

# Contents

<b>1</b>	<b>Chapter 4 Solution</b>	<b>2</b>
1.1	Problem 4.3 . . . . .	2
1.2	Problem 4.4 . . . . .	3
1.3	Problem 4.5 . . . . .	3
<b>2</b>	<b>Chapter 5 Solution</b>	<b>5</b>
2.1	Problem 5.3 . . . . .	5
2.2	Problem 5.4 . . . . .	5
2.3	Problem 5.5 . . . . .	5
2.4	Problem 5.6 . . . . .	6
2.5	Problem 5.7 . . . . .	6
2.6	Problem 5.8 . . . . .	6
2.7	Problem 5.10 . . . . .	6
2.8	Problem 5.16 . . . . .	7
2.9	Problem 5.20 . . . . .	7
2.10	Problem 5.21 . . . . .	7
2.11	Problem 5.23 . . . . .	8
2.12	Problem 5.29 . . . . .	8
<b>3</b>	<b>Chapter 6 Solution</b>	<b>9</b>
3.1	Problem 6.2 . . . . .	9
3.2	Problem 6.3 . . . . .	9
3.3	Problem 6.4 . . . . .	9
3.4	Problem 6.6 . . . . .	10
3.5	Problem 6.7 . . . . .	10
3.6	Problem 6.8 . . . . .	10
3.7	Problem 6.9 . . . . .	11
3.8	Problem 6.10 . . . . .	11
3.9	Problem 6.11 . . . . .	11
3.10	Problem 6.12 . . . . .	11
3.11	Problem 6.13 . . . . .	11

# Chapter 1

## Distance in graph

### 1.1 Problem 4.3

Modify the breadth first search algorithm as follows. Here each node has a property `dist`, the length of the shortest path from the source node to it, and a property `n`, the number of shortest paths from the source node to it. We update both as we visit each node. If we find a node whose `dist = n = 2` then there exists a cycle of length 4.

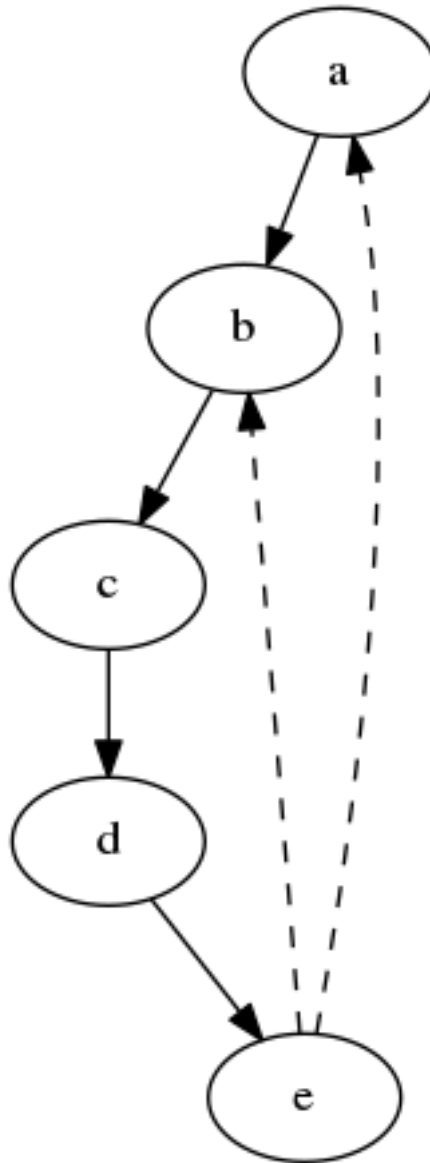
---

```
bfsModified(Node u):
    For all node v:
        v.dist = INFINITY
        v.n = 0
    u.dist = 0
    H = makeQueue(u)
    while H is not empty:
        Node s = h.poll()
        for all node v adjacent to s:
            if v.dist == INFINITY:
                v.n ++
                v.dist = s.dist + 1
                H.add(v)
            else:
                if s.dist + 1 == v.dist:
                    // found another shortest path from u to v
                    v.n ++
                if v.n == 2 && v.dist == 2:
                    // found a simple cycle of four
                    return TRUE
    return FALSE
```

---

We call `bfsModified` on all vertices in the graph. The complexity is therefore  $O(|V|(|E| + |V|)) = O(|V|^3)$  because  $|E| = O(|V|^2)$ .

