path is alternatively unmatched and matched.

(b) The path is between two free nodes.

Clearly, on an augmenting path, turn all matched edges to unmatched and turn all unmatched edges to matched. Then considered matching will become a matching with one more edges. Therefore, if a matching M has an augmenting path, then M cannot be maximum. The following theorem indicates that the inverse holds.

Theorem 5.3.2. *A matching M is maximum if and only if M does not have an augmenting path.*

Proof. Let M be a matching without augmenting path. For contradiction,

suppose M is not maximum. Let M^* be a maximum matching. Then $|M| < |M^*|$. Consider $M \oplus M^* = (M \setminus M^*) \cup (M^* \setminus M)$in which every node has degree at most two (Fig. 5.9).

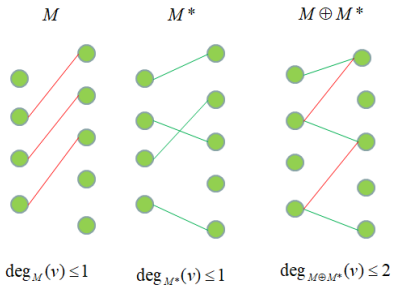


Figure 5.9: $M \oplus M^*$.

Hence, it is disjoint union of paths and cycles. Since each node with degree two must be incident to two edges belonging in M and M', respectively. Those paths and cycles must be alternative. They can be classified into four types as shown in Fig. 5.10.

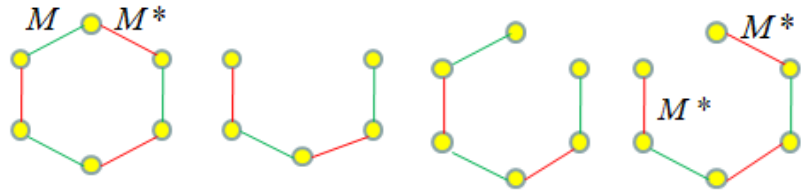


Figure 5.10: Connected components of $M \oplus M^*$.

Note that in each of the first three types of connected components, the number of edges in M is not less than the number of edges in M^*. Since $|M| < |M^*|$, we have $|M \setminus M^*| < |M^* \setminus M|$. Therefore, the connected component of the fourth type must exist, that is, M has an augmenting path, a contradiction. □

We now return to the question on augmentation of several paths at the same time. The following algorithm is the result of positive answer.

Algorithm 17 Hopcroft-Karp Algorithm for Maximum Bipartite Matching

Input: A bipartite graph $G = (U, V, E)$.
Output: A maximum matching M.

1: $M \leftarrow$any edge;
2: **while** there exists an augmenting path **do**
3: find a maximal set of disjoint augmenting paths $\{P_1, P_2, ..., P_k\}$;
4: $M \leftarrow M \oplus (P_1 \cup P_2 \cup \cdots P_k)$;
5: **end while**
6: **return** M.

We next analyze Hopcroft-Karp algorithm.

Lemma 5.3.3. *In each iteration, the length of the shortest augmenting path is increased by at least two.*

Proof. Suppose matching M' is obtained from matching M through augmentation on a maximal set of shortest augmenting paths, $\{P_1, P_2, ..., P_k\}$, for M. Let P be a shortest augmenting path for M'. If P is disjoint from $\{P_1, P_2, ..., P_k\}$, then P is also an augmenting path for M. Hence, the length of P is longer than the length of P_1. Note that the augmenting path must have odd length. Therefore, the length of P at-least-two longer than the length of P_1.

Next, assume that P has an edge lying in P_i for some i. Note that every augmenting path has two endpoints in U and V, respectively. Let u and v be two endpoints of P, and u_i and v_i two endpoints of P_i where $u, u_i in U$ and $v, v_i \in V$. Without loss of generality, assume that (x, y) is the edge lying on P and also on some P_i such that no such edge exists from y to v. Clearly,

$$\text{dist}_P(y, v) \geq \text{dist}_{P_i}(y, v_i), \tag{5.1}$$

where $\text{dist}_P(y, v)$ denotes the distance between y and v on path P. In fact, if $\text{dist}_P(y, v) < \text{dist}_{P_i}(y, v_i)$, then replacing the piece of P_i between y and v_i by the piece of P between y and v, we obtain an augmenting path for M, shorter than P_i, contradicting to shortest property of P_i. Now, we claim that following holds.

$$\text{dist}_{P_i}(u_i, y) + 1 = \text{dist}_{P_i}(u_i, x) \leq \text{dist}_P(u, x) = \text{dist}_P(u, y) - 1. \tag{5.2}$$

To prove this claim, we may put the bipartite graph into a flow network as shown in Fig. 5.8. Then every augmenting path receive a direction from U to V and the claim can be proved as follows.

First, note that on path P, we assumed that the piece from y to v is disjoint from all $P_1, P_2, ..., P_k$. This assumption implies that edge (x, y) is in direction from x to y on P, so that $\text{dist}_P(u, x) = \text{dist}_P(u, y) - 1$.

Secondly, note that edge (x, y) also appears on P_i and after augmentation, every edge in P_i must change its direction. Thus, edge (x, y) is in direction from y to x on P_i. Hence, $\text{dist}_{P_i}(u_i, y) + 1 = \text{dist}_{P_i}(u_i, x)$.

Thirdly, by Lemma 5.2.1, we have $\text{dist}_{P_i}(u_i, x) \leq \text{dist}_P(u, x)$.

Finally, putting (5.1) and (5.2) together, we obtain

$$\text{dist}_{P_i}(u_i, v_i) + 2 \leq \text{dist}_P(u, v).$$

□

Theorem 5.3.4. *Hopcroft-Karp algorithm computes a maximum bipartite matching in time* $O(|E|\sqrt{|V|})$.

Proof. In each iteration, it takes $O(|E|)$ time to find a maximal set of shortest augmenting paths and to perform augmentation on these paths. (We will give more explanation after the proof of this theorem.) Let M be the matching obtained through $\sqrt{|V|}$ iterations. Let M^* be the maximum matching. Then $M \oplus M^*$ contains $|M^*| \setminus |M|$ augmenting path, each of length at least $1 + 2\sqrt{|V|}$ by Lemma 5.3.3. Therefore, each takes at least $2 + 2\sqrt{|V|}$ nodes. This implies that the number of augmenting paths in $M \oplus M^*$ is upper bounded by

$$|V|/(2 + 2\sqrt{|V|}) < \sqrt{|V|}/2.$$

Thus, M^* can be obtained from M through at most $\sqrt{|V|}/2$ iterations. Therefore, M^* can be obtained within at most $\frac{3}{2} \cdot \sqrt{|V|}$ iterations. This completes the proof. □

There are two steps in finding a maximal set of disjoint augmenting paths for a matching M in bipartite graph $G = (U, V, E)$.

In the first step, employ the breadth-first-search to put nodes into different levels as follows. Initially, select all free nodes in U and put them in the first level. Next, put in the second level all nodes each with a unmatched edge connecting to a node in the first level. Then, put in the third level all nodes each with a matched edge connecting to a node in the second level. Continue in this alternating ways, until a free node in V is discovered, say in the kth level (Fig. 5.11). Let F be all free nodes in the kth level and H the obtained subgraph. If the breadth-first-search comes to the end and still cannot find a free node in V, then this means that there is no augmenting path and a maximum matching has already obtained by Hopcroft-Karp algorithm.

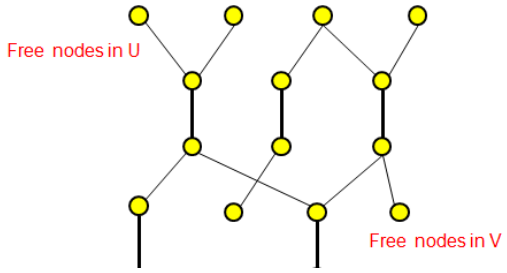


Figure 5.11: The breadth-first search.

In the second step, employ the depth-first-search to find path from each node in F to a node in the first level. Such paths will be search one by one in H and once a path is obtained, all nodes on this depth-first-search path will be deleted from H, until no more such path can be found.

Since both steps can work in $O(|E|)$ time, the total time for finishing this task is $O(|E|)$.

5.4 Dinitz Algorithm for Maximum Flow

The idea in Hopcroft-Karp algorithm can be extended from matching to flow. This extension gives a variation of Edmonds-Karp algorithm, called Dinitz algorithm.

Consider a flow network $G = (V, E)$. The algorithm starts with a zero flow $f(u, v) = 0$ for every arc (u, v). In each substantial iteration, consider residual network G_f for flow f. Start from source node s to do the breadth-first-search until node t is reached. If t cannot be researched, then algorithm stops and the maximum flow is already obtained. If t is reached with distance ℓ from node s, then the breadth-first-search tree contains ℓ level and its nodes are divided into ℓ classes $V_0, V_1, ..., V_\ell$ where V_i is the set of all nodes each with distance i from s and $\ell \leq |V|$. Collect all arcs from V_i to V_{i+1} for $i = 0, 1, ..., \ell - 1$. Let $L(s)$ be the obtained a levelable subnetwork. Above computation can be done in $O(|E|)$ time.

Next, the algorithm finds augmenting paths to do augmentations in following way.

Step 1. Iteratively, for $v \neq t$ and $u \neq s$, remove, from $L(s)$, every arc (u, v) with no coming arc at u or no out-going arc at v. Denote by $\hat{L}(s)$ the obtained levelable network.

Step 2. If $\hat{L}(s)$ is empty, then this iteration is completed and go to next iteration. If $\hat{L}(s)$ is not empty, then it contains a path of length ℓ, from s to t. Find such a path P by using the depth-first-search. Do augmentation along the path P. Update $L(s)$ by using $\hat{L}(s)$ and deleting all critical arcs on P. Go to Step 1.

This algorithm has following property.

Lemma 5.4.1. *Let $\delta_f(s,t)$ denote the distance from s to t in residual graph G_f of flow f. Suppose flow f' is obtained from flow f through an iteration of Dinitz' algorithm. Then $\delta_{f'}(s,t) \geq \delta_f(s,t) + 2$.*

Proof. The proof is similar to the proof of Lemma 5.2.1. □

The correctness of Dinitz' algorithm is stated in following theorem.

Theorem 5.4.2. *Dinitz' algorithm produces a maximum flow in $O(|V|^2|E|)$ time.*

Proof. By Lemma 5.4.1, Dinitz' algorithm runs within $O(|V|)$ iterations. Let us estimate the running time in each iteration.

- The construction of $L(s)$ spends $O(|E|)$ time.
- It needs $O(|V|)$ time to find each augmenting path and to do augmentation. Since each augmentation will remove at least one critical arc, there are at most $O(|E|)$ augmentations. Thus, the total time for augmentations is $O(|V| \cdot |E|)$.
- Amortizing all time for removing arcs, it is at most $O(|E|)$.

Therefore, each iteration runs in $O(|V| \cdot |E|)$ time. Hence, Dinitz algorithm runs in $O(|V|^2|E|)$ time. At end of algorithm, G_f does not contain a path from s to t. Thus, f is a maximum flow. □

5.5 Minimum Cost Maximum Flow

The following problem is closely related to maximum flow.

Problem 5.5.1 (Minimum Cost Maximum Flow)**.** *Given a flow network $G = (V, E)$ with capacity $c(u, v)$ and cost $a(u, v)$, a source s and a sink t, find a maximum flow f with the minimum total cost $cost(f) = \sum_{(u,v)\in E} a(u, v) \cdot f(u, v)$.*

There is a solution (Algorithm 18) similar to Edmonds-Karp algorithm.

Algorithm 18 Algorithm for Minimum Cost Maximum Flow

Input: A flow network $G = (V, E)$ with nonnegative capacity $c(u, v)$ and nonnegative cost $a(u, v)$ for each arc (u, v), a source s and a sink t.
Output: A flow f.

1: $G_f \leftarrow G$;
2: $f \leftarrow 0$; (i.e., $\forall u, v \in V, f(u, v) = 0$)
3: **while** there exists a path from s to t in G_f **do**
4: find a minimum cost path P from s to t;
5: set $\delta \leftarrow \min\{c(u, v) \mid (u, v) \in P\}$ and
6: send a flow f' with value δ from s to t along path P;
7: $G_f \leftarrow (G_f)_{f'}$;
8: $f \leftarrow f + f'$;
9: **end while**
10: **return** f.

In this algorithm, arc cost in G_f needs some explanation. For arc (v, u) added from a flow on arc (u, v), its cost $a(v, u) = -a(u, v)$. In particular, when G contains two arc (u, v) and (v, u), G_f may contains multi-arcs (v, u) and $(v, u)^f$. (v, u) has given capacity $c(v, u)$ and cost $a(u, v)$. However, $(v, u)^f$ has capacity $f(u, v)$ and cost $-a(u, v)$.

Clearly, Algorithm 18 produces a maximum flow. However, for correctness, we have to show this maximum flow has the minimum cost. To do so, let us first establish an optimality condition.

A directed cycle is called a *negative cost cycle* in residual graph G_f if the total arc cost of the cycle is negative.

Lemma 5.5.2. *A maximum flow f has the minimum cost if and only if its residual graph G_f does not contain a negative cost cycle.*

Proof. If G_f contains a negative cost cycle, then the cost can be reduced by adding a flow along this cycle. Next, assume that G_f for a maximum flow f does not contain a negative cost cycle. We show that f has the minimum cost. For contradiction, assume that f does not reach the minimum cost, so that its cost is larger than the cost of a maximum flow f'. Note that

every flow can be decomposed into disjoint union of several path-flows. This fact implies that f contains a path-flow P has cost larger that the cost of a path-flow P' in f'. Let $\hat{P}$ be obtained from P by reversing its direction. Then $\hat{P} \cup P'$ forms a negative cost cycle, which may be decomposed into several simple cycles and one of them must also have negative cost. This simple cycle must be contained in G_f, a contradiction. □

Lemma 5.5.3. *Suppose that the residual graph G_f of maximum flow f does not contain a negative cost cycle. Let the maximum flow f' be obtained from f through one augmentation in Algorithm 18. Then $G_{f'}$ does not contain a negative cost cycle.*

Proof. For contradiction, suppose $G_{f'}$ contains a negative cost cycle Q. Since G_f does not contain a negative cost cycle, Q must contain some arcs which are reverse of some arcs on the augmenting path P in G_f. In P, replacing those arcs by remaining arcs in Q, we will obtain a path with path in G_f, with cost smaller than the cost of P, contradicting the rule for choosing augmenting path in Algorithm 18. □

Now, we show the correctness of Algorithm 18.

Theorem 5.5.4. *Suppose that the input flow network $G = (V, E)$ has nonnegative capacity and nonnegative cost on each arc. Then Algorithm 18 ends at a minimum cost maximum flow.*

Proof. It follow immediately from Lemmas 5.5.3 and 5.5.2. □

There is an interesting observation that Algorithm 18 is not in class of incremental methods. In fact, its feasible domain is the set of maximum flows. The incremental method should move from moving from a maximum flow to another maximum flow with cost reducing. Such an example is the cycle canceling algorithm as shown in Algorithm 19.

Both Algorithms 18 and 19 run in pseudo-polynomial time when all capacities are integers. In such a case, the flow in the algorithms are always integers and hence in each iteration, the flow value is increased by at least one. Since the maximum flow value is bounded $O(|V|C)$ where $Cisthemaximumarccapacity$. Therefore, the number of iterations is at most $O(|V|C)$. In case that arc capacities are not integers, we even do not know whether these two algorithms will halt. In the literature, there are strong polynomial-time algorithms for the minimum cost maximum flow problem. The reader may refer the historical notes for references.

The minimum cost maximum flow has many applications. Here, let us mention an example.

Algorithm 19 Cycle Canceling for Minimum Cost Maximum Flow

Input: A flow network $G = (V, E)$ with nonnegative capacity $c(u, v)$ and nonnegative cost $a(u, v)$ for each arc (u, v), a source s and a sink t.
Output: A maximum flow f.

1: Compute a maximum flow f with Admonds-Karp algorithm;
2: **while** G_f contains a negative cost cycle Q **do**
3: $f \leftarrow f \cup Q$;
4: **end while**
5: **return** f.

Problem 5.5.5 (Minimum Cost Perfect Matching in Bipartite Graph)**.** *Given a complete bipartite graph $G = (U, V, E)$ ($|U| = |V|$) with cost $a(u, v)$ for each edge $(u, v) \in E$, find a perfect matching with the minimum total edge cost.*

This problem can be transformed into an instance of minimum cost maximum flow in following way:

- Add a source node s and a sink node t and add arcs from s to every node in U and arcs from every node in V to t. Every of those added arcs has capacity one and cost zero.
- For every edge in E, assign a direction from U to V.

This problem can also be reduced to the maximum weight bipartite matching problem as follows: Let $a_{max} = max_{e \in E} a(e)$. Define a new edge cost $a'(e) = a_{max} - a(e)$ for $e \in E$. Since the perfect matching always contains $|U|$ edges, a perfect matching reaches the minimum edge cost with $a(e)$ if and only if it reaches the maximum edge cost with $a'(e)$. Thus, the minimum cost perfect matching problem is transformed to the maximum weight bipartite matching problem.

5.6 Chinese Postman and Graph Matching

The technique at end of last section can also be employed to solve following problem.

Problem 5.6.1 (Chinese Postman Problem)**.** *Given a graph $G = (V, E)$ with edge cost a, find a postman tour to minimize total edge cost where a postman tour is a cycle passing through every edge at least once.*

A cycle is called as a *Euler tour* if it passes through every edge exactly once. In graph theory, it has been proved that a connected graph has an Euler tour if and only if every node has even degree. A node is called as an *odd node* if its degree is odd. Note that the number of odd degree for any graph is even. Therefore, we may solve the Chinese postman problem in following way.

- Construct a complete graph H on all old nodes and assign the distance between any two nodes u and v with the shortest distance between u and v in G.
- Find the minimum cost perfect matching M in H.
- Add M to input graph G and the Euler tour in $G \cup M$ is the optimal solution.

In this method, the minimum cost perfect matching in graph H can be transformed to the maximum weight matching problem in H by changing the edge cost by employing the same technique at last section.

Note that currently, we do not know how to reduce the maximum weight matching problem into a network flow problem. Therefore, we may like to employ alternating path and cycle again.

This time, an alternating path is an augmenting path if it is maximal and the total weight of its matched edges is less than the total weight of its unmatched edges; an alternating cycle is an augmenting cycle if it is maximal and the total weight of its matched edges is less than the total weight of its unmatched edges.

Actually, maximum matching in general graph is in similar situation. Let us explore more alternating path method for this problem.

Problem 5.6.2 (Maximum Graph Matching)**.** *Given a graph $G = (V, E)$, find a matching with maximum cardinality.*

Recall that an *augmenting path* is defined to be a path satisfying following two conditions:

(a) It is an *alternating path*, that is, edges on the path is alternatively unmatched and matched.

(b) The path is between two free nodes.

Now, proof of Theorem 5.3.2 can be applied to the graph matching without any change. Therefore, we obtained following algorithm for the maximum graph matching problem.

Algorithm 20 Algorithm for Maximum Graph Matching

Input: A graph $G = (V, E)$.
Output: A maximum matching M.

1: $M \leftarrow$any edge;
2: **while** there exists an augmenting path P **do**
3: $\quad M \leftarrow M \oplus P$;
4: **end while**
5: **return** M.

How to find an augmenting path for matching in a general graph $G = (V, E)$? Let us introduce the Blossom algorithm. A *blossom* is an almost alternating odd cycle as shown in Fig.5.12. The Blossom algorithm is similar

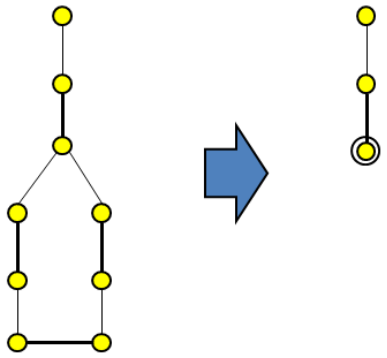


Figure 5.12: A Blossom shrinks into a node.

to the first step of augmenting-path finding in Hopcroft-Karp algorithm, i.e., employ the breadth-first-search by using unmatched edge and matched edge alternatively. However, start from one free node x at a time. In the search, algorithm may stop at another free node y, i.e., an augmenting path is found, or determine that no augmenting path exists with x as an end. In the search process, a Blossom may be found. In such a case, shrink the Blossom into a node. Why a Blossom can be shrink into a point? It is because the alternating path can be extended passing through a Blossom out-reach to its any connection (Fig. 5.13). Clearly, this algorithm runs in $O(|V| \cdot |E|)$ time. Thus, we have following.

Theorem 5.6.3. *The maximum cardinality matching in graph $G = (V, E)$ can be found in $O(|V|^2 \cdot |E|)$ time.*

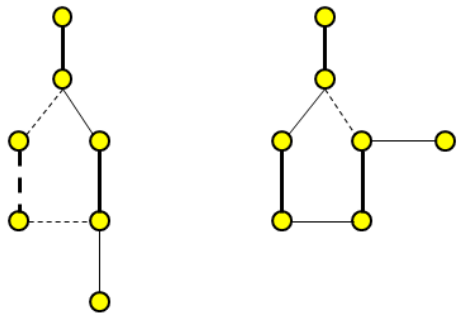


Figure 5.13: A alternating path passes a Blossom.

Exercises

1. A conference organizer wants to set up a review plan. There are m submitted papers and n reviewers. Each reviewer has made p papers as "prefer to review". Each paper should have at least q review reports. Find a method to determine whether such a review plan exists or not.

2. A conference organizer wants to set up a review plan. There are m submitted papers and n reviewers. Each reviewer is allowed to made at least p_1 papers as "prefer to review" and at least p_2 papers as "likely to review". Each paper should have at least q_1 review reports and at most q_2 review reports. Please give a procedure to make the review plan.

3. Suppose there exist two distinct maximum flows f_1 and f_2. Show that there exist infinitely many maximum flows.

4. Consider a directed graph G with a source s, a sink t and nonnegative arc capacities. Find a polynomial-time algorithm to determine whether G contains a unique s-t cut.

5. Consider a flow network $G = (V, E)$ with a source s, a sink t and nonnegative capacities. Suppose a maximum flow f is given. If an arc is broken, find a fast algorithm to compute a new maximum flow based on f. A favorite algorithm will run in $O(|E| \log |V|)$ time.

6. Consider a flow network $G = (V, E)$ with a source s, a sink t and

nonnegative integer capacities. Suppose a maximum flow f is given. If the capacity of an arc is increased by one, find a fast algorithm to update the maximum flow. A favorite algorithm runs in $O(|E| + |V|)$ time.

7. Consider a directed graph $G = (V, E)$ with a source s and a sink t. Instead of arc capacity, assume that there is the nonnegative integer node capacity $c(v)$ on each node $v \in V$, that is, the total flow passing node v cannot exceed $c(v)$. Show that the maximum flow can be computed in polynomial-time.

8. Show that the maximum flow of a flow network $G = (V, E)$ can be decomposed into at most $|E|$ path-flows.

9. Suppose a flow network $G = (V, E)$ is symmetric, i.e., $(u, v) \in E$ if and only if $(v, u) \in E$ and $c(u, v) = c(v, u)$. Show that Edmonds-Karp algorithm terminates within at most $|V| \cdot |E|/4$ iterations.

10. Consider a directed graph G. A node-disjoint set of cycles is called a *cycle-cover* if it covers all nodes. Find a polynomial-time algorithm to determine whether a given graph G has a cycle-cover or not.

11. Consider a graph G. Given two nodes s and t, and a positive integer k, find a polynomial time algorithm to determine whether there exist or not k edge-disjoint paths between s and t.

12. Consider a graph G. Given two nodes s and t, and a positive integer k, find a polynomial time algorithm to determine whether there exist or not k node-disjoint paths between s and t.

13. Consider a graph G. Given three nodes x, y, z, find a polynomial algorithm to determine whether there exists a simple path from x to z passing through y.

14. Prove or disprove (by counterexample) following statement.

    (a) If a flow network has unique maximum flow, then it has unique minimum s-t cut.

    (b) If a flow network has unique minimum s-t cut, then it has unique maximum flow.

    (c) A maximum flow must associate with a minimum s-t cut such that the flow passes through the minimum s-t cut.

(d) A minimum s-t cut must associate with a maximum flow such that he flow passes through the minimum s-t cut.

15. Let M be a maximal matching of a graph G. Show that for any matching M' of G, $|M'| \leq 2 \cdot |M|$.

16. We say that a bipartite graph $G = (L, R, E)$ is d-regular if every vertex $v \in L \cup R$ has degree exactly d. Prove that every d-regular bipartite graph has a matching of size $|L|$.

17. There are n students who studied at a late-night study for final exam. The time has come to order pizzas. Each student has his own list of required toppings (e.g. mushroom, pepperoni, onions, garlic, sausage, etc). Everyone wants to eat at least half a pizza, and the topping of that pizza must be in his reqired list. A pizza may have only one topping. How to compute the minimum number of pizzas to order to make everyone happy?

18. Consider bipartite graph $G = (U, V, E)$. Let $\mathcal{H}$ be the collection of all subgraphs H that for every $u \in U$, H has at most one edge incident to u. Let $E(H)$ denote the edge set of H and $\mathcal{I} = \{E(H) \mid H \in \mathcal{H}\}$. Show that (a) $(E, \mathcal{I})$ is a matroid and (b) all matchings in G form an intersection of two matroids.

19. Consider a graph $G = (V, E)$ with nonnegative integer function $c : V \to N$. Find an augmenting path method to compute a subgraph $H = (V, F)$ $(F \subseteq E)$ with maximum number of edges such that for every $v \in V$, $deg(v) \leq c(v)$.

20. A conference with a program committee of 30 members received 100 papers. The PC chair wants to make an assignment. He first asked all PC members each to choose 15 prefered papers. Based on what PC members choose, the PC chair wants to find an assignment such that each PC member reviews 10 papers among 15 chosen ones and each paper gets 3 PC members to review. How do we figure out whether such an assignment exists? Please design a maximum flow formulation to answer this question.

21. Let $U = \{u_1, u_2, ..., u_n\}$ and $V = \{v_1, v_2, ..., v_n\}$. A bipartite graph $G = (U, V, E)$ is *convex* if $(u_i, v_k), (u_j, v_k) \in E$ with $i < j$ imply $(u_h, v_k) \in E$ for all $h = i, i + 1, ..., j$. Find a greedy algorithm to compute the maximum matching in a convex bipartite graph.

22. Consider a bipartite graph $G = (U, V, E)$ and two node subsets $A \subseteq U$ and $B \subseteq V$. Show that if there exist a matching M_A covering A and a matching M_B covering B, then there exists a matching $M_{A \cup B}$ covering $A \cup B$.

23. An *edge-cover* C of a graph $G = (V, E)$ is a subset of edges such that every vertex is incident to an edge in C. Design a polynomial-time algorithm to find the minimum edge-cover, i.e., an edge-cover with minimum cardinality.

24. (König Theorem) Show that the minimum size of vertex cover is equal to the maximum size of matching in bipartite graph.

25. Show that the vertex-cover problem in bipartite graphs can be solved in polynomial-time.

26. A matrix with all entries being 0 or 1 is called a 0-1 matrix. Consider a positive integer d and a 0-1 matrix M that each row contains exactly two 1s. Show a polynomail-time algorithm to find a minimum number of rows to form a submatrix such that for every $d+1$ columns C_0, C_1, ..., C_d, there exists a row at which C_0 has entry 1, but all C_1, ..., C_d have entry 0 (such a matrix is called a *d-disjunct* matrix).

27. Design a cycle canceling algorithm for the Chinese postman problem.

28. Design a cycle canceling algorithm for the minimum spanning tree problem.

29. Consider a graph $G = (V, E)$ with nonnegative edge distance $d(e)$ for $e \in E$. There are m source nodes $s_1, s_1, ..., s_m$ and n sink nodes $t_1, t_2, ..., t_n$. Suppose these source are required to provide those sink nodes with certain type of product. Suppose that s_i is required to provide a_i products and t_j requires b_j products. Assume $\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j$. The target is to find a transportation plan to minimize the total cost where on each edge, the cost is the multiplication of the distance and the amount of products passing through the edge. Show that a transportation plan is minimum if and only if there is no cycle such that the total distance of unloaded edges is less than the total distance of loaded edges.

30. Consider m sources $s_1, s_1, ..., s_m$ and n sinks $t_1, t_2, ..., t_n$. These source are required to provide those sink nodes with certain type of product. s_i is required to provide a_i products and t_j requires b_j product-

s. Assume $\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j$. Given a distance table (d_{ij} between sources s_i and sinks t_j, the target is to find a transportation plan to minimize the total cost where on each edge, the cost is the multiplication of the distance and the amount of products passing through the edge. Show that a transportation plan is minimum if and only if there is no circuit $[(i_1, j_1), (i_2, j_1), (i_2, j_2), ..., (i_1, j_k)]$ such that $(i_1, j_1), (i_2, j_2), ..., (i_k, j_k)$ are loaded, $(i_2, j_1), (i_3, j_2), ..., (i_1, j_k)$ are unloaded, and $\sum_{h=1}^{k} d(i_h, j_h) > \sum_{h=1}^{k} d(i_h, j_{h-1})$ $(j_0 = j_k)$. Here, (i, j) is said to be loaded if there is at least one product transported from s_i to t_j.

Historical Notes

Maximum flow problem was proposed by T. E. Harris and F. S. Ross in 1955 [39, 40] and was first solved by L.R. Ford and D.R. Fulkerson in 1956 [41]. However, Ford-Fulkerson algorithm is a pseudo polynomial-time algorithm when all arc capacities are integers. If arc capacities may not be integers, the termination of the algorithm may meet a trouble. The first strong polynomial-time was designed by Edmonds and Karp [42]. Later, various designs appeared in the literature, including Dinitz' algorithm [43, 44], Goldberg-Tarjan push-relabel algorithm [45], Goldberg-Rao algorithm [46], Sherman algorithm [47], algorithm of Kelner, Lee, Orecchia and Sidford [48]. Currently, the best running time is $O(|V||E|)$. This record is kept by Orlin algorithm [49].

Minimum cost maximum flow is studied following up with maximum flow problem. Similarly, earlier algorithms run in pseudo polynomial-time such as out-of-kilter algorithm [56], cheapest path augmentation [60], cycle canceling [57], and successive shortest path [58]. Polynomial-time algorithms were found later such as minimum mean cycle canceling [55] and speed-up successive shortest path. Currently, the fastest strong polynomial-time algorithm has running time is $O(|E|^2 \log^2 |V|)$. This record is also kept by an algorithm of Orlin [59].

