# Solutions of Introduction to Algorithms

Saman Saadi

# Contents

# Chapter 1

# Dynamic Programming

## 1.1 Rod cutting

**Exercise 2** No it cannot always produce an optimal solution. Consider the following example.

| $l_i$ | 1 | 2 | 3 |
|---|---|---|---|
| $p_i$ | 1 | 50 | 72 |
| $\frac{p_i}{l_i}$ | 1 | 25 | 24 |

For n = 3 the greedy approach cut the rod in 2 pieces. The length of one of them is 2 and the other's is 1. So the profit is $50\$ + 1\$ = 51\$$. But the optimal solution is to keep the rod intact so the profit is 72$.

**Exercise 3** We can keep the rod intact so we don't need to incur the fixed cost c or we can have at least one cut. We need to choose the best solution among all of them:

$$r(i) = \begin{cases} \max\limits_{1 \le k < n} (p_i, r(i-k) + p_k - c) & i > 0 \\ 0 & i = 0 \end{cases}$$

So the solution is $r(n)$. We have n distinct subproblem. In each step we need to choose between keeping the rod intact or have at least one cut which divide the rod into two pieces. The length of one of them is k and the other's $n - k$. We don't know the exact value of k so we need to try all possible values. This can be done in $O(n)$. Therefore the overall running time is $O(n^2)$

```
function F(p, n, c)
    let r[0..n] be a new array
    r[0] ← 0
    for j from 1 to n do
        q ← p[j]
        for i from 1 to j − 1 do
            q = max(q, r[j − i] + p[i] − c)
        end for
        r[j] = q
    end for
    return r[n]
end function
```

## 1.2 Matrix-chain multiplication

**Exercise 4** I've used the following equations:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1.1}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.2}$$

Each node of the graph represents a distinct sub-problem. Suppose we have two nodes v and u. There is an edge from v to u, if the solution of subproblem v is depended on subproblem u. In other words, there is an edge from m[i, j] to all m[i, k] and m[k + 1, j] for $i \leq k < j$.
Usually $|V|$ determines space complexity and $|V| + |E|$ time complexity. we know for every subprobelm m[i, j], $j \geq i$. Hence we have $n - i + 1$ subproblems which starts with $A_i$. So the number of vertices is:

$$\begin{aligned} |V| &= \sum_{i=1}^{n} n - i + 1 \\ &= \sum_{i=1}^{n} i \\ &= \frac{n(n+1)}{2} \end{aligned} \tag{1.3}$$

Hence the space complexity is is $O(n^2)$. We don't use all of the array cells when $j < i$. So we waste $\frac{n^2-n}{2}$ of allocated array. By analyzing lines 5 - 10 of MATRIX-CHAIN-ORDER pseudocode in the text book we can compute the number of edges. As you can see in line 10, m[i, j] is depends on two subproblem m[i, k] and m[k + 1, j]. We visit each distinct subproblem exactly once. So by counting the outdegree of each node we can calculate the number of edges in a

```
for l = 2 to n                    // l is the chain length
    for i = 1 to n - l + 1
        j = i + l - 1
        m[i, j] = ∞
        for k = i to j - 1
            q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
```

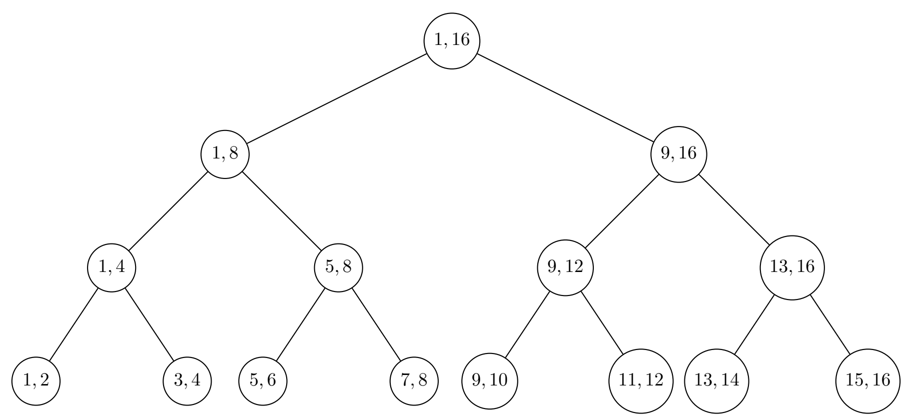
directed graph:

$$
\begin{aligned}
|E| &= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \sum_{k=i}^{i+l-2} 2 \\
&= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} 2(l-1) \\
&= 2\sum_{l=2}^{n}(n-l+1)(l-1) \\
&= 2\sum_{l=2}^{n}(n-(l-1))(l-1) \\
&= 2\sum_{l=1}^{n-1}(n-l)l \\
&= 2(\sum_{l=1}^{n-1} nl - \sum_{l=1}^{n-1} l^2) \\
&= 2(n\sum_{l=1}^{n-1} l - \sum_{l=1}^{n-1} l^2) \\
&= 2[n\frac{(n-1)n}{2} - \frac{(n-1)(n)(2n-1)}{6}] \\
&= n^2(n-1) - \frac{n(n-1)(2n-1)}{3} \\
&= \frac{3n^2(n-1) - n(n-1)(2n-1)}{3} \\
&= \frac{n(n-1)(3n-2n+1)}{3} \\
&= \frac{n(n-1)(n+1)}{3} \\
&= \frac{n(n^2-1)}{3} \\
&= \frac{n^3-n}{3}
\end{aligned} \tag{1.4}
$$

So the running time is $|V| + |E| = \frac{n^2+n}{2} + \frac{n^3-n}{3} = O(n^3)$

## 1.3 Elements of dynamic programming

**Exercise 2** Each node is filled with (p, r). p is the index of leftmost element and r is the index of rightmost element of array which the subprolem wants to sort. As you can see there is no overlapping between subproblems so dynamic programming is not a good idea for merge sort. In other words, we don't see a previously solved subproblem again and we only waste memory. As a general rule if the subproblem graph is a tree, dynamic programming cannot be applied.



# Chapter 2

# Amortized Analysis

## 2.1 Aggregate analysis

**Exercise 1** No it doesn't hold. The maximum number of pops, including multipop, is proportional to the number of previous push operations. If we can only push one item, the number of pushed elements is at most $n$. If we add a new operation named multipush, then the number of pushed items is at most $n \times k$. So the amortized cost is $O(k)$. For example we can have two operations. One is multipushing $10^9$ items and the other is multipopping $10^9$ items. It is obvious the total cost is not $O(n) = O(2)$ and is $O(nk) = O(2 \times 10^9)$.

**Exercise 2** The following pseudo-code explains how to implement DECREMENT. The worst case happens when we start with 0 and then decrements it

```
function DECREMENT(A)
    i = 0
    while i < A.length and A[i] == 0 do
        A[i] = 1
        i = i + 1
    end while
    if i < A.length then
        A[i] = 0
    end if
end function
```

to get $2^k - 1$ which all bits are set to 1 and then increments it to get 0. We repeat this loop until we have n operations. For $n = 4$ and $k = 3$ we have:

$$\begin{array}{c} 000 \\ 111 \\ 000 \\ 111 \end{array}$$

## 2.2 The accounting method

For another example of ”the accounting method” see exercise 5 of 3.1.