## 1.2 Problem 4.4



In the above tree, the cycle edges are  $EB$  and  $EA$ .  $EB$  will give  $\text{level}[E] - \text{level}[B] + 1 = 4 - 1 + 1 = 4$ , while  $EA$  will give  $\text{level}[E] - \text{level}[A] + 1 = 4 - 0 + 1 = 5$ . The shortest cycle, however, is  $ABE$  with length 3.

The reason is that dfs can only detect all back edges but not all cycles; a back edge might belong to more than one cycle.

## 1.3 Problem 4.5

Use the `bfsModified` algorithm from 4.3:

---

```
bfs_modified(Node u, Node target):  
    For all node v:  
        v.dist = INFINITY  
        v.n = 0  
    u.dist = 0  
    H = makeQueue(u)  
    while H is not empty:  
        Node s = h.poll()  
        for all node v adjacent to s:
```

```

if v.dist == INFINITY:
    // first time v is discovered, set number of shortest path to 1
    v.n += 1
    v.dist = s.dist + 1
    H.add(v)
else:
    if s.dist + 1 == v.dist:
        // found another shortest path from u to v
        v.n += 1
return target.n

```

---

## 1.4 Problem 4.6

Consider the directed graph  $G'$  formed by taking all the edges  $\overrightarrow{(u, \text{prev}[u])}$ . We note that the `prev` array is set in Dijkstra when

---

```

s = H.remove()
for v adjacent to s:
    if v.dist > s.dist + l(v,s):
        v.dist = s.dist + l(v,s)
        v.prev = s

```

---

In other words, if we have an edge  $\overrightarrow{(a, b)}$  in  $G'$ , then  $b = \text{prev}[a]$  so  $b.\text{dist} > a.\text{dist}$ . So if there is a path from a node  $a$  to a node  $c$  then  $c.\text{dist} > a.\text{dist}$ . If there is a cycle then there is an edge from a node  $c$  to a node  $a$  while there is also a path from  $a$  to  $c$ . The former implies  $a.\text{dist} > c.\text{dist}$  while the latter implies  $c.\text{dist} > a.\text{dist}$ , so we have a contradiction.

## 1.5 Problem 4.7

# Chapter 2

## Greedy algorithm

### 2.1 Problem 5.3

Run DFS on the graph to detect a cycle edge. Return YES as soon as a cycle edge is found. Else, if there is no cycle edge, return NO.

This algorithm has  $O(|V|)$  runtime because we note that  $G$  is either a tree (in which case  $|E| = |V| - 1$ ) or it is not (in which case  $|E| > |V| - 1$ ).

- If  $G$  is a tree then we will not be able to detect any back edge. DFS will traverse the entire graph, which takes  $O(|V| + |E|)$ , but because  $|E| = |V| - 1$ , this is  $O(|V|)$ .
- If  $G$  is not a tree then we can find a back edge after traversing at most  $|V|$  edges because the edges picked by DFS form a tree, and any tree in the original graph can have at most  $|V|$  vertices.

### 2.2 Problem 5.4

We note that a connected component with  $m$  vertices must have at least  $m - 1$  edges<sup>1</sup>.

Let the number of vertices in component  $i$  be  $m_i$ ,  $i = 1, 2, \dots, k$ . We have  $\sum_{i=1}^k m_i = n$ .