Matching is a classical subject in graph theory. Both maximum (cardinality) matching and minimum cost perfect matching problems in bipartite graphs can be easily transformed to maximum flow problems. However, they can also solved with alternating path methods. So far, Hopcroft-Karp algorithm [53] is the fastest algorithm for the maximum bipartite matching. In general graph, they have to be solved with alternating path method since currently, no reduction have beeb found to transform matching problem to

flow problem. Those algorithms were designed by Edmonds [52]. An extension of Hopcroft-Karp algorithm was made by Micali and Vazirani [54], which runs in $O(\sqrt{|E|}|V|)$ time.

For maximum weight matching, nobody has found any method to transform it to a flow problem. Therefore, we have to employ the alternating path and cycle method [52], too.

Chinese postman problem was proposed by Kwan [50] and the first polynomial-time solution was given by Edmonds and Johnson[51] with minimum cost perfect matching in complete graph with even number of nodes.

Chapter 8

NP-hard Problems and Approximation Algorithms

> "The biggest difference between time and space is that you can't reuse time."
> - Merrick Furst

8.1 What is the class NP?

The class P consists of all polynomial-time solvable decision problems. What is the class NP? There are two popular misunderstandings:

(1) NP is the class of problems which are not polynomial-time solvable.

(2) A decision problem belongs to the class NP if its answer can be checked in polynomial-time.

The misunderstanding (1) comes from misexplanation of NP as the brief name for "Not Polynomial-time solvable". Actually, it is polynomial-time solvable, but in a wide sense of computation, nondeterministic computation, that is, **NP is the class of all nondeterministic polynomial-time solvable decision problems.** Thus, NP is the brief name of "Nondeterministic Polynomial-time".

What is the nondeterministic computation? Let us explain it starting from computation model, Turing machine (TM). A TM consists of three parts, a tape, a head, and a finite control (Fig. 8.1).

The tape has the left end and infinite long in the right direction, which is divided into infinitely many cells. Each cell can hold a symbol. All symbols possibly on the tape form an alphabet Γ, called the *alphabet of tape symbols*. In Γ, there is a special symbol B, called the *blank symbol*, which

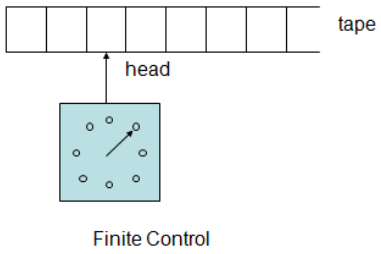


Figure 8.1: One-tape Turing machine.

means the cell is actually empty. Initially, an input string is written on the tape. All symbols possibly in the input string form another alphabet Σ, called the *alphabet of input symbols*. Assume that both Γ and Σ are finite and $B \in \Gamma \setminus \Sigma$.

The head can read, erase, and write symbols on the tape. Moreover, it can move to left and right. In each move, the head can shift a distance of one cell. Please note that in classical one-tape TM, the head is not allowed to stay in the place without move before the TM halts.

The finite control contains a finite number of states, forming a set Q. The TM's computation depends on function $\delta : Q \times \Gamma \to Q \times \Gamma \times D$ where $D = \{R, L\}$ is the set of possible moving directions and R means moving to right while L means moving to left. This function δ is called the *transaction function*. For example, $\delta(q, a) = (p, b, L)$ means that when TM in state q reads symbol a, it will change state to p, change symbol a to b, and them move to the left (the upper case in 8.2); $\delta(q, a) = (p, b, R)$ means that when TM in state q reads symbol a, it will change state to p, change symbol a to b, and them move to the right (the lower case in 8.2); Initially, on an input x, the TM is in a special state s, called the *initial state*, and its head is located at the leftmost cell, which contains the first symbol of x if x is not empty. The TM stops moving if and only if it enters another special state h, called the *final state*. An input x is said to be *accepted* if on x, the TM will finally stop. All accepted inputs form a language, which is called the language accepted by the TM. The language accepted by a TM M is denoted by $L(M)$.

From above description, we see that each TM can be described by the following parameters, an alphabet Σ of input symbols, an alphabet Γ of tape symbols, a finite set Q of states in finite control, a transition function δ, and

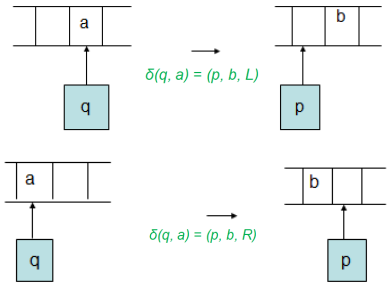


Figure 8.2: One move to the right.

an initial state s.

The computation time of an TM M on an input x is the number of moves from initial state to final state, denoted by $Time_M(x)$. A TM M is said to be *polynomial-time bounded* if there exists a polynomial p such that for every input $x \in L(M)$, $Time_m(x) \leq p(|x|)$. So far, what we described TM is the deterministic TM (DTM), that is, for each move, there exists at most one transition determined by the transition function. All languages accepted by polynomial-time bounded DTM form a class, denoted by P.

There are many variations of the TM, in which the TM has more freedom. For example, the head is allowed to stay at the same cell during a move, the tape may have no left-end, and multiple tapes exist (Fig. 8.3). However, in term of polynomial-time computability, all of them have been proved to have the same power. Based on such experiences, one made following conclusion.

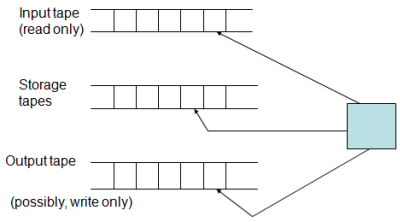


Figure 8.3: A multi-tape TM.

Extended Church-Turing Thesis. *A function computable in polynomial-time in any reasonable computational model using a reasonable time complexity measure is computable by a deterministic TM in polynomial-time.*

Extended Church-Turing Thesis is a natural law of computation. It is similar to physics laws, which cannot have a mathematical proof, but is obeyed by the natural world. By Extended Church-Turing thesis, the class P is independent from computational models. In the statement, "reasonable" is an important word. Are there unreasonable computational models? The answer is Yes. For example, the nondeterministic Turing machine (NTM) is an important one among them. In an NTM, for each move, there may

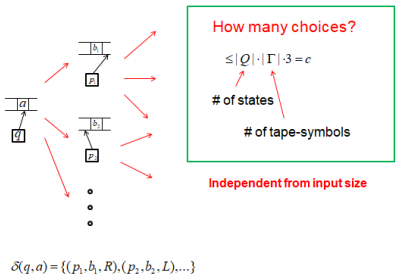


Figure 8.4: There are many possible transitions for each move in an NTM.

exist many possible transitions (Fig. 8.4) and the NTM can use any one of them. Therefore, transition function δ in an NTM is a mapping from $Q \times \Gamma$ to $2^{Q \times \Gamma \times \{R,L\}}$, that is, $\delta(q, a)$ is the set of all possible transitions. When the NTM in state q reads symbol a, it can choose any one transition from $\delta(q, a)$ to implement.

It is worth to mentioning that for each nondeterministic move of one-tape NTM, the number of possible transitions is upper-bounded by $|Q| \times |\Gamma| \times 3$ where Q is the set of states, Γ is the alphabet of tape symbols, and 3 is a upper-bound for the number of moving choices. $|Q| \times |\Gamma| \times 3$ is a constant independent from input size $|x|$.

The computation process of the DTM can be represented by a path while the computation process of the NTM has to be represented by a tree. When an input x is accepted by an NTM? The definition is that as long as there is a path in the computation tree, leading to final state, then x is accepted. Suppose that at each move, we make a guess for choice of possible transitions. This definition means that if there exists a correct guess which

leads to final state, we will accept the input. Let us look at an example. Consider following problem.

Problem 8.1.1 (Hamiltonian Cycle)**.** *Given a graph $G = (V, E)$, does G contain a Hamiltonian cycle? Here, a Hamiltonian cycle is a cycle passing through each vertex exactly once.*

The following is a nondeterministic algorithm for the Hamiltonian cycle problem.

input a graph $G = (V, E)$.
step 1 guess a permutation of all vertices.
step 2 check if guessed permutation gives a Hamitonian cycle.
if yes, **then** accept input.

In step 1, the guess corresponds to nondeterministic moves in the NTM. Note that in step 2, if the outcome of checking is no, then we cannot give any conclusion and hence nondeterministic computation gets stuck. However, a nondeterministic algorithm is considered to solve a decision problem correctly if there exists a guessed result leading to correct yes-answer. For example, in above algorithm, if input graph contains a Hamiltonian cycle, then there exists a guessed permutation which gives a Hamiltonian cycle and hence gives yes-answer. Therefore, it is a nondeterministic algorithm which solves the HAMILTONIAN CYCLE problem.

Now, let us recall the second popular misunderstanding of NP mentioned at beginning of this section:

(2) A decision problem belongs to the class NP if its answer can checked in polynomial-time.

Why (2) is wrong? This is because not only checking step is required to be polynomial-time computable, but also guessing step is required to be polynomial-time computable. How do we estimate guessing time? Let us explain this starting from what is a legal guess. Note that in an NTM, each nondeterministic move can select a choice of transition from a pool with size upper-bound independent from input size. Therefore, **A legal guess is a guess from a pool with size independent from input size.** For example, in above algorithm, guessing in step 1 is not legal because the number of permutation of n vertices is $n!$ which depends on input size.

What is the running time of step 1? It is the number of legal guesses spent in implementation of the guess in step 1. To implement the guess in step 1, we may encode each vertex into a binary code of length $\lceil \log_2 n \rceil$. Then each permutation of n vertices is encoded into a binary code of length

$O(n \log n)$. Now, guessing a permutation can be implemented by $O(n \log n)$ legal guesses each of which chooses either 0 or 1. Therefore, the running time of step 1 is $O(n \log n)$.

In many cases, the guessing step is easily implemented by a polynomial number of legal guesses. However, there are some exceptions; one of them is the following.

Problem 8.1.2. *Given an $m \times n$ integer matrix A and an n-dimensional integer vector b, determine whether there exists a m-dimensional integer vector x such that $Ax \geq b$.*

In order to prove that Problem 8.1.2 is in NP, we may guess an n-dimensional integer vector x and check whether x satisfies $Ax \geq b$. However, we need to make sure that guessing can be done in nondeterministic polynomial-time. That is, we need to show that if the problem has a solution, then there is a solution of polynomial size. Otherwise, our guess cannot find it. This is not an easy job. We include the proof into the following three lemmas.

Let α denote the maximum absolute value of elements in A and b. Denote $q = \max(m, n)$.

Lemma 8.1.3. *If B is a square submatrix of A, then $|\det B| \leq (\alpha q)^q$.*

Proof. Let k be the order of B. Then $|\det B| \leq k!\alpha^k \leq k^k\alpha^k \leq q^q\alpha^q = (q\alpha)^q$. □

Lemma 8.1.4. *If $rank(A) = r < n$, then there exists a nonzero vector z such that $Az = 0$ and every component of z is at most $(\alpha q)^q$.*

Proof. Without loss of generality, assume that the left-upper $r \times r$ submatrix B is nonsingular. Set $x_{r+1} = \cdots = x_{n-1} = 0$ and $x_n = -1$. Apply Cramer's rule to system of equations

$$B(x_1, \cdots, x_r)^T = (a_{1n}, \cdots, a_{rn})^T$$

where a_{ij} is the element of A on the ith row and the jth column. Then we can obtain $x_i = \det B_i / \det B$ where B_i is a submatrix of A. By Lemma 3.1, $|\det B_i| \leq (\alpha q)^q$. Now, set $z_1 = \det B_1, \cdots, z_r = \det B_r, z_{r+1} = \cdots = z_{n-1} = 0$, and $z_n = \det B$. Then $Az = 0$. □

Lemma 8.1.5. *If $Ax \geq b$ has an integer solution, then it must have an integer solution whose components of absolute value not exceed $2(\alpha q)^{2q+1}$.*

Proof. Let a_i denote the ith row of A and b_i the ith component of b. Suppose that $Ax \geq b$ has an integer solution. Then we choose a solution x such that the following set gets the maximum number of elements.

$$\mathcal{A}_x = \{a_i \mid b_i \leq a_i x \leq b_i + (\alpha q)^{q+1}\} \cup \{e_i \mid |x_i| \leq (\alpha q)^q\},$$

where $e_i = (\underbrace{0, \cdots, 0, 1}_{i}, 0, \cdots, 0)$. We first prove that the rank of $\mathcal{A}_x$ is n. For otherwise, suppose that the rank of $\mathcal{A}_x$ is less than n. Then we can find nonzero integer vector z such that for any $d \in \mathcal{A}_x$, $dz = 0$ and each component of z does not exceed $(\alpha q)^q$. Note that $e_k \in \mathcal{A}_x$ implies that kth component z_k of z is zero since $0 = e_k z = z_k$. If $z_k \neq 0$, then $e_k \notin \mathcal{A}_x$, so, $|x_k| > (\alpha q)^q$. Set $y = x + z$ or $x - z$ such that $|y_k| < |x_k|$. Then for every $e_i \in \mathcal{A}_x$, $y_i = x_i$, so, $e_i \in \mathcal{A}_y$, and for $a_i \in \mathcal{A}_x$, $a_i y = a_i x$, so, $a_i \in \mathcal{A}_y$. Thus, $\mathcal{A}_y$ contains $\mathcal{A}_x$. Moreover, for $a_i \notin \mathcal{A}_x$, $a_i y \geq a_i x - |a_i z| \geq b_i + (\alpha q)^{q+1} - n\alpha(\alpha q)^q \geq b_i$. Thus, y is an integer solution of $Ax \geq b$. By the maximality of $\mathcal{A}_x$, $\mathcal{A}_y = \mathcal{A}_x$. This means that we can decrease the value of the kth component again. However, it cannot be decreased forever. Finally, a contradiction would appear. Thus, $\mathcal{A}_x$ must have rank n.

Now, choose n linearly independent vectors $d_1, \cdots, d_n$ from $\mathcal{A}_x$. Denote $c_i = d_i x$. Then $|c_i| \leq \alpha + (\alpha q)^{q+1}$. Applying Cramer's rule to the system of equations $d_i x = c_i$, $i = 1, 2, \cdots, n$, we obtain a representation of x through c_i's: $x_i = \det D_i / \det D$ where D is a square submatrix of $(A^T, I)^T$ and D_i is a square matrix obtained from D by replacing the ith column by vector $(c_1, \cdots, c_n)^T$. Note that the determinant of any submatrix of $(A^T, I)^T$ equals to the determinant of a submatrix of A. By Laplac expansion, it is easy to see that $|x_i| \leq |\det D_i| \leq (\alpha q)^q(|c_1| + \cdots + |c_n|) \leq (\alpha q)^q n(\alpha + (\alpha q)^{q+1}) \leq 2(\alpha q)^{2q+1}$. □

By Lemma 8.1.5, it is enough to guess a solution x whose total size is at most $n \log_2(2(\alpha q)^{2q+1})) = O(q^2(\log_2 q + \log_2 \alpha)$. Note that the input A and b have total length at least $\beta = \sum_{i=1}^{m} \sum_{j=1}^{n} \log_2 |a_{ij}| + \sum_{j=1}^{n} \log_2 |b_j| \geq mn + \log_2 \alpha \geq q+$ IP is in NP. □

Theorem 8.1.6. *Problem 8.1.2 is in NP.*

Proof. It follows immediately from Lemma 8.1.5. □

The definition of the class NP involves three concepts, nondeterministic computation, polynomial-time, and decision problems. The first two concepts have been explained as above. Next, we explain what is the decision problem.

A problem is called a *decision problem* if its answer is "Yes" or "No". Each decision problem corresponds to the set of all inputs which receive yes-answer, which is a language when each input is encoded into a string. For example, the Hamiltonian cycle problem and Problem 8.1.2 are decision problems. In case of no confusion, we sometimes use the same notation to denote a decision problem and its corresponding language. For example, we may say that a decision problem A has its characteristic function

$$\chi_A(x) = \begin{cases} 1 & \text{if } x \in A, \\ 0 & \text{otherwise.} \end{cases}$$

Actually, we mean that the corresponding language of decision problem A has characteristic function χ_A.

Usually, combinatorial optimization problems are not decision problems. However, every combinatorial optimization problem can be transformed into a decision version. For example, consider following.

Problem 8.1.7 (Traveling Salesman)**.** *Given n cities and a distance table between n cities, find the shortest Hamiltonian tour where a Hamitonian tour is a Hamitonian cycle in the complete graph on the n cities.*

Its decision version is as follows.

Problem 8.1.8 (Decision Version of Traveling Salesman)**.** *Given n cities, a distance table between n cities, and an integer $K > 0$, is there a Hamiltonian tour with total distance at most K?*

Clearly, if the traveling salesman problem can be solved in polynomial-time, so is its decision version. Conversely, if its decision version can be solved in polynomial-time, then we may solve the traveling salesman problem in the following way within polynomial-time.

Let us assume that all distances between cities are integers. [1] Let d_{min} and d_{max} be the smallest distance and the maximum distance between two cities. Let $a = nd_{min}$ and $b = nd_{max}$. Set $K = \lceil (a+b)/2 \rceil$. Determine whether there is a tour with total distance at most K by solving the decision version of the traveling salesman problem. If answer is Yes, then set $b \leftarrow K$; else set $a \leftarrow K$. Repeat this process until $|b - a| \leq 1$. Then, compute the exact optimal objective function value of the traveling salesman problem by solving its decision version twice with $K = a$ and $K = b$, respectively. In

[1] If they are rational numbers, then we can transform them into integers. If some of them are irrational numbers, then we have to touch the complexity theory of real number computation, which is out of scope of this book.

this way, suppose the decision version of the traveling salesman problem can be solved in polynomial-time $p(n)$. Then the traveling salesman problem can be solved in polynomial-time $O(\log(nd_{max})p(n))$.

Now, we may find that actually, Problem 8.1.2 is closely related to the decision version of following integer program.

Problem 8.1.9 (0-1 Integer Program)**.**

$$\begin{aligned} \max \quad & cx \\ \textit{subject to} \quad & Ax \geq b \\ & x \in \{0,1\}^n, \end{aligned}$$

where A is an $m \times n$ integer matrix, c is an n-dimensional integer row vector, and b is an m-dimensional integer column vector.

8.2 What is NP-completeness?

In 1965, J. Admonds conjectured following.

Conjecture 8.2.1. *The traveling salesman problem does not have a polynomial-time solution.*

In study of this conjecture, S. Cook introduced the class NP and showed the first NP-complete problem in 1971.

A problem is NP-hard if the existence of polynomial-time solution for it implies the existence of polynomial-time solution for every problem in NP. An NP-hard problem is NP-complete if it also belongs to the class NP.

To introduce Cook's result, let us recall some knowledge on Boolean algebra.

A Boolean function is a function whose variable values and function value all are in $\{true, false\}$. Here, we would like to denote true by 1 and false by 0. In the following table, there are two boolean functions of two variables, *conjunction* $\wedge$ and *disjunction* $\vee$, and a Boolean function of a variable, *negation* $\neg$.

x	y	$x \wedge y$	$x \vee y$	$\neg x$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

For simplicity, we also write $x \wedge y = xy$, $x \vee y = x + y$ and $\neg x = \bar{x}$. The conjunction and disjunction follow the commutative, associative, and distributive laws. An interesting and important law about negation is De Morgan's law, i.e.

$$\overline{xy} = \bar{x} + \bar{y} \text{ and } \overline{x + y} = \bar{x}\bar{y}.$$

The SAT problem is defined as follows:

Problem 8.2.2 (Satisfiability (SAT)). *Given a Boolean formula F, is there a satisfied assignment for F?*

Here, an assignment to variables of F is *satisfied* if the assignment makes F equal to 1. A Boolean formula F is *satisfiable* if there exists a satisfied assignment for F.

The SAT problem has many applications. For example, the following puzzle can be formulated into an instance od SAT.

Example 8.2.3. *After three men interviewed, the department Chair said:"We need Brown and if we need John then we need David, if and only if we need either Brown or John and don't need David." If this department actually need more than one new faculty, which ones were they?*

Solution. Let B, J, and D denote respectively Brown, John, and David. What Chair said can be written as a Boolean formula

$$\begin{aligned} & [[B(\bar{J} + D)][(B + J)\bar{D}] + \overline{B(\bar{J} + D)} \cdot \overline{(B + J)\bar{D}} \\ = \; & B\bar{D}\bar{J} + (\bar{B} + J\bar{D})(\bar{B}\bar{J} + D) \\ = \; & B\bar{D}\bar{J} + \bar{B}\bar{J} + \bar{B}D \end{aligned}$$

Since this department actually need more than one new faculty, there is only one way to satisfy this Boolean formula, that is, $B = 0$, $D = J = 1$. Thus, John and David will be hired. □

Now, we are ready to state Cook's result.

Theorem 8.2.4 (Cook Theorem). *The* SAT *problem is NP-complete.*

After the first NP-complete problem is discovered, there are a large number of problems have been found to be NP-hard or NP-complete. Indeed, there are many tools passing the NP-hardness from one problem to another problem. We introduce one of them as follows.

Consider two decision problems A and B. A is said to be polynomial-time many-one reducible to B, denoted by $A \leq_m^p B$, if there exists a polynomial-time computable function f mapping from all inputs of A to inputs of B such

that A receives yes-answer on input x if and only if B receives yes-answer on input $f(x)$ (Fig. 8.5).

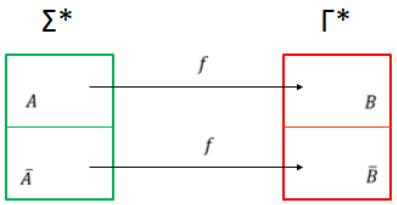


Figure 8.5: Polynomial-time many-one reduction.

For example, we have

Example 8.2.5. *The Hamiltonian cycle problem is polynomial-time many-one reducible to the decision version of the traveling salesman problem.*

Proof. To construct this reduction, for each input graph $G = (V, E)$ of the Hamiltonian cycle problem, we consider V as the set of cities and define a distance table D by setting

$$d(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ |V| + 1 & \text{otherwise.} \end{cases}$$

Moreover, set $K = |V|$. If G contains a Hamiltonian cycle, this Hamiltonian cycle would give a tour with total distance $|V| = K$ for the traveling salesman problem on defined instance. Conversely, if the traveling salesman problem on defined instance has a Hamiltonian tour with total distance at most K, then this tour cannot contain an edge $(u, v) \notin E$ and hence it induces a Hamiltonian cycle in G. Since the reduction can be constructed in polynomial-time, it is a polynomial-time many-one reduction from the Hamiltonian cycle problem to the decision version of the traveling salesman problem. □

There are two important properties of the polynomial-time many-one reduction.

Proposition 8.2.6. *(a) If $A \leq_m^p B$ and $B \leq_m^p C$, then $A \leq_m^p C$.*
(b) If $A \leq_m^p B$ and $B \in P$, then $A \in P$.

Proof. . (a) Let $A \leq_m^p B$ via f and $B \leq_m^p C$ via g. Then $A \leq_m^p C$ via h where $h(x) = g(f(x))$. Let f and g be computable in polynomial-times $p(n)$ and $q(n)$, respectively. Then for any x with $|x| = n$, $|f(x)| \leq p(n)$. Hence, h can be computed in time $p(n) + q(p(n))$ (b) Let $A \leq_m^p B$ via f. f

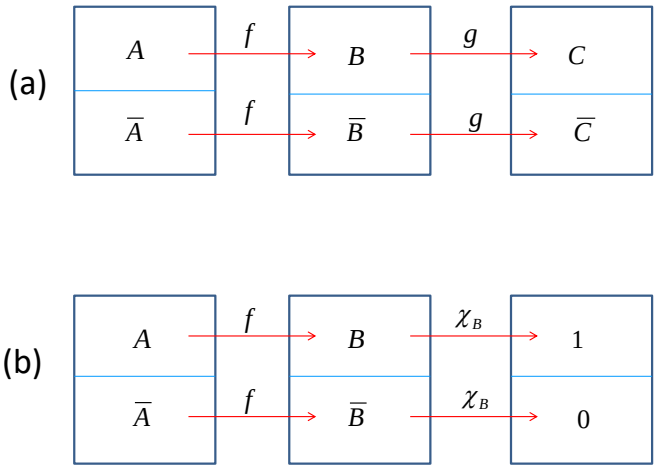


Figure 8.6: The proof of Proposition 8.2.6.

is computable in polynomial-time $p(n)$ and B can be solved in polynomial-time $q(n)$. Then A can be solved in polynomial-time $p(n) + q(p(n))$. □

Property (a) indicates that $\leq_m^p$ is a partial ordering. Property (b) gives us a simple way to establish the NP-hardness of a decision problem. To show the NP-hardness of a decision problem B, it suffices to find an NP-complete problem A and prove $A \leq_m^p B$. In fact, if $B \in P$, then $A \in P$. Since A is NP-complete, every problem in NP is polynomial-time solvable. Therefore, B is NP-hard.

The SAT problem is the root to establish the NP-hardness of almost all other problems. However, it is hard to use the SAT problem directly to construct reduction. Often, we use an NP-complete special case of the SAT problem. To introduce this special case, let us first explain a special type of Boolean formulas, 3CNF.

A *literal* is either a Boolean variable or the negation of a Boolean variable.

An *elementary sum* is a sum of several literals. Consider an elementary sum c and a Boolean function f. If $c = 0$ implies $f = 0$, then c is called a *clause* of f. A CNF (conjunctive normal form) is a product of its clauses. A CNF is called a 3CNF if each clause of the CNF contains exactly three distinct literals about three variables.

Problem 8.2.7. 3SAT*: Given a 3CNF F, determine whether the F is satisfiable.*

Theorem 8.2.8. *The 3SAT problem is NP-complete.*

Proof. It is easy to see that the 3SAT problem belongs to NP. Next, we show $SAT \leq_m^p 3SAT$.

First, we show two facts.

(a) $w = x + y$ if and only if $p(w, x, y)$ is satisfiable where

$$p(w, x, y) = (\bar{w} + x + y)(w + \bar{x} + y)(w + x + \bar{y})(w + \bar{x} + \bar{y}).$$

(b) $w = xy$ if and only if $q(w, x, y)$ is satisfiable where

$$q(w, x, y) = p(\bar{w}, \bar{x}, \bar{y}).$$

To show (a), we note that $w = x + y$ if and only if $\bar{w}\bar{x}\bar{y} + w(x + y) = 1$. Moreover, we have

$$\begin{aligned} & \bar{w}\bar{x}\bar{y} + w(x + y) \\ = \ & (\bar{w} + x + y)(\bar{x}\bar{y} + w) \\ = \ & (\bar{w} + x + y)(\bar{x} + w)(\bar{y} + w) \\ = \ & (\bar{w} + x + y)(w + \bar{x} + y)(w + x + \bar{y})(w + \bar{x} + \bar{y}) \\ = \ & q(w, x, y). \end{aligned}$$

Therefore, (a) holds.

(b) can be derived from (a) by noting that $w = xy$ if and only if $\bar{w} = \bar{x} + \bar{y}$.

Now, consider a Boolean formula F. F must contain a term xy or $x + y$ where x and y are two literals. In the former case, replace xy by a new variable w in F and set $F \leftarrow q(w, x, y)F$. In the latter case, replace $x + y$ by a new variable w in F and set $F \leftarrow p(w, x, y)F$. Repeat this operation until F becomes a literal z. Let u and v be two new variables. Finally, set $F \leftarrow F(z + u + v)(z + \bar{u} + v)(z + u + \bar{v})(z + \bar{u} + \bar{v})$. Then the original F is satisfiable if and only if the new F is satisfiable. □

Starting from the 3SAT problem through polynomial-time many-one reduction, there are a very large number of combinatorial optimization problems; their decision versions have been proved to be NP-complete. Moreover, none of them have been found to have a polynomial-time solution. If one of them has a polynomial-time solution, so do others. This fact makes one confidently say: An NP-hard problem is **unlikely** to have a polynomial-time solution. This "unlikely" can be removed only if P≠NP is proved, which is a big open problem in the literature.

Since for NP-hard combinatorial optimization problems, they are unlikely to have polynomial-time exact solution, we have to move our attention from exact solutions to approximation solutions. How do we design and analyze approximation solution? Those techniques will be studied systematically in next a few chapters. Before to do so, we would touch a few fundamental NP-complete problems and their related combinatorial optimization problems with their approximation solution in later sections of this chapter.

To end of this section, let us mention a rough way to judge whether a problem has a possible polynomial-time solution or not. Note that in many cases, it is easy to judge whether a problem belongs to NP or not. For a decision problem A in NP, if it is hard to find a polynomial-time solution, then we may study its complement $\bar{A} = \{x \mid x \notin A\}$. If $\bar{A} \in$ NP, then we may need to try hard to find a polynomial-time solution. If is is hard to show $\bar{A} \in NP$, then we may try to show NP-hardness of problem A.

Actually, let co-NP denote the class consisting of all complements of decision problems in NP. Then class P is contained in the intersection of NP and co-NP (Fig. 8.7). So far, no natural problem has been found to exist in

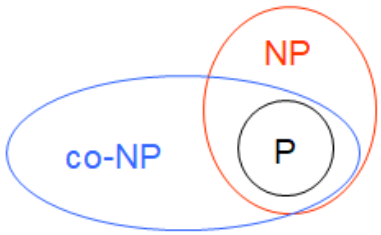


Figure 8.7: Intersection of NP and co-NP.

(NP∩co-NP)\P.

In the history, there are two well-known open problems existing in NP∩co-

NP, and was unknown to have polynomial-time solutions for many years. They are the primality test and the decision version of linear program. Finally, they both have been found to have polynomial-time solutions.

8.3 Hamiltonian Cycle

Consider an NP-complete decision problem A and a possible NP-hard decision problem B. How do we construct a polynomial-time many-one reduction? Every reader who has no experience would like to know the answer of this question. Of course, we would not have an efficient method to produce such a reduction. Indeed, we do not know if such an method exists. However, we may give some idea to follow.

Let us recall how to show a polynomial-time many-one reduction from A to B.

(1) Construct a polynomial-time computable mapping f from all inputs of problem A to inputs of problem B.

(2) Prove that problem A on input x receives yes-answer if and only if problem B on input $f(x)$ receives yes-answer.

Since the mapping f has to satisfy (2), the idea is to find the relationship of output of problem A and output of problem B, that is, **find the mapping from inputs to inputs through the relationship between outputs of two problems.** Let us explain this idea through an example.

Theorem 8.3.1. *The Hamiltonian cycle problem is NP-complete.*

Proof. We already proved previously that the Hamiltonian cycle problem belongs to NP. Next, we are going to construct a polynomial-time many-one reduction from the NP-complete 3SAT problem to the Hamiltonian cycle problem.