# Chapter 3

# Elementary Graph Algorithms

## 3.1 Representation of graphs

**Exercise 1** We know that $adj[u]$ is a list. Depends on the list implementation, it can take $O(1)$ to determine its size. In that case the running time for finding the out-degree of each vertex is $O(V)$. If we cannot determine size of the list in $O(1)$, then the overall running time of algorithm is $O(V + E)$.
The running time for finding in-degree of each vertex is $O(V + E)$.

**Exercise 3** For adjacency-matrix it takes $O(V^2)$ and for adjacency-list it takes $O(V + E)$.

**Algorithm 1** $G'$ using adjacency matrix

```
function TransposeGraph(G)
    Let G' be a new graph
    G' ← G
    for all u ∈ V do
        for all v ∈ V do
            G'.A[v][u] = G.A[u][v]
        end for
    end for
    return G'
end function
```

**Algorithm 2** $G'$ using adjacency list

```
1: function TransposeGraph(G)
2:     Let G' be a new graph
3:     G'.V = G.V
4:     for all u ∈ G.V do
5:         for all v ∈ G.Adj[u] do
6:             G'.Adj[v].insert(u)
7:         end for
8:     end for
9: end function
```

**Exercise 4** We create a new adjacency-list for $G'$ called $adj$. For each vertex $u$ in $G$, suppose $v$ is its neighbor. If $u \neq v$, then $adj[u].insert(v)$ and $adj[v].insert(u)$. If there are multiple edges between $u$ and $v$, we see $v$ as $u$'s neighbor more than once. So if the last element if $adj[v]$ is $u$, it means there are more than one edges between them so we shouldn't insert $v$ again. Traversing $G$ takes $O(V+E)$. Finding out there are more than one edge between two vertices is $O(1)$. So the overall running time is $O(V+E)$. Note that I supposed $G$ is also undirected.

```
1: function F(G)
2:     let G' be a new graph
3:     G'.V = G.V
4:     for all u ∈ G.V do
5:         for all v ∈ G.adj[u] do
6:             if u ≠ v ∧ G'.adj[v].last() ≠ u then
7:                 G'.adj[v].insert(u)
8:             end if
9:         end for
10:    end for
11:    return G'
12: end function
```

**Exercise 5** The running time of matrix-list implementation is $O(V^3)$. For analyzing the running time of adjacency-list implementation we can use amortized analysis. We use "accountant method".

> $\boldsymbol{in_u}$ **:** The number of edges that enter $u$
>
> $\boldsymbol{out_u}$ **:** The number of edges that leave $u$
>
> $\boldsymbol{e_u}$ **:** And edge from u to an arbitrary vertex $v \neq u$

We assign to all $e_u$ cost $c_{e_u} = 1 + in_u$. Because by traversing the graph, we visit $e_u$ at least once (line 6). For each edge that enters $u$ we visit or revisit $e_u$ (lines

7 - 8). We know that $\sum_{u=1}^{|V|} in_u + \sum_{u=1}^{|V|} out_u = 2|E|$. So we can easily calculate the total cost.

$$\begin{aligned}
\sum_{e_u \in E} c_{e_u} &= \sum_{e_u \in E} 1 + in_u \\
&= \sum_{e_u \in E} 1 + \sum_{u=1}^{|V|} in_u \\
&= |E| + \sum_{u=1}^{|V|} in_u \\
&\leq 3|E|
\end{aligned}$$

We execute line 6 at most $|E|$ times and lines 7 - 8 at most $2|E|$ times. So the total running time of algorithm using adjacency-list is $O(|V|+3|E|) = O(V+E)$.

**Algorithm 3** Finding square graph using matrix-list

```
function MakeSquareGraph(G)
    Let G' be a new Graph                          ▷ G.A[1..|V|, 1..|V|]
    for all u ∈ G.V do
        for all v ∈ G.V do
            G'.A[u][v] = G.A[u][v]                 ▷ 1-edge paths
            if G.A[u][v] = 1 then
                for all k ∈ G.V do
                    G'.A[u][k] = G.A[v][k]         ▷ 2-edge paths
                end for
            end if
        end for
    end for
end function
```

**Algorithm 4** Finding square graph using adjacency-list

```
function MakeSqureGraph(G)
    Let G′ be a new graph
    G′.V = G.V
    for all u ∈ G.V do
        for all v ∈ G.Adj[u] do
            G′.Adj[u].insert(v)                ▷ 1-edge paths
            for all w ∈ G.Adj[v] do
                G′.Adj[u].insert(w)            ▷ 2-edge paths
            end for
        end for
    end for
end function
```

**Exercise 6** Suppose $A$ is an adjacency matrix for $G$.

$$A[i,j] = \begin{cases} 1 & \text{i cannot be a universal sink} \\ 0 & \text{j cannot be a universal sink} \end{cases}$$

The following algorithm find the universal sink in $O(V)$. In each step we remove one vertex from all candidates for ”universal sink”. It takes $O(V)$ to have only one candidate. To determine that candidate is indeed a universal sink we need $O(2V)$ operations. So the overall running time of algorithm is $O(V) + O(2V) = O(V)$.

```
function getUniversalSink(G)
    A = G.A                                        ▷ A[1..|V|, 1..|V|]
    u ← 1
    while u ≤ |V| do
        v ← u + 1
        sink ← u               ▷ Vertices from sink to |V| can be universal sink
        while v ≤ |V| ∧ A[u, v] = 0 do
            v ← v + 1                              ▷ v cannot be a universal sink
        end while
        u ← v                             ▷ u to v − 1 cannot be a universal sink
    end while
    for c from 1 to sink − 1 do
        if A[sink, c] ≠ 0 then
            return "No universal sink"
        end if
    end for
    for r ∈ V − {sink} do
        if A[r, sink] ≠ 1 then
            return "No universal sink"
        end if
    end for
    return sink
end function
```

**Exercise 7** We know that $B$ is an $V \times E$ matrix which we show it as $B_{V\times E}$. By definition $B^T$ is an $E \times V$ matrix which we show it as $B^T_{E\times V}$. We define $P_{V\times V} = B_{V\times E} \times B^T_{E\times V}$.

$$\begin{aligned} p[i,j] &= \sum_{k=1}^{|E|} b[i,k] \times b^T[k,j] \\ &= \sum_{k=1}^{|E|} b[i,k] \times b[j,k] \end{aligned}$$

There are two cases:

1. $\boldsymbol{i \neq j}$ **:**

$$b[i,k] \times b[j,k] = \begin{cases} -1 \times 1 & k = (i,j) \in E \\ 1 \times -1 & k = (j,i) \in E \\ 0 & k = (u,v) \in E \wedge (u \neq i \vee v \neq j) \end{cases}$$

So $p[i,j]$ is the number of edges between $i$ and $j$.

2. $\boldsymbol{i = j}$ **:**

$$b[i,k] \times b[j,k] = b[i,k] \times b[i,k] = \begin{cases} -1 \times -1 & k = (i,u) \in E \\ 1 \times 1 & k = (u,i) \in E \\ 0 \times 0 & k = (u,v) \in E \wedge (u \neq i \wedge v \neq i) \end{cases}$$

In this case $p[i,i]$ is the sum of all edges that enter and leave the vertex $i$.

We can summarize the answer

$$p[i,j] = \begin{cases} \text{number of edges between i and j} & i \neq j \\ indegree(i) + outdegree(i) & i = j \end{cases}$$

## 3.2 Breadth-first search

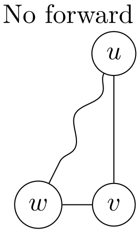
**BFS algorithm** For understanding the following algorithm see Lemma 22.3 from the textbook.