The number of edges in the graph is the total number of edges in all components, which is at least

$$\sum_{i=1}^k (m_i - 1) = n - k.$$

### 2.3 Problem 5.5

- (a) We follow Kruskal's algorithm to build the minimum spanning tree: at each step, pick the edge with the least weight that does not create a cycle. Because all edge weights are increased by 1, the weight of any edge relative to all other edges is the same, so Kruskal's will produce the same result.
- (b) The shortest path will change. Consider the quadrilateral  $ABCD$  with

$$AB = BC = CD = 2, AD = 7.$$

Currently the shortest path from  $A$  to  $D$  is  $A \rightarrow B \rightarrow C \rightarrow D$ . If we increase the weight of all edges by 1 then  $AB = BC = CD = 3, DA = 8$ , so the shortest path from  $A$  to  $D$  is  $A \rightarrow D$ .

---

<sup>1</sup>since the component is connected, we can build a minimum spanning tree in it; this tree has  $m$  vertices and  $m - 1$  edges, so the number of edges in the component must be at least  $m - 1$ .

## 2.4 Problem 5.6

Sort the edge by their weights

$$e_1 < e_2 < \dots < e_n.$$

Kruskal's algorithm will iterate  $n$  times and, at time  $i$ , pick edge  $e_i$ , or the  $i$ -th smallest edge in the graph. Because the edge weights are distinct, the  $i$ -th smallest edge in the graph is distinct for all  $1 \leq i \leq n$ . Therefore the minimum spanning tree is unique.

## 2.5 Problem 5.7

Negate all the edge weights in the input graph and use Kruskal's algorithm to find the minimum spanning tree in the new graph. This tree is the maximum spanning tree in the original graph.

## 2.6 Problem 5.8

We use Prim's algorithm to build a minimum spanning tree in  $G$ .

Start building the MST from a vertex that is not  $S$ . Now assume we have reached the step where we have the tree current tree  $T$  and we are about to add  $S$  to  $T(V)$ . In other words, the next edge to be added to  $T(E)$  is the edge  $SA$  connecting  $S$  to a vertex  $A \in T(V)$ , which means  $SA$  is the lightest edge that connects  $T$  a vertex outside of  $T$ . (\*)

We prove that  $SA$  is also the shortest path from  $S$  to  $A$  and therefore  $T$  and the tree of shortest paths from  $S$  share the same edge  $SA$ .

Assume otherwise, the shortest path from  $S$  to  $A$  is not  $SA$ . Then there exists a vertex  $B \neq A$  such that the shortest path from  $S$  to  $A$  consists of the shortest path from  $S$  to  $B$  and  $BA$ . So

$$d(S, B) + BA < SA.$$

We consider two cases:

- If  $B \notin T(V)$  then because  $BA < SA$ ,  $SA$  is not the lightest edge that connects  $T$  to a vertex outside of it.
- If  $B \in T(V)$ , call  $I$  and  $K$  two vertices on the path from  $S$  to  $B$  such that  $I \in T(V)$ ,  $K \notin T(V)$  ( $I$  can be  $B$  and  $K$  can be  $S$ ). Then  $SA > d(S, B) = d(S, K) + IK + d(I, B) \geq IK$ , so again  $SA$  is not the lightest edge.

Both cases contradict (\*). Thus we have proven that the minimum spanning tree and the tree of shortest paths from a vertex  $S$  always share an edge.

## 2.7 Problem 5.10

Start with a MST  $T_H \in MST_H$  and  $T_G \in MST_G$ .

While there is an edge  $e \in T_G \cap H$  such that  $e \notin T_H$  do:

1. Add  $e$  to  $T_H$  to create a cycle  $C$ .
2. We see that for all  $e' \in C$ ,  $e' \neq e$  we have  $w(e') \leq w(e)$ . Otherwise if  $w(e') > w(e)$  we should have picked  $e$ , not  $e'$  when building the MST  $T_H$ , according to Kruskal's algorithm.
3. Let  $e = (u, v)$ , so  $u, v \in H$ . We see that  $e \in T_G$  so it connects two previously separate connected components, which we call  $U$  and  $V$ , and assume that  $u \in U$ ,  $v \in V$ . Because  $u$  and  $v$  are in  $H$  and  $T_H$  is the MST of  $H$ , there exists an edge  $e'' \in C \cap T_H$  that connects  $U$  and  $V$ .