The input of the 3SAT problem is a 3CNF F and the input of the Hamiltonian cycle problem is a graph G. We need to find a mapping f such that for any 3CNF F, $f(F)$ is a graph such that F is satisfiable if and only if $f(F)$ contains a Hamiltonian cycle. What can make F satisfiable? It is a satisfied assignment. Therefore, our construction should give a relationship between assignments and Hamiltonian cycles. Suppose F contains n variables $x_1, x_2, ..., x_n$ and m clauses $C_1, C_2, ..., C_m$. To do so, we first build a ladder H_i with $4m+2$ levels, corresponding to a variable x_i as shown in Fig. 8.8. In this ladder, there are exactly two Hamiltonian paths corresponding two values 0 and 1 for x_i. Connect n ladders into a cycle as shown in Fig. 8.9. Then we obtained a graph H with exactly 2^n Hamiltonian cycles corresponding to 2^n assignments of F.

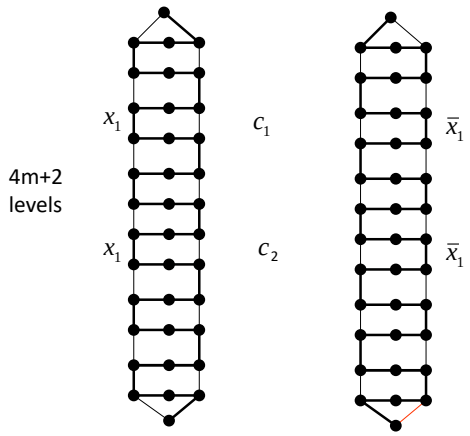


Figure 8.8: A ladder H_i contains exactly two Hamiltonian paths between two ends.

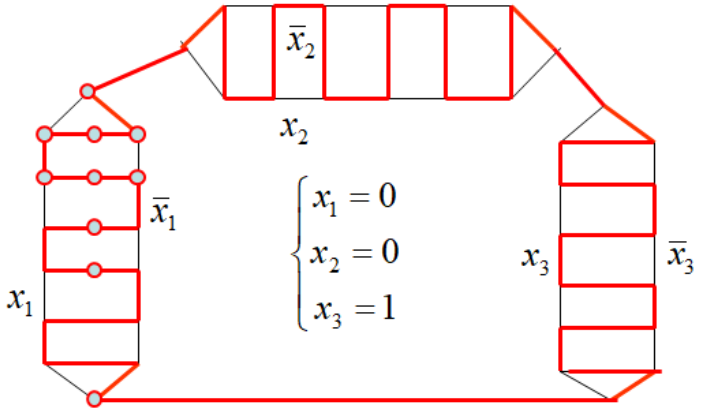


Figure 8.9: Each Hamiltonian cycle of graph H represents an assignment.

Now, we need to find a way to involve clauses. An idea is to represent each clause C_j by a point and represent the fact "clause C_j is satisfied under an assignment" by the fact "point C_j is included in the Hamiltonian cycle corresponding to the assignment". To realize this idea, for each literal x_i in clause C_j, we connected point C_j to two endpoints of an edge, between the $(4j-1)$th level and the $(4j)$th level, on the path corresponding to $x_i = 1$ (Fig. 8.10), and for each $\bar{x}_i$ in clause C_j, we connected point C_j to two endpoints of an edge on the path corresponding to $x_i = 0$. This completes our construction for graph $f(F) = G$.

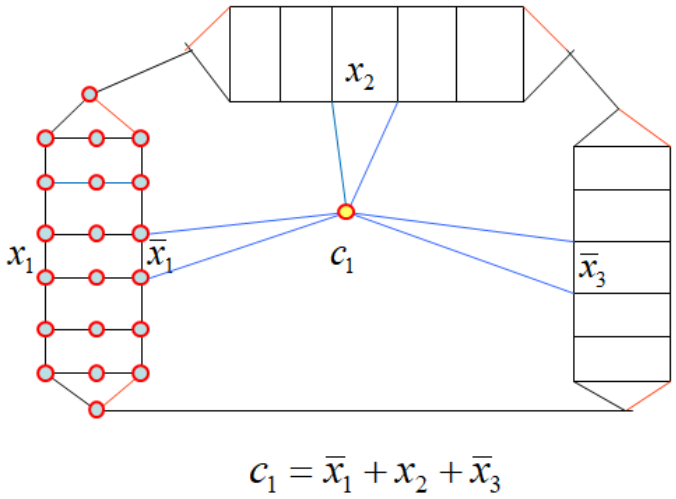


Figure 8.10: A point C_1 is added.

To see this construction meeting our requirement, we first assume F has a satisfied assignment σ and show that G has a Hamiltonian cycle. To this end, we find the Hamiltonian cycle C in H corresponding to the satisfied assignment. Note that each clause C_j contains a literal $y = 1$ under assignment σ. Thus, C_j is connected to endpoints of an edge (u, v) on the path corresponding to $y = 1$. Replacing this edge (u, v) by two edges (C_j, u) and (C_j, v) would include point C_j into the cycle, which would becomes a Hamiltonian cycle of G when all points C_j are included.

Conversely, suppose G has a Hamiltonian cycle C. We claim that in C, each point C_j must connect to two endpoints of an edge (u, v) in H. If our claim holds, then replace two edges (C_j, u) and (C_j, v) by edge (u, v). We would obtain a Hamiltonian cycle of graph H, corresponding an assignment

of F, which makes every clause C_j satisfied.

Now, we show the claim. For contradiction, suppose that cycle C contains its two edges (C_j, u) and (C_j, v) for some clause C_j such that u and v are located in different H_i and $H_{i'}$, respectively, with $i \neq i'$. To find a contradiction, we look at closely the local structure of vertex u as shown in Fig. 8.11. Note that each ladder is constructed with length longer enough

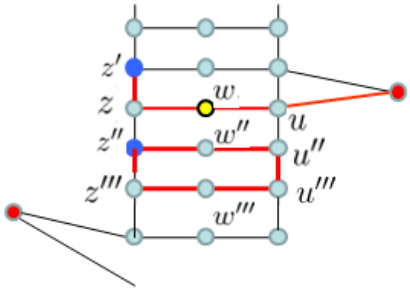


Figure 8.11: Local structure near vertex u.

so that every clause C_j has a special location in ladder H_i and locations for different clauses with at least distance three away each other (see Fig. 8.8). This makes that at vertex u, edges possible in C form a structure as shown in Fig. 8.11. In this local structure, since cycle C contains vertex w, C must contain edges (u, w) and (w, z), which imply that (u, u') and (u, u'') are not in C. Note that either (z, z') or (z, z'') is not in C. Without loss of generality assume that (z, z'') is not in C. Then edges possible in C form a structure as shown in 8.11. Since Hamiltonian cycle C contains vertices u'', w'', and z'', C must contain edges (u'', u'''), (u'', w''), (w'', z''), (z'', z'''). Since C contains vertex w''', C must contain edges (u''', w''') and (w''', z'''). This means that C must contain the small cycle $(u'', w'', z'', z''', w''', u''')$. However, a Hamiltonian cycle is a simple cycle which cannot properly contain a small cycle, a contradiction. □

Next, we give some examples in each of which the NP-hardness is established by reductions from the Hamiltonian cycle problem.

Problem 8.3.2 (Hamiltonian Path)**.** *Given a graph $G = (V, E)$, does G contains a Hamiltonian path? Here, a* Hamiltonian path *of a graph G is a simple path on which every vertex appears exactly once.*

Theorem 8.3.3. *The Hamiltonian path problem is NP-complete.*

Proof. The Hamiltonian path problem belongs to NP because we can guess a permutation of all vertices in $O(n \log n)$ time and then check, in $O(n)$ time, whether guessed permutation gives a Hamiltonian path. To show the NP-hardness of the Hamiltonian path problem, we may modify the proof of Theorem 8.3.1, to construct a reduction from the 3SAT problem to the Hamiltonian path problem by making a little change on graph H, which is obtained from connecting all H_i into a path instead of a cycle. However, in the following we would like to give a simple proof by reducing the Hamiltonian cycle problem to the Hamiltonian path problem.

We are going to find a polynomial-time computable mapping f from graphs to graphs such that G contains a Hamiltonian cycle if and only if $f(G)$ contains a Hamiltonian path. Our analysis starts from how to build a relationship between a Hamiltonian cycle of G and a Hamiltonian path of $f(G)$. If $f(G) = G$, then from a Hamiltonian cycle of G we can find a Hamiltonian path of $f(G)$ by deleting an ege; however, from a Hamiltonian path of $f(G)$ we may not be able to find a Hamiltonian cycle of G. To have "if and only if" relation, we first consider a simple case that there is an edge (u, v) such that if G contains a Hamiltonian cycle C, then C must contains edge (u, v). In this special case, we may put two new edges (u, u') and (v, v') at u and v, respectively.

For simplicity of speaking, we may call these two edges as two *horns*. Now, if G has the Hamiltonian cycle C, then $f(G)$ has a Hamiltonian path between endpoints of two horns, u' and v', Conversely, if $f(G)$ has a Hamiltonian path, then this Hamiltonian path must have two endpoints u' and v'; hence we can get back C by deleting two horns and putting back edge (u, v).

Now, we consider the general case that such an edge (u, v) may not exists. Note that for any vertex u of G, suppose u have k neighbors $v_1, v_2, ..., v_k$. Then a Hamiltonian cycle of G must contain one of edges (u, v_1), (u, v_2), ..., (u, v_k). Thus, we may first connect all $v_1, v_2, ..., v_k$ to a vertex u' and put two horns (u, w) and (u', w') (Fig. 8.12). This construction would work similarly as above. □

As a corollary of Theorem 8.3.1, we have

Corollary 8.3.4. *The traveling salesman problem is NP-hard.*

Proof. In Example 8.2.5, a polynomial-time many-one reduction has been constructed from the Hamiltonian cycle problem to the traveling salesman problem. □

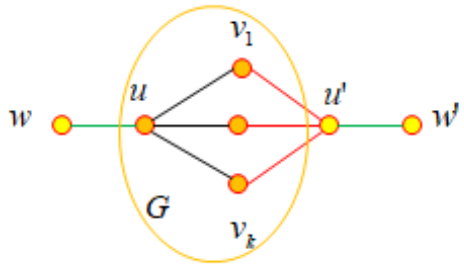


Figure 8.12: Install two horns at u and its copy u'.

The longest path problem is a maximization problem as follows:

Problem 8.3.5 (Longest Path)**.** *Given a graph $G = (V, E)$ with positive edge length $c : E \to R^+$, and two vertices s and t, find a longest simple path between s and t.*

As another corollary of Theorem 8.3.1, we have

Corollary 8.3.6. *The longest path problem is NP-hard.*

Proof. We will construct a polynomial-time many-one reduction from the Hamiltonian cycle problem to the decision version of the longest path problem as follows: *Given a graph $G = (V, E)$ with positive edge length $c : E \to R^+$, two vertices s and t, and an integer $K > 0$, is there a simple path between s and t with length at least K?*.

Let graph $G = (V, E)$ be an input of the Hamiltonian cycle problem. Choose a vertex $u \in V$. We make a copy of u by adding a new vertex u' and connecting u' to all neighbors of u. Add two new edges (u, s) and (u', t). Obtained graph is denoted by $f(G)$. Let $K = |V| + 2$. We show that G contains a Hamiltonian cycle if and only if $f(G)$ contains a simple path between s and t with length at most K.

First, assume that G contains a Hamiltonian cycle C. Break C at vertex u by replacing an edge (u, v) with (u', v). We would obtain a simple path between u and u' with length $|V|$. Extend this path to s and t. We would obtain a simple path between s and t with length $|V| + 2 = K$.

Conversely, assume that $f(G)$ contains a simple path between s and t with length at most K. Then this path contains a simple subpath between

u and u' with length $|V|$. Merge u and u' by replacing edge (u', v), on the subpath, with edge (u, v). Then we would obtain a Hamiltonian cycle of G. □

For NP-hard optimization problems like the traveling salesman problem and the longest path problem, it is unlikely to have an efficient algorithm to compute their exact optimal solution. Therefore, one usually study algorithms which produce approximation solutions for them. Such algorithms are simply called *approximations*.

For example, let us study the traveling salesman problem. When the given distance table satisfies the triangular inequality, that is,

$$d(a, b) + d(b, c) \geq d(a, c)$$

for any three vertices a, b and c where $d(a, b)$ is the distance between a and b, there is an easy way to obtain a tour (i.e, a Hamiltonian cycle) with total distance within twice from the optimal.

To do so, at the first compute a minimum spanning tree in the input graph and then travel around the minimum spanning tree (see Fig. 8.13). During this trip, a vertex which appearing at the second time can be skipped without increasing the total distance of the trip due to the triangular inequality. Note that the length of a minimum spanning tree is smaller than the minimum length of a tour. Moreover, this trip uses each edge of the minimum spanning tree exactly twice. Thus, the length of the Hamiltonian cycle obtained from this trip is within twice from the optimal.

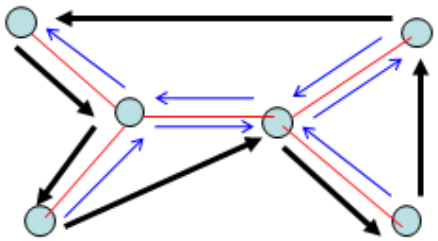


Figure 8.13: Travel around the minimum spanning tree.

Christofids in 1976 introduced an idea to improve above approximation. After computing the minimum spanning tree, he consider all vertices of odd degree (called *odd vertices*) in the tree and compute a minimum perfect matching among these odd vertices. Because in the union of the minimum spanning tree and the minimum perfect matching, every vertex has even degree, one can travel along edges in this union using each edge exactly once. This trip, called *Euler tour*, can be modified into a traveling salesman tour (Fig.8.14), without increasing the length by the triangular inequality. Thus, an approximation is produced with length bounded by the length of

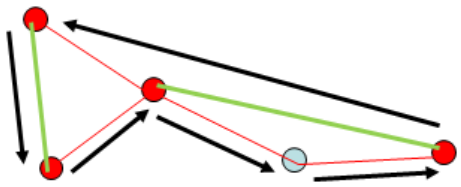


Figure 8.14: Christofids approximation.

minimum spanning tree plus the length of the minimum perfect matching on the set of vertices with odd degree. We claim that each Hamiltonian cycle (namely, a traveling salesman tour) can be decomposed into a disjoint union of two parts that each is not smaller than the minimum perfect matchings for vertices with odd degree. To see this, we first note that the number of vertices with odd degree is even since the sum of degrees over all vertices in a graph is even. Now, let $x_1, x_2, \cdots, x_{2k}$ denote all vertices with odd degree in clockwise ordering of the considered Hamiltonian cycle. Then $(x_1, x_2), (x_3, x_4), \cdots, (x_{2k-1}, x_{2k})$ form a perfect matching for vertices with odd degree and $(x_2, x_3), (x_4, x_5), \cdots, (x_{2k}, x_1)$ form the other perfect matching. The claim then follows immediately from the triangular inequality. Thus, the length of the minimum matching is at most half of the length of the minimum Hamiltonian cycle. Therefore, Christofids gave an approximation within a factor of 1.5 from the optimal.

From the above example, we see that the ratio of objective function values between approximation solution and optimal solution is a measure

for the performance of an approximation.

For a minimization problem, the *performance ratio* of an approximation algorithm A is defined as follows:

$$r(A) = \sup_I \frac{A(I)}{opt(I)}$$

where I is over all possible instances and $A(I)$ and $opt(I)$ are respectively the objective function values of the approximation produced by algorithm A and the optimal solution with respect to instance I.

For a maximization problem, the performance ratio of an approximation algorithm A is defined by

$$r(A) = \inf_I \frac{A(I)}{opt(I)}.$$

For example, the performance ratio of Christofids approximation is at most $3/2$ as we showed in the above. Actually, the performance ratio of Christofids approximation is exactly $3/2$. To see this, we consider $2n+1$ points (vertices) with distances as shown in Figure 8.15. The minimum spanning tree of these $2n+1$ points has distance $2n$. It has only two odd vertices with distance $n(1+\varepsilon)$. Hence, the length of Christofids approximation is $2n + n(1+\varepsilon)$. Moreover, the minimum tour has length $(2n-1)(1+\varepsilon)+2$. Thus, in this example, $A(I)/opt(I) = (3n+n\varepsilon)/(2n+1+(2n-1)\varepsilon))$ which is appoarch to $3/2$ as ε goes to 0 and n goes to infinity.

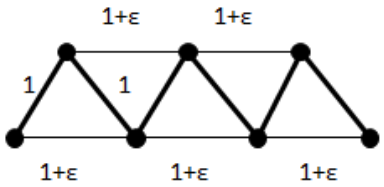


Figure 8.15: Extremal case for Christofids approximation.

Theorem 8.3.7. *For the traveling salesman problem in metric space, the Christofids approximation A has the performance ratio $r(A) = 3/2$.*

For simplicity, an approximation A is said to be α-approximation if $r(A) \leq \alpha$ for minimizations and $r(A) \geq \alpha$ for maximizations, that is, for every input I,

$$opt(I) \leq A(I) \leq \alpha \cdot opt(I)$$

for minimizations, and

$$opt(I) \geq A(I) \geq \alpha \cdot opt(I).$$

For example, Christofids approximation is a 1.5-approximation, but not α-approximation of the traveling salesman problem in metric space for any constant $\alpha < 1.5$.

Not every problem has a polynomial-time approximation with constant performance ratio. An example is the traveling salesman problem without triangular inequality condition on distance table. In fact, for contradiction, suppose that its performance ratio $r(A) \leq K$ for a constant K. Then we can show that the Hamiltonian cycle problem can be solved in polynomial time. For any graph $G = (V, E)$, define that for any pair of vertices u and v,

$$d(u, v) = \begin{cases} 1 & \text{if } \{u, v\} \in E \\ K \cdot |V| & \text{otherwise} \end{cases}$$

This gives an instance I for the traveling salesman problem. Then, G has a Hamiltonian cycle if and only if for I, the travel salesman has a tour with length at most $K|V|$. The optimal tour has length $|V|$. Applying approximation algorithm A to I, we will obtain a tour of length at most $K|V|$. Thus, G has a Hamiltonian cycle if and only if approximation algorithm A produces a tour of length at most $K|V|$. This means that the Hamiltonian cycle problem can be solved in polynomial time. Because the Hamiltonian cycle problem is NP-complete, we obtain a contradiction. The above argument proved the following.

Theorem 8.3.8. *If $P \neq NP$, then no polynomial-time approximation algorithm for the traveling salesman problem in general case has a constant performance ratio.*

For the longest path problem, there exists also a negative result.

Theorem 8.3.9. *For any $\varepsilon > 0$, the longest path problem has no polynomial-time $n^{1-\varepsilon}$-approximation unless $NP = P$.*

8.4 Vertex Cover

A vertex subset C is called a *vertex cover* if every edge has at least one endpoint in C. Consider following problem.

Problem 8.4.1 (Vertex Cover)**.** *Given a graph $G = (V, E)$ and a positive integer K, is there a vertex cover of size at most K?*

The vertex-cover problem is the decision version of the minimum vertex cover problem as follows.

Problem 8.4.2 (Minimum Vertex Cover)**.** *Given a graph $G = (V, E)$, compute a vertex cover with minimum cardinality.*

Theorem 8.4.3. *The vertex-cover problem is NP-complete.*

Proof. It is easy to show that the vertex cover problem is in NP. This can be done by guessing a vertex subset within $O(n \log n)$ time and checking whether obtained vertex subset is a vertex cover or not. Next, we show that the vertex cover problem is NP-hard.

Let F be a 3-CNF with m clauses $C_1, ..., C_m$ and n variables $x_1, ..., x_n$. We construct a graph $G(F)$ of $2n+3m$ vertices as follows: For each variable x_i, we give an edge with two endpoints labeled by two literals x_i and $\bar{x}_i$. For each clause $C_j = x + y + z$, we give a triangle $j_1 j_2 j_3$ and connect j_1 to literal x, j_2 to literal y and j_3 to literal z (Fig.8.16). Now, we prove that F is satisfiable if and only if $G(F)$ has a vertex cover of size at most $n + 2m$.

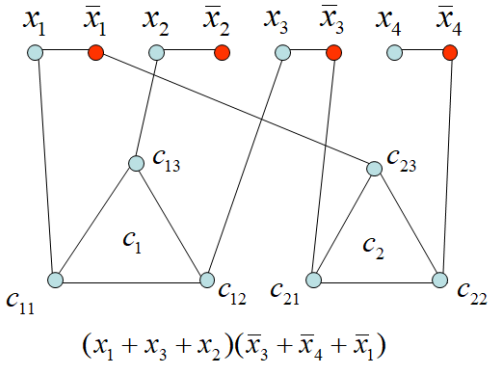


Figure 8.16: G(F)

First, suppose that F is satisfiable. Consider an assignment satisfying F. Let us construct a vertex cover S as follows: (1) S contains all truth

literals; (2) for each triangle $j_1j_2j_3$, put two vertices into S such that the remainder j_k is adjacent to a truth literal. Then S is a vertex cover of size exactly $n + 2m$.

Conversely, suppose that $G(F)$ has a vertex cover S of size at most $n + 2m$. Since each triangle $j_1j_2j_3$ must have at least two vertices in S and each edge $(x_i, \bar{x}_i)$ has at least one vertex in S, S must contain exactly two vertices in each triangle $j_1j_2j_3$ and exactly one vertex for each edge $(x_i, \bar{x}_i)$. Set

$$x_i = \begin{cases} 1 & \text{if } x_i \in S, \\ 0 & \text{if } \bar{x}_i \in S. \end{cases}$$

Then each clause C_j must have a truth literal which is the one adjacent to the j_k not in S. Thus, F is satisfiable.

The above construction is clearly polynomial-time computable. Hence, the 3SAT problem is polynomial-time many-one reducible to the vertex cover problem. □

Corollary 8.4.4. *The minimum vertex cover problem is NP-hard.*

Proof. It is NP-hard since its decision version is NP-complete. □

There are two combinatorial optimization problems closely related to the minimum vertex cover problem.

Problem 8.4.5 (Maximum Independent Set)**.** *Given a graph $G = (V, E)$, find an independent set with maximum cardinality.*

Here, an independent set is a subset of vertices such that no edge exists between two vertices in the subset. A subset of vertices is an independent set if and only if its complement is a vertex cover. In fact, from the definition, every edge has to have at least one endpoint in the complement of an independent set, which means that the complement of an independent set must be a vertex cover. Conversely, if the complement of a vertex subset I is a vertex cover, then every edge has an endpoint not in I and hence I is independent. Furthermore, it is easy to see that a vertex subset I is the maximum independent set if and only if the complement of I is the minimum vertex cover.

Problem 8.4.6 (Maximum Clique)**.** *Given a graph $G = (V, E)$, find a clique with maximum size.*

Here, a clique is a complete subgraph of input graph G and its size is the number of vertices in the clique. Let $\bar{G}$ be the complementary graph of G,

that is, an edge e is in $\bar{G}$ if and only if e is not in G. Then a vertex subset I is induced a clique in G if and only if I is an independent set in $\bar{G}$. Thus, a subgraph on a vertex subset I is a maximum clique in G if and only if I is a maximum independent set in $\bar{G}$.

From their relationship, we see clearly following.

Corollary 8.4.7. *Both the maximum independent set problem and the maximum clique problem are NP-hard.*

Next, we study the approximation of the minimum vertex cover problem.

Theorem 8.4.8. *The minimum vertex cover problem has a polynomial-time 2-approximation.*

Proof. Compute a maximal matching. The set of all endpoints of edges in this maximal matching form a vertex cover which is a 2-approximation for the minimum vertex cover problem since each edge in the matching must have an endpoint in the minimum vertex cover. □

The minimum vertex cover problem can be generalized to hypergraphs. This generalization is called the hitting set problem as follows:

Problem 8.4.9 (Hitting Set)**.** *Given a collection $\mathcal{C}$ of subsets of a finite set X, find a minimum subset S of X such that every subset in $\mathcal{C}$ contains an element in S. Such a set S is called a* hitting set*.*

For the maximum independent set problem and the maximum clique problem, there are negative results on their approximation.

Theorem 8.4.10. *For any $\varepsilon > 0$, the maximum independent set problem has no polynomial-time $n^{1-\varepsilon}$-approximation unless $NP = P$.*

Theorem 8.4.11. *For any $\varepsilon > 0$, the maximum clique problem has no polynomial-time $n^{1-\varepsilon}$-approximation unless $NP = P$.*

8.5 Three-Dimensional Matching

Consider another well-known NP-complete problem.

Problem 8.5.1 (Three-Dimensional Matching (3DM))**.** *Consider three disjoint sets X, Y, Z each with n elements and 3-sets each consisting three elements belonging to X, Y, and Z, respectively. Given a collection $\mathcal{C}$ of 3-sets, determine whether $\mathcal{C}$ contains a three-dimensional matching, where*

a subcollection $\mathcal{M}$ of $\mathcal{C}$ is called a three-dimensional matching *if $\mathcal{M}$ consists of n 3-sets such that each element of $X \cup Y \cup Z$ appears exactly once in 3-sets of $\mathcal{M}$.*

Theorem 8.5.2. *The 3DM problem is NP-complete.*

Proof. First, the 3DM problem belongs to NP because we can guess a collection of n 3-sets within $O(n \log n)$ time and check, in $O(n+m)$ time, whether obtained collection is a three-dimensional matching in given collection $\mathcal{C}$.

Next, we show the NP-hardness of the 3DM problem by constructing a polynomial-time many-one reduction from the 3SAT problem to the 3DM problem. Consider an input 3CNF F of the 3SAT problem. Suppose that F contains n variables $x_1, ..., x_n$ and m clauses $C_1, ..., C_m$. Construct a collection $\mathcal{C}$ of 3-sets as follows.

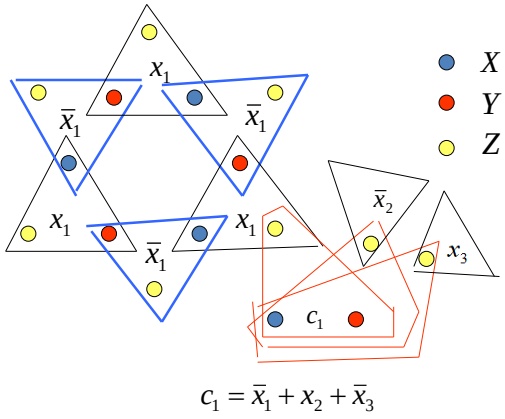


Figure 8.17: Proof of Theorem 8.5.2.

- For each variable x_i, construct $2m$ 3-sets $\{x_{i1}, y_{i1}, z_{i1}\}$, $\{x_{i2}, y_{i1}, z_{i2}\}$, $\{x_{i2}, y_{i2}, z_{i3}\}$, ..., $\{x_{i1}, y_{im}, z_{2m}\}$. They form a cycle as shown in Fig.8.17.
- For each clause C_j consisting variables $x_{i_1}, x_{i_2}, x_{i_3}$, construct three 3-sets, $\{x_{0j}, y_{0j}, z_{i_1 k_1}\}$, $\{x_{0j}, y_{0j}, z_{i_2 k_2}\}$, and $\{x_{0j}, y_{0j}, z_{i_3 k_3}\}$ where for $h = 1, 2, 3$,
$$k_h = \begin{cases} 2j-1 & \text{if } C_j \text{ contains } x_{i_h}, \\ 2j & \text{if } C_j \text{ contains } \bar{x}_{i_h}. \end{cases}$$
- For each $1 \le h \le m(n-1)$, $1 \le i \le n$, $1 \le k \le 2m$, construct 3-set $\{x_{n+1,h}, y_{n+1,h}, z_{ik}\}$.

- Collect all above x_{pq}, y_{pq}, and z_{pq} to form sets X, Y, and Z, respectively.

Now, suppose $\mathcal{C}_F$ has a three-dimensional matching $\mathcal{M}$. Note that each element appears in $\mathcal{M}$ exactly once. For each variable x_i, $\mathcal{M}$ contains either

$$P_i = \{\{\{x_{i1}, y_{i1}, z_{i1}\}, \{x_{i2}, y_{i2}, z_{i3}, ..., \{x_{im}, y_{im}, z_{i,2m-1}\}\}$$

or

$$Q_i = \{\{\{x_{i1}, y_{i2}, z_{i2}\}, \{x_{i2}, y_{i3}, z_{i4}, ..., \{x_{im}, y_{i1}, z_{i,2m}\}\}$$

Define

$$x_i = \begin{cases} 1 & \text{if } P_i \subseteq \mathcal{M}, \\ 0 & \text{if } Q_i \subseteq \mathcal{M}. \end{cases}$$

Then this assignment will satisfy F. In fact, for any clause C_j. In order to have elements x_{0j} and y_{0j} appear in $\mathcal{M}$, $\mathcal{M}$ must contain 3-set $\{x_{0j}, y_{0j}, z_{i_h k_h}\}$ for some $h \in \{1, 2, 3\}$. This assignment will assign 1 to the hth literal of C_j according to the construction.

Conversely, suppose F has a satisfied assignment. We can construct a three-dimensional matching $\mathcal{M}$ as follows.

- if $x_i = 1$, then put P_i into $\mathcal{M}$. If $x_i = 0$, then put Q_i into $\mathcal{M}$.
- If hth literal of clause C_j is equal to 1, then put 3-set $\{x_{0j}, y_{0j}, z_{i_h k_h}\}$ into $\mathcal{M}$.
- So far, all elements in $X \cup Y$ have been covered by 3-sets put in $\mathcal{M}$. However, there are $m(n-1)$ elements of Z are left outside. We now use 3-sets $\{x_{n+1,h}, y_{n+1,h}, z_{ik}\}$ to play a role of garbage collector. For each z_{ik} not appearing in 3-sets in $\mathcal{M}$, select a pair of $x_{n+1,h}$ and $y_{n+1,h}$, and then put 3-set $\{x_{n+1,h}, y_{n+1,h}, z_{ik}\}$ into $\mathcal{M}$.

□

There is a combinatorial optimization problem closely related to the three-dimensional matching problem.

Problem 8.5.3 (Set Cover)**.** *Given a collection $\mathcal{C}$ of subsets of a finite set X, find a minimum set cover $\mathcal{A}$ where a set cover $\mathcal{A}$ is a subcollection of $\mathcal{C}$ such that every element of X is contained in a subset in $\mathcal{A}$.*

Theorem 8.5.4. *The set cover problem is NP-hard.*

Proof. Note that the decision version of the set cover problem is as follows: Given a collection $\mathcal{C}$ of subsets of a finite set X and a positive integer $k \leq |X|$, determine whether there exists a set cover of size at most k.