**Lemma 22.3:** Suppose that during the execution of BFS on graph G = (V, E), the queue Q contains the vertices $[v_1, v_2, \ldots, v_r]$, where $v_1$ is the head of Q and $v_r$ is the tail. Then $v_r.d \leq v_1.d$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \ldots, r-1$.

As you can see $(u,v)$ cannot be a forward edge (u is the ancestor of v) in BFS algorithm.

**Proof:** Suppose in BFS first we discover $u$ and $(u,v) \in E$ and $w$ is reachable from $u$. The shortest path from $u$ to $w$ is $\{v_1, v_2, \ldots, v_k\}$ for $k > 2$. See figure 3.1. In order $(u,v)$ be a forward edge, we must discover $w$ before $v$. According to BFS properties it's $v$ which is discovered first.

Figure 3.1: No forward edge in BFS



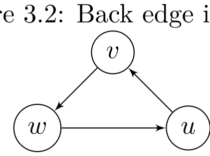
**Algorithm 5** BFS algorithm

```
function BFS(G, s)
    for all u ∈ G.V − {s} do
        u.color = WHITE
        u.d = ∞
        u.π = NIL
    end for
    s.color = GRAY
    s.d = 0
    s.π = NIL
    Q = ∅
    Enqueue(Q, s)
    while Q ≠ ∅ do
        u = Dequeue(Q)
        for all v ∈ G.adj[u] do
            if v.color == WHITE then                ▷ edge (u, v) is a tree edge
                v.color = GRAY
                v.d = u.d + 1
                v.π = u
                Enqueue(Q, v)
            else if u.d == v.d + 1 then                          ▷ back edge
                Assert(v.color == BLACK)
            else if v.d == u.d + 1 then                          ▷ cross edge
                Assert(v.color == GRAY)
            else if u.d == v.d then                              ▷ cross edge
                Assert(v.color == GRAY)
            end if
        end for
        u.color = BLACK
    end while
end function
```

### BFS characteristics

- Note that in BFS if $(u, v)$ is a back edge (v is the ancestor of u in BFS tree), then the color of v is black (in DFS it's gray). As you can see in figure 3.2, BFS starts from $v$ and when we navigate edge $(u, v)$ the color of $v$ is black. In other words $w.d = v.d + 1 \wedge u.d = w.d + 1$.

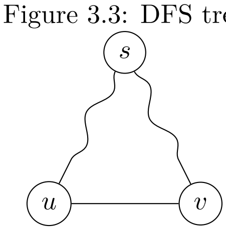
Figure 3.2: Back edge in BFS



- If the graph is undirected, after running BFS algorithms, for all back edges $(u, v)$, $v$ is the parent of $u$.
- If $(u, v)$ is a cross edge, the color of $v$ is gray. Unlike DFS in which the color of $v$ is black.
- In BFS both directed and undirected graphs can have cross edges.
- If the graph is undirected, a cross edge in BFS means we have an undirected cycle in graph.
- Since in an undirected graph $(u, v) \in E$ means $(v, u) \in E$, we always have back edges. In other words for all tree edges $(u, v)$, $(v, u)$ is a back edge. This is true for both BFS and DFS algorithms.
- It's better to use BFS for finding cycles in undirected graphs. If we encounter a cross edge, we have a cycle. For directed graphs, it's better to use DFS. If we have a back edge, it means we have a cycle.

**Exercise 7** We need to determine whether an undirected graph is bipartite or not. We can paint the vertices of a bipartite graph with two colors in such a way that no two adjacent vertices share the same color.
We can easily prove that if there is a cycle in graph in which the number of edges is odd, then the graph cannot be bipartite.
We can use BFS. We know that in BFS algorithm we can only have tree and back and cross edges (see 3.2). Note that in BFS we can have cross edges whether or not the graph is directed. We run BFS on an arbitrary vertex $s$. Suppose $u$ is reachable from $s$. If $u.d$ is even we color that vertex "blue" otherwise we color it "red". For tree edges we don't have any problem. Since in back edges in BFS there is a parent-child relationship between two vertices of an endge, we don't have any problem with back edges. We need to think about cross edges. We know that $\delta(s, u) = u.d$ which is the shortest path from $s$ to $u$.

Figure 3.3: DFS tree




**Lemma 22.3:** Suppose that during the execution of BFS on graph G = (V, E), the queue Q contains the vertices $[v_1, v_2, \ldots, v_r]$, where $v_1$ is the head of Q and $v_r$ is the tail. Then $v_r.d \leq v_1.d$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \ldots, r - 1$.

For more information about Lemma 22.3 see the textbook. Suppose $Q = \{v_1, v_2, \ldots, v_r\}$ as we defined it in Lemma 22.3. Note that if we encounter a cross edge, it connects two vertices (u, v) that $u \in Q \wedge v \in Q$. To be more specific, first we pop $u$ from queue and then we find out $(u, v)$ is a cross edge. In that moment $u \notin Q \wedge v \in Q$.

Suppose $(u, v)$ is a cross edge. According to Lemma 22.3 from the textbook, $u.d \leq v.d \wedge v.d \leq u.d + 1$. If $v.d = u.d + 1$, then $u$ and $v$ have different colors. So we only need to consider $u.d = v.d$. In that case both $u$ and $v$ have the same color and we need to prove that this graph cannot be bipartite. When we have a cross edge in an undirected graph, it means that we have a cycle (see figure 3.3). The number of edges in this cycle is $u.d + v.d + 1 = 2 \times u.d + 1$ which is odd. So the graph cannot be bipartite. Note that the graph can have more than one connected component so it is possible we need to run BFS more than once.

**Algorithm 6** Determining whether a graph is bipartite or not

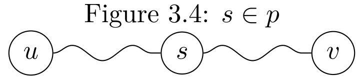
```
function IsBipartiteGraph(G)
    for all u ∈ G.V do
        u.color = WHITE
        u.d = ∞
        u.π = NIL
    end for
    for all u ∈ G.V do
        if u.color == WHITE ∧ BFS(G, u) == FALSE then
            return FALSE
        end if
    end for
    return TRUE
end function
```

```
function BFS(G, s)
    s.color = GRAY
    s.d = 0
    Q = ∅
    Enqueue(Q, s)
    while Q ≠ ∅ do
        u = Dequeue(Q)
        for all v ∈ G.adj[u] do
            if v.color == WHITE then            ▷ edge (u, v) is a tree edge
                v.color = GRAY
                v.d = u.d + 1
                v.π = u
                Enqueue(Q, v)
            else if u.d == v.d + 1 then         ▷ v.color is Black
                continue
            else if v.d == u.d + 1 then         ▷ v.color is GRAY (cross edge)
                Continue
            else if u.d == v.d then             ▷ v.color is GRAY (cross edge)
                return FALSE
            end if
        end for
        u.color = BLACK
    end while
    return TRUE
end function
```

**Exercise 8** Suppose the maximum distance is path $p = (v_0, v_1, \ldots, v_k)$ in which $u = v_0$ and $v = v_k$. Consider an arbitrary vertex $s$. We know that there is exactly one path between every two vertices in a tree. We have two cases.