4. From Step 2 we get  $w(e'') \leq w(e)$ . If  $w(e'') < w(e)$  then when we built  $T_G$  we should have picked  $e''$  instead of  $e$  to connect  $U$  and  $V$ . Therefore  $w(e'') = w(e)$ .
5. Let  $T'_H = T_H \cup \{e\} - \{e''\}$  then  $T_H$  is also a MST in  $H$ .
6. Rename  $T'_H \rightarrow T_H$  and check the loop condition.

After the loop we have  $T_G \cap H \subset T_H$ .

## 2.8 Problem 5.16

[http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-assignments/MIT6\\_046JS12\\_ps9\\_sol.pdf](http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-assignments/MIT6_046JS12_ps9_sol.pdf)

## 2.9 Problem 5.20

### Perfect matching.

Let  $G(V, E)$  be the input graph,  $S$  be the queue that contains all leaf nodes and  $M$  be the perfect matching.

The algorithm is as follows:

---

```

all leaf nodes to S.
while S is not empty:
    u = S.pop()
    v = u.adjacent
    M.add((u,v))
    Remove u, v from V and all edges connected to v from E
    Add new leaf nodes to S
return |V| = 0

```

---

### Feedback edge set.

Let  $F$  be the feedback edge set. Negate all the edges in  $G$  to get a graph  $G'$ . Use Kruskal's algorithm to build a minimum spanning tree  $T$  in  $G'$ . At each step, after we pop an edge  $e$ , if adding  $e$  to  $T$  does not form a cycle, then add  $e$  to  $T$ , else add  $e$  to  $F$ . As we go back to the original graph,  $T$  is the maximum spanning tree and  $F$  is the feedback set edge with minimum weight.

## 2.10 Problem 5.21

- (a) If no MST contain  $e$ , we are done. If **there is a MST  $T$  that contains  $e$** , removing  $e$  from  $T$  will separate it into two connected components  $M$  and  $N$ . Since  $e$  is part of a cycle, there exists another edge  $e'$  that connects  $M$  and  $N$ . Then  $T \cup \{e'\} - \{e\}$  is also a MST with  $w(T') \leq w(T)$ .
  - If  $w(T') < w(T)$  then we have a contradiction because  $T$  is a MST, so no MST can contain  $e$ .
  - If  $w(T') = w(T)$  then  $e$  and  $e'$  are both the heaviest edges in the cycle.  $T$  contains  $e$  and not  $e'$ , while  $T'$  contains  $e'$  and not  $e$ . In other words, there is a MST that does not contain the heaviest edge in a cycle.
- (b) If we get an edge  $e$  that is part of a cycle, then because the edges are ordered in decreasing weights,  $e$  is the heaviest edge in that cycle. By the property in (a),  $e$  cannot belong a MST. Therefore we can remove it.
- (c) To check if an edge  $e$  belongs to a cycle:
  - Let  $e = (u, v)$ .

- Remove  $e$  from the graph and DFS on  $u$ .
  - If  $v$  can be reached, return YES. Else, return NO.
- (d) The loop executes  $E - (|V| - 1)$  times because that is the number of edges we need to remove to get a MST with  $|V|$  vertices and  $|V| - 1$  edges. Each iteration takes  $O(|V| + |E|)$  times because of DFS. Therefore the total runtime is

$$O((|V| + |E|) \cdot (|E| - |V| + 1)).$$

## 2.11 Problem 5.23

The algorithm can be described as follows:

- Let  $E'$  be the set of all edges that touch a vertex in  $U$ .
- Build a MST  $T$  in  $G'(V - U, E - E')$ .
- For each node  $u$  in  $U$ , add the lightest edge that connects  $u$  and  $T$  to  $T$ .
- Return  $T$ .