We construct a polynomial-time many-one reduction from the three-dimensional matching problem to the decision version of the set cover problem. Let $(X, Y, Z, \mathcal{C})$ be an instance of the three-dimensional matching problem. Construct an instance $(X, \mathcal{C}, k)$ of the decision version of the set cover problem by setting

$$\begin{aligned} X &\leftarrow X \cup Y \cup Z, \\ \mathcal{C} &\leftarrow \mathcal{C}, \\ k &\leftarrow |X \cup Y \cup Z|. \end{aligned}$$

Clearly, for instance $(X, Y, Z, \mathcal{C})$, a three-dimensional matching exists if and only if for instance $(X, \mathcal{C}, k)$, a set cover of size k exists. □

For any subcollection $\mathcal{A} \subseteq \mathcal{C}$, define

$$f(\mathcal{A}) = |\cup_{A \in \mathcal{A}} A|.$$

The set cover problem has a greedy approximation as follows:

Algorithm 21 Greedy Algorithm SC.

Input: A finite set X and a collection of subsets of X.
Output: A subcollection $\mathcal{A}$ of $\mathcal{C}$.

1: $\mathcal{A} \leftarrow \emptyset$.
2: **while** $f(\mathcal{A}) < |X|$ **do**
3: choose $A \in \mathcal{C}$ to maximize $f(\mathcal{A} \cup \{A\})$
4: and set $\mathcal{A} \leftarrow \mathcal{A} \cup \{A\}$
5: **end while**
6: **return** $\mathcal{A}$.

This approximation can be analyzed as follows:

Lemma 8.5.5. *For any two subcollections $\mathcal{A} \subset \mathcal{B}$ and any subset $A \subseteq X$,*

$$\Delta_A f(\mathcal{A}) \geq \Delta_A f(\mathcal{B}), \tag{8.1}$$

where $\Delta_A f(\mathcal{A}) = f(\mathcal{A} \cup \{A\}) - f(\mathcal{A})$.

Proof. Since $\mathcal{A} \subset \mathcal{B}$, we have

$$\Delta_A f(\mathcal{A}) = |A \setminus \cup_{S \in \mathcal{A}} S| \geq |A \setminus \cup_{S \in \mathcal{B}} S| = \Delta_A f(\mathcal{B}).$$

□

Theorem 8.5.6. *Greedy Algorithm SC is a polynomial-time* $(1 + \ln \gamma)$*-approximation for the set cover problem, where* γ *is the maximum cardinality of a subset in input collection* $\mathcal{C}$*.*

Proof. Let $A_1, ..., A_g$ be subsets selected in turn by Greedy Algorithm SC. Denote $\mathcal{A}_i = \{A_1, ..., A_i\}$. Let *opt* be the number of subsets in a minimum set-cover.

Let $\{C_1, ..., C_{opt}\}$ be a minimum set-cover. Denote $\mathcal{C}_j = \{C_1, ..., C_j\}$.

By the greedy rule,

$$f(\mathcal{A}_{i+1}) - f(\mathcal{A}_i) = \Delta_{A_{i+1}} f(\mathcal{A}_i) \geq \Delta_{C_j} f(\mathcal{A}_i)$$

for $1 \leq j \leq opt$. Therefore,

$$f(\mathcal{A}_{i+1}) - f(\mathcal{A}_i) \geq \frac{\sum_{j=1}^{opt} \Delta_{C_j} f(\mathcal{A}_i)}{opt}.$$

On the other hand,

$$\begin{aligned} \frac{|X| - f(\mathcal{A}_i)}{opt} &= \frac{f(\mathcal{A}_i \cup \mathcal{C}_{opt}) - f(\mathcal{A}_i)}{opt} \\ &= \frac{\sum_{j=1}^{opt} \Delta_{C_j} f(\mathcal{A}_i \cup \mathcal{C}_{j-1})}{opt}. \end{aligned}$$

By Lemma 8.5.5,

$$\Delta_{C_j} f(\mathcal{A}_i) \geq \Delta_{C_j} f(\mathcal{A}_i \cup \mathcal{C}_{j-1}).$$

Therefore,

$$f(\mathcal{A}_{i+1}) - f(\mathcal{A}_i) \geq \frac{|X| - f(\mathcal{A}_i)}{opt}, \tag{8.2}$$

that is,

$$\begin{aligned} |X| - f(\mathcal{A}_{i+1}) &\leq (|X| - f(\mathcal{A}_i))(1 - \frac{1}{opt}) \\ &\leq |X|(1 - \frac{1}{opt})^{i+1} \\ &\leq |X| e^{-(i+1)/opt}. \end{aligned}$$

Choose i such that $|X| - f(\mathcal{A}_{i+1}) < opt \leq |X| - f(\mathcal{A}_i)$. Then

$$g \leq i + opt$$

and

$$opt \leq |X| e^{-i/opt}.$$

Therefore,

$$g \leq opt(1 + \ln \frac{|X|}{opt}) \leq opt(1 + \ln \gamma).$$

□

The following theorem indicates that above greedy approximation has the best possible performance ratio for the set cover problem.

Theorem 8.5.7. *For $\rho < 1$, there is no polynomial-time $(\rho \ln n)$-approximation for the set cover problem unless $NP = P$ where $n = |X|$.*

In the worst case, we may have $\gamma = n$. Therefore, this theorem indicates that the performance of Greedy algorithm is tight in some sense.

The hitting set problem is equivalent to the set cover problem. To see this equivalence, for each element $x \in X$, define $\mathcal{S}_x = \{C \in \mathcal{C} \mid x \in C\}$. Then the set cover problem on input $(X, \mathcal{C})$ is equivalent to the hitting set problem on input $(\mathcal{C}, \{\mathcal{S}_x \mid x \in X\})$. In fact, $\mathcal{A} \subseteq \mathcal{C}$ covers X if and only if $\mathcal{A}$ hits every S_x. From this equivalence, following is obtained immediately.

Corollary 8.5.8. *The hitting set problem is NP-hard and has a greedy $(1 + \ln \gamma)$-approximation. Moreover, for any $\rho < 1$, it has no polynomial-time $\rho \ln \gamma$-approximation unless NP=P.*

8.6 Partition

The partition problem is defined as follows.

Problem 8.6.1 (Partition)**.** *Given n positive integers $a_1, a_2, ..., a_n$, is there a partition (N_1, N_2) of $[n]$ such that $\sum_{i \in N_1} a_i = \sum_{i \in N_2} a_i$?*

To show NP-completeness of this problem, we first study another proble.

Problem 8.6.2 (Subsum)**.** *Given $n+1$ positive integers $a_1, a_2, ..., a_n$ and L where $1 \leq L \leq S = \sum_{i=1}^{n} a_i$, is there a subset N_1 of $[n]$ such that $\sum_{i \in N_1} a_i = L$?*

Theorem 8.6.3. *The subsum problem is NP-complete.*

Proof. The subsum problem belongs to NP because we can guess a subset N_1 of $[n]$ in $O(n)$ time and check, in polynomial-time, whether $\sum_{i \in N_1} a_i = L$.

Next, we show $3SAT \leq_m^p$ SUBSUM. Let F be a 3CNF with n variables $x_1, x_2, ..., x_n$ and m clauses $C_1, C_2, ..., C_m$. For each variable x_i, we construct two positive decimal integers b_{x_i} and $b_{\bar{x}_i}$, representing two literals x_i and $\bar{x}_i$,

respectively. Each b_{x_i} ($b_{\bar{x}_i}$) contains $m+n$ digits. Let $b_{x_i}[k]$ ($b_{\bar{x}_i}[k]$) be the kth rightmost digit of b_{x_i} ($b_{\bar{x}_i}$). Set

$$b_{x_i}[k] = b_{\bar{x}_i}[k] = \begin{cases} 1 & \text{if } k = i, \\ 0 & \text{otherwise} \end{cases}$$

for recording the ID of variable x_i. To record information on relationship between literals and clauses, set

$$b_{x_i}[n+j] = \begin{cases} 1 & \text{if } x_i \text{ appears in clause } C_j, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$b_{\bar{x}_i}[n+j] = \begin{cases} 1 & \text{if } \bar{x}_i \text{ appears in clause } C_j, \\ 0 & \text{otherwise.} \end{cases}$$

Finally, define $2m+1$ additional positive integers c_j, c'_j for $1 \le j \le m$ and L as follows:

$$c_j[k] = c'_j[k] = \begin{cases} 1 & \text{if } k = n+j, \\ 0 & \text{otherwise.} \end{cases}$$

$$L = \overbrace{3...3}^{m}\overbrace{1...1}^{n}.$$

For example, if $F = (x_1 + x_2 + \bar{x}_3)(\bar{x}_2 + \bar{x}_3 + x_4)$, then we would construct the following $2(m+n)+1 = 13$ positive integers.

$$\begin{aligned} b_{x_1} &= 010001, & b_{\bar{x}_1} &= 000001, \\ b_{x_2} &= 010010, & b_{\bar{x}_2} &= 100010, \\ b_{x_3} &= 000100, & b_{\bar{x}_3} &= 110100, \\ b_{x_4} &= 101000, & b_{\bar{x}_4} &= 001000, \\ c_1 = c'_1 &= 010000, & c_2 = c'_2 &= 100000, \\ L &= 331111. \end{aligned}$$

Now, we show that F has a satisfied assignment if and only if $A = \{b_{i,j} \mid 1 \le n, j = 0, 1\} \cup \{c_j, c'_j \mid 1 \le j \le m\}$ has a subset A' such that the sum of all integers in A' is equal to L.

First, suppose F has a satified assignment σ. For each variable x_i, put b_{x_i} into A' if $x_i = 1$ under assignment σ and put $b_{\bar{x}_i}$ into A' if $x_i = 0$ under assignment σ. For each clause C_j, put both c_j and c'_j into A' if C_j contains exactly one satisfied literal under assignment σ, put only c_j into A' if C_j contains exactly two satisfied literal under assignment σ, and put neither

c_j nor c'_j into A' if all three literals in C_j are satisfied under assignment σ. Clearly, obtained A' meets the condition that the sum of all numbers in A' isequal to L.

Conversely, suppose that there exists a subset A' of A such that the sum of all numbers in A is equal to L. Since $L[i] = 1$ for $1 \leq i \leq n$, A' contains exactly one of b_{x_i} and $b_{\bar{x}_i}$. Define an assignment σ by setting

$$x_i = \begin{cases} 1 & \text{if } b_{x_i} \in A', \\ 0 & \text{if } b_{\bar{x}_i} \in A'. \end{cases}$$

We claim that σ is a satisfied assignment for F. In fact, for any clause C_j, since $L[n + j] = 3$, there must be a b_{x_i} or $b_{\bar{x}_i}$ in A' whose the $(n + j)$th leftmost digit is 1. This means that there is a literal with assignment 1, appearing C_j, i.e., making C_j satisfied. □

Now, we show the NP-completeness of the partition problem.

Theorem 8.6.4. *The partition problem is NP-complete.*

Proof. The partition problem can be seen as the subsum problem in the special case that $L = S/2$ where $S = a_1 + a_2 + \cdots + a_n$. Therefore, it is in NP. Next, we show Subsum $\leq_m^p$ Partition.

Consider an instance of the subsum problem, consisting of $n+1$ positive integers $a_1, a_2, ..., a_n$ and L where $0 < L \leq S$. Since the partition problem is equivalent to the subsum problem with $2L = S$, we may assume without of generality that $2L \neq S$. Now, consider an input for the partition problem, consisting of $n+1$ positive integers $a_1, a_2, ..., a_n$ and $|2L - S|$. We will show that there exists a subset N_1 of $[n]$ such that $\sum_{i \in N_1} a_i = L$ if and only if $A = \{a_1, a_2, ..., a_n, |2L - S|\}$ has a partition (A_1, A_2) such the sum of all numbers in A_1 equals the sume of all numbers in A_2. Consider two cases.

Case 1. $2L > S$. First, suppose there exists a subset N_1 of $[n]$ such that $\sum_{i \in N_1} a_i = L$. Let $A_1 = \{a_i \mid i \in N_1\}$ and $A_2 = A - A_1$. Then, the sum of all numbers in A_2 is equal to

$$\sum_{i \in [n] - N_1} a_i + 2L - S = S - L + 2L - S = L = \sum_{i \in N_1} a_i.$$

Conversely, suppose A has a partition (A_1, A_2) such the sum of all numbers in A_1 equals the sume of all numbers in A_2. Without loss of generality, assume $2L - S \in A_2$. Note that the sum of all numbers in A equals $S + 2L - S = 2L$. Therefore, the sum of all numbers in A_1 equals L.

Case 2. $2L < S$. Let $L' = S - L$ and $N_2 = [n] - N_1$. Then $2L' - S > 0$ and $\sum_{i \in N_1} a_i = L$ if and only if $\sum_{i \in N_2} a_i = L'$. Therefore, this case can be done in a way similar to Case 1 by replacing L and N_1 with L' and N_2, respectively. □

We next study an optimization problem.

Problem 8.6.5 (Knapsack)**.** *Suppose you get in a cave and find n items. However, you have only a knapsack to carry them and this knapsack cannot carry all of them. The knapsack has a space limit S and the ith item takes space a_i and has value c_i. Therefore, you would face a problem of choosing a subset of items, which can be put in the knapsack, to maximize the total value of chosen items. This problem can be formulated into the following linear 0-1 programming.*

$$\begin{aligned} \max \quad & c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ \textit{subject to} \quad & a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq S \\ & x_1, x_2, ..., x_n \in \{0, 1\} \end{aligned}$$

In this 0-1 linear programming, variable x_i is an indicator that $x_i = 1$ if the ith item is chosen, and $x_i = 0$ if the ith item is not chosen.

Theorem 8.6.6. *The knapsack problem is NP-hard.*

Proof. The decision version of the knapsack problem is as follows: Given positive integers $a_1, a_2, ..., a_n$, $c_1, c_2, ..., c_n$, S and k, does following system of inequalities have 0-1 solution?

$$\begin{aligned} c_1x_1 + c_2x_2 + \cdots + c_nx_n &\geq k, \\ a_1x_1 + a_2x_2 + \cdots + a_nx_n &\leq S. \end{aligned}$$

We construct a polynomial-time many-one reduction from the partition problem to the decision version of the knapsack problem. Consider an instance of the partition problem, consisting positive integers $a_1, a_2, ..., a_n$. Define an instance of the decision version of the knapsack problem by setting

$$\begin{aligned} c_i &= a_i \text{ for } 1 \leq i \leq n \\ k = S &= \lfloor (a_1 + a_2 + \cdots + a_n)/2 \rfloor. \end{aligned}$$

Then the partition problem receives yes-answer if and only if the decision version of knapsack problem receives yes-answer. □

Algorithm 22 2-Approximation for Knapsack.

Input: n items $1, 2, ..., n$ and a knapsack with volume S. Each item i is associated with a positive volume a_i and a positive value c_i. Assume $a_i \leq S$ for all $i \in [n]$.

Output: A subset A of items with total value c_G.

sort all items into ordering $c_1/a_1 \geq c_2/a_2 \geq \cdots \geq c_n/a_n$;
$A \leftarrow \emptyset$, $k \leftarrow 1$;
if $\sum_{i=1}^{n} a_i \leq S$ **then**
  $A \leftarrow [n]$
else
  while $\sum_{i \in A} \leq S$ and $k < n$ **do**
    $k \leftarrow k + 1$
  end while
  if $\sum_{i=1}^{k_1} c_i > c_k$ **then**
    $A \leftarrow [k-1]$
  else
    $A \leftarrow \{k\}$
  end if
end if
$c_G \leftarrow \sum_{i \in A} c_i$;
return A and c_G.

The knapsack problem has a simple 2-approximation (Algorithm 22).

Without loss of generality, assume $a_i \leq S$ for every $1 \leq i \leq n$. Otherwise, item i can be removed from our consideration because it cannot be put in the knapsack. First, Sort all items into ordering $\frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \cdots \geq \frac{c_n}{a_n}$. Then put items one by one into knapsack according to this ordering, until no more item can be put in. Suppose that above process stops at the kth item, that is, either $k = n$ or first k items have been placed into the knapsack and the $(k+1)$th item cannot put in. In the former case, all n items can be put in the knapsack. In the latter case, if $\sum_{i=1}^{k} c_i > c_{k+1}$, then take the first k items to form a solution; otherwise, take the $(k+1)$th item as a solution.

Theorem 8.6.7. *Algorithm 22 produces a 1/2-approximation for the knapsack problem.*

Proof. If all items can be put in the knapsack, then this will give a simple optimal solution. If not, then $\sum_{i=1}^{k} c_i + c_{k+1} > opt$ where opt is the objective function value of an optimal solution. Hence $\max(\sum_{i=1}^{k} c_i, c_{k+1}) \geq 1/2 \cdot opt$. □

From above 2-approximation, we may have following observation: For an item selected into the knapsack, two facts are considered.

- The first fact is the ratio c_i/a_i. The larger ratio means that volume is used for higher value.
- The second fact is the c_i. When putting an item with small c_i and bigger c_i/a_i into the knapsack may affect the possibility of putting items with bigger c_i and smaller c_i/a_i, we may select the one with bigger c_i.

By properly balancing consideration on these two facts, we can obtain a construction for $(1+\varepsilon)$-approximation for any $\varepsilon > 0$ (Algorithm 23).

Denote $\alpha = c_G \cdot \frac{2\varepsilon}{1+\varepsilon}$ where c_G is the total value of a 2-approximation solution obtained by Algorithm 22. Classify all items into two sets A and B. Let A be the set of all items each with value $c_i < \alpha$ and B the set of all items each with value $c_i \geq \alpha$. Suppose $|A| = m$. Sort all items in A in ordering $c_1/a_1 \geq c_2/a_2 \geq \cdots \geq c_m/a_m$.

For any subset I of B, with $|I| \leq 1 + 1/\varepsilon$, if $\sum_{i \in I} a_i > S$, then define $c(I) = 0$; otherwise, select the largest $k \leq m$ satisfying $\sum_{i=1}^{k} a_i \leq S - \sum_{i \in I} c_i$ and define $c(I) = \sum_{i \in I} c_i + \sum_{i=1}^{k} c_i$.

Set $c_\varepsilon = \max_I c(I)$.

Algorithm 23 2-Approximation for Knapsack.

Input: n items $1, 2, ..., n$, a knapsack with volume S and a positive number ε. Each item i is associated with a positive volume a_i and a positive value c_i. Assume $a_i \leq S$ for all $i \in [n]$.

Output: A subset A_ε of items with total value c_ε.

run Algorithm 22 to obtain c_G;
$\alpha \leftarrow c_G \cdot \frac{2\varepsilon}{1+\varepsilon}$;
classify all items into A and B where
$A \leftarrow \{i \in [n] \mid c_i < \alpha\}$, $B \leftarrow \{i \in [n] \mid c_i \geq \alpha\}$;
sort all items of A into ordering $c_1/a_1 \geq c_2/a_2 \geq \cdots \geq c_m/a_m$;
$A_\varepsilon \leftarrow \emptyset$, $k \leftarrow 1$;
$\mathcal{B} \leftarrow \{IsubseteqB \mid |I| \leq 1 + 1/\varepsilon\}$;
for each $I \in \mathcal{B}$ **do**
  if $\sum_{i \in I} > S$ **then**
    $\mathcal{B} \leftarrow \mathcal{B} \setminus \{I\}$
  else
    $S \leftarrow S - \sum_{i \in I} c_i$;
    if $\sum_{i \in A} a_i > S$ **then**
      while $\sum_{i=1}^{k} a_i \leq S$ and $k < m$ **do**
        $k \leftarrow k + 1$
      end while
      if $\sum_{i=1}^{k_1} c_i > c_k$ **then**
        $A(I) \leftarrow [k-1]$
      else
        $A(I) \leftarrow \{k\}$
      end if
    else
      $A(I) \leftarrow A$
    end if
  end if
  $c(I) \leftarrow \sum_{i \in I \cup A(I)} c_i$;
end for
$I \leftarrow \text{argmax}_{I \in \mathcal{B}} c(I)$;
$A_\varepsilon \leftarrow I \cup A(I)$;
$c_\varepsilon \leftarrow c(I)$;
return A_ε and c_ε.

Lemma 8.6.8.

$$c_\varepsilon \geq \frac{1}{1+\varepsilon} \cdot opt$$

where opt is the objective function value of an optimal solution.

Proof. Let $I_b = B \cap OPT$ and $I_a = A \cap OPT$ where OPT is an optimal solution. Note that for $I \subseteq B$ with $|I| > 1 + 1/\varepsilon$, we have

$$\begin{aligned} \sum_{i \in I} a_i &> \alpha \cdot (1 + 1/\varepsilon) \\ &= c_G \cdot \frac{2\varepsilon}{1+\varepsilon} \cdot (1 + 1/\varepsilon) \\ &\geq opt. \end{aligned}$$

Thus, we must have $|I_b| \leq 1 + 1/\varepsilon$ and hence $c_\varepsilon \geq c(I_b)$. Moreover, we have

$$\begin{aligned} c(I_b) &= \sum_{i \in I_b} c_i + \sum_{i \in I_a} c_i \\ &\geq opt - \alpha \\ &= opt - c_G \cdot \frac{2\varepsilon}{1+\varepsilon} \\ &\geq opt - \frac{opt}{2} \cdot \frac{2\varepsilon}{1+\varepsilon} \\ &= opt \cdot \frac{1}{1+\varepsilon}. \end{aligned}$$

Therefore, $c_\varepsilon \geq opt \cdot \frac{1}{1+\varepsilon}$. □

Lemma 8.6.9. *Algorithm 23 runs in $O(n^{2+1/\varepsilon})$ time.*

Proof. Note that there are at most $n^{1+1/\varepsilon}$ subsets I of B with $|I| \leq 1 + 1/\varepsilon$. For each such I, the algorithm runs in $O(n)$ time. Hence, the total time is $O(n^{2+1/\varepsilon})$. □

An optimization problem is said to have PTAS (polynomial-time approximation scheme) if for any $\varepsilon > 0$, there is a polynomial-time $(1 + \varepsilon)$-approximation for the problem. By Lemmas 8.6.8 and 8.6.9, the knapsack problem has a PTAS.

Theorem 8.6.10. *Algorithm 23 provides a PTAS for the knapsack problem.*

A PTAS is called a FPTAS (fully polynomial-time approximation scheme) if for any $\varepsilon > 0$, there exists a $(1+\varepsilon)$-approximation with running time which is a polynomial with respect to $1/\varepsilon$ and the input size. Actually, the knapsack problem also has a FPTAS. To show it, let us first study exact solutions for the knapsack problem

Let $opt(k, S)$ be the objective function value of an optimal solution of the following problem:

$$\begin{aligned} \max \quad & c_1x_1 + c_2x_2 + \cdots + c_kx_k \\ \text{subject to} \quad & a_1x_1 + a_2x_2 + \cdots + a_kx_k \\ & x_1, x_2, ..., x_k \in \{0, 1\}. \end{aligned}$$

Then

$$opt(k, S) = \max(opt(k-1, S), c_k + opt(k-1, S-a_k)).$$

This recursive formula gives a dynamic programming to solve the knapsack problem within $O(nS)$ time. This is a pseudopolynomial-time algorithm, not a polynomial-time algorithm because the input size of S is $\lceil \log_2 S \rceil$, not S.

To construct a PTAS, we need to design another pseudopolynomial-time algorithm for the knapsack problem.

Let $c(i, j)$ denote a subset of index set $\{1, \ldots, i\}$ such that
(a) $\sum_{k \in c(i,j)} c_k = j$ and
(b) $\sum_{k \in c(i,j)} s_k = \min\{\sum_{k \in I} s_k \mid \sum_{k \in I} c_k = j, I \subseteq \{1, ..., i\}\}$.

If no index subset satisfies (a), then we say that $c(i, j)$ is undefined, or write $c(i, j) = nil$. Clearly, $opt = \max\{j \mid c(n, j) \neq nil$ and $\sum_{k \in c(i,j)} s_k \leq S\}$. Therefore, it suffices to compute all $c(i, j)$. The following algorithm is designed with this idea.

Initially, compute $c(1, j)$ for $j = 0, \ldots, c_{sum}$ by setting

$$c(1, j) := \begin{cases} \emptyset & \text{if } j = 0, \\ \{1\} & \text{if } j = c_1, \\ nil & \text{otherwise,} \end{cases}$$

where $c_{sum} = \sum_{i=1}^{n} c_i$.
Next, compute $c(i, j)$ for $i \geq 2$ and $j = 0, \ldots, c_{sum}$.

for $i = 2$ **to** n **do**
  for $j = 0$ **to** c_{sum} **do**
    case 1 $[c(i-1, j-c_i) = nil]$

set $c(i,j) = c(i-1,j)$
case 2 $[c(i-1,j-c_i) \neq nil]$
and $[c(i-1,j) = nil]$
set $c(i,j) = c(i-1,j-c_i) \cup \{i\}$
case 3 $[c(i-1,j-c_i) \neq nil]$
and $[c(i-1,j) \neq nil]$
if $[\sum_{k \in c(i-1,j)} s_k > \sum_{k \in c(i-1,j-c_i)} s_k + s_i]$
then $c(i,j) := c(i-1,j-c_i) \cup \{i\}$
else $c(i,j) := c(i-1,j)$;
Finally, set $opt = \max\{j \mid c(n,j) \neq nil \text{ and } \sum_{k \in c(i,j)} s_k \leq S\}$.

This algorithm computes the exact optimal solution for the knapsack problem with running time $O(n^3 M \log(MS))$ where $M = \max_{1 \leq k \leq n} c_k$, because the algorithm contains two loops, the outside loop runs in $O(n)$ time, the inside loop runs in $O(nM)$ time, and the central part runs in $O(n \log(MS))$ time. This is a pseudopolynomial-time algorithm because the input size of M is $\log_2 M$, the running time is not a polynomial with respect to input size.

Now, we use the second pseudopolynomial-time algorithm to design a FPTAS.

For any $\varepsilon > 0$, choose integer $h > 1/\varepsilon$. Denote $c'_k = \lfloor c_k n(h+1)/M \rfloor$ for $1 \leq k \leq n$ and consider a new instance of the knapsack problem as follows:

$$\begin{aligned} \max \quad & c'_1 x_1 + c'_2 x_2 + \cdots + c'_n x_n \\ \text{subject to} \quad & s_1 x_1 + s_2 x_2 + \cdots + s_n x_n \leq S \\ & x_1, x_2, \ldots, x_n \in \{0,1\}. \end{aligned}$$

Apply the second pseudopolynomial-time algorithm to this new problem. The running time will be $O(n^4 h \log(nhS))$, a polynomial-time with respect to n, h, and $\log S$. Suppose x^h is an optimal solution of this new problem. Set $c^h = c_1 x^* h 1 + \cdots + c_n x_n^h$. We claim that

$$\frac{c^*}{c^h} \leq 1 + \frac{1}{h},$$

that is, x^h is a $(1 + 1/h)$-approximation.

To show our claim, let $I^h = \{k \mid x_k^h = 1\}$ and $c^* = \sum_{k \in I^*} c_k$. Then, we

have

$$\begin{aligned} c^h &= \sum_{k\in I^h} \frac{c_k n(h+1)}{M} \cdot \frac{M}{n(h+1)} \\ &\geq \sum_{k\in I^h} \lfloor \frac{c_k n(h+1)}{M} \rfloor \cdot \frac{M}{n(h+1)} \\ &= \frac{M}{n(h+1)} \sum_{k\in I^h} c'_k \\ &\geq \frac{M}{n(h+1)} \sum_{k\in I^*} c'_k \\ &\geq \frac{M}{n(h+1)} \sum_{k\in I^*} (\frac{c_k n(h+1)}{M} - 1) \\ &\geq opt - \frac{M}{h+1} \\ &\geq opt(1 - \frac{1}{h+1}). \end{aligned}$$

Theorem 8.6.11. *The knapsack problem has FPTAS.*

For an application of this result, we study a scheduling problem.

Problem 8.6.12 (Scheduling $P\|C_{\max}$)**.** *Suppose there are m identical machines and n jobs $J_1, ..., J_n$. Each job J_i has a processing time a_i, which does not allow preemption, i.e., the processing cannot be cut. All jobs are available at the beginning. The problem is to find a scheduling to minimize the complete time, called* makespan*.*

Theorem 8.6.13. *The scheduling $P\|C_{\max}$ problem is NP-hard.*

Proof. For $m = 2$, this problem is equivalent to find a partition (N_1, N_2) for $[n]$ to minimize $\max(\sum_{i\in N_1} a_i, \sum_{i\in N_2} a_i)$. Thus, it is easy to reduce the partition problem to the decision version of this problem by requiring the makespan not exceed $\lfloor (\sum_{i=1}^n a_i)/2 \rfloor$. □

For $m = 2$, we can also obtain a FPTAS from the FPTAS of the knapsack problem.

To this end, we consider the following instance of the knapsack problem:

$$\begin{aligned} \max \quad & a_1x_1 + a_2x_2 + \cdots + a_nx_n \\ \text{subject to} \quad & a_1x_1 + a_2 + \cdots + a_nx_n \leq S/2 \\ & x_1, x_2, ..., x_n \in \{0, 1\} \end{aligned}$$

where $S = a_1 + a_2 + \cdots + a_n$. It is easy to see that if opt_k is the objective function value of an optimal solution for this knapsack problem, then $opt_s = S - opt_k$ is the objective function value of an optimal solution of above scheduling problem.

Applying the FPTAS to above instance of the knapsack problem, we may obtain a $(1+\varepsilon)$-approximation solution $\hat{x}$. Let $N_1 = \{i \mid \hat{x}_i = 1\}$ and $N_2 = \{i \mid \bar{x}_i = 0\}$. Then (N_1, N_2) is a partition of $[n]$ and moreover, we have

$$\max(\sum_{i\in N_1} a_i, \sum_{i\in N_2} a_i) = \sum_{i\in N_2} a_i = S - \sum_{i\in N_1} a_i$$

and

$$\frac{opt_k}{\sum_{i\in N_1} a_i} \leq 1+\varepsilon.$$

Therefore,

$$\frac{S - opt_s}{S - \sum_{i\in N_2} a_i} \leq 1+\varepsilon,$$

that is,

$$S - \sum_{i\in N_2} a_i \geq (S - opt_s)/(1+\varepsilon).$$

Thus,

$$\sum_{i\in N_2} a_i \leq \frac{\varepsilon S + opt_s}{1+\varepsilon} \leq \frac{\varepsilon \cdot 2opt_s + opt_s}{1+\varepsilon} \leq opt_s(1+\varepsilon).$$

Therefore, (N_1, N_2) is a $(1+\varepsilon)$-approximation solution for the scheduling problem.