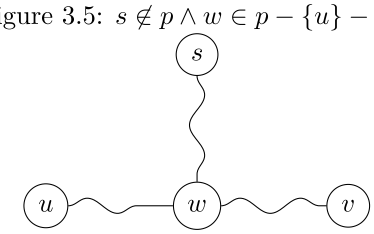
1. $\boldsymbol{s \in p}$

Figure 3.4: $s \in p$




2. $\boldsymbol{s \notin p}$ **:** In this case there is exactly one path between $s$ and $w \in p - \{u\} - \{v\}$. For example if $w = u$ then the diameter is between $s$ and $v$.

Figure 3.5: $s \notin p \wedge w \in p - \{u\} - \{v\}$




If we run BFS on $s$, then $\max\limits_{x \in G.V}(x.d)$ belongs to either $u$ or $v$. Otherwise the diameter is not between $u$ and $v$. Without loss of generality suppose it is $u.d = \max\limits_{x \in G.V}(x.d)$. Then we run another BFS on $u$ to get $v$ in a similar manner. The running time of algorithm is $O(2V + 2E) = O(V + E)$. Since in a tree $|E| = |V| - 1$ the running time is $O(V)$.

**function** FINDDIAMETER(G)
  Let $s$ be an arbitrary vertex such that $s \in G.V$
  INITBFS(G)
  u = BFS(G, s)
  INITBFS(G)
  v = BFS(G, u)
  **return** $u, v, v.d$
**end function**

```
function BFS(G, s)
    s.color = GRAY
    s.d = 0
    Q = ∅
    Enqueue(Q, s)
    max = −∞
    while Q ≠ ∅ do
        u = Dequeue(Q)
        for all v ∈ G.adj[u] do
            if v.color == WHITE then
                v.color = GRAY
                v.d = u.d + 1
                if v.d > max then
                    max = v.d
                    z = v
                end if
                v.π = u
                Enqueue(Q, v)
            end if
        end for
        u.color = BLACK
    end while
    return z
end function
```

**exercise 9** This undirected graph is equivalent to a directed graph which for all $u, v \in V$, $(u, v), (v, u) \in E$. We can use a modified version of DFS. Because we have both edges $(u, v)$ and $(v, u)$, we don't have "cross edges". We need to choose between "forward edges" or "back edges". In the following algorithm we use "forward edges" and skip "back edges".

```
function DFS(G, u)
    u.color ← Gray
    paths ← ϕ
    for all v ∈ G.Adj[u] do
        if v.color = White then                         ▷ Tree edge
            paths ← {(u, v)} ∪ DFS(G, v) ∪ {(v, u)}
        else if v.color = Black then                    ▷ Forward edge
            paths ← paths ∪ {u, v} ∪ {v, u}
        end if
    end for
    u.color ← Black
    return paths
end function
```

## 3.3 Depth-first search

### Edge classification

**Tree edges** are edges in the depth-first forest $G_\pi$. Edge $(u, v)$ is a tree edge if v was discovered by exploring edge $(u, v)$

**Back edges** are those edges $(u, v)$ connecting a vertex $u$ to **an ancestor** $v$ in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
Note that a directed graph is acyclic if and only if a depth-first search yields no back edges.

Undirected graphs are tricky. Since $(u, v) \in E \wedge (v, u) \in E$, $(u, v)$ is a back edge if and only if $(v, u)$ is a tree edge.

**Forward edges** are those nontree edges $(u, v)$ connecting a vertex $u$ to a descendant $v$ in a depth-first tree.

**Cross edges** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

The DFS algorithm has enough information to classify some edges as it encounters them. The key idea is that when we first explore an edge $(u, v)$, the color of vertex $v$ tells us something about the edge:

1. **WHITE** indicates a **tree** edge

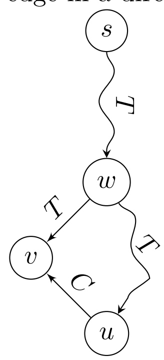
2. **GRAY** indicates a **back** edge

3. **BLACK** indicates a **forward or cross** edge. It's **forward** edge If $u.d < v.d$ and it's **cross** edge if $u.d > v.d$.

Suppose $s$ is the root of DFS or BFS tree.

- **Tree edge**
  - Directed graph
    * DFS: We can have tree edges
    * BFS: We can have tree edges
  - Undirected graph
    * DFS: We can have tree edges
    * BFS: We can have tree edges
- **Forward edge**
  - Directed graph
    * DFS: We can have forward edges
    * BFS: We can't have forward edges. Refer to the beginning of chapter 3.2
  - Undirected graph
    * DFS: We can't have forward edges
    * BFS: We can't have forward edges
- **Back edge**
  - Directed graph
    * DFS: We can have back edges
    * BFS: We can have back edges.
  - Undirected graph
    * DFS: We can have back edges
    * BFS: We can have back edges. Suppose $(u, v)$ is a back edge, then $v$ is the parent of $u$ for more information see the beginning of chapter 3.2.
- **Cross edge**
  - Directed graph
    * DFS: We can have cross edges
    * BFS: We can have cross edges. refer to the beginning of chapter 3.2. For an example see figure 3.6
  - Undirected graph
    * DFS: We can't have cross edges
    * BFS: We can have cross edges. See the beginning of chapter 3.2.

Figure 3.6: BFS – cross edge in a directed graph. $v.d \leq u.d + 1$




**Exercise 1** You can use the following facts. Suppose we have edge (u, v) and we consider loops as back edges (If back edge is a loop then $v.d = u.d \wedge v.f = u.f$).

**Tree edge:** $u.d < v.d < v.f < u.f$

**Forward edge:** $u.d < v.d < v.f < u.f$

**Back edge:** $v.d \leq u.d < u.f \leq v.f$

**Cross edge:** $v.d < v.f < u.d < u.f$

Note that when the graph is undirected we don't have "forward edge" and "cross edge". Because they are equivalent to "back edge" and "tree edge" respectively.

Table 3.1: Directed graph

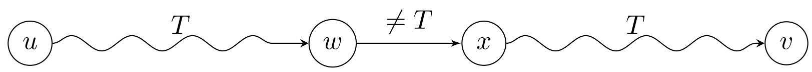
| | white | gray | black |
|---|---|---|---|
| white | tree, back, forward, cross | back, cross | cross |
| gray | tree, forward | tree, back, forward | tree, forward, cross |
| black | impossible | back | tree, back, forward, cross |

Table 3.2: Undirected graph

| | white | gray | black |
|---|---|---|---|
| white | tree, back | tree, back | impossible |
| gray | tree, back | tree, back | tree, back |
| black | impossible | tree, back | tree, back |

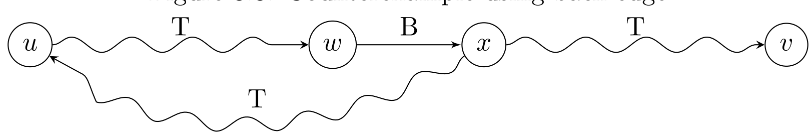
**Excercise 8** We need to show some examples which there is only one path from $u$ to $v$ and at least of one the edges in this path is non-tree edge. Without loss of generality, suppose in this path except $e = (w, x)$ which is a non-tree edge, all other edges are tree ones. We consider all possible types.