## 2.12 Problem 5.29

[www.ece.northwestern.edu/~dda902/336/hw5-sol.pdf](http://www.ece.northwestern.edu/~dda902/336/hw5-sol.pdf)



# Chapter 3

## Dynamic programming

### 3.1 Problem 6.2

Let  $b[i]$  be the minimum total penalty for stopping at hotel  $a_i$ ,  $1 \leq i \leq n$ . We have

$$b[i] = \min_{1 \leq j < i} \{b[j] + (200 - (a[i] - a[j]))^2\}.$$

Also record the value  $j$  which yields  $\min_{1 \leq j < i} \{b[j] + (200 - (a[i] - a[j]))^2\}$  and set  $b[i].prev = b[j]$ . Backtrack from  $b[n]$  to get the sequence of hotels to stop by.

### 3.2 Problem 6.3

Let  $S[i]$  be the maximum total profit we get from building some restaurants in  $\{m_1, m_2, \dots, m_i\}$ . Consider 2 cases:

- (1) If restaurant  $m_i$  should not be built, then  $S[i] = S[i - 1]$ .
- (2) If restaurant  $m_i$  should be built, then let  $c_i$  be the maximum index  $j$  which yields  $m_i - m_j \geq k$ . We then have  $S[i] = p_i + S[c_i]$ .

Therefore in general,

$$S[i] = \max\{S[i - 1], p_i + S[c_i]\}.$$

To get the sequence of restaurants, keep an array  $R[n]$  such that  $R[i] = 1$  if restaurant  $i$  is built in the optimal solution, and  $R[i] = 0$  otherwise. When we calculate  $S[i]$ , if the max falls to case (1),  $R[i] = 0$ . Else,  $R[i] = 1$ . Output all the  $R[i]$ s that are 1.

### 3.3 Problem 6.4

Consider an array  $S[n]$  where  $S[i] = \text{true}$  if the substring  $s_1s_2 \dots s_i$  is a valid string, and  $\text{false}$  otherwise. We have  $S[1] = \text{dict}(s[1])$  and

$$\begin{aligned} S[i] = & (S[1] \ \&\& \ \text{dict}(s[2 \dots i]) \\ & || (S[2] \ \&\& \ \text{dict}(s[3 \dots i]) \\ & || \dots \\ & || (S[i - 1] \ \&\& \ \text{dict}(s[i \dots i]))), \end{aligned}$$

where  $s[j \dots i]$  is  $s_js_{j+1} \dots s_i$ .

### 3.4 Problem 6.6

Let the input string be  $x_1x_2 \dots x_n$ .

Let  $Z = \{a, b, c\}$  and let  $T[i, j] \subset Z$  be the set of the possible values that the product  $x_i x_{i+1} \dots x_j$  can yield with all possible parenthesizations.

We see that  $T[i, i] = x_i$  for all  $1 \leq i \leq n$ . We need to compute  $T[1, n]$ .

Define  $A \times B$  as  $\{a \cdot b \mid a \in A, b \in B\}$ .

We note that  $T[i, i+1] = T[i, i] \cup T[i+1, i+1]$  and  $T[i, i+2] = (T[i, i] \times T[i+1, i+2]) \cup (T[i, i+1] \times T[i+2, i+2])$  (to put it another way,  $abc$  can be written as  $(a)(bc)$  or  $(ab)(c)$ ).

We therefore see that we already have

$$T[1, 1], T[2, 2], T[3, 3], \dots,$$

from which we can calculate

$$T[1, 2], T[2, 3], T[3, 4], \dots,$$

from which we can calculate

$$T[1, 3], T[2, 4], T[3, 5], \dots$$

and eventually we can expand to  $T[1, n]$ , which is what we need to find.

In other words,

$$T[i, i+s] = \bigcup_{i \leq k < i+s} (T[i, k] \times T[k+1, i+s]).$$

The algorithm is as follows:

---

```

for i = 1 to n: T[i,i] = x[i].
for s = 1 to n-1:
  for i = 1 to n - s:
    T[i, i + s] = empty
    for k = 1 to i + s - 1:
      T[i, i + s] = T[i, i + s] UNION (T[i, k] * T[k+1,s])
If a is in T[1,n] return true. Else, return false.
```