8.7 Planar 3SAT

A CNF F is *planar* if graph $G*(F)$, defined as follows, is planar.

- The vertex set consists of all variables $x_1, x_2, ..., x_n$ and all clauses $C_1, C_2, ..., C_m$.
- The edge set $E(G^*(F)) = \{(x_i, C_j) \mid x_i \text{ appears in } C_j\}$.

A CNF F is *strongly planar* if graph $G(F)$, defined as follows, is planar.

- The vertex set consists of all literals $x_1, \bar{x}_1, x_2, \bar{x}_2, ..., x_n, \bar{x}_n$ and all clauses $C_1, C_2, ..., C_m$.

- The edge set $E(G^*(F)) = \{(x_i, \bar{x}_i) \mid i = 1, 2, ..., m\} \cup \{(x_i, C_j) \mid x_i \in C_j\} \cup \{(\bar{x}, C_j) \mid \bar{x}_i \in C_j\}$.

Corresponding two types of planar CNF, there are two problems.

Problem 8.7.1 (Planar 3SAT)**.** *Given a planar 3CNF F, determine whether F is satisfiable.*

Problem 8.7.2 (Strongly Planar 3SAT)**.** *Given a strongly planar 3CNF F, determine whether F is satisfiable.*

Theorem 8.7.3. *The planar 3STA problem is NP-complete.*

Proof. The problem is a special case of the 3SAT problem and hence longs to NP. We next construct a reduction to witness 3SAT $\leq_m^p$ Planar 3SAT. To do so, consider a 3CNF F and $G^*(F)$. $G^*(F)$ may contains many crosspoints. For each crosspoint, we use a crosser to remove it. As shown in Fig.8.18, this crosser is constructed with three $\oplus$ operations each defined by

$$x \oplus y = x\bar{y} + \bar{x}y.$$

We next show that for each $\oplus$ operation $x \oplus y = z$, there exists a planar

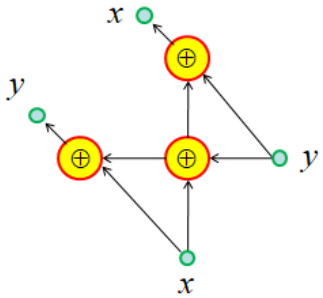


Figure 8.18: A crosser.

3CNF $F_{x\oplus y=z}$ such that

$$x \oplus y = z \Leftrightarrow F_{x\oplus y=z} \in SAT$$

that is, $F_{x\oplus y=z}$ is satisfiable.

Note that a CNF $c(x, y, z) = (x + y + \bar{z})(\bar{x} + z)(\bar{(}y) + z)$ is planar as shown in Fig.8.19. Moreover, we have

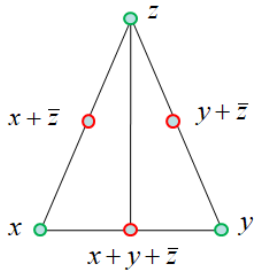


Figure 8.19: CNF $c(x, y, z)$ is planar.

$$\begin{aligned} x + y = z &\Leftrightarrow c(x, y, z) \in SAT, \\ x \cdot y = z &\Leftrightarrow c(\bar{x}, \bar{y}, \bar{z}) \in SAT. \end{aligned}$$

Since

$$x \oplus y = (x + y) \cdot \bar{y} + \bar{x} \cdot (x + y),$$

we have

$$x \oplus y = z \Leftrightarrow F'_{x \oplus y = z} = c(x, y, u)c(\bar{u}, y, \bar{v})c(x, \bar{u}, \bar{w})c(v, w, z) \in SAT.$$

As show in Fig.8.20, $F'_{x \oplus y = z}$ is planar. $F'_{x \oplus y = z}$ contains some clauses with

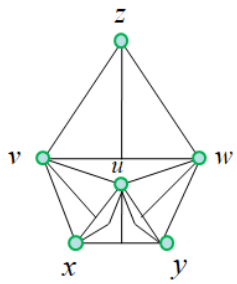


Figure 8.20: CNF $F'_{x \oplus y = z}$ is planar.

two literals. Each such clause $x + y$ can be replaced by two clauses $(x + y + w)(x + y + \bar{w})$ with a new variable w as shown in Fig.8.21. Then we can obtain a planar 3CNF $F_{x \oplus y = z}$ such that

$$x \oplus y = z \Longleftrightarrow F_{x \oplus y = z} \in SAT.$$

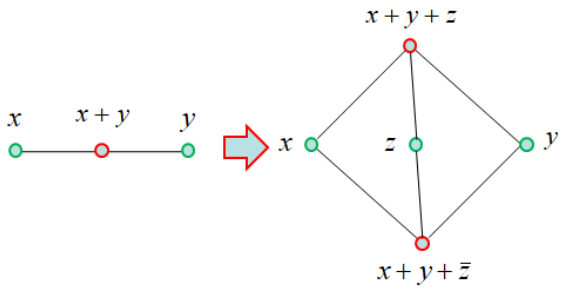


Figure 8.21: A clause of size 2 can be replaced by two clauses of size 3.

Finally, look back at the instance 3cNF F of the 3SAT problem at the beginning. Let F^* be the product of F and all 3CNFs for all $\oplus$ operations appearing in crossers used for removing crosspoints in $G^*(F)$. Then F^* is planar and

$$F \in SAT \Longleftrightarrow F^* \in SAT.$$

This completes our reduction from the 3SAT problem to the planar 3SAT problem. □

Theorem 8.7.4. *The strongly planar 3STA problem is NP-complete.*

Proof. Clearly, the strongly planar 3SAT problem belongs to NP. Next, we construct a polynomial-time many-one reduction from the planar 3SAT problem to the strongly plana 3SAT problem. Consider a planar 3CNF F. In $G^*(F)$, if we replace each vertex labeled with variable x by an edge with endpoints x and $\bar{x}$, then some crosspoints will be introduced.

To overcome this trouble, we replace the vertex with label x in $G^*(F)$ by a cycle $G(F_x)$ (Fig.8.22) where

$$F_x = (x + \bar{w}_1)(w_1 + \bar{w}_2)\cdots(w_k + \bar{x}),$$

and k is selected in the following way: For each edge (x, C_j), label it with x if C_j contains lateral X, and $\bar{x}$ if C_j contains literal $\bar{x}$. Select k to be the number of changes from edge x to $\bar{x}$ when travel around vertex x. Note that

$$(x + \bar{w}_1)(w_1 + \bar{w}_2)\cdots(w_k + \bar{x}) = 1 \Rightarrow x = w_1 = \cdots = w_k.$$

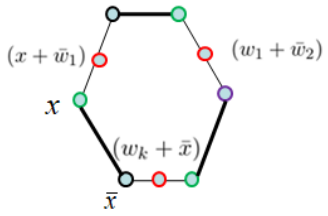


Figure 8.22: Cycle $G(F_x)$.

Now, each edge (x, C_j) in $G^*(F)$ is properly replaced by an edge (x, C_j) or (w_i, C_j) (Fig.8.23). We will obtain a planar $G(F')$ where $F' = F \cdot \prod_x F_x$ and F' is a strongly planar CNF. Note that F' contains some clauses of size 2. Finally, we can replace them by clauses of size 3 as shown in Fig.8.21. □

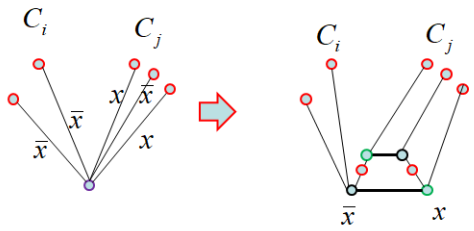


Figure 8.23: Each vertex x is replaced properly by a cycle $G(F_x)$.

As an application, let us consider a problem in planar graphs.

Problem 8.7.5 (Planar Vertex Cover)**.** *Given a planar graph G, find a minimum vertex cover of G.*

Theorem 8.7.6. *The planar vertex cover problem is NP-hard.*

Proof. We construct a polynomial-time many-one reduction from the strongly planar 3SAT problem to the planar vertex cover problem. Let F be a

strongly planar 3CNF with n variables $x_1, x_2, ..., x_n$ and m clauses $C_1, C_2, ..., C_m$. Consider $G(F)$. For each clause C_j, replace vertex C_j properly by a triangle

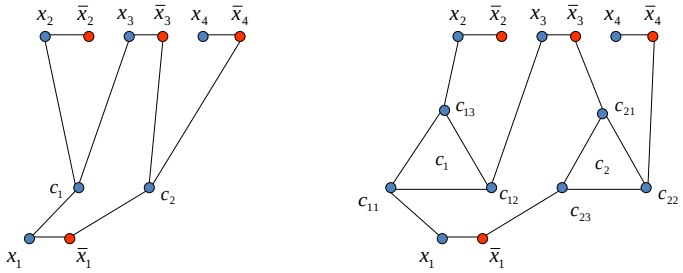


Figure 8.24: Each vertex C_j is replaced properly by a triangle $C_{j1}C_{j2}C_{j3}$.

$C_{j1}c_{j2}C_{j3}$ (Fig.8.24) and we will obtain a planar graph G' such that G' has a vertex cover of size at most $2m + n$ if and only if F is satisfiable. □

The planar vertex cover problem has PTAS. Actually, a lot of combinatorial optimization problem in planar graph and geometric plan or space have PTAS. We will discuss them in later chapters.

8.8 Complexity of Approximation

In previous sections, we studied several NP-hard combinatorial optimization problems. Based on their approximation solutions, they may be classified into following four classes.

1. PTAS, consisting of all combinatorial problems each of which has a PTAS, e.g., the knapsack problem and the planar vertex cover problem.

2. APX, consisting of all combinatorial optimization problems each of which has a polynomial-time $O(1)$-approximation, e.g., the vertex cover problem and the Hamiltonian cycle problem with triangular inequality.

3. Log-APX, consisting of all combinatorial optimization problems each of which has a polynomial-time $O(\ln n)$-approximation for minimization, or $(1/O(\ln n))$-approximation for maximization, e.g., the set cover problem.

4. Poly-APX, consisting of all combinatorial optimization problems each of which has a polynomial-time $p(n)$-approximation for minimization, or $(1/p(n))$-approximation for maximization for some polynomial $p(n)$, e.g., the maximum independent set problem and the longest path problem.

Clearly, PTAS $\subset$ APX $\subset$ Log-APX $\subset$ Poly-APX. Moreover, we have

Theorem 8.8.1. *If NP $\neq$ P, then PTAS $\neq$ APX $\neq$ Log-APX $\neq$ Poly-APX.*

Actually from previous sections we know that the set cover problem has $(1+\ln n)$-approximation and if NP $\neq$ P, then it has no polynomial-time $O(1)$-approximation. Hence, the set cover problem separates APX and Log-APX. The maximum independent set problem has a trivial $1/n$-approximation by taking a single vertex as solution and hence it belongs to Poly-APX. Moreover, if NP $\neq$ P, then it has no polynomial-time $n^{\varepsilon-1}$-approximation. Hence, the maximum independent set problem separates Log-APX from Poly-APX. Next, we study a problem which separates PTAS from APX.

Problem 8.8.2 (k-Center)**.** *Given a set C of n cities with a distance table, find a subset S of k cities as centers to minimize*

$$\max_{c\in C}\min_{s\in S} d(c,s)$$

where $d(c,s)$ is the distance between c and s.

Theorem 8.8.3. *The k-center problem with triangular inequality has a polynomial-time 2-approximation.*

Proof. Consider following algorithm.

Initially, choose arbitrarily a vertex $s_1 \in C$ and set
$S_1 \leftarrow \{s_1\}$;
for $i = 2$ **to** k **do**
select $s_i = \text{arcmax}_{c\in C} d(c, S_{i-1})$, and set
$S_i \leftarrow S_{i-1} \cup \{s_i\}$; **output** S_k.

We will show that this algorithm gives a 2-approximation.

Let S^* be an optimal solution. Denote

$$opt = \max_{c\in C} d(c, S^*).$$

Classify all cities into k clusters such that each cluster contains a center $s^* \in S^*$ and $d(c, s^*) \leq d^*$ for every city c in the cluster. Now, we consider two cases.

Case 1. Every cluster contain a member $s_i \in S_k$. Then for each city c in the cluster with center s^*, $d(c, s_i) \leq d(c, s^*) + d(s^*, s_i) \leq 2 \cdot opt$.

Case 2. There is a cluster containing two members $s_i, s_j \in S_k$ with $i < j$. Suppose the center of this cluster is s^*. Then for any $c \in C$,

$$\begin{aligned} d(c, S_k) &\leq d(c, S_{j-1}) \\ &\leq d(s_j, S_{j-1}) \\ &\leq d(s_j, s_i) \\ &\leq d(s_i, s^*) + d(s^*, s_j) \\ &\leq 2 \cdot opt. \end{aligned}$$

□

A corollary of Theorem 8.8.3 is that the k-center problem belongs to APX. Before we show that the k-center problem does not belong to PTAS unless NP=P, let us study another problem.

Problem 8.8.4 (Dominating Set)**.** *Given a graph G, find the minimum dominating set where a dominating set is a subset of vertices such that every vertex is either in the subset or adjacent to a vertex in the subset.*

Lemma 8.8.5. *The decision version of the dominating set problem is NP-complete.*

Proof. Consider an input graph $G = (V, E)$ of the vertex cover problem. For each edge (u, v), create a new vertex x_{uv} together with two edges (u, x_{uv}) and (x_{uv}, v) (Fig.8.25). Then we obtain a modified graph G'. If G has a

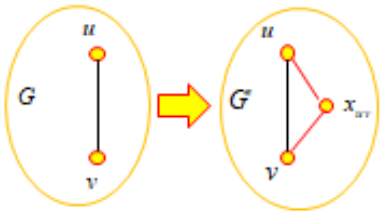


Figure 8.25: For each edge (u, v), add a new vertex x_{uv} and two edges (x_{uv}, u) and $x_{uv}, v)$.

vertex cover of size $\leq k$, then the same vertex subset must be a dominating set of G', also of size $\leq k$.

Conversely, if G' has a dominating set D of size $\leq k$, then without loss of generality, we may assume $D \subseteq E$. In fact, if $x_{uv} \in D$, then we can replace x_{uv} by either u or v, which results in a dominating set of the same size. Since $D \subseteq E$ dominating all x_{uv} in G', D covers all edges in G. □

Now, we come back to the k-center problem.

Theorem 8.8.6. *For any $\varepsilon > 0$, the k-center problem with triangular inequality does not have a polynomial-time $(2-\varepsilon)$-approximation unless NP=P.*

Proof. Suppose that the k-center problem has a polynomial-time $(2-\varepsilon)$-approximation algorithm A. We use algorithm A to construct a polynomial-time algorithm for the decision version of the dominating set problem.

Consider an instance of the decision version version of the dominating set problem, consisting of a graph $G = (V, E)$ and a positive integer k. Construct an instance of the k-center problem by choosing all vertices as cities with distance table defined as follows:

$$d(u,v) = \begin{cases} 1 & \text{if } (u,v) \in E, \\ |V|+1 & \text{otherwise.} \end{cases}$$

If G has a dominating set of size at most k, then the k-center problem will have an optimal solution with $opt = 1$. Therefore, algorithm A produces a solution with objective function value at most $(2-\varepsilon)$, actually has to be one. If G does not have a dominating set of size at most k, then the k-center problem will have its optimal solution with $opt \geq 2$. Hence, algorithm A produces a solution with objective function value at least two. Therefore, from objective function value of solution produced by algorithm A, we can determine whether G has a dominating set od size $\leq k$ or not. By Lemma 8.8.5, we have NP=P. □

By Theorems 8.8.3 and 8.8.6, the k-center problem with triangular inequality separates PTAS and APX.

Actually, APX is a large class which contains many problems not in PTAS if NP≠P. Those are called APX-complete problems. There are several reductions to establish the APX-completeness. Let us introduce a popular one, the polynomial-time L-reduction.

Consider two combinatorial optimization problems Π and Γ. Π is said to be polynomial-time L-reducible to Γ, written as $\Pi \leq_L^p \Gamma$, if there exist two

polynomial-time computable functions h and g, and two positive constants a and b such that

(L1) h maps from instances x of Π to instances $h(x)$ of Γ such that

$$opt_\Gamma(h(x)) \leq a \cdot opt_\Pi(x)$$

where $opt_\Pi(x)$ is the objective function value of an optimal solution for Π on instance x;

(L2) g maps from feasible solutions y of Γ on instance $h(x)$ to feasible solutions $g(y)$ of Π on instance x such that

$$|obj_\Pi(g(y)) - opt_\Pi(x)| \leq b \cdot |obj_\Gamma(y) - opt_\Gamma(h(x))|$$

where $obj_\Gamma(y)$ is the objective function value of feasible solution y for Γ (Fig.8.26).

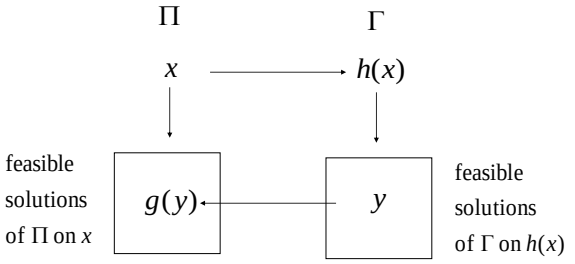


Figure 8.26: Definition of L-reduction.

This reduction has following properties.

Theorem 8.8.7. $\Pi \leq_L^p \Gamma, \Gamma \leq_L^p \Lambda \Leftarrow \Pi \leq_L^p \Lambda$.

Proof. As shown in Fig.8.27, we have

$$opt_\Lambda(h'(h(x))) \leq a' \cdot opt_\Gamma(h(x)) \leq a'a \cdot opt_\Pi(x)$$

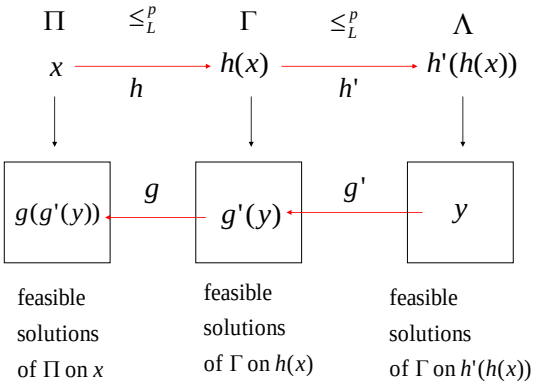


Figure 8.27: The proof of Theorem 8.8.7.

and

$$\begin{aligned} & |obj_\Pi(g(g'(y))) - opt_\Pi(x)| \\ \leq\ & b \cdot |obj_\Gamma(g'(y)) - opt_\Gamma(h(x))| \\ \leq\ & bb' \cdot |obj_\Lambda(y) - opt_\Lambda(h'(h(x)))|. \end{aligned}$$

□

Theorem 8.8.8. *If* $\Pi \leq_L^p \Gamma$ *and* $\Gamma \in PTAS$*, then* $\Pi \in PTAS$*.*

Proof. Consider four cases.

Case 1. Both Π and Γ are minimization problems.

$$\begin{aligned} \frac{obj_\Pi(g(y))}{opt_\Pi(x)} &= 1 + \frac{obj_\Pi(g(y)) - opt_\Pi(x)}{opt_\Pi(x)} \\ &\leq 1 + \frac{ab(obj_\Gamma(y) - opt_\Gamma(h(x)))}{opt_\Gamma(h(x))}. \end{aligned}$$

If y is a polynomial-time $(1+\varepsilon)$-approximation for Γ, then $g(y)$ is a polynomial-time $(1 + ab\varepsilon)$-approximation for Π.

Case 2. Π is a minimization and Γ is a maximization.

$$\begin{aligned}\frac{obj_\Pi(g(y))}{opt_\Pi(x)} &= 1+\frac{obj_\Pi(g(y))-opt_\Pi(x)}{opt_\Pi(x)}\\ &\le 1+\frac{ab(opt_\Gamma(h(x))-obj_\Gamma(y))}{opt_\Gamma(h(x))}\\ &\le 1+\frac{ab(opt_\Gamma(h(x))-obj_\Gamma(y))}{obj_\Gamma(y)}.\end{aligned}$$

If y is a polynomial-time $(1+\varepsilon)^{-1}$-approximation for Γ, then $g(y)$ is a polynomial-time $(1+ab\varepsilon)$-approximation for Π.

Case 3. Π is a maximization and Γ is a minimization.

$$\begin{aligned}\frac{obj_\Pi(g(y))}{opt_\Pi(x)} &= 1-\frac{opt_\Pi(x)-obj_\Pi(g(y))}{opt_\Pi(x)}\\ &\ge 1-\frac{ab(obj_\Gamma(y)-opt_\Gamma(h(x)))}{opt_\Gamma(h(x))}.\end{aligned}$$

If y is a polynomial-time $(1+\varepsilon)$-approximation for Γ, then $g(y)$ is a polynomial-time $(1-ab\varepsilon)$-approximation for Π.

Case 4. Both Π and Γ are maximization.

$$\begin{aligned}\frac{obj_\Pi(g(y))}{opt_\Pi(x)} &= 1-\frac{opt_\Pi(x)-obj_\Pi(g(y))}{opt_\Pi(x)}\\ &\ge 1-\frac{ab(opt_\Gamma(h(x))-obj_\Gamma(y))}{opt_\Gamma(h(x))}\\ &\ge 1-\frac{ab(opt_\Gamma(h(x))-obj_\Gamma(y))}{obj_\Gamma(y)}.\end{aligned}$$

If y is a polynomial-time $(1+\varepsilon)^{-1}$-approximation for Γ, then $g(y)$ is a polynomial-time $(1-ab\varepsilon)$-approximation for Π. □

Let us look at some examples for APX-complete problems.

Problem 8.8.9 (MAX3SAT-3)**.** *Given a 3CNF F that each variable appears in at most three clauses, find an assignment to maximize the number of satisfied clauses.*

Theorem 8.8.10. *The MAXT3SAT-3 problem is APX-complete.*

Let us use this APX-completeness as a root to derive others.

Problem 8.8.11 (MI-b)**.** *Given a graph G with vertex-degree upper-bounded by b, find a maximum independent set.*

Theorem 8.8.12. *The MI-4 problem is APX-complete.*

Proof. First, we show MI-4 $\in$ APX. Given a graph $G = (V, E)$, construct a maximal independent set by selecting vertices iteratedly and at each iteration, select an vertex and delete it together with its neighbors until no vertex is left. Clearly, this maximal independent set contains at least $|V|/5$ vertices. Therefore, it gives a polynomial-time 1/5-approximation.

Next, we show MAX3SAT $\leq_L^p$ MI-4. Consider an instance 3CNF F of MAX3SAT, with n variables $x_1, x_2, ..., x_n$ and m clauses $C_1, C_2, ..., C_m$. For each clause C_j, create a triangle with three vertices C_{j1}, C_{j2}, C_{j3} labeled by three literals of C_j, respectively. Then connect every vertex with label x_i to every vertex with label $\bar{x}_i$ as shown in Fig.8.28. This graph is denoted by

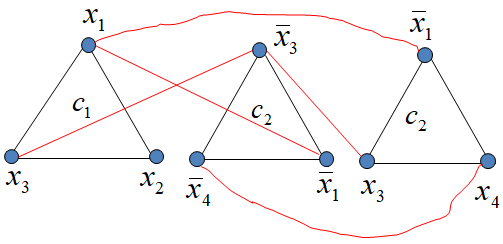


Figure 8.28: The proof of Theorem 8.8.12.

$h(F)$. For each independent set y of $h(F)$, we define an assignment $g(y)$ to make every vertex in the independent set with a true literal. Thus, F has at least $|y|$ clauses satisfied.

To show (L1), we claim that

$$opt_{MAX3SAT}(F) = opt_{MI-4}(h(F)).$$

Suppose x^* is an optimal assignment. Construct an independent set y^* by selecting one vertex with true label in each satisfied clause. Then

$$opt_{MAX3SAT}(F) = |y^*| \leq opt_{MI-4}(h(F)).$$

Conversely, suppose $y*$ is a maximum independent set of $h(F)$. We have F have at least $|y^*|$ satisfied clauses with assignment $g(y^*)$. Therefore,

$$opt_{MAX3SAT}F \geq |y^*| = opt_{MI-4}(h(F)).$$

To see (L2), we note that

$$|opt_{MAX3SAT}(F) - obj_{MAX3SAT}(g(y))| \leq |opt_{MI-4}(h(F)) - obj_{MI-4}(y)|$$

since $obj_{MAX3SAT}(g(y)) \geq obj_{MI-4}(y)$. □

Theorem 8.8.13. *The MI-3 problem is APX-complete.*

Proof. Since MI-4 ∈ APX, so is MI-3. We next show MI-4 $\leq_L^p$ MI-3. Consider a graph $G = (V, E)$ with vertex-degree at most 4. For each vertex u with degree 4, we put a path $(u_1, v_1, u_2, v_2, u_3, v_3, u_4)$ as shown in Fig.8.29. Denote obtained graph by $G' = h(G)$. Then (L1) holds since

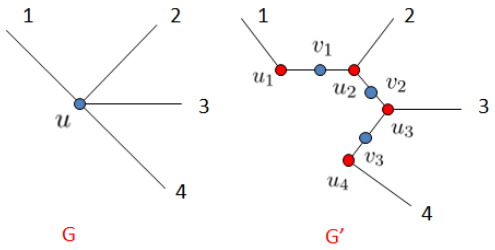


Figure 8.29: Vertex u is replaced by a path $(u_1, v_1, u_2, v_2, u_3, v_3, u_4)$.

$$opt_{MI_3}(G') \leq |V(G')| \leq 7|V(G)| \leq 28 \cdot opt_{MI-4}(G)$$

where $V(G)$ denotes the vertex set of graph G. To see (L2), note that the path $(u_1, v_1, u_2, v_2, u_3, v_3, u_4)$ has unique maximum independent set $I_u = \{u_1, u_2, u_3, u_4\}$. For any independent set I of G', define $g(I)$ to be obtained from I by replace set I_u by vertex u and removing all other vertices not in G. Then $g(I)$ is an independent set of G. We claim

$$opt_{MI-4}(G) - |g(I)| \leq opt_{MI-3}(G') - |I|.$$

To show it, define I' to be obtained from I by $\{v_1, v_2, v_3\}$ if I contains one of v_1, v_2, v_3. Clearly, I' is still an independent set of G' and $|I| \leq |I'|$ and $g(I) = g(I')$. Then, we have

$$\begin{aligned} opt_{M-4}(G) - g(I) &= opt_{MI-4}(G) - g(I') \\ &= opt_{MI-3}(G') - |I'| \\ &\leq opt_{MI-3}(G') - |I|. \end{aligned}$$

□

Problem 8.8.14 (VC-b)**.** *Given a graph G with vertex-degree upper-bounded by b, find the minimum vertex cover.*

Theorem 8.8.15. *The VC-3 problem is APX-complete.*

Proof. Since the vertex cover problem has a polynomial-time 2-approximation, so does the VC-3 problem. Hence, VC-3 $\in$ APX. Next, we show MI-3 $\leq_L^p$ VC-3.

For any graph $G = (V, E)$ with vertex-degree at most 3, define $h(G) = G$. To see (L1), note that $opt_{MI-3} \geq |V|/4$ and $|E| \leq 3|V|/2 = 1.5|V|$. Hence

$$opt_{VC-3}(G) = |V| - opt_{MI-3}(G) \leq |V| - |V|/4 = (3/4)|V| \leq 3 \cdot opt_{MI-3}(G).$$

Now, for any vertex cover C, define $g(C)$ to be the complement of C. Then we have

$$\begin{aligned} opt_{MI-3}(G) - |g(C)| &= |V| - opt_{VC-3}(G)) - (|V| - |C|) \\ &= |C| - opt_{VC-3}(G). \end{aligned}$$

Therefore, (L2) holds. □

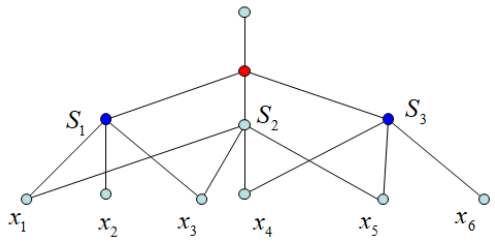
There are also many problems in Log-APX \ APX, such as various optimization problems on covering and dominating. The lower bound for their approximation performance is often established based on Theorem 8.5.7 with a little modification.

Theorem 8.8.16. *Theorem 8.5.7 still holds in special case that each input consisting of a collection $\mathcal{C}$ of subsets of a final set X with condition $|\mathcal{C}| \leq |X|$, that is, in this case, we still have that for any $0 < \rho < 1$, the set cover problem does not have a polynomial-time $(\rho \ln n)$-approximation unless NP=P where $n = |X|$.*

Here is an example.

Theorem 8.8.17. *For any $0 < \rho < 1$, the dominating set problem does not have a polynomial-time $(\rho \ln n)$-approximation unless NP=P where n is the number of vertices in input graph.*

Proof. Suppose there exists a polynomial-time $(\rho \ln n)$-approximation for the dominating set problem. Consider instance $(X, \mathcal{C})$ of the set cover problem with $|\mathcal{C}| \leq |X|$. Construct a bipartite graph $G = (\mathcal{C}, X, E)$. For $S \in \mathcal{C}$ and



$S_1 = \{x_1, x_2, x_3\}, S_2 = \{x_1, x_3, x_4, x_5\},$
$S_3 = \{x_4, x_5, x_6\}.$

Figure 8.30: The proof of Theorem 8.8.17.

$x \in X$, there exists an edge $(S, x) \in E$ if and only if $x \in S$. Add two new vertices o and o' together with edges (o, o') and (S, o) for all (S, o). Denote this new graph by G' as shown in Fig.8.30.

First, note that

$$opt_{ds}(G') \leq opt_{sc}(X, \mathcal{C}) + 1$$

where $opt_{ds}(G')$ denotes the size of minimum dominating set of G' and $opt_{sc}(G)$ denotes the cardinality of minimum set cover on input $(X, \mathcal{C})$. In fact, suppose that $\mathcal{C}^*$ is a minimum set cover on input $(X, |\mathcal{C})$. Then $\mathcal{C}^{\cup}\{o\}$ is a dominating set of G'.