Figure 3.7: Edge e = (w, x) is a non-tree edge




- **Forward Edge:** If $(w, x)$ is forward ege, then $w$ is an ancestor of $x$ which leads to $v$ be a descendant of $u$. So it cannot be a forward edge
- **Cross edge:** If $(w, x)$ is a cross edge, then $x$ finishes before the discovery of $w$. In other words, all reachable vertices from $x$, including $v$, will be discover before $w$ and $u$. So it cannot be a cross edge
- **Back edge:** Consider the following example which the root of DFS tree is vertex $x$ and it discover $u$ before $v$.

Figure 3.8: Counterexample using back edge




**Excercise 9** Suppose we have edge (u, v) and we consider loops as back edges.

$$\begin{aligned}
\textbf{Tree edge: } & u.d < v.d < v.f < u.f \\
\textbf{Forward edge: } & u.d < v.d < v.f < u.f \\
\textbf{Back edge: } & v.d \leq u.d < u.f \leq v.f \\
\textbf{Cross edge: } & v.d < v.f < u.d < u.f
\end{aligned}$$

As you can see only in cross edge the discovery of one endpoint is after the other finished. We need to prove that if there is a path from $u$ to $v$, it is possible to have $v.d > u.f$. In other words, edge $(v, u)$ is a cross edge and there is a path from $u$ to $v$ in which there is at least one edge which is not a tree edge. We can make counterexample even simpler by removing the cross edge. Note that the root of DFS tree is vertex $s$. In general if there is a cycle from $s$ to $u$ and $u$ to $s$, then it is possible $v.d > u.f$.

Figure 3.9: Counterexample using cross edge

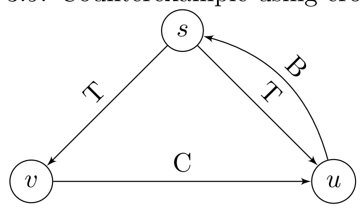



Figure 3.10: Counterexample without cross edge

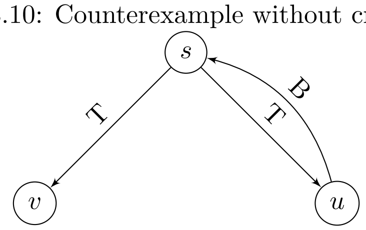
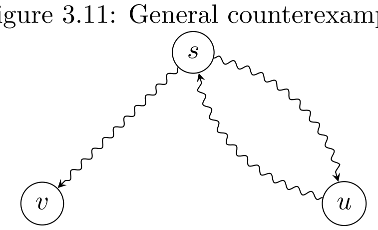
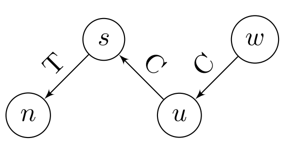



Figure 3.11: General counterexample




**Exercise 10** In an undirected graph forward edges are equivalent to back edges and cross edges are equivalent to tree edges. Although the following modification works for both directed and undirected graph, you can remove portion of code that is related to "forward" and "cross" edges to save space.

**Exercise 11** If both incoming and outgoing edges are cross, that happens. Consider the following example. Suppose DFS starts at $s$, then $u$ and finally at $w$.



```
function DFS-VISIT(G, u)
    time = time + 1
    u.d = time
    u.color = GRAY
    for all v ∈ G.adj[u] do
        if v.color == WHITE then                ▷ edge (u, v) is a tree edge
            PRINT-EDGE(u, v, TREE)
            v.π = u
            DFS-VISIT(G, v)
        else if v.color == GRAY then            ▷ edge u, v is a back edge
            PRINT-EDGE(u, v, BACK)
        else if u.d < v.d then
            PRINT-EDGE(u, v, FORWARD)
        else
            PRINT-EDGE(u, v, CROSS)
        end if
    end for
    u.color = BLACK
    time = time + 1
    u.f = time
end function
```

**Exercise 12** There exists at least one path between every two vertices in a connected component. So if we perform DFS on an arbitrary vertex $u$, it will discover all reachable vertices from $u$. After DFS visit all vertices in $u$'s connected component it terminates. So the number of DFS trees is equal to the number of connect components in undirected graph. This is not true for directed graphs. Consider figure 3.12. If we start from $u$ we only have one DFS tree. But if we start from $v$, $x_1$, $x_2$, $x_3$, $x_4$ and finally $u$, the number of DFS trees are equal to the number of vertices. In other words, all DFS trees are consist of only a vertex and we don't have any tree edges.

**Algorithm 7** Connected components in an undirected graph

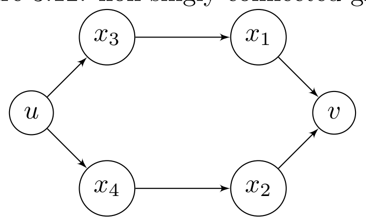
```
function DFS(G)
    for all u ∈ G.V do
        u.color = WHITE
        u.π = NIL
    end for
    time = 0
    ccn = 0                    ▷ ccn is the number of connected components
    for all u ∈ G.V do
        if u.color == WHITE then
            ccn = ccn + 1
            DFS-VISIT(G, u)
        end if
    end for
end function

function DFS-VISIT(G, u)
    time = time + 1
    u.d = time
    u.cc = ccn
    u.color = GRAY
    for all v ∈ G.adj[u] do
        if v.color == WHITE then
            v.π = u
            DFS-VISIT(G, v)
        end if
    end for
    u.color = BLACK
    time = time + 1
    u.f = time
end function
```

**Exercise 13** It is obvious that we should only have tree and back edges. It is important to start from the right vertex. Consider figure 3.12. There is exactly two distinct paths between $u$ and $v$. If we start the DFS from $u$, in the first run we can detect that the graph is not singly connected. I thought I can design an $O(V + E)$ algorithm to solve this problem. I was wrong.

**Wrong idea** Start DFS from an arbitrary vertex $s$. If you found ”forward” or ”cross” edges then it is not singly connected so the algorithm terminates. After DFS finished, it is possible we have unvisited vertices.

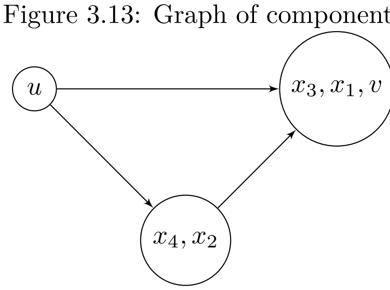
**component:** All undiscovered vertices which will be discover in one DFS run. For example if we run DFS from $x_3$ in figure 3.12, $x_3$, $x_1$ and $u$ are belong to the same component.

Figure 3.12: non singly connected graph




We can determine and number the components (similar to connected components). If we find "forward" or "cross" edges within each component, the algorithm terminates. Note that "cross" edges between components is not trivial. We can have singly connected graph which has at least one cross edge between its components. We can create a new graph which its vertices are the components of input graph and its edges are the cross edges between components. Suppose in figure 3.12 we run DFS first on $x_3$, then $x_4$ and finally $u$. You can see the result in figure 3.13. This algorithm is not always correct. Suppose we run DFS first on $v$, $x_1$, $x_2$, $x_3$, $x_4$ and finally $u$. As you can see we don't have any tree edges and the graph of components is same as original one.

Figure 3.13: Graph of components



# Chapter 4

# Minimum Spanning Trees

For questions similar to Prim's minimum spanning tree see "Variants" in chapter 5.