---

### 3.5 Problem 6.7

Let the input string be  $x_1x_2 \dots x_n$ . Let  $T[i, j]$  be the length of the longest palindromic subsequence in  $x[i..j]$ . We have

$$\begin{cases} T[i,i] = 1 \\ T[i, i+1] = 2 \text{ if } x[i] = x[i+1] \text{ and } 0 \text{ if } x[i] \neq x[i+1] \\ T[i, j] = T[i+1, j-1] + 2 \text{ if } x[i] = x[j] \text{ and } \max\{T[i+1, j], T[i, j-1]\} \text{ else} \end{cases}$$

### 3.6 Problem 6.8

Let  $E[i, j]$  be the length of the largest common substring of  $x_1x_2 \dots x_i$  and  $y_1y_2 \dots y_j$  such that  $x_i = y_j$ .

We see that  $E[1, j] = 1$  if  $x_1 = y_j$  and 0 otherwise. Similarly,  $E[j, 1] = 1$  if  $y_1 = x_j$  and 0 otherwise.

In general, we have

$$E[i, j] = \begin{cases} E[i-1, j-1] + 1 & \text{if } x_i = y_j \\ 0 & \text{if } x_i \neq y_j \end{cases}$$

### 3.7 Problem 6.9

Let the input string be  $x[0..n-1]$  and the input breakpoint array be  $y[1..m]$ . Convert  $y$  to  $y[0..m+1]$  and let  $y[0] = -1, y[m+1] = n-1$ .

Let  $M(i, j)$  be

$$\begin{cases} M(i, i) = 0, \forall i : 0 \leq i \leq m+1 \\ M(i, i+1) = 0, \forall i : 0 \leq i \leq m+1 \\ M(i, j) = (y[j] - y[i]) + \min_{l:i < l < j} \{M(i, l) + M(l, j)\} \end{cases}$$

### 3.8 Problem 6.10

Let  $E[i, j]$  be the probability of obtaining exactly  $i$  heads when  $j$  coins  $c_1, c_2, \dots, c_j$  with head-probability  $p_1, p_2, \dots, p_j$  are tossed. We have

$$\begin{cases} E[0, 0] = 1 \\ E[0, j] = E[0, j-1] \cdot (1 - p_j) \quad \forall 1 \leq j \leq n \\ E[i, 0] = 0 \quad \forall 1 \leq i \leq k \\ E[i, j] = p_j \cdot E[i-1, j-1] + (1 - p_j) \cdot E[i, j-1] \end{cases}$$

### 3.9 Problem 6.11

Let  $E[i, j]$  be the longest common subsequence of  $x_1x_2 \dots x_j$  and  $y_1y_2 \dots y_j$ .

We have  $E[1, j] = 1$  if  $x_1 = y_j$ , for all  $1 \leq j \leq m$ . Similarly,  $E[j, 1] = 1$  if  $x_j = y_1$ , for all  $1 \leq j \leq n$ .

In general we have

$$E[i, j] = \begin{cases} E[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\{E[i-1, j] + E[i, j-1]\} & \text{if } x_i \neq y_j. \end{cases}$$

### 3.10 Problem 6.12

Let  $d[i, j]$  be the distance between point  $i$  and  $j$ . We have

$$\begin{cases} A[i, i] = 0 \\ A[i, i+1] = 0 \\ A[i, i+2] = 0 \\ A[i, j] = \min_{i < k < j} \{A[i, k] + A[k, j] + d[i, k] + d[k, j]\} \end{cases}$$

### 3.11 Problem 6.13

A sequence where a greedy approach would fail is

$$1000 \quad 2000 \quad 1.$$

Let  $E[i, j]$  be the maximum value the first player can have by picking cards from the set of cards  $s_i, s_{i+1}, \dots, s_j$ .

We see that  $E[i, j] = 0$  if  $i \leq j$ . In general,

$$E[i, j] = \max\{v_i + \min\{E[i+2, j] - v_{i+1}, E[i+1, j-1] - v_j\}, \\ v_j + \min\{E[i+1, j-2] - v_{j-1}, E[i+2, j-1] - v_{i+1}\}\}.$$