Next, consider a dominating set D of G', generated by the polynomial-time $(\rho \ln n)$-approximation for the dominating set problem. Then, we have $|D| \leq (\rho \ln(2|X| + 2))opt_{ds}(G')$. Construct a set cover $\mathcal{S}$ as follows:

Step 1. If D does not contains o, then add o. If D contains a vertex with label $x \in X$, then replace x by a vertex with label $C \in \mathcal{C}$ such that $x \in C$. We will obtain a dominating set D' of G' with size $|D'| \leq |D|+1$ and without vertex labeled by element in X.

Step 2. Remove o and o' from D'. We will obtain a set cover $\mathcal{S}$ for X with size $|\mathcal{S}| \leq |D|$.

Note that

$$\begin{aligned} |\mathcal{S}| &\leq |D| \\ &\leq (\rho \ln(2|X|+2)) opt_{ds}(G') \\ &\leq (\rho \ln(2|X|+2))(1+opt_{sc}(X,\mathcal{C})) \\ &= \frac{\ln(2|X|+2)}{\ln|X|} \cdot (1+\frac{1}{opt_{sc}(X,\mathcal{C})}) \cdot (\rho \ln|X|) opt_{sc}(X,\mathcal{C}). \end{aligned}$$

Select two sufficiently large positive constants α and β such that

$$\rho' = \frac{\ln(2\alpha+2)}{\ln\alpha} \cdot (1+\frac{1}{\beta}) \cdot \rho < 1.$$

Then for $|X| \geq \alpha$ and $opt_{sc}(X,\mathcal{C}) \geq \beta$,

$$|\mathcal{S}| \leq (\rho' \ln|X|) \cdot opt_{sc}(X,\mathcal{C}).$$

For $|X| < \alpha$ or $opt_{sc}(X,\mathcal{C}) < \beta$, an exactly optimal solution can be computed in polynomial-time. Therefore, there exists a polynomial-time $(\rho' \ln n)$-approximation for the set cover problem and hence NP=P by Theorem 8.8.16. □

Class Poly-APX may also be further divided into several levels.

Polylog-APX, consisting of all combinatorial optimization problems each of which has a polynomial-time $O(\ln^i n)$-approximation for minimization, or $(1/O(\ln^i n))$-approximation for maximization for some $i \geq 1$.

Sublinear-APX, consisting of all combinatorial optimization problems each of which has a polynomial-time $O(n^a)$-approximation for minimization, or $(1/n^a)$-approximation for maximization for some $0 < a < 1$.

Linear-APX, consisting of all combinatorial optimization problems each of which has a polynomial-time $O(n)$-approximation for minimization, or $(1/n)$-approximation for maximization.

In the literature, we can find the group Steiner tree problem [71, 72] and the connected set cover problem [73] in Polylog-APX \ Log-APX unless some complexity class collapses; the directed Steiner tree problem [74] and the densest k subgraph problem [75] in Sublinear-APX \ Log-APX. However, there are quite a few problems in Linear-APX \ Sublinear-APX. Especially, we may meet such a problem in the real world. Next, we give an example in study of wireless sensors.

Consider a set of wireless sensors lying in a rectangle which is a piece of boundary area of region of interest. The region is below the rectangle and outside is above the rectangle. The monitoring area of each sensor is a unit disk, i.e., a disk with radius of unit length. A point is said to be covered by a sensor if it lies in the monitoring disk of the sensor. The set of sensors is called a *barrier cover* if they can cover a line (not necessarily straight) connecting two vertical edges (Fig.8.31) of the rectangle. The barrier cover is used for protecting any intruder coming from outside. Sensors are powered

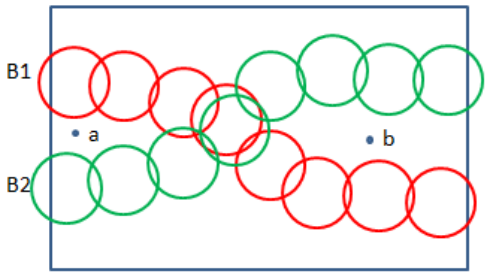


Figure 8.31: Sensor barrier covers.

with batteries and hence lifetime is limited. Assume that all sensors have unit lifetime. When several disjoint barrier covers are available, they are often working one by one, so that a security problem is raised.

In Fig.8.31, a point a lies behind barrier cover B_1 and in frond of barrier cover B_2. Suppose B_2 works first and after B_2 stops, B_1 starts to work. Then the intruder can go point a during the period that B_2 works. After B_2 stops, the intruder enters the area of interest without getting monitored by any sensor. Thus, scheduling (B_2, B_1) is not secure. The existence of point b in Fig.8.31 indicates that scheduling (B_1, B_2) is not secure, neither. Thus, in Fig.8.31, secure scheduling can contain only one barrier cover. In general, we have following problem.

Problem 8.8.18 (Secure Scheduling)**.** *Given n disjoint barrier covers $B_1, B_2, ..., B_n$, find a longest secure scheduling.*

Following gives a necessary and sufficient condition for a secure scheduling.

Lemma 8.8.19. *A scheduling $(B_1, B_2, .., B_k)$ is secure if and only if for any $1 \leq i \leq k-1$, there is not point a lying above B_i and below B_{i+1}.*

Proof. If such a point a exists, then the scheduling is not secure since the intruder can walk to point a during B_i works and enters into the area of interest during B_{i+1} works. Thus, the condition is necessary.

For sufficiency, suppose the scheduling is not secure. Consider the moment at which the intruder gets the possibility to enter the area of interest and the location a where the intruder lies. Let B_i works before this moment. Then a must lie above B_i and below B_{i+1}. □

This lemma indicates that the secure scheduling can be reduced to the longest path problem in directed graphs in the following way.

- Construct a directed graph G as follows. For each barrier cover B_i, create a node i. For two barrier covers B_i and B_j, if there exists a point a lying above barrier cover B_i and below barrier cover B_j, add an arc (i, j).
- Construct the complement $\bar{G}$ of graph G, that is, $\bar{G}$ and G have the same node set and ar arc in $\bar{G}$ if and only if it is not in G.

By Lemma 8.8.19, each secure scheduling of barrier covers corresponding a simple path in $\bar{G}$ and a secure scheduling is maximum if and only if corresponding simple path is the longest one. Actually, the longest path problem can also be reduced to the secure scheduling problem as shown in the proof of following theorem.

Theorem 8.8.20. *For any $\varepsilon > 0$, the secure scheduling problem has no polynomial-time $n^{1-\varepsilon}$-approximation unless $NP = P$.*

Proof. Let us reduce the longest path problem in directed graph to the secure scheduling problem. Consider a directed graph $G = (V, E)$. Let $\bar{G} = (V, \bar{E})$ be the complement of G, i.e., $\bar{E} = \{(i, j) \in V \times V \mid (i, j) \notin E\}$. Draw a horizontal line L and for each arc $(i, j) \in \bar{E}$, create a point (i, j) on the line L. All points (i, j) are apart from each other with distance 6 units (Fig.8.32). At each point (i, j), add a disk S_{ij} with center (i, j) and unit radius. Cut line L into a segment L' to include all disks between two endpoints. Add more unit disks with centers on the segment L' to cover the uncovered part of L' such that point (i, j) is covered only by S_{ij}. Let B_0 denote the set of sensors with constructed disks as their monitoring areas.

Now, let B_i be obtained from B_0 by following way:

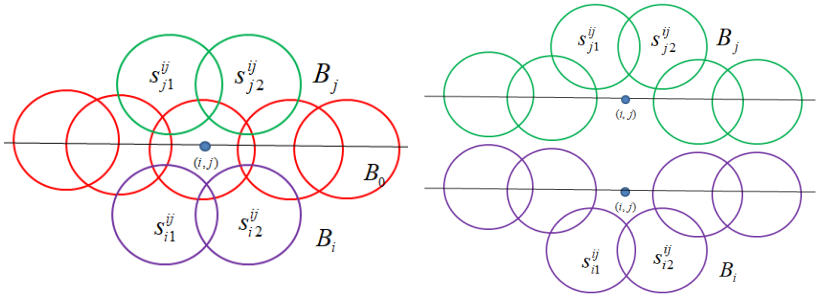


Figure 8.32: Sensor barrier covers.

- For any $(i,j) \in \bar{E}$, remove S_{ij} to break B_0 into two parts. Add two unit disks S_{i1}^{ij} and S_{i2}^{ij} to connect the two parts, such that point (i,j) lies above them.
- For any $(j,i) \in \bar{E}$, remove S_{ji} to break B_0 into two parts. Add two unit disks S_{i1}^{ij} and S_{i2}^{ij} to connect the two parts, such that point (i,j) lies below them.
- To make all constructed barrier covers disjoint, unremoved disks in B_0 will be made copies and put those copies into B_i (see Fig.8.32).

Clearly, G has a simple path $(i_1, i_2, ..., i_k)$ if and only if there exists a secure scheduling $(B_{i_1}, B_{i_2}, ..., B_{i_k})$. Therefore, our construction gives a reduction from the longest path problem to the secure scheduling problem. Hence, this theorem can be obtained from Theorem 8.3.9 for the longest path problem[2]. □

Exercises

1. For any language A, Kleene closure $A^* = A^0 \cup A^1 \cup A^2 \cup \cdots$. Solve following:
   (a) Design a deterministic Turing machine to accept language $\emptyset^*$.
   (b) Show that if $A \in$ P, then $A^* \in$ P.
   (c) Show that if $A \in$ NP, then $A^* \in$ NP.

[2]Theorem 8.3.9 states for undirected graphs. The same theorem also holds for directed graphs since the graph can be seen as special case of directed graphs.

2. Given a graph $G = (V, E)$ with edge weight $w : E \to R^+$, assign each vertex u with a weight x_u to satisfy $x_u + x_v \geq w(u, v)$ for every edge $(u, v) \in E$, and to minimize $\sum_{u \in V} x_u$. Find a polynomial-time solution and a faster 2-approximation.

3. Given a graph $G = (V, E)$ and a positive integer k, find a set C of k vertices to cover the maximum number of edges. Show following.

   (a) This problem has a polynomial-time (1/2)-approximation.

   (b) If this problem has a polynomial-time $1/\gamma$-approximation, then the minimum vertex cover problem has a polynomial-time γ-approximation.

4. Show that following problems are NP-hard:

   (a) Given a directed graph, find the minimum subset of edges such that every directed cycle contains at least one edge in the subset.

   (b) Given a directed graph, find the minimum subset of vertices such that every directed cycle contains at least one vertex in the subset.

5. Show NP-hardness of following problem: Given a graph G and an integer $k > 0$, determine whether G has a vertex cover C of size at most k, satisfying the following conditions:

   (a) The subgraph $G|_C$ induced by C has no isolated point.

   (b) Every vertex in C is adjacent to a vertex not in C.

6. Show that all internal nodes of a depth-first search tree form a vertex cover, which is 2-approximation for the minimum vertex cover problem.

7. Given a directed graph, find an acyclic subgraph containing maximum number of arcs. Design a polynomial-time 1/2-approximation for this problem

8. A wheel is a cycle with a center (not on the cycle) which is connected every vertex on the cycle. Prove the NP-completeness of following problem: Given a graph G, does G have a spanning wheel?

9. Given a 2-connected graph G and a vertex subset A, find the minimum vertex subset B such that $A \cup B$ induces a 2-connected subgraph. Show that this problem is NP-hard.

10. Show that following problems are NP-hard:

    (a) Given a graph G, find a spanning tree with minimum number of leaves.

    (b) Given a graph G, find a spanning tree with maximum number of leaves.

11. Given two graphs G_1 and G_2, show following:

    (a) It is NP-complete to determine whether G_1 is isomorphic to a subgraph of G_2 or not.

    (b) It is NP-hard to find a subgraphs H_1 of G_1 and a subgraph H_2 of G_1 such that H_1 is isomorphic to H_2 and $|E(H_1)| = |E(H_2)|$ reaches the maximum common.

12. Given a collection $\mathcal{C}$ of subsets of three elements in a finite set X, show following:

    (a) It is NP-complete to determine whether there exists a set cover consisting of disjoint subsets in $\mathcal{C}$.

    (b) It is NP-hard to find a minimum set cover, consisting of subsets in $\mathcal{C}$.

13. Given a graph, find the maximum number of vertex-disjoint paths with length two. Show following.

    (a) This problem is NP-hard.

    (b) This problem has a polynomial-time 2-approximation.

14. Design a polynomial-time 2-approximation for following problem: Given a graph, find a maximal matching with minimum cardinality.

15. (Maximum 3DM) Given 3 disjoint sets X, Y, Y with $|X| = |Y| = |Z|$ and a collection $\mathcal{C}$ of 3-sets, each 3-set consisting of exactly one element in X, one element in Y, and one element in Z, find the maximum number of disjoint 3-sets in $\mathcal{C}$. Show following.

    (a) This problem is NP-hard.

    (b) This problem has polynomial-time 3-approximation.

    (c) This problem is APX-complete.

16. There are n students who studied at a late-night for an exam. The time has come to order pizzas. Each student has his own list of required toppings (e.g., pepperoni, sausage, mushroom, etc). Everyone wants to eat at least one third of a pizza, and the topping of the pizza must be in his required list. To save money, every pizza may have only one topping. Find the minimum number of pizzas to order in order to make everybody happy. Please answer following questions.

    (a) Is it an NP-hard problem?

    (b) Does it belong to APX?

    (c) If everyone wants to eat at least a half of a pizza, is there a change about the answer for above questions?

17. Show that following is an NP-hard problem: Given two collections $\mathcal{C}$ and $\mathcal{D}$ of subsets of X and an positive integer d, find a subset A with at most d elements of X to minimize the total number of subsets in $\mathcal{C}$ not hit by A and subsets in $\mathcal{D}$ hit by A, i.e., to minimize

$$|\{S \in \mathcal{C} \mid S \cap A = \emptyset\} \cup \{Sin\mathcal{D} \mid S \cap A \neq \emptyset\}|.$$

18. Design a FPTAS for following problem: Consider n jobs and m identical machine. Assume that m is a constant. Each job j has a processing time p_j and a weight w_j. The processing does not allow preemption. The problem is to find a scheduling to minimize $\sum_j w_j C_j$ where C_j is the completion time of job j.

19. Design a FPTAS for following problem: Consider a directed graph with a source node s and a sink node t. Each edge e has an associated cost $c(e)$ and length $\ell(e)$. Given a length bound L, find a minimum-cost path from s to t of total length at most L.

20. Show the NP-completeness of following problem: Given n positive integers $a_1, a_2, ..., a_n$, is there a partition (I_1, I_2) of $[n]$ such that $|\sum_{i \in I_1} a_i - \sum_{i \in I_2} a_i| \leq 2$.

21. (Ron Graham's Approximation for Scheduling $P||C_{\max}$) Show that following algorithm gives a 2-approximation for the scheduling $P||C_{\max}$ problem:

    - List all jobs. Process them according to the ordering in the list.
    - Whenever a machine is available, move the first job from the list to the machine until the list becomes empty.

22. In proof of Theorem 8.7.4, if let k be the degree of vertex x, then the proof can also work. Please complete the construction of replacing vertex x by cycle $G(F_x)$.

23. (1-in-3SAT) Given a 3CNF F, is there an assignment such that for each clause of F, exactly one literal gets value 1? This is called the 1-in-3SAT problem. Show following.

    (a) The 1-in-3SAT problem is NP-complete.

    (b) The planar 1-in-3SAT problem is NP-complete.

    (c) The strongly planar 1-in-3SAT is NP-complete.

24. (NAE3SAT) Given a 3CNF F, determine whether there exists an assignment such that for each clause of F, is there an assignment such that for each clause of F, not all 3 literals are equal? This is called the NAE3SAT problem. Show following.

    (a) The NAE3SAT problem is NP-complete.

    (b) The planar NAE3SAT is in P.

25. (Planar 3SAT with Variable Cycle) Given a 3CNF F which has $G^*(F)$ with property that all variables can be connected into a cycle without crossing, is F satisfiable.

    (a) Show that this problem is NP-complete.

    (b) Show that the planar Hamiltonian cycle problem is NP-hard.

26. Show that the planar dominating set problem is NP-hard.

27. Show that following are APX-complete problems:

    (a) (Maximum 1-in-3SAT) Given a 3CNF F, find an assignment to maximize the number of 1-in-3 clauses, i.e., exactly one literal equal to 1.

    (b) (Maximum NAE3SAT) Given a 3CNF F, find an assignment to maximize the number of NAE clauses, i.e., either one or two literals equal to 1.

28. (Network Steiner Tree) Given a network $G = (V, E)$ with nonnegative edge weight, and a subset of nodes, P, find a tree interconnecting all nodes in P, with minimum total edge weight. Show that this problem is APX-complete.

29. (Rectilinear Steiner Arborescence) Consider a rectilinear plan with origin O. Given a finite set of terminals in the first of this plan, find the shortest arborescence to connect all terminals, that is, the shortest directed tree rooted at origin O such that for each terminal t, there is path from O to t and the path is allowed to go only to the right or upward. Show that this problem is NP-hard.

30. (Connected Vertex Cover) Given a graph $G = (V, E)$, find a minimum vertex cover which induces a connected subgraph. Show that this problem has a polynomial-time 3-approximation.

31. (Weighed Connected Vertex Cover) Given a graph $G = (V, E)$ with nonnegative vertex weight, find a minimum total weight vertex cover which induces a connected subgraph. Show following.

    (a) This problem has a polynomial-time $O(\ln n)$-approximation where $n = |V|$.

    (b) For any $0 < \rho < 1$, this problem has no polynomial-time $(\rho \ln n)$-approximation unless NP=P.

32. (Connected Dominating Set) In a graph G, a subset C is called a *connected dominating set* if C is a dominating set and induces a connected subgraph. Given a graph, find a minimum connected dominating set. Show that for any $0 < \rho < 1$, this problem has no polynomial-time $(\rho \ln n)$-approximation unless NP=P where n is the number of vertices in input graph.

33. Show that following problem is APX-complete: Given a graph with vertex-degree upper-bounded by a constant b, find a clique of the maximum size.

34. Show that the traveling salesman problem does not belong to Poly-APX if the distance table is not required to satisfy the triangular inequality.

Historical Notes

Bibliography

[1] Cook, William J.; Cunningham, William H.; Pulleyblank, William R.; Schrijver, Alexander (1997). Combinatorial Optimization. Wiley.

[2] Lawler, Eugene (2001). Combinatorial Optimization: Networks and Matroids.

[3] Lee, Jon (2004). A First Course in Combinatorial Optimization. Cambridge University Press.

[4] Papadimitriou, Christos H.; Steiglitz, Kenneth (July 1998). Combinatorial Optimization : Algorithms and Complexity. Dover.

[5] Schrijver, Alexander (2003). Combinatorial Optimization: Polyhedra and Efficiency. Algorithms and Combinatorics. 24. Springer.

[6] Gerard Sierksma; Yori Zwols (2015). Linear and Integer Optimization: Theory and Practice. CRC Press.

[7] Sierksma, Gerard; Ghosh, Diptesh (2010). Networks in Action; Text and Computer Exercises in Network Optimization. Springer.

[8] Pintea, C-M. (2014). Advances in Bio-inspired Computing for Combinatorial Optimization Problem. Intelligent Systems Reference Library. Springer.

[9] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009). Introduction to Algorithms, Third Edition (3rd ed.).

[10] Anany V. Levitin, Introduction to the Design and Analysis of Algorithms (Addison Wesley, 2002).

[11] Vazirani, Vijay V. (2003). Approximation Algorithms. Berlin: Springer.

[12] Williamson, David P.; Shmoys, David B. (April 26, 2011), The Design of Approximation Algorithms, Cambridge University Press.

[13] Wen Xu, Weili Wu: Optimal Social Influence, Springer, 2020.

[14] Weili Wu, Zha Zhang Wonjun Lee, Ding-Zhu Du: Optimal Coverage in Wireless Sensor Networks, Springer, 2020.

[15] Ding-Zhu Du, Panos M. Pardalos, Weili Wu: Mathematical Theory of Optimization, Springer, 2010.

[16] Ding-Zhu Du, Ker-I Ko, Xiaodong Hu, *Desin and Analysis of Approximation Algorithms*, (Springer 2012).

[17] Heideman, M. T., D. H. Johnson, and C. S. Burrus, "Gauss and the history of the fast Fourier transform", IEEE ASSP Magazine, 1, (4), 14C21 (1984).

[18] Donald E. Knuth, The Art of Computer Programming: Volume 3, Sorting and Searching, second edition (Addison-Wesley, 1998).

[19] "Sir Antony Hoare". Computer History Museum. Archived from the original on 3 April 2015. Retrieved 22 April 2015.

[20] Hoare, C. A. R. (1961). "Algorithm 64: Quicksort". Comm. ACM. 4 (7): 321

[21] Seward, H. H. (1954), "2.4.6 Internal Sorting by Floating Digital Sort", Information sorting in the application of electronic digital computers to business operations (PDF), Master's thesis, Report R-232, Massachusetts Institute of Technology, Digital Computer Laboratory, pp. 25C28.

[22] Stuart Dreyfus: Richard Bellman on the birth of dynamic programming, Operations Research 50 (1) (2002) 48-51.

[23] F.F. Yao, Efficient dynamic programming using quadrangle inequalities, in: Proc. 12th Ann. ACM Symp. on Theory of Computing (1980) 429-435.

[24] Al Borchers, Prosenjit Gupta: Extending the Quadrangle Inequality to Speed-Up Dynamic Programming. Inf. Process. Lett. 49(6): 287-290 (1994)

[25] S K. Rao, P. Sadayappan, F.K. Hwang and P.W. Shor: The Rectilinear Steiner Arborescence Problem, Algorithmica, 7 (2-3) (1992) 277-288.

[26] R. Bellman, On a routing problem, Quarterly of Applied Mathematics 16 (1958) 87C90.

[27] E.W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik, 1 (1959) 269C271.

[28] M.L. Fredman, R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, Journal of the Association for Computing Machinery 34 (1987) 596C615

[29] A. Schrijver: On the history of the shortest path problem, Documenta Math, Extra Volume ISMP (2012) 155-167.

[30] Chr. Wiener, Ueber eine Aufgabe aus der Geometria situs, Mathematik Annalen 6 (1873) 29C30.

[31] Dial, Robert B. (1969), "Algorithm 360: Shortest-Path Forest with Topological Ordering [H]", Communications of the ACM, 12 (11): 632C633.

[32] Shimbel, Alfonso (1953). "Structural parameters of communication networks". Bulletin of Mathematical Biophysics. 15 (4): 501C507.

[33] Floyd, Robert W. (June 1962). "Algorithm 97: Shortest Path". Communications of the ACM. 5 (6): 345.

[34] Warshall, Stephen (January 1962). "A theorem on Boolean matrices". Journal of the ACM. 9 (1): 11C12.

[35] Graham, R. L.; Hell, Pavol (1985), "On the history of the minimum spanning tree problem", Annals of the History of Computing, 7 (1): 43C57

[36] Chazelle, Bernard (2000), "A minimum spanning tree algorithm with inverse-Ackermann type complexity", Journal of the Association for Computing Machinery, 47 (6): 1028C1047

[37] Chazelle, Bernard (2000), "The soft heap: an approximate priority queue with optimal error rate" (PDF), Journal of the Association for Computing Machinery, 47 (6): 1012C1027

[38] Otakar Boruvka on Minimum Spanning Tree Problem (translation of both 1926 papers, comments, history) (2000) Jaroslav Nesetril, Eva Milková, Helena Nesetrilová. (Section 7 gives his algorithm, which looks like a cross between Prim's and Kruskal's.)

[39] T.E. Harris, F.S. Ross: Fundamentals of a Method for Evaluating Rail Net Capacities, *Research Memorandum* 1955.

[40] A. Schrijver: On the history of the transportation and maximum flow problems, *Mathematical Programming*, 91 (3) (2002): 437C445.

[41] L.R. Ford, D.R. Fulkerson: Maximal flow through a network *Canadian Journal of Mathematics* 8: 399C404 (1956).

[42] J. Edmonds, R. Karp: Theoretical improvements in algorithmic efficiency for network flow problems, *Journal of the ACM* 19 (2): 248C264 (1072).

[43] E.A. Dinic: Algorithm for solution of a problem of maximum flow in a network with power estimation". *Soviet Mathematics - Doklady*, 11: 1277C1280 (1970).

[44] Yefim Dinitz: Dinitz' Algorithm: The Original Version and Even's Version, in Oded Goldreich, Arnold L. Rosenberg, Alan L. Selman (eds.), *Theoretical Computer Science: Essays in Memory of Shimon Even*. (Springer, 2006): pp. 218C240.

[45] A.V. Goldberg, R.E. Tarjan: A new approach to the maximum-flow problem, *Journal of the ACM* 35 (4): 921 (1988).

[46] A.V. Goldberg, S. Rao: Beyond the flow decomposition barrier, *Journal of the ACM*, 45 (5): 783 (1998).

[47] J. Sherman: Nearly Maximum Flows in Nearly Linear Time, *Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS'2013)*, pp. 263C269.

[48] J.A. Kelner, Y.T. Lee, L. Orecchia, A. Sidford: An Almost-Linear-Time Algorithm for Approximate Max Flow in Undirected Graphs, and its Multicommodity Generalizations, *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'2014)*, pp. 217.

[49] J.B. Orlin: Max flows in O(nm) time, or better, *Proceedings of the 45th annual ACM symposium on Symposium on theory of computing (STOC '13)*, pp. 765C774.

[50] M.K. Kwan: Graphic Programming Using Odd or Even Points, *Chinese Math.* 1: 273-277 (1962).

[51] J. Edmonds, E.L. Johnson: Matching, Euler Tours, and the Chinese Postman, *Math. Programm.* 5 : 88-124 (1973).

[52] J. Edmonds: Paths, Trees and Flowers, *Canadian Journal of Mathematics*, 17: 449C467 (1965).

[53] J.E. Hopcroft, R.M. Karp: An n5/2 algorithm for maximum matchings in bipartite graphs, *SIAM Journal on Computing*, 2 (4): 225C231 (1973).

[54] S. Micali, V.V. Vazirani: An $O(\sqrt{|V|}\cdot|E|)$ algorithm for finding maximum matching in general graphs, *Proc. 21st IEEE Symp. Foundations of Computer Science*, pp. 17C27 (1980).

[55] Andrew V. Goldberg, Robert E. Tarjan: Finding minimum-cost circulations by canceling negative cycles, *Journal of the ACM* 36 (4): 873C886 (1989).

[56] D.R. Fulkerson: An out-of-kilter method for minimal cost flow problems, *Journal of the Society for Industrial and Applied Mathematics* 9(1):18-27 (1961).

[57] Morton Klein: A primal method for minimal cost flows with applications to the assignment and transportation problems, *Management Science* 14 (3): 205C220 (1967).

[58] M.A. Engquist: A successive shortest path algorithm for the assignment problem. Research Report, Center for Cybernetic Studies (CCS) 375, University of Texas, Austin; 1980.

[59] James B. Orlin: A polynomial time primal network simplex algorithm for minimum cost flows, *Mathematical Programming* 78 (2): 109C129 (1997).

[60] R.G. Busacker, P.G. Gowen: A procedure for determining a family of minimum cost network flow patterns, Operations Research Office Technical Report 15, John Hopkins University, Baltimore; 1961.

[61] V. Klee, G. Minty: How Good Is the Simplex Algorithm? In *Inequalities*, (Academic Press, New York, 1972).

[62] A. Charnes: Optimality and degeneracy in linear programming, *Economics* 2: 160-170 (1952).

[63] G.B. Dantzig: A. Orden, P. Wolfe: Note on linear programming, *Pacific J. Math* 5: 183-195 (1955).

[64] Robert G. Bland: New finite pivoting rules for the simplex method, *Mathematics of Operations Research* 2 (2): 103C107 (1977).

[65] E.M. Beale: Cycling in dual simplex algorithm, *Navel Research Logistics Quarterly* 2: 269-276 (1955).

[66] L.G. Karchiyan: A polynomial algorithm for linear progranning, *Doklady Akad. Nauk. USSR. Sci.* 244: 1093-1096 (1979).

[67] N. Karmakkar: A new polynomial-time algorithm for linear programming, *Proceedings of the 16th Annual ACM Symposium on the Theory of Computing*, 302-311, 1984.

[68] Jianzhong Zhang, Shaoji Xu: *Linear Programming*, Schiece Press, 1987.

[69] L.V. Kantorovich: A new method of solving some classes of extremal problems, *Doklady Akad Sci SSSR* 28: 211C214 (1940).

[70] G.B Dantzig: Maximization of a linear function of variables subject to linear inequalities, 1947. Published pp. 339C347 in T.C. Koopmans (ed.):Activity Analysis of Production and Allocation, New York-London 1951 (Wiley & Chapman-Hall).

[71] N. Garg, G. Konjevod, R. Ravi, A polylogarithmic approximation algorithm for the group Steiner tree problem, SODA 2000.

[72] Halperin, E. and Krauthgamer, R. [2003], Polylogarithmic inapproximability, Proceedings, 35th ACM Symposium on Theory of Computing, pp. 585C594.

[73] Wei Zhang, Weili Wu, Wonjun Lee, Ding-Zhu Du: Complexity and approximation of the connected set-cover problem. J. Glob. Optim. 53(3): 563-572 (2012)

[74] M. Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed Steiner problems. Journal of Algorithms, 33:73C91, 1999.

[75] Bhaskara, Aditya; Charikar, Moses; Chlamtac, Eden; Feige, Uriel; Vijayaraghavan, Aravindan (2010), "Detecting high log-densitiesan O(n1/4) approximation for densest k-subgraph", S-TOC'10Proceedings of the 2010 ACM International Symposium on Theory of Computing, ACM, New York, pp. 201C210

[76] Lidong Wu, Hongwei Du, Weili Wu, Deying Li, Jing Lv, Wonjun Lee: Approximations for Minimum Connected Sensor Cover. INFOCOM 2013: 1187-1194

[77] Zhao Zhang, Xiaofeng Gao, Weili Wu: Algorithms for connected set cover problem and fault-tolerant connected set cover problem. Theor. Comput. Sci. 410(8-10): 812-817 (2009)

[78] Lidong Wu, Huijuan Wang, Weili Wu: Connected Set-Cover and Group Steiner Tree. Encyclopedia of Algorithms 2016: 430-432

[79] Zhao Zhang, Weili Wu, Jing Yuan, Ding-Zhu Du: Breach-Free Sleep-Wakeup Scheduling for Barrier Coverage With Heterogeneous Wireless Sensors. IEEE/ACM Trans. Netw. 26(5): 2404-2413 (2018)

[80] Ambühl, C. [2005], An optimal bound for the MST algorithm to compute energy efficient broadcast trees in wireless networkds, Proceedings, 32nd International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 3580, Springer, pp. 1139C1150.