## 4.1 Problems

**Problem 3 Bottleneck spanning tree**

**a.** Consider an arbitrary MST $T$. Suppose the maximum-weight edge in $T$ is $e$. If we remove that edge we have a forest of two trees $C_1$ and $C_2$. Now consider MST $T'$ whose maximum-weight edge, $e'$, is less that $e$. There should be an edge in $T'$ that connects $C_1$ to $C_2$. The weight of that edge should be less than $e$. In other words, $e$ is not a light edge which contradicts $T$ is an MST.

**b.** We remove all edges in $G.E$ which their weight are higher than $b$. We called the modified graph $G_b$. It is obvious that $G_b.V = G.V$ and $G_b.E = \{(u, v) \in G.E : w(u, v) \leq b\}$. If $G_b$ remains connected then every spanning tree of $G_b$ doesn't have an edge whose weight is greater than b. The running time of this algorithm is $O(V + E)$. Since G should be a connected graph, $|E| \geq |V| - 1$. So we can say the running time is $O(E)$.

```
function VALID-BST-VALUE(G, b)
    G′ = REMOVE-EDGES(G, b)
    InitDFS(G′)
    c = 0                                  ▷ The number of connected components
    for all u ∈ G′.V do
        if u.color == WHITE then
            c = c + 1
            DFS(G′, u)
        end if
    end for
    return (c == 1)
end function
```

```
function REMOVE-EDGES(G, b)
    G_b.V = G.V
    G_b.E = ∅
    for all (u, v) ∈ G.E do
        if (u, v).c ≤ b then                          ▷ (u, v).c ≡ w(u, v)
            G_b.E = G_b.E ∪ {(u, v)}
        end if
    end for
    return G_b
end function
```

**c.** We need some definitions:

- $\boldsymbol{G_b}$ **:** a new sub-graph of $G$ which $G_b.V = G.V$ and $G_b.E = \{(u,v) \in G.E : w(u,v) \leq b\}$
- $\boldsymbol{comp(u)}$ **:** Suppose connected components $c_1, c_2, \ldots, c_k$ make the Graph $G_b$ and vertex $u$ belongs to the i$^{\text{th}}$ component. Then $comp(u) = i$
- $\boldsymbol{G'_b}$ **:** Suppose $C = \{c_1, c_2, \ldots, c_k\}$ is the set of connected components of $G_b$. Then $G'_b.V = C$ and $G'_b.E = \{(u,v) \in G.E : comp(u) \neq comp(v)\}$. It is possible that there are more than one edge between $c_i$ and $c_j$. In this case we choose the light edge (minimum weight)

We want to find $b_m = \min\limits_{b \in G.E \wedge |G'_b.V|=1}(b)$. In other words, we want to find the minimum of $b \in G.E$ which $G_b$ is a connected graph. Consider an arbitrary edge $(u,v)$. Suppose $w(u,v) = b$ There are two cases:

1. $\boldsymbol{G_b}$ **is connected:** It means $b_m < b$. More precisely, $b_m \in G_b.E$
2. $\boldsymbol{G_b}$ **is not connected:** It means $b_m \geq b$. More precisely, $b_m \in G'_b.E$.

Consider set $W = w(u,v) : (u,v) \in G.E$. We can find the median of $W$ in $O(E)$ by ”median of medians” algorithm. After finding the median, we can divide the edges into two equal sets: $G_b.E$ and $G'_b.E$. In each step we eliminate half of edges. For simplicity we assume edge $(u, v)$ has an attribute $c$ such that $(u, v).c = w(u, v)$.

```
function BST(G)
    if |G.E| == 1 then
        return G.E
    end if
    m = FIND-MEDIAN(G.E)                                ▷ O(E)
    if VALID-BST-VALUE(G, m) then
        G_b = REMOVE-EDGES(G, m)                        ▷ O(E)
        R = BST(G_b)
    else
        G'_b = MAKE-G'_b(G)                             ▷ O(E)
        R = BST(G'_b)
    end if
    return R
end function
```

```
function MAKE-G'_b(G)
    C, comp = CC(G)
    G'_b.V = C
    for all (u, v) ∈ G.E do
        if comp[u] ≠ comp[v] then
            if (comp[u], comp[v]) ∉ G'_b.E then
                (comp[u], comp[v]).c = (u, v).c
                G'_b.E = G'_b.E ∪ {(comp[u], comp[v])}
            else if (u, v).c < (comp[u], comp[v]).c then
                (comp[u], comp[v]).c = (u, v).c
            end if
        end if
    end for
    return G'_b
end function
```

```
function CC(G)
    Let comp[1..G.V] be a new array
    C = ∅                                  ▷ The set of connected components
    c = 0                                  ▷ The number of connected components
    INIT-DFS(G)
    for all u ∈ G.V do
        if u.color == WHITE then
            c = c + 1
            C = C ∪ {c}
            DFS(G, u, c, comp)
        end if
    end for
    return C, comp
end function
```

```
function DFS(G, u, c, comp)
    u.color = GRAY
    comp[u] = c
    for all v ∈ G.adj[u] do
        if v.color == WHITE then
            DFS(G, v, c, comp)
        end if
    end for
end function
```

So the total running time of algorithm is $O(E)$:

$$
\begin{aligned}
T(E) &= T(\frac{E}{2}) + O(E) \\
&= O(E) + O(\frac{E}{2}) + O(\frac{E}{4}) + \cdots + O(\frac{E}{2^i}) + \cdots + O(1) \\
&= O(\frac{E}{2^0}) + O(\frac{E}{2^1}) + \cdots + O(\frac{E}{2^{log_2 E}}) \\
&= \frac{E(\frac{1}{2})^{log_2 E+1} - E}{\frac{1}{2} - 1} \\
&= 2E - 1
\end{aligned}
$$

For the analysis of run-time we assumed $G$ is connected so $|E| \geq |V| - 1$.

# Chapter 5

# Single-Source Shortest Path

**Variants**

1. **Single-source shortest-path problem:**

   (a) Given unweighted undirected graph $G$ and vertices $s$ and $t$. Find the number of shortest path between $s$ and $t$.

   **Solution** It's similar to all-pair shortest path problem but we solve it through Single-source shortest-path problem. Since the graph is unweighted we use BFS. We define a new attirbute $u.num$ which is the number of shortest path from $s$ to $u$. Because the graph is unweighted we only deal with tree and back edges. It is possible a tree edge be another shortest path which we need to consider it.

```
function FIND-SHORTEST-PATH-COUNT(G, s, t)
    INIT-BFS(G)
    s.d = 0
    s.num = 1
    s.color = GREY
    Q = ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅ do
        u = DEQUEUE(Q)
        for all v ∈ G.adj[u] do
            if u.color == WHITE then                ▷ (u, v) is a tree edge
                v.color = GREY
                v.d = u.d + 1
                v.num = u.num
                ENQUEUE(G, v)
            else if u.d + 1 == v.d then             ▷ (u, v) is a back edge
                v.num = v.num + u.num
            end if
        end for
    end while
    return t.num
end function
```

```
function INIT-BFS(G)
    for all u ∈ G.V do
        u.d = ∞
        u.color = WHITE
        u.num = 0
    end for
end function
```

(b) Consider two arbitrary vertices $u$ and $v$. Suppose there is path $p$ between $u$ and $v$. We define $m = \min\limits_{(u,v)\in p} (w(u,v))$ and $M = \max\limits_{(u,v)\in p} (w(u,v))$.

i. Find a path between $u$ and $v$ which has the maximum $m$ among all possible paths
   - **Solution** We can use an algorithm similar to Dijkstra's shortest path for solving this problem

ii. Find a path between $u$ and $v$ which has the minimum $M$ among all possible paths
   - **Solution** We can use an algorithm similar to Dijkstra's shortest path for solving this problem

iii. Find a path between $u$ and $v$ which has the maximum $M$ among all possible paths

- **Solution** We can't use an algorithm similar to Dijkstra's shortest path. Instead we use an algorithm similar to Bellman-Ford shortest path. It is possible the path has at least one cycle.

iv. Find a path between $u$ and $v$ which has the minimum $m$ among all possible paths

- **Solution** We can't use an algorithm similar to Dijkstra's shortest path. Instead we use an algorithm similar to Bellman-Ford shortest path. It is possible the path has at least one cycle.

(c) acm-icpc World Finals 2002 question C, Crossing the Desert: You can see the problem statement in "DESERT Problem in SPOJ" and "Problem 1011 in UVa" online judges.
In this problem, you will compute how much food you need to purchase for a trip across the desert on foot.
At your starting location, you can purchase food at the general store and you can collect an unlimited amount of free water. The desert may contain oases at various locations. At each oasis, you can collect as much water as you like and you can store food for later use, but you cannot purchase any additional food. You can also store food for later use at the starting location. You will be given the coordinates of the starting location, all the oases, and your destination in a two-dimensional coordinate system where the unit distance is one mile.
For each mile that you walk, you must consume one unit of food and one unit of water. Assume that these supplies are consumed continuously, so if you walk for a partial mile you will consume partial units of food and water. You are not able to walk at all unless you have supplies of both food and water. You must consume the supplies while you are walking, not while you are resting at an oasis. Of course, there is a limit to the total amount of food and water that you can carry. This limit is expressed as a carrying capacity in total units. At no time can the sum of the food units and the water units that you are carrying exceed this capacity.
You must decide how much food you need to purchase at the starting location in order to make it to the destination. You need not have any food or water left when you arrive at the destination. Since the general store sells food only in whole units and has only one million food units available, the amount of food you should buy will be an integer greater than zero and less than or equal to one million.

**Input** The first line of input in each trial data set contains n ($2 \leq n \leq 20$), which is the total number of significant locations in the desert, followed by an integer that is your total carrying capacity in units of food and water. The next n lines contain pairs of integers

that represent the coordinates of the n significant locations. The first significant location is the starting point, where your food supply must be purchased; the last significant location is the destination; and the intervening significant locations (if any) are oases. You need not visit any oasis unless you find it helpful in reaching your destination, and you need not visit the oases in any particular order.
The input is terminated by a pair of zeroes.

**Output** For each trial, print the trial number followed by an integer that represents the number of units of food needed for your journey. Use the format shown in the example. If you cannot make it to the destination under the given conditions, print the trial number followed by the word "Impossible."
Place a blank line after the output of each test case.

**Example**

**Input**
4 100
10 -20
-10 5
30 15
15 35
2 100
0 0
100 100
0 0

**Output**
Trial 1: 136 units of food
Trial 2: Impossible

**Solution:** First we make question simpler. So we suppose it is impossible to leave food on oases or starting location and possibly return and collect them.
We can model this problem to an undirected graph. The vertices are the starting location, oases and the destination. There is an edge between $u$ and $v$, if the amount of required food and water doesn't exceed $C$.

- $\boldsymbol{f_{u,v}}$ **:** The amount of required food from $u$ to $v$
- $\boldsymbol{a_{u,v}}$ **:** The amount of required water from $u$ to $v$

We define weight function $w$:

$$w(u,v) = \begin{cases} f_{u,v} & f_{u,v} + a_{u,v} \leq C \\ \infty & f_{u,v} + a_{u,v} > C \end{cases}$$

Unlike food, we can pick up water in every oases. So we need to order all required food in the starting location. Because we cannot leave food anywhere in the desert, the final path should be simple. Otherwise we have at least one cycle. If we remove that cycle we obtain an equivalent path which required less food. Suppose path $p$ which connects the starting location to the target is an optimal path. We define $a_m = \max\limits_{(u,v)\in p}(a(u,v))$. We called $p$ a valid path if $\sum_{(u,v)\in p} f(u,v) + a_m \leq C$. The required food for $p$ must be minimum among all valid paths from the starting location to the destination. We can solve this problem with a greedy algorithm similar to Dijkstra's shortest path. $u.d$ is the amount of required food from the starting location to $u$. We define a new attribute $u.a_m$ which we described it before. $s$ is the starting location and $t$ is the target location. The running time of algorithm is like Dijkstra's shortest path which can be $O(E \log V)$.

```
function DESERT(G, w, C, s, t)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅ do
        u = EXTRACT-MIN(G)
        S = S ∪ {u}
        for all v ∈ G.Adj[u] do
            RELAX(u, v, w, C)
        end for
    end while
    if t.d < ∞ then
        return t.d
    else
        "IMPOSSIBLE"
    end if
end function
```

```
function RELAX(u, v, w, C)
    m = max(w(u, v), u.a_m)
    food = u.d + w(u, v)
    if (food + m) ≤ C ∧ food < v.d then
        v.d = food
        v.a_m = m
        v.π = u
    end if
end function
```

```
function INITIALIZE-SINGLE-SOURCE(G, s)
    for all v ∈ G.V do
        v.d = ∞
        v.π = NIL
        v.a_m = −∞
    end for
    s.d = 0
end function
```

2. **Single-destination shortest-path problem:**

   (a) You are given flight schedules between a set of $n$ cities. For each pair of cities $(i, j)$ between which there is a direct flight, you are given the pair $(d_{ij}, a_{ij})$, the departure and arrival time of the flight from city $i$ to city $j$. Assume that there is at most one flight from city $i$ to city $j$ per day. Suppose you start at city $A$ and want to reach city $B$. You have an important meeting in city $B$ that you need to attend, and you need to reach city $B$ latest by time $t$. Give an algorithm that outputs a possible sequence of flights you could take starting from city $A$ as late as possible and reaching city $B$ before time $t$, with at least one hour layover between any two consecutive flights.

**Solution:** We don't know which flight in $A$ we should choose. We can solve the problem if we consder flights in $B$. Given the graph $G$, we need to change that to graph $G'$ such that $G'.V = G.V$ and $G'.E = \{(u, v) : (v, u) \in V.E\}$. Hence if $(i, j) \in G'.E$, there is a flight from $j$ to $i$ in which the departure time is $d_{ji}$ and arrival time is $a_{ji}$. In $B$ we only need to consider all flights $C = \{(B, u) \in G'.E : a_{uB} \leq t\}$ and choose the edge with latest departure time $(\max\limits_{(B,u)\in C}(d_{uB}))$. Because if we arrive at $u$, flight $(u, B)$ has the latest departure time and it doesn't make any sense to go from $u$ to $B$ through other intermediate vertices (this description is not entirely correct see figure 5.1 for more information). So we add edge $(u, B)$ as an optimal answer between $u$ and $B$ (in $G'$ we should say edge $(B, u)$). This algorithm is similar to Prim's minimum spanning tree. We can call it "Single-destination latest-departure problem". We calculate the best possible sequence of flights from $u$ to $B$. Eventually we calculate an optimal path from $A$ to $B$. By "best" we mean the departure time of the first flight is as late as possible and the arrival time is at most $t$ and there is a layover of at least one hour between two consecutive flights. $u.d$ store the latest possible departure from $u$ to $B$. We add a dummy flight from $B$ to an unknown place with departure $t+1$ to discard all those flights to $B$ with arrival time greater than $t$. $Q$ contains all those vertices which we don't know yet an optimal flight sequence from them to $B$. On the other hand, $S$ contains all those vertices which we found out an optimal flight sequence from them to $B$.

```
function SCHEDULING(G, A, B, d, a, t)
    G′ = REVERSE-GRAPH-EDGES(G)
    INITIALIZE-SINGLE-SOURCE(G′, s)
    B.d = t + 1                ▷ To make sure we arrive at B no more than t
    S = ∅
    Q = G′.V
    while Q ≠ ∅ do
        u = EXTRACT-MAX(G′)
        S = S ∪ {u}
        for all v ∈ G′.Adj[u] do
            RELAX(u, v, d, a)
        end for
    end while
    if A.d > −∞ then
        u = A
        while u ≠ B do
            PRINT(u, u.π)
            u = u.π
        end while
    else
        PRINT("IMPOSSIBLE")
    end if
end function
```

```
function RELAX(u, v, d, a)
    if (a_vu + 1) ≤ u.d ∧ d_vu > v.d then
        v.d = d_vu
        v.π = u
    end if
end function
```