[81] P.K. Agarwal, M. van Kreveld, S. Suri, Label placement by maximum independent set in rectangles, *Comput. Geom. Theory Appl.*, 11 (118) 209-218.

[82] C. Ambühl, T. Erlebach, M. Mihalák and M. Nunkesser, Constant-approximation for minimum-weight (connected) dominating sets in unit disk graphs, *Proceedings of the 9th International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX 2006)*, LNCS 4110, Springer, 2006, pp. 3-14.

[83] B.S. Baker, Approximation algorithms for NP-complete problems on planar graphs, *Proc. FOCS*, 1983, pp. 265-273.

[84] B.S. Baker, Approximation algorithms for NP-complete problems on planar graphs, *J. ACM* 41(1) (1994) 153-180.

[85] P. Berman, B. Basgupta, S. Muthukrishnan, S. Ramaswami, Efficient approximation algorithms for tiling and packing problem with rectangles, *J. Algorithms*, 41 (2001) 178-189.

[86] T.M. Chan, Polynomial-time approximation schemes for picking and piercing fat objects, *J. Algorithms*, 46 (2003) 178-189.

[87] T.M. Chan, A note on maximum independent sets in rectangle intersection graphs, *Information Processing Letters*, 89 (2004) 19-23.

[88] Xiuzhen Cheng, Xiao Huang, Deying Li, Weili Wu, Ding-Zhu Du: A polynomial-time approximation scheme for the minimum-connected dominating set in ad hoc wireless networks. Networks 42(4): 202-208 (2003)

[89] D. Dai and C. Yu, A 5-Approximation Algorithm for Minimum Weighted Dominating Set in Unit Disk Graph, to appear in *Theoretical Computer Science*.

[90] T. Erlebach, K. Jansen, and E. Seidel, Polynomial-time approximation schemes for geometric graphs, *Proc. of 12th SODA*, 2001, 671-679.

[91] X. Gao, Y. Huang, Z. Zhang and W. Wu, $(6 + \varepsilon)$-approximation for minimum weight

dominating set in unit disk graphs, COCOON'08, pp. 551-557.

[92] D.S. Hochbaum and W. Maass, Approximation schemes for covering and packing problems in image processing and VLSI, *J.ACM* 32 (1985) 130-136.

[93] H.B. Hunt III, M.V. Marathe, V. Radhakrishnan, S.S. Ravi, D.J. Rosenkrantz, and R.E. Stearns, Efficient approximations and approximation schemes for geometric problems, *Journal of Algorithms*, 26(2) (1998) 238-274.

[94] T. Jiang and L. Wang, An approximation scheme for some Steiner tree problems in the plane, *Lecture Notes in Computer Science*, Vol 834 (1994) 414-427.

[95] T. Jiang, E.B. Lawler, and L. Wang, Aligning sequences via an evolutionary tree: complexity and algorithms, *Proc. 26th STOC*, 1994, .

[96] D.S. Johnson and M. Garey, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, New York, 1979).

[97] R.M. Karp, Probabilistic analysis of partitioning algorithms for the traveling salesman problem in the plane, *Mathematics of Operations Research* 2 (1977).

[98] S. Khanna, S. Muthukrishnan, and M. Paterson, On approximating rectangle tiling and packing, *Proc. 9th ACM-SIAM Symp. on Discrete Algorithms*, 1998, pp. 384-393.

[99] J. Komolos and M.T. Shing, Probabilistic partitioning algorithms for the rectilinear Steiner tree problem, *Networks* 15 (1985) 413-423.

[100] M. Min, S.C.-H. Huang, J. Liu, E. Shragowitz, W. Wu, Y. Zhao, and Y. Zhao, An approximation scheme for the rectilinear Steiner minimum tree in presence of obstructions, *Novel Approaches to Hard Discrete Optimization*, Fields Institute Communications Series, American Math. Society, vol 37(2003), pp. 155-163.

[101] F. Nielsen, Fast stabbing of boxes in high dimensions,

Theoret. Comput. Sci., 246 (2000) 53-72.

[102] S.A. Vavasis, Automatic domain partitioning in tree dimensions, *SIAM J. Sci. Stat. Comput.* 12(4) (1991) 950-970.

[103] D. Sankoff, Minimal mutation trees of sequences, *SIAM J. Appl. Math.*, 28 (1975) 35-42.

[104] L. Wang, T. Jiang, and E.L. Lawler, Approximation algorithms for tree alignment with a given phylogeny, *Algorithmica* 16 (1996) 302-315.

[105] L. Wang and T. Jiang, An approximation scheme for some Steiner tree problems in the plane, *Networks* 28 (1996) 187–193.

[106] L. Wang, T. Jiang, and D. Gusfield, A more efficient approximation scheme for tree alignment, *Proceedings of the first annual international conference on computational biology*, 1997, 310-319.

[107] Zhao Zhang, Xiaofeng Gao, Weili Wu, Ding-Zhu Du: A PTAS for minimum connected dominating set in 3-dimensional Wireless sensor networks. J. Glob. Optim. 45(3): 451-458 (2009)

[108] Feng Zou, Xianyue Li, Donghyun Kim and Weil Wu, Two Constant Approximation Algorithms for Node-Weighted Steiner Tree in Unit Disk Graphs", COCOA2008, St. John's, Newfoundland, Canada, August 21-24.

[109] Feng Zou, Xiayue Li, Sugang Gao, Yuexian Wang, Weili Wu, $(6+\varepsilon)$-approximation for weighted connected dominating set in unit disk graphs, submitted to *Theoretical Computer Science*.

[110] Hongwei Du, Qiang Ye, Jiaofei Zhong, Yuexuan Wang, Wonjun Lee, Haesun Park: Polynomial-time approximation scheme for minimum connected dominating set under routing cost constraint in wireless sensor networks. Theor. Comput. Sci. 447: 38-43 (2012).

[111] E.M. Arkin, J.S.B. Mitchell and G. Narasimhan, Resource-constructed geometric network optimization, *Proc. of 14th Annual Symposium on Computational Geometry*, Minneapolis, 1998, pp.307-316.

[112] Arora, S. [1996], Polynomial-time approximation schemes for Euclidean TSP and other geometric problems, *Proc. 37th IEEE Symp. on Foundations of Computer Science*, pp. 2-12.

[113] Arora, S. [1997], Nearly linear time approximation schemes for Euclidean TSP and other geometric problems, I*Proc. 38th IEEE Symp. on Foundations of Computer Science*, pp. 554-563.

[114] Arora, S. [1998], Polynomial-time approximation schemes for Euclidean TSP and other geometric problems, *Journal of the ACM* 45 (1998) 753-782.

[115] Arora, S., Raghavan, P. and Rao, S. [1998], Polynomial Time Approximation Schemes for Euclidean k-medians and related problems, ACM STOC.

[116] Arora, S., Grigni, M., Karger, D., Klein, P. and Woloszyn, A. [1998], Polynomial time approximation scheme for Weighted Planar Graph TSP, roc. SIAM SODA.

[117] Cheng, X., Kim, J.-M. and Lu, B. [2001], A polynomial time approximation scheme for the problem of interconnecting highways, *Journal of Combinatorial Optimization* 5: 327-343.

[118] Cheng, X., DasGupta, B. and Lu, B. [2000], A polynomial time approximation scheme for the symmetric rectilinear Steiner arborescence problem, to appear in *Journal of Global Optimization.*

[119] Du, D.Z., Hwang, F.K., Shing, M.T. and Witbold, T. [1988], Optimal routing trees, *IEEE Transactions on Circuits* 35: 1335-1337.

[120] Du, D.-Z., Pan, L.Q., and Shing, M.-T. [1986], Minimum edge length guillotine rectangular partition, Technical Report 0241886, Math. Sci. Res. Inst., Univ. California, Berkeley.

[121] Du, D.-Z., Hsu, D.F. and Xu, K.-J. [1987], Bounds on guillotine ratio, *Congressus Numerantium* 58: 313-318.

[122] Du, D.-Z. [1986], On heuristics for minimum length rectangular partitions, Technical Report, Math. Sci. Res. Inst., Univ. California, Berkeley.

[123] Du, D.-Z. and Zhang, Y.-J. [1990], On heuristics for minimum length rectilinear partitions, *Algorithmica*, 5: 111-128.

[124] Gonzalez, T. and Zheng, S.Q. [1985], Bounds for partitioning rectilinear polygons, *Proc. 1st Symp. on Computational Geometry.*

[125] Gonzalez, T. and Zheng, S.Q. [1989], Improved bounds for rectangular and guillotine partitions, *Journal of Symbolic Computation* 7: 591-610.

[126] Lingas, A., Pinter, R.Y., Rivest, R.L. and Shamir, A. [1982], Minimum edge length partitioning of rectilinear polygons, *Proc. 20th Allerton Conf. on Comm. Control and Compt.*, Illinos.

[127] Lingas, A. [1983], Heuristics for minimum edge length rectangular partitions of rectilinear figures, *Proc. 6th GI-Conference*, Dortmund, (Springer-Verlag).

[128] Levcopoulos, C. [1986], Fast heuristics for minimum length rectangular partitions of polygons, *Proc 2nd Symp. on Computational Geometry.*

[129] Lu, B. and Ruan, L. [2000], Polynomial time approximation scheme for the rectilinear Steiner arborescence problem, *Journal of Combinatorial Optimization* 4: 357-363.

[130] Mitchell, J.S.B. [1996a], Guillotine subdivisions approximate polygonal subdivisions: A simple new method for the geometric k-MST problem. *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pp. 402-408.

[131] Mitchell, J.S.B., Blum, A., Chalasani, P., Vempala, S. [1999], A constant-factor approximation algorithm for the geometric k-MST problem in the plane *SIAM J. Comput.* 28: 771–781.

[132] Mitchell, J.S.B. [1999], Guillotine subdivisions approximate polygonal subdivisions: Part II - A simple polynomial-time approximation scheme for geometric k-MST, TSP, and related problem, *SIAM J. Comput.* 28: 1298-1307.

[133] Mitchell, J.S.B. [1997], Guillotine subdivisions approximate polygonal subdivisions: Part III - Faster polynomial-time approximation scheme for geometric network optimization, *Proc. ninth Canadian conference on computational geometry*, 229-232.

[134] Rao, S.B. and W.D. Smith [1998], Approximating geometrical graphs via "spanners" and "banyans", ACM STOC'98, pp. 540-550.

[135] Xiaofeng Gao, Weili Wu, Xuefei Zhang, Xianyue Li: A constant-factor approximation for d-hop connected dominating sets in unit disk graph. Int. J. Sens. Networks 12(3): 125-136 (2012)

[136] Lidong Wu, Hongwei Du, Weili Wu, Yuqing Zhu, Ailian Wang, Wonjun Lee: PTAS for routing-cost constrained minimum connected dominating set in growth bounded graphs. J. Comb. Optim. 30(1): 18-26 (2015)

[137] Zhao Zhang, Xiaofeng Gao, Weili Wu, Ding-Zhu Du: PTAS for Minimum Connected Dominating Set in Unit Ball Graph. WASA 2008: 154-161

[138] A. Borchers and D.-Z. Du, The k-Steiner ratio in graphs, *Proceedings of 27th ACM Symposium on Theory of Computing*, 1995.

[139] F.R.K. Chung and E.N. Gilbert, Steiner trees for the regular simplex, *Bull. Inst. Math. Acad. Sinica*, 4 (1976) 313-325.

[140] F.R.K. Chung and R.L. Graham, A new bound for euclidean Steiner minimum trees, *Ann. N.Y. Acad. Sci.*, 440 (1985) 328-346.

[141] F.R.K. Chung and F.K. Hwang, A lower bound for the Steiner tree problem, *SIAM J.Appl.Math.*, 34 (1978) 27-36.

[142] R. Crourant and H. Robbins, *What Is Mathematics?*, (Oxford Univ. Press, New York, 1941).

[143] D.E. Drake and S. Hougardy, On approximation algorithms for the terminal Steiner tree problem, *Information Processing Letters*, 89 (2004) 15-18.

[144] D.-Z. Du, R.L. Graham, P.M. Pardalos, P.-J. Wan, Weili Wu and W. Zhao, Analysis of greedy approximations with nonsubmodular potential functions, *Proc. ACM-SIAM Symposiun on Discrete Algorithms (SODA)*, 2008, pp. 167-175.

[145] D.-Z. Du and F.K. Hwang, The Steiner ratio conjecture of Gilbert-Pollak is true, *Proceedings of National Academy of Sciences*, 87 (1990) 9464-9466.

[146] D.-Z. Du, Y. Zhang, and Q. Feng, On better heuristic for euclidean Steiner minimum trees, *Proceedings 32nd FOCS* (1991).

[147] L.R. Foulds and R.L. Graham, The Steiner problem in Phylogeny is NP-complete, *Advanced Applied Mathematics*, 3 (1982) 43-49.

[148] M.R. Garey, R.L. Graham and D.S. Johnson, The complexity of computing Steiner minimal trees, *SIAM J. Appl. Math.*, 32 (1977) 835-859.

[149] M.R. Garey and D.S. Johnson, The rectilinear Steiner tree is NP-complete, *SIAM J. Appl. Math.*, 32 (1977) 826-834.

[150] E.N. Gilbert and H.O. Pollak, Steiner minimal trees, *SIAM J. Appl. Math.*, 16 (1968) 1-29.

[151] R.L. Graham and F.K. Hwang, Remarks on Steiner minimal trees, *Bull. Inst. Math. Acad. Sinica*, 4 (1976) 177-182.

[152] S.Y. Hsieh and S.-C. Yang, Approximating the selected-internal Steiner tree, *Theoretical Computer Science*, 38 (2007) 288-291.

[153] F.K. Hwang, On Steiner minimal trees with rectilinear distance, *SIAM J. Appl. Math.*, 30 (1972) 104-114.

[154] R.M. Karp, Reducibility among combinatorial problems, in R.E. Miller and J.W. Tatcher (ed.), *Complexity of Computer Computation*, Plenum Press, New York, (1972) 85-103.

[155] G.-H. Lin and G. Xue, Steiner tree problem with minimum number of Steiner points and bounded edge-length, *Information Processing Letters*, 69 (1999) 53-57.

[156] G.-H. Lin and G. Xue, On the terminal Steiner tree problem, *Information Processing Letters*, 84 (2002) 103-107.

[157] I. Mandoiu and A. Zelikovsky, A Note on the MST heuristic for bounded edge-length Steiner trees with minimum number of Steiner points, *Information Processing Letters*, 75(4) (2000) 165-167.

[158] R. Ravi and J. D. Kececioglu, Approximation methods for sequence alignment under a fixed evolutionary tree, *Proc. 6th Symp. on Combinatorial Parrern Matching. Springer LNCS*, 937 (1995) 330-339.

[159] G. Robin and A. Zelikovsky, Improved Steiner trees approximation in graphs, *SIAM-ACM Symposium on Discrete Algorithms (SODA)*, San Francisco, CA, January 2000, pp. 770-779.

[160] J.H. Rubinstein and D.A. Thomas, The Steiner ratio conjecture for six points, *J. Combinatoria Theory, Ser.A*, 58 (1991) 54-77.

[161] P. Schreiber, On the history of the so-called Steiner weber problem, *Wiss. Z. Ernst-Moritz-Arndt-Univ. Greifswald, Math.-nat.wiss. Reihe*, 35, no.3 (1986).

[162] L. Wang and D.-Z. Du, Approximations for bottleneck Steiner trees, *Algorithmica*, 32 (2002) 554-561.

[163] L. Wang and D. Gusfield, Improved approximation algorithms for tree alignment, *Proc. 7th Symp. on Combinatorial Parrern Matching. Springer LNCS*, 1075 (1996) 220-233.

[164] A. Zelikovsky, The 11/6-approximation algorithm for the Steiner problem on networks, *Algorithmica*, 9 (1993) 463-470.

[165] A. Zelikovsky, A series of approximation algorithms for the acyclic airected Steiner tree Problem, *Algorithmica*, 18 (1997) 99-110.

[166] Jiawen Gao, Suogang Gao, Wen Liu, Weili Wu, Ding-Zhu Du, Bo Hou: An approximation algorithm for the k-generalized Steiner forest problem. Optim. Lett. 15(4): 1475-1483 (2021)

[167] Feng Zou, Yuexuan Wang, XiaoHua Xu, Xianyue Li, Hongwei Du, Peng-Jun Wan, Weili Wu: New approximations for minimum-weighted dominating sets and minimum-weighted connected dominating sets on unit disk graphs. Theor. Comput. Sci. 412(3): 198-208 (2011)

[168] Ling Ding, Weili Wu, James Willson, Lidong Wu, Zaixin Lu, Wonjun Lee: Constant-approximation for target coverage problem in wireless sensor networks. INFOCOM 2012: 1584-1592

[169] Zaixin Lu, Weili Wu, Wei Wayne Li: Target coverage maximisation for directional sensor networks. Int. J. Sens. Networks 24(4): 253-263 (2017)

[170] Zaixin Lu, Travis Pitchford, Wei Li, Weili Wu: On the Maximum Directional Target Coverage Problem in Wireless Sensor Networks. MSN 2014: 74-79

[171] Biaofei Xu, Yuqing Zhu, Deying Li, Donghyun Kim, Weili Wu: Minimum (k, ω)-angle barrier coverage in wireless camera sensor networks. Int. J. Sens. Networks 21(3): 179-188 (2016)

[172] Maggie Xiaoyan Cheng, Lu Ruan, Weili Wu: Achieving minimum coverage breach under bandwidth constraints in wireless sensor networks. INFOCOM 2005: 2638-2645

[173] Maggie Xiaoyan Cheng, Lu Ruan, Weili Wu: Coverage breach problems in bandwidth-constrained sensor networks. ACM Trans. Sens. Networks 3(2): 12 (2007)

[174] Ling Guo, Deying Li, Yongcai Wang, Zhao Zhang, Guangmo Tong, Weili Wu, Ding-Zhu Du: Maximisation of the number of β-view covered targets in visual sensor networks. Int. J. Sens. Networks 29(4): 226-241 (2019)

[175] Weili Wu, Zhao Zhang, Chuangen Gao, Hai Du, Hua Wang, Ding-Zhu Du: Quality of barrier cover with wireless sensors. Int. J. Sens. Networks 29(4): 242-251 (2019)

[176] Mihaela Cardei, My T. Thai, Yingshu Li, Weili Wu: Energy-efficient target coverage in wireless sensor networks. INFOCOM 2005: 1976-1984

[177] Zhao Zhang, Weili Wu, Jing Yuan, Ding-Zhu Du: Breach-Free Sleep-Wakeup Scheduling for Barrier Coverage With Heterogeneous Wireless Sensors. IEEE/ACM Trans. Netw. 26(5): 2404-2413 (2018)

[178] Donghyun Kim, Wei Wang, Junggab Son, Weili Wu, Wonjun Lee, Alade O. Tokuta: Maximum Lifetime Combined Barrier-Coverage of Weak Static Sensors and Strong Mobile Sensors. IEEE Trans. Mob. Comput. 16(7): 1956-1966 (2017)

[179] Chen Wang, My T. Thai, Yingshu Li, Feng Wang, Weili Wu: Minimum Coverage Breach and Maximum Network Lifetime in Wireless Sensor Networks. GLOBECOM 2007: 1118-1123

[180] Chen Wang, My T. Thai, Yingshu Li, Feng Wang, Weili Wu: Optimization scheme for sensor coverage scheduling with bandwidth constraints. Optim. Lett. 3(1): 63-75 (2009)

[181] Zhao Zhang, James Willson, Zaixin Lu, Weili Wu, Xuding Zhu, Ding-Zhu Du: Approximating Maximum Lifetime k-Coverage Through Minimizing Weighted k-Cover in Homogeneous Wireless Sensor Networks. IEEE/ACM Trans. Netw. 24(6): 3620-3633 (2016)

[182] James Willson, Zhao Zhang, Weili Wu, Ding-Zhu Du: Fault-tolerant coverage with maximum lifetime in wireless sensor networks. INFOCOM 2015: 1364-1372

[183] Hongwei Du, Panos M. Pardalos, Weili Wu, Lidong Wu: Maximum lifetime connected coverage with two active-phase sensors. J. Glob. Optim. 56(2): 559-568 (2013)

[184] Lidong Wu, Hongwei Du, Weili Wu, Deying Li, Jing Lv, Wonjun Lee: Approximations for Minimum Connected Sensor Cover. INFOCOM 2013: 1187-1194

[185] N. Garg, J. Köemann, Faster and simpler algorithms for multicommodity flows and other fractional packing problems, in Proceedings of the 39th Annual Symposium on the Foundations of Computer Science (1998), pp. 300C309.

[186] J. Byrka, F. Grandoni, T. Rothvoss, L. Sanita, An improved LP-based approximation for Steiner tree, in STOC 10: Proceedings of the Forty-Second ACM Symposium on Theory of Computing, June 5C8 (2010), pp. 583C592

[187] P. Berman, G. Calinescu, C. Shah, A. Zelikovsky, Efficient energy management in sensor networks, in Ad Hoc and Sensor Networks, Wireless Networks and Mobile Computing, vol. 2, ed. by Y. Xiao, Y. Pan (Nova Science Publishers, Hauppauge, 2005)

[188] T. Erlebach, M. Mihal, A $(4 + \varepsilon)$-approximation for the minimum-weight dominating set problem in unit disk graphs, in WAOA (2009), pp. 135C146

[189] T. Erlebach, T. Grant, F. Kammer, Maximising lifetime for fault tolerant target coverage in sensor networks. Sustain. Comput. Inform. Syst. 1, 213C225 (2011)

[190] J. Willson, W. Wu, L. Wu, L. Ding, D.-Z. Du, New approximation for maximum lifetime coverage. Optimization 63(6), 839C847 (2014)

[191] J. Li, Y. Jin, A PTAS for the weighted unit disk cover problem, in Automata, Languages, and Programming. ICALP (2015), pp. 898C909.

[192] Manki Min, Hongwei Du, Xiaohua Jia, Christina Xiao Huang, Scott C.-H. Huang, Weili Wu: Improving Construction for Connected Dominating Set with Steiner Tree in Wireless Sensor Networks. J. Glob. Optim. 35(1): 111-119 (2006)

[193] Donghyun Kim, Zhao Zhang, Xianyue Li, Wei Wang, Weili Wu, Ding-Zhu Du: A Better Approximation Algorithm for Computing Connected Dominating Sets in Unit Ball Graphs. IEEE Trans. Mob. Comput. 9(8): 1108-1118 (2010)

[194] Weili Wu, Hongwei Du, Xiaohua Jia, Yingshu Li, Scott C.-H. Huang: Minimum connected dominating sets and maximal independent sets in unit disk graphs. Theor. Comput. Sci. 352(1-3): 1-7 (2006)

[195] Blum, Arim; Jiang, Tao; Li, Ming; Tromp, John; Yannakakis [1994], Linear approximation of shortest superstrings, *Journal of ACM*, vol 41, no 4, 630-647.

[196] Chvátal, V. [1979], A greedy heuristic for the set-covering problem, *Mathematics of Operations Research*, vol 4, no 3, 233-235.

[197] Du, Ding-Zhu and Miller, Zevi [1988], Matroids and subset interconnection design, *SIAM J. Discrete Math.*, vol 1 no 4, 416-424.

[198] Du, D.-Z.; Graham, R.L; Pardalos, P.M.; Wan, P.-J.; Wu, Weili and Zhao, Wenbo [2008], Analysis of greedy approximation with nonsubmodular potential functions, to appear in *Proc. of SODA*.

[199] Du, Xiufeng; Wu, Weili; Kelley, Dean F. [1998], Approximations for subset interconnection designs, *Theoretical Computer Science*, vol 207 no 1, 171-180.

[200] Feige, Uriel, A threshold of $\ln n$ for approximating set cover, *J. ACM* vol 45, no 4, 634-652.

[201] M.L. Fisher, G.L. Nemhauser and L.A. Wolsey, An alalysis of approximations for maximizing submodular set functions - II, *Math. Prog. Study*, 8 (1978) 73-87.

[202] Guha, S. and Khuller, S. [1998], Approximation algorithms for connected dominating sets, *Algorithmca* vol 20, no 4, 374-387.

[203] Hausmann, D.; Korte, B.; Jenkyns, T.A. [1980], Worst case analysis of greedy type algorithms for independence systems, *Mathematical Programming Study* no 12, 120-131.

[204] Jenkyns, Thomas A [1976], The efficacy of the "greedy" algorithm, *Congressus Numerantium*, no 17, 341-350.

[205] Johnson, D.S. [1974], Approximation algorithms for combinatorial problems, *Journal of Computer and System Sciences*, vol 9 no 3, 256-278.

[206] Lovász, L. [1975], On the ratio of optimal integral and fractional covers, *Discrete Mathematics*, vol 13, 383-390.

[207] Lund, C, and Yanakakis, M. [1994], On the hardness of approximating minimization problems, *J. ACM*, vol 41 no 5, 960-981.

[208] Korte, Bernhard and Hausmann, Dirk [1978], An analysis of the greedy heuristic for independence systems, Ann. Discrete Math. vol 2, 65-74.

[209] Korte, B and Vygen, J. [2002], *Combinatorial Optimization*, Springer.

[210] Prisner, Erich [1992], Two algorithms for the subset interconnection design problem, *Networks*, vol 22 no 4, 385-395.

[211] Slavik, Petr [1997], A tight analysis of the greedy algorithm for set cover, *Journal of Algorithms*, vol 25, no 2, 237-254.

[212] Tarhio, J. and Ukkonen, E. [1988], A greedy approximation algorithm for constructing shortest common superstrings, *Theoretical Computer Science*, vol 57, no 1, 131-145.

[213] Turner, J.S. [1989], Approximation algorithms for the shortest common superstring problem, *Information and Computation*, vol 83, no 1, 1-20.

[214] Wolsey, Laurence A. [1982], An analysis of the greedy algorithm for submodular set covering problem, *Combinatorica*, vol 2 no 4, 385-393.

[215] L.M. Kirousis, E. Kranakis, D. Krizanc, and A. Pelc, Power consumption in packer radio networks, Theoretical Computer Science 243 (2000) 289-305.

[216] W.T. Chen and N.F. Huang, The Strongly connection problem on multihop packet radio networks, IEEE Transactions on Communications, vol. 37, no. 3 (1989) pp. 293-295.

[217] Kirk Pruhs: Speed Scaling. Encyclopedia of Algorithms 2016: 2045-2047.

[218] Jeff Edmonds, Kirk Pruhs: Scalably scheduling processes with arbitrary speedup curves. ACM Trans. Algorithms 8(3): 28 (2012).

[219] H. Lin and J. Bilmes, Optimal selection of limited vocabulary speech corpora. In Interspeech, 2011

[220] Rishabh K. Iyer, Stefanie Jegelka, Jeff A. Bilmes: Fast Semidifferential-based Submodular Function Optimization. ICML (3) 2013: 855-863.

[221] K. Nagano, Y. Kawahara, and K. Aihara: Size-constrained submodular minimization through minimum norm base, in Proceedings of the 28th International Conference on Machine Learning, Bellevue, WA, USA, 2011.

[222] Anton Barhan, Andrey Shakhomirov: Methods for Sentiment Analysis of Twitter Messages, Proceeding of the 12th Conference of Fruct Association, 2012, pp. 216-222.

[223] Baoyuan Wu, Siwei Lyu, Bernard Ghanem: Constrained Submodular Minimization for Missing Labels and Class Imbalance in Multi-label Learning. AAAI 2016: 2229-2236.

[224] M. Grötschel, L. Lovász, and A. Schrijver: Geometric Algorithms and Combinatorial Optimization. Springer-Verlag, 2nd edition, 1988.

[225] J. B. Orlin, A faster strongly polynomial time algorithm for submodular function minimization. Mathematical Programming, 118:237C251, 2009.

[226] A. Schrijver: A combinatorial algorithm minimizing submodular functions in strong polynomial time, J. Combinatorial Theory (B), 80 (2000): 346-355.

[227] George L. Nemhauser, Laurence A. Wolsey, and Marshall L. Fisher, An analysis of approximations for maximizing submodular set functions - I. Mathematical Programming, 14(1): 265-294 (1978).

[228] M. Sviridenko, A Note on Maximizing a Submodular Set Function Subject to Knapsack Constraint. Operations Research Letters, 32: 41-43 (2004).

[229] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance, Cost-effective outbreak detection in networks. Pages 420-429 of: KDD'07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. New York, NY, USA: ACM, 2007.

[230] G. Calinescu, C. Chekuri, M. Pl and J. Vondrk, Maximizing a submodular set function subject to a matroid constraint, SIAM J. Comp. 40:6 (2011), 1740-1766.

[231] M. L. Fisher, G. L. Nemhauser, and L. A. Wolsey. An analysis of approximations for maximizing submodular set functions C II. In Polyhedral Combinatorics, volume 8 of Mathematical Programming Study, pages 73C87. North-Holland Publishing Company, 1978.

[232] Laurence A. Wolsey, An analysis of the greedy algorithm for the submodular set covering problem. Combinatorica, 2(4):385-393 (1982).

[233] U. Feige, V. Mirrokni, and J. Vondrak, Maximizing nonmonotone submodular functions, in Proceedings of the IEEE Foundations of Computer Science, 2007, pp. 461C471.

[234] J. Lee, V. Mirrokni, V. Nagarajan, and M. Sviridenko, Nonmonotone submodular maximization under matroid and knapsack constraints, in Proceedings of the ACM Symposium on Theory of Computing, 2009, pp. 323C332.

[235] M. Feldman, J. Naor, R. Schwartz: A unified continuous greedy algorithm for submodular maximization, IEEE FOCS 2011, pp. 570-579.

[236] Z. Svitkina and L. Fleischer, Submodular approximation: Sampling-based algorithms and lower bounds, SIAM Journal on Computing (2011).

[237] Ruiqi Yang, Shuyang Gu, Chuangen Gao, Weili Wu, Hua Wang, Dachuan Xu: A constrained two-stage submodular maximization. Theor. Comput. Sci. 853: 57-64 (2021).

[238] Shuyang Gu, Ganquan Shi, Weili Wu, Changhong Lu: A fast double greedy algorithm for non-monotone DR-submodular function maximization. Discret. Math. Algorithms Appl. 12(1): 2050007:1-2050007:11 (2020)

[239] Lei Lai, Qiufen Ni, Changhong Lu, Chuanhe Huang, Weili Wu: Monotone submodular maximization over the bounded integer lattice with cardinality constraints. Discret. Math. Algorithms Appl. 11(6): 1950075:1-1950075:14 (2019)

[240] Chenfei Hou, Suogang Gao, Wen Liu, Weili Wu, Ding-Zhu Du, Bo Hou: An approximation algorithm for the submodular multicut problem in trees with linear penalties. Optim. Lett. 15(4): 1105-1112 (2021).

[241] Zhao Zhang, Joonglyul Lee, Weili Wu, Ding-Zhu Du: Approximation for minimum strongly connected dominating and absorbing set with routing-cost constraint in disk digraphs. Optim. Lett. 10(7): 1393-1401 (2016)

[242] Zhao Zhang, Weili Wu, Lidong Wu, Yanjie Li, Zongqing Chen: Strongly connected dominating and absorbing set in directed disk graph. Int. J. Sens. Networks 19(2): 69-77 (2015)

[243] Hongjie Du, Weili Wu, Shan Shan, Donghyun Kim, Wonjun Lee: Constructing weakly connected dominating set for secure clustering in distributed sensor network. J. Comb. Optim. 23(2): 301-307 (2012)

[244] Deying Li, Donghyun Kim, Qinghua Zhu, Lin Liu, Weili Wu: Minimum Total Communication Power Connected Dominating Set in Wireless Networks. WASA 2012: 132-141

[245] Xiaofeng Gao, Wei Wang, Zhao Zhang, Shiwei Zhu, Weili Wu: A PTAS for minimum d-hop connected dominating set in growth-bounded graphs. Optim. Lett. 4(3): 321-333 (2010)

[246] Jiao Zhou, Zhao Zhang, Weili Wu, Kai Xing: A greedy algorithm for the fault-tolerant connected dominating set in a general graph. J. Comb. Optim. 28(1): 310-319 (2014)

[247] Shan Shan, Weili Wu, Wei Wang, Hongjie Du, Xiaofeng Gao, Ailian Jiang: Constructing minimum interference connected dominating set for multi-channel multi-radio multi-hop wireless network. Int. J. Sens. Networks 11(2): 100-108 (2012)

[248] Deying Li, Hongwei Du, Peng-Jun Wan, Xiaofeng Gao, Zhao Zhang, Weili Wu: Minimum Power Strongly Connected Dominating Sets in Wireless Networks. ICWN 2008: 447-451

[249] Donghyun Kim, Xianyue Li, Feng Zou, Zhao Zhang, Weili Wu: Recyclable Connected Dominating Set for Large Scale Dynamic Wireless Networks. WASA 2008: 560-569

[250] Ning Zhang, Incheol Shin, Feng Zou, Weili Wu, My T. Thai: Tradeoff scheme for fault tolerant connected dominating sets on size and diameter. FOWANC 2008: 1-8

[251] Ling Ding, Xiaofeng Gao, Weili Wu, Wonjun Lee, Xu Zhu, Ding-Zhu Du: Distributed Construction of Connected Dominating Sets with Minimum Routing Cost in Wireless Networks. ICDCS 2010: 448-457

[252] Deying Li, Hongwei Du, Peng-Jun Wan, Xiaofeng Gao, Zhao Zhang, Weili Wu: Construction of strongly connected dominating sets in asymmetric multihop wireless networks. Theor. Comput. Sci. 410(8-10): 661-669 (2009)

[253] Feng Zou, Xianyue Li, Donghyun Kim, Weili Wu: Construction of Minimum Connected Dominating Set in 3-Dimensional Wireless Network. WASA 2008: 134-140

[254] Xianyue Li, Xiaofeng Gao, Weili Wu: A Better Theoretical Bound to Approximate Connected Dominating Set in Unit Disk Graph. WASA 2008: 162-175

[255] Lu Ruan, Hongwei Du, Xiaohua Jia, Weili Wu, Yingshu Li, Ker-I Ko: A greedy approximation for minimum connected dominating sets. Theor. Comput. Sci. 329(1-3): 325-330 (2004)

[256] Hongwei Du, Weili Wu, Qiang Ye, Deying Li, Wonjun Lee, Xuepeng Xu: CDS-Based Virtual Backbone Construction with Guaranteed Routing Cost in Wireless Sensor Networks. IEEE Trans. Parallel Distributed Syst. 24(4): 652-661 (2013)

[257] Hongwei Du, Qiang Ye, Weili Wu, Wonjun Lee, Deying Li, Ding-Zhu Du, Stephen Howard: Constant approximation for virtual backbone construction with Guaranteed Routing Cost in wireless sensor networks. INFOCOM 2011: 1737-1744

[258] I. Dinur, D. Steurer, Analytical approach to parallel repetition, in STOC 14: Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing (2014), pp. 624C633

[259] Chuanwen Luo, Wenping Chen, Deying Li, Yongcai Wang, Hongwei Du, Lidong Wu, Weili Wu: Optimizing flight trajectory of UAV for efficient data collection in wireless sensor networks. Theor. Comput. Sci. 853: 25-42 (2021)

[260] Xingjian Ding, Jianxiong Guo, Deying Li, Weili Wu: Optimal wireless charger placement with individual energy requirement. Theor. Comput. Sci. 857: 16-28 (2021)

[261] Chuanwen Luo, Lidong Wu, Wenping Chen, Yongcai Wang, Deying Li, Weili Wu: Trajectory Optimization of UAV for Efficient Data Collection from Wireless Sensor Networks. AAIM 2019: 223-235

[262] Donghyun Kim, Wei Wang, Deying Li, Joonglyul Lee, Weili Wu, Alade O. Tokuta: A joint optimization of data ferry trajectories and communication powers of ground sensors for long-term environmental monitoring. J. Comb. Optim. 31(4): 1550-1568 (2016)

[263] Donghyun Kim, R. N. Uma, Baraki H. Abay, Weili Wu, Wei Wang, Alade O. Tokuta: Minimum Latency Multiple Data MULE Trajectory Planning in Wireless Sensor Networks. IEEE Trans. Mob. Comput. 13(4): 838-851 (2014)

[264] Donghyun Kim, Wei Wang, Weili Wu, Deying Li, Changcun Ma, Nassim Sohaee, Wonjun Lee, Yuexuan Wang, Ding-Zhu Du: On bounding node-to-sink latency in wireless sensor networks with multiple sinks. Int. J. Sens. Networks 13(1): 13-29 (2013)

[265] Donghyun Kim, Baraki H. Abay, R. N. Uma, Weili Wu, Wei Wang, Alade O. Tokuta: Minimizing data collection latency in wireless sensor network with multiple mobile elements. INFOCOM 2012: 504-512

[266] Donghyun Kim, Wei Wang, Nassim Sohaee, Changcun Ma, Weili Wu, Wonjun Lee, Ding-Zhu Du: Minimum Data-Latency-Bound k-Sink Placement Problem in Wireless Sensor Networks. IEEE/ACM Trans. Netw. 19(5): 1344-1353 (2011)

[267] Deying Li, Qinghua Zhu, Hongwei Du, Weili Wu, Hong Chen, Wenping Chen: Conflict-Free Many-to-One Data Aggregation Scheduling in Multi-Channel Multi-Hop Wireless Sensor Networks. ICC 2011: 1-5

[268] Donghyun Kim, Wei Wang, Ling Ding, Jihwan Lim, Heekuck Oh, Weili Wu: Minimum average routing path clustering problem in multi-hop 2-D underwater sensor networks. Optim. Lett. 4(3): 383-392 (2010)

[269] Wei Wang, Donghyun Kim, James Willson, Bhavani M. Thuraisingham, Weili Wu: A Better Approximation for Minimum Average Routing Path Clustering Problem in 2-d Underwater Sensor Networks. Discret. Math. Algorithms Appl. 1(2): 175-192 (2009)

[270] Wei Wang, Donghyun Kim, Nassim Sohaee, Changcun Ma, Weili Wu: A PTAS for Minimum d-Hop Underwater Sink Placement Problem in 2-d Underwater Sensor Networks. Discret. Math. Algorithms Appl. 1(2): 283-290 (2009)

[271] Zhao Zhang, Xiaofeng Gao, Xuefei Zhang, Weili Wu, Hui Xiong: Three Approximation Algorithms for Energy-Efficient Query Dissemination in Sensor Database System. DEXA 2009: 807-821

[272] Weili Wu, Xiuzhen Cheng, Min Ding, Kai Xing, Fang Liu, Ping Deng: Localized Outlying and Boundary Data Detection in Sensor Networks. IEEE Trans. Knowl. Data Eng. 19(8): 1145-1157 (2007)

[273] Guanfeng Li, Hui Ling, Taieb Znati, Weili Wu: A Robust on-Demand Path-Key Establishment Framework via Random Key Predistribution

for Wireless Sensor Networks. EURASIP J. Wirel. Commun. Netw. 2006 (2006)

[274] Xingjian Ding, Jianxiong Guo, Yongcai Wang, Deying Li, Weili Wu: Task-driven charger placement and power allocation for wireless sensor networks. Ad Hoc Networks 119: 102556 (2021).

[275] Guoyao Rao, Yongcai Wang, Wenping Chen, Deying Li, Weili Wu: Matching influence maximization in social networks. Theor. Comput. Sci. 857: 71-86 (2021)

[276] Jianxiong Guo, Weili Wu: Adaptive Influence Maximization: If Influential Node Unwilling to Be the Seed. ACM Trans. Knowl. Discov. Data 15(5): 84:1-84:23 (2021)

[277] Guoyao Rao, Yongcai Wang, Wenping Chen, Deying Li, Weili Wu: Maximize the Probability of Union-Influenced in Social Networks. COCOA 2021: 288-301.

[278] Jianxiong Guo, Weili Wu: Influence Maximization: Seeding Based on Community Structure. ACM Trans. Knowl. Discov. Data 14(6): 66:1-66:22 (2020)

[279] Jing Yuan, Weili Wu, Wen Xu: Approximation for Influence Maximization. Handbook of Approximation Algorithms and Metaheuristics (2) 2018

[280] Zaixin Lu, Zhao Zhang, Weili Wu: Solution of Bharathi-Kempe-Salek conjecture for influence maximization on arborescence. J. Comb. Optim. 33(2): 803-808 (2017)

[281] Yuqing Zhu, Weili Wu, Yuanjun Bi, Lidong Wu, Yiwei Jiang, Wen Xu: Better approximation algorithms for influence maximization in online social networks. J. Comb. Optim. 30(1): 97-108 (2015)

[282] Wen Xu, Zaixin Lu, Weili Wu, Zhiming Chen: A novel approach to online social influence maximization. Soc. Netw. Anal. Min. 4(1): 153 (2014)

[283] Zaixin Lu, Wei Zhang, Weili Wu, Joonmo Kim, Bin Fu: The complexity of influence maximization problem in the deterministic linear threshold model. J. Comb. Optim. 24(3): 374-378 (2012)

[284] Zaixin Lu, Wei Zhang, Weili Wu, Bin Fu, Ding-Zhu Du: Approximation and Inapproximation for the Influence Maximization Problem in Social Networks under Deterministic Linear Threshold Model. ICDCS Workshops 2011: 160-165

[285] Jianxiong Guo, Weili Wu: Continuous Profit Maximization: A Study of Unconstrained Dr-Submodular Maximization. IEEE Trans. Comput. Soc. Syst. 8(3): 768-779 (2021)

[286] Bin Liu, Xiao Li, Huijuan Wang, Qizhi Fang, Junyu Dong, Weili Wu: Profit Maximization problem with Coupons in social networks. Theor. Comput. Sci. 803: 22-35 (2020)

[287] Tiantian Chen, Bin Liu, Wenjing Liu, Qizhi Fang, Jing Yuan, Weili Wu: A random algorithm for profit maximization in online social networks. Theor. Comput. Sci. 803: 36-47 (2020)

[288] Bin Liu, Yuxia Yan, Qizhi Fang, Junyu Dong, Weili Wu, Huijuan Wang: Maximizing profit of multiple adoptions in social networks with a martingale approach. J. Comb. Optim. 38(1): 1-20 (2019)

[289] Yuqing Zhu, Deying Li, Ruidong Yan, Weili Wu, Yuanjun Bi: Maximizing the Influence and Profit in Social Networks. IEEE Trans. Comput. Soc. Syst. 4(3): 54-64 (2017)

[290] Yuqing Zhu, Zaixin Lu, Yuanjun Bi, Weili Wu, Yiwei Jiang, Deying Li: Influence and Profit: Two Sides of the Coin. ICDM 2013: 1301-1306

[291] Bin Liu, Xiao Li, Huijuan Wang, Qizhi Fang, Junyu Dong, Weili Wu: Profit Maximization Problem with Coupons in Social Networks. AAIM 2018: 49-61

[292] A.A. Ageev and M. Svirdenko, Pipage rounding: a new method of constructing algorithms with proven performance guarantee, *Journal of Combinatorial Optimization*, 8 (2004) 307-328.

[293] Bellare, M., Goldreich, O., and Sudan, M., Free bits and nonapproximability, *Proc. 36th FOCS*, 1995, pp.422-431.

[294] Dimitris Bertsimas, Chung-Piaw Teo, Rakesh Vohra: On dependent randomized rounding algorithms, *Oper. Res. Lett.* 24(3): 105-114 (1999).

[295] Bland, G. G. [1977], New finite pivoting rules of the simplex method, *Mathematics of Operations Research* Vol 2, 103-107.

[296] J. Bar-LLan, G. Kortsarz and D. Prleg, Generalized submodular cover problem and applications, *Theoretical Computer Science*, 250 (2001) 179-200.

[297] Gruia Calinescu, Chandra Chekuri, Martin P?, Jan Vondr?: Maximizing a Submodular Set Function Subject to a Matroid Constraint, IPCO 2007: 182-196.

[298] Charnes, A. [1952], Optimality and degeneracy in linear programming, *Econometrica* 20, 160-170.

[299] Jing-Chao Chen: Iterative Rounding for the Closest String Problem CoRR abs/0705.0561: (2007).

[300] Cheriyan, J.; Vempala, S.; Vetta, A.: Network design via iterative rounding of setpair relaxations, *Combinatorica*, Volume 26, Issue 3, p.255-275 (2006)

[301] Chvátal, V. [1979], A greedy heuristic for the set-covering problem, *Mathematics of Operations Research*, 4: 233-235.

[302] Dantzig, G.B. [1951], Maximization of a linear function of variables subject to linear inequalities, Chap. XXI of *Activity Analysis of Production and Allocation*, (Cowles Commission Monograph 13), T.C. Koopmans (ed.), John-Wiley, New York, 1951.

[303] Lisa Fleischer, Kamal Jain, David P. Williamson, An iterative rounding 2-approximation algorithm for the element connectivity problem, 42nd Annual IEEE Symposium on Foundations of Computer Science, 2001.

[304] Harold N. Gabow, Suzanne Gallagher: Iterated rounding algorithms for the smallest k-edge connected spanning subgraph. SODA 2008: 550-559.

[305] Harold N. Gabow, Michel X. Goemans, Evá Tardos, David P. Williamson: Approximating the smallest k-edge connected spanning subgraph by LP-rounding, *Networks* 53(4): 345-357 (2009).

[306] Dongdong Ge, Yinyu Ye, Jiawei Zhang, the fixed-hub single allocation problem: a geometric rounding approach, working paper, 2007.

[307] Dongdong Ge, Simai Hey, Zizhuo Wang, Yinyu Ye, Shuzhong Zhang, Geometric rounding: a dependent rounding scheme for allocation problems, working paper, 2008.

[308] M. X. Goemans, A. Goldberg, S. Plotkin, D. Shmoys, E. Tardos, and D. P. Williamson: Approximation algorithms for network design problems, *SODA*, 1994 pp.223-232.

[309] M.X. Goemans and D.P. Williamson, New $\frac{3}{4}$-approximation algorithms for the maximum satisfiability problem, *SIAM Journal on Discrete Mathematics*, 7 (1994) 656-666.

[310] Rajiv Gandhi, Samir Khuller, Srinivasan Parthasarathy, Aravind Srinivasan: Dependent rounding and its applications to approximation algorithms. J. ACM 53(3): 324-360 (2006).

[311] D. Gusfield and L. Pitt, A bounded approximation for the minimum cost 2-sat problem, *Algorithmica*,8 (1992) 103-117.

[312] Hochbaum, D.S. [1997], Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems, in D.S. Hochbaum (ed.) *Approximation Algorithms for NP-Hard Problems*, PWS Publishing Company, Boston, pp.94-143.

[313] K. Jain, A factor 2 approximation algorithm for the generalized Steiner network problem, *Combinatorica* 21 (2001), pp. 39-60.

[314] D.S. Johnson, Approximation algorithms for combinatorial problems, *Journal of Computer and System Sciences*, 9 (1974) 256-278.

[315] Karmarkar, N. [1984], A new polynomial-time algorithm for linear programming, *Proc. 16th STOC*, 302-311.

[316] Khachiyan, L.G. [1979], A polynomial algorithm for linear programming, *Doklad. Akad. Nauk.* USSR Sec. 244, 1093-1096.

[317] Klee, V.L. and Minty, G.J. [1972], How good is the simplex algorithm, in O. Shisha (ed.) *Inequalities* 3, Academic, New York.

[318] J.K. Lenstra, D.B. Shmoys and E. Tardos, Approximation algorithms for scheduling unrelated parallel machines, *Mathematical Programming*, 46 (1990) 259-271.

[319] Lovasz, L. [1975], On the ratio of optimal integral and fractional covers, *Discrete Mathematics*, 13: 383-390.

[320] V. Melkonian and E. Tardos, Algorithms for a network design problem with crossing supermodular demands, Networks 43, (2004), 256.265.

[321] G.L. Namhause and L.E. Trotter, Vertex packings: structural properties and algorithms, *Math. Program.* 8 (1975) 232-??.

[322] Wolsey, L.A. [1980], Heuristic analysis, linear programming and branch and bound, *Mathematical Programming Study* 13: 121-134.

[323] Laurence A. Wolsey: Maximizing real-valued submodular function: primal and dual heuristics for location problems, *Math. of Operations Research* 7 (1982) 410-425.

[324] M. Yannakakis, On the approximation of maximum satisfiability, *Journal of Algorithms*, 3 (1994) 475-502.

[325] Wenguo Yang, Jianmin Ma, Yi Li, Ruidong Yan, Jing Yuan, Weili Wu, Deying Li: Marginal Gains to Maximize Content Spread in Social Networks. IEEE Trans. Comput. Soc. Syst. 6(3): 479-490 (2019)

[326] Yapu Zhang, Jianxiong Guo, Wenguo Yang, Weili Wu: Targeted Activation Probability Maximization Problem in Online Social Networks. IEEE Trans. Netw. Sci. Eng. 8(1): 294-304 (2021)

[327] Ruidong Yan, Yi Li, Weili Wu, Deying Li, Yongcai Wang: Rumor Blocking through Online Link Deletion on Social Networks. ACM Trans. Knowl. Discov. Data 13(2): 16:1-16:26 (2019)

[328] Guangmo Amo Tong, Ding-Zhu Du, Weili Wu: On Misinformation Containment in Online Social Networks. NeurIPS 2018: 339-349

[329] Ruidong Yan, Deying Li, Weili Wu, Ding-Zhu Du, Yongcai Wang: Minimizing Influence of Rumors by Blockers on Social Networks: Algorithms and Analysis. IEEE Trans. Netw. Sci. Eng. 7(3): 1067-1078 (2020)

[330] Jianming Zhu, Smita Ghosh, Weili Wu: Robust rumor blocking problem with uncertain rumor sources in social networks. World Wide Web 24(1): 229-247 (2021)

[331] Jianxiong Guo, Yi Li, Weili Wu: Targeted Protection Maximization in Social Networks. IEEE Trans. Netw. Sci. Eng. 7(3): 1645-1655 (2020)

[332] Jianxiong Guo, Tiantian Chen, Weili Wu: A Multi-Feature Diffusion Model: Rumor Blocking in Social Networks. IEEE/ACM Trans. Netw. 29(1): 386-397 (2021)

[333] Yapu Zhang, Wenguo Yang, Weili Wu, Yi Li: Effector Detection Problem in Social Networks. IEEE Trans. Comput. Soc. Syst. 7(5): 1200-1209 (2020)

[334] Luobing Dong, Qiumin Guo, Weili Wu: Speech corpora subset selection based on time-continuous utterances features. J. Comb. Optim. 37(4): 1237-1248 (2019)

[335] Luobing Dong, Qiumin Guo, Weili Wu, Meghana N. Satpute: A semantic relatedness preserved subset extraction method for language corpora based on pseudo-Boolean optimization. Theor. Comput. Sci. 836: 65-75 (2020)

[336] Jianxiong Guo, Weili Wu: A Novel Scene of Viral Marketing for Complementary Products. IEEE Trans. Comput. Soc. Syst. 6(4): 797-808 (2019)

[337] Wenguo Yang, Jing Yuan, Weili Wu, Jianmin Ma, Ding-Zhu Du: Maximizing Activity Profit in Social Networks. IEEE Trans. Comput. Soc. Syst. 6(1): 117-126 (2019)

[338] Jianming Zhu, Smita Ghosh, Weili Wu: Group Influence Maximization Problem in Social Networks. IEEE Trans. Comput. Soc. Syst. 6(6): 1156-1164 (2019)

[339] Jianming Zhu, Junlei Zhu, Smita Ghosh, Weili Wu, Jing Yuan: Social Influence Maximization in Hypergraph in Social Networks. IEEE Trans. Netw. Sci. Eng. 6(4): 801-811 (2019)

[340] U. Feige and R. Izsak, Welfare maximization and the supermodular degree, in ACM ITCS, 2013, pp. 247C256.

[341] M. Feldman and R. Izsak, Constrained monotone function maximization and the supermodular degree, in ACM-SIAM SODA, 2014.

[342] M. Narasimhan and J. Bilmes, A submodular-supermodular procedure with applications to discriminative structure learning, In Proc. UAI (2005).

[343] R. Iyer and J. Bilmes, Algorithms for Approximate Minimization of the Difference between Submodular Functions, In Proc. UAI (2012).

[344] R. Iyer and J. Bilmes, Submodular Optimization Subject to Submodular Cover and Submodular Knapsack Constraints, In Advances of NIPS (2013).

[345] Chenchen Wu, Yishui Wang, Zaixin Lu, P.M. Pardalos, Dachuan Xu, Zhao Zhang, Ding-Zhu Du, Solving the degree-concentrated fault-tolerant spanning subgraph problem by DC programming, submitted for publication.

[346] Takanori Maehara, Kazuo Murota: A framework of discrete DC programming by discrete convex analysis. Math. Program. 152(1-2): 435-466 (2015).

[347] W. H. Cunningham. Decomposition of submodular functions. Combinatorica, 3(1):53C68, 1983.

[348] Yuqing Zhu, Zaixin Lu, Yuanjun Bi, Weili Wu, Yiwei Jiang, Deying Li: Influence and Profit: Two Sides of the Coin. ICDM 2013: 1301-1306.

[349] Yingfan L. Du, Hongmin W. Du: A new bound on maximum independent set and minimum connected dominating set in unit disk graphs. J. Comb. Optim. 30(4): 1173-1179 (2015).

[350] S. Fujishige. Submodular functions and optimization, volume 58. Elsevier Science, 2005.

[351] Wei Lu, Wei Chen, Laks V.S. Lakshmanan, From competition to complementarity: comparative influence diffusion and maximization, Proc. the VLDB Endowsment, Vol 9, No. 2: 60-71 (2015).

[352] Wei Chen, Tian Lin, Zihan Tan, Mingfei Zhao, Xuren Zhou, Robus influence maximization, KDD'16, San Francisco, CA, USA, 2016,

[353] Zhefeng Wang, Yu Yang, Jian Pei, and Enhong Chen, Activity maximization by effective information diffusion in social networks, arXiv: 1610.07754v1 [cs.SI] 25 Oct 2016.

[354] Jianming Zhu, Junlei Zhu, Smita Ghosh, Weili Wu, and Jing Yuan Social influence maximization in hypergraph in social networks, IEEE Transactions on Network Science and Engineering, online.

[355] Lidan Fan, Zaixin Lu, Weili Wu, Bhavani M. Thuraisingham, Huan Ma, Yuanjun Bi: Least Cost Rumor Blocking in Social Networks. ICDCS 2013: 540-549.

[356] Lidan Fan, Weili Wu: Rumor Blocking. Encyclopedia of Algorithms 2016: 1887-1892.

[357] Guangmo Amo Tong, Weili Wu, Ding-Zhu Du: Distributed Rumor Blocking in Social Networks: A Game Theoretical Analysis, IEEE Transactions on Computational Social Systems, online 2018.

[358] Xin Chen, Qingqin Nong, Yan Feng, Yongchang Cao, Suning Gong, Qizhi Fang, Ker-I Ko: Centralized and decentralized rumor blocking problems. J. Comb. Optim. 34(1): 314-329 (2017).

[359] Guangmo Amo Tong, Weili Wu, Ling Guo, Deying Li, Cong Liu, Bin Liu, Ding-Zhu Du: An efficient randomized algorithm for rumor blocking in online social networks. INFOCOM 2017: 1-9.

[360] Lidan Fan, Weili Wu, Xuming Zhai, Kai Xing, Wonjun Lee, Ding-Zhu Du: Maximizing rumor containment in social networks with constrained time. Social Netw. Analys. Mining 4(1): 214 (2014).

[361] Ailian Wang, Weili Wu, Junjie Chen: Social Network Rumors Spread Model Based on Cellular Automata. MSN 2014: 236-242.

[362] Lidan Fan, Weili Wu, Kai Xing, Wonjun Lee: Precautionary rumor containment via trustworthy people in social networks. Discrete Math., Alg. and Appl. 8(1) (2016).

[363] Ling Gai, Hongwei Du, Lidong Wu, Junlei Zhu, Yuehua Bu: Blocking Rumor by Cut. J. Comb. Optim. 36(2): 392-399 (2018).

[364] S. Fujishige. Submodular functions and optimization, volume 58. Elsevier Science, 2005.

[365] Anton Barhan, Andrey Shakhomirov: Methods for Sentiment Analysis of Twitter Messages, Proceeding of the 12th Conference of Fruct Association, 2012, pp. 216-222.

[366] Jeff Edmonds, Kirk Pruhs: Scalably scheduling processes with arbitrary speedup curves. ACM Trans. Algorithms 8(3): 28 (2012).

[367] Baoyuan Wu, Siwei Lyu, Bernard Ghanem: Constrained Submodular Minimization for Missing Labels and Class Imbalance in Multi-label Learning. AAAI 2016: 2229-2236.

[368] Jianxiong Guo, Weili Wu: Viral marketing with complementary products, in this book.

[369] Jianming Zhu, Junlei Zhu, Smita Ghosh, Weili Wu, Jing Yuan: Social influence maximization in hypergraph in social networks, IEEE Trans. Network Science and Engineering, online, 2018.

[370] Guangmo Amo Tong, Ding-Zhu Du, Weili Wu: On Misinformation Containment in Online Social Networks. NeurIPS 2018: 339-349.

[371] U. Feige and R. Izsak, Welfare maximization and the supermodular degree, in ACM ITCS, 2013, pp. 247C256.

[372] M. Feldman and R. Izsak, Constrained monotone function maximization and the supermodular degree, in ACM-SIAM SODA, 2014.

[373] Moran Feldman, Rani Izsak: Building a good team: Secretary problems and the supermodular degree. SODA 2017: 1651-1670.

[374] R. Iyer and J. Bilmes, Submodular Optimization Subject to Submodular Cover and Submodular Knapsack Constraints, In Advances of NIPS (2013).

[375] Wenruo Bai, Jeffrey A. Bilmes: Greed is Still Good: Maximizing Monotone Submodular+Supermodular (BP) Functions. ICML 2018: 314-323.

[376] Peng-Jun Wan, Ding-Zhu Du, Panos M. Pardalos, Weili Wu: Greedy approximations for minimum submodular cover with submodular cost. Comp. Opt. and Appl. 45(2): 463-474 (2010)

[377] Hongjie Du, Weili Wu, Wonjun Lee, Qinghai Liu, Zhao Zhang, Ding-Zhu Du: On minimum submodular cover with submodular cost. J. Global Optimization 50(2): 229-234 (2011).

[378] M. Narasimhan and J. Bilmes, A submodular-supermodular procedure with applications to discriminative structure learning, In Proc. UAI (2005).

[379] R. Iyer and J. Bilmes, Algorithms for Approximate Minimization of the Difference between Submodular Functions, In Proc. UAI (2012).

[380] Chenchen Wu, Yishui Wang, Zaixin Lu, P.M. Pardalos, Dachuan Xu, Zhao Zhang, Ding-Zhu Du, Solving the degree-concentrated fault-tolerant spanning subgraph problem by DC programming, submitted for publication.

[381] Takanori Maehara, Kazuo Murota: A framework of discrete DC programming by discrete convex analysis. Math. Program. 152(1-2): 435-466 (2015).

[382] Wei Lu, Wei Chen, Laks V.S. Lakshmanan, From competition to complementarity: comparative influence diffusion and maximization, Proc. the VLDB Endowsment, Vol 9, No. 2: 60-71 (2015).

[383] Wei Chen, Tian Lin, Zihan Tan, Mingfei Zhao, Xuren Zhou, Robus influence maximization, KDD'16, San Francisco, CA, USA, 2016,

[384] Zhefeng Wang, Yu Yang, Jian Pei, and Enhong Chen, Activity maximization by effective information diffusion in social networks, arXiv: 1610.07754v1 [cs.SI] 25 Oct 2016.

[385] Guoyao Rao, Yongcai Wang, Wenping Chen, Deying Li, Weili Wu: Maximize the Probability of Union-Influenced in Social Networks. COCOA 2021: 288-301

[386] Luobing Dong, Meghana N. Satpute, Weili Wu, Ding-Zhu Du: Two-Phase Multidocument Summarization Through Content-Attention-Based Subtopic Detection. IEEE Trans. Comput. Soc. Syst. 8(6): 1379-1392 (2021)

[387] Yapu Zhang, Jianxiong Guo, Wenguo Yang, Weili Wu: Mixed-case community detection problem in social networks: Algorithms and analysis. Theor. Comput. Sci. 854: 94-104 (2021)

[388] Qiufen Ni, Smita Ghosh, Chuanhe Huang, Weili Wu, Rong Jin: Discount allocation for cost minimization in online social networks. J. Comb. Optim. 41(1): 213-233 (2021)

[389] Shuyang Gu, Chuangen Gao, Ruiqi Yang, Weili Wu, Hua Wang, Dachuan Xu: A general method of active friending in different diffusion models in social networks. Soc. Netw. Anal. Min. 10(1): 41 (2020)

[390] Jianxiong Guo, Weili Wu: Discount advertisement in social platform: algorithm and robust analysis. Soc. Netw. Anal. Min. 10(1): 57 (2020)