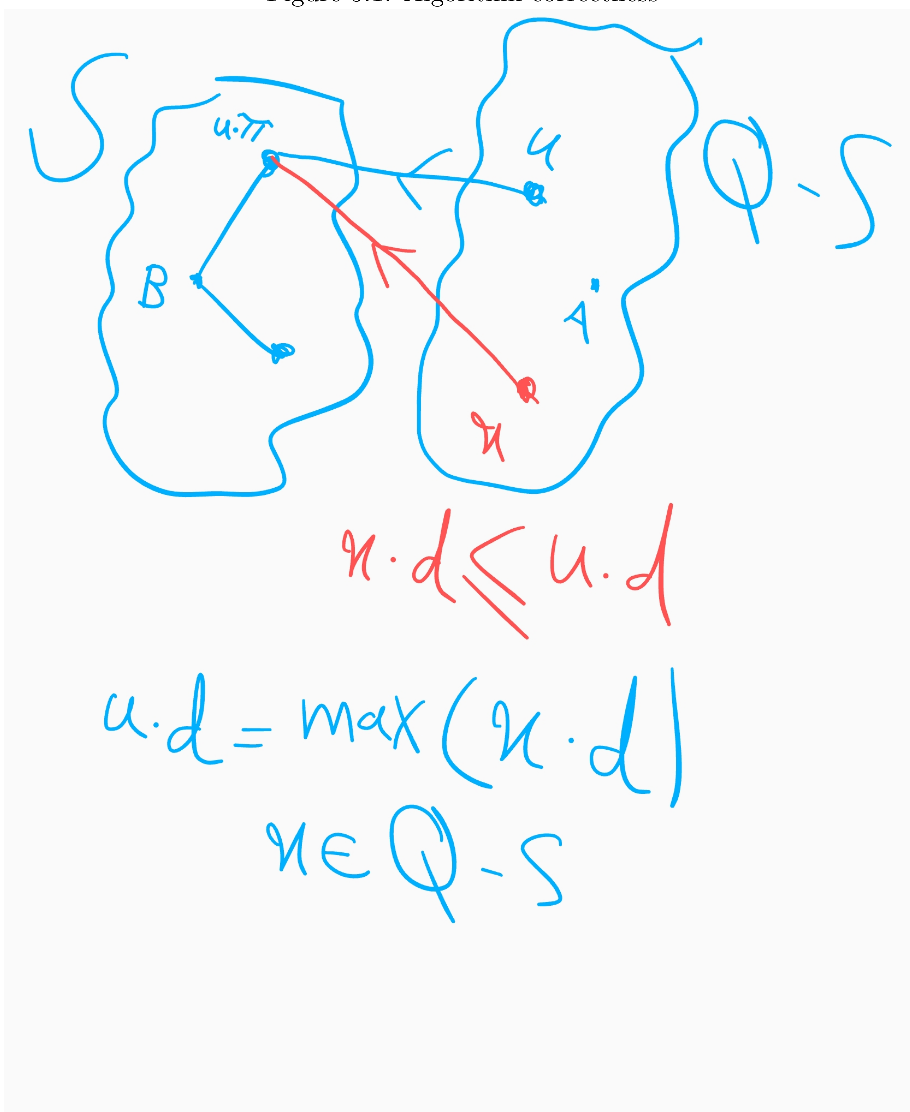



Figure 5.1: Algorithm correctness

```
function INITIALIZE-SINGLE-SOURCE(G, s)
    for all v ∈ G.V do
        v.d = −∞
        v.π = NIL
    end for
end function
```

3. **Single-pair shortest-path problem:** Many problems for previous sections actually belong here.
4. **All-pair shortest-path problem:**

## 5.1 The Bellman-Ford algorithm

### 5.1.1 Exercises

**Exercise 3** Since finding all shortest-paths require at most $m$ steps, it is obvious in $(m+1)^{\text{th}}$ iteration no $u.d$ will be updated for all $u \in V$.

```
function BELLMAN-FORD2(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i = 1  to  |G.V| − 1 do
        updated = FALSE
        for all edge (u, v) ∈ G.E do
            if RELAX(u, v, w) == TRUE then
                updated = TRUE
            end if
        end for
        if updated == FALSE then
            break
        end if
    end for
end function
```

```
function RELAX(u, v, w)
    if v.d > u.d + w(u, v) then
        v.d = u.d + w(u, v)
        v.π = u
        return TRUE
    else
        return FALSE
    end if
end function
```

**Exercise 4** When we detect a negative cycle, we need to update $u.d = -\infty$. Then we need to update $d$ attribute for all reachable vertices from $u$ to $-\infty$.

```
function BELLMAN-FORD-MODIFIED(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i = 1 to |G.V| − 1 do
        for all edge (u, v) ∈ G.E do
            RELAX(u, v, w)
        end for
    end for
    INITIALIZE-DFS(G)
    for each edge (u, v) ∈ G.E do
        if v.d > u.d + w(u, v) then
            v.d = −∞
            DFS(G, v)
        end if
    end for
end function
```

```
function DFS(G, u)
    u.color = GREY
    for all v ∈ G.adj[u] do
        if v.color == WHITE then
            v.d = −∞
            v.color = GREY
            DFS(G, v)
        end if
    end for
end function
```

**Exercise 5** In the Bellman-Ford algorithm $s.d = 0$ and other vertices have value $\infty$ for their $d$ attribute. If we set $u.d = 0$ for all $u \in V$ we solved this problem. Note that $\delta^*(v) = v.d$.

```
function BELLMAN-FORD-MODIFIED(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for all u ∈ G.V do
        u.d = 0
    end for
    for i = 1 to |G.V| − 1 do
        for all edge (u, v) ∈ G.E do
            RELAX(u, v, w)
        end for
    end for
end function
```

**Exercise 6** We add a new attribute $u.mark$ for avoiding printing vertices in a negative cycle more than once. Suppose $v_0, v_1, \ldots, v_k$ are vertices in a negative cycle and $v_0 = v_k$. The Bellman-Ford algorithm expands shortest-path tree in each step. Suppose $v_i.\pi = v_{i-1}$ for $1 \leq i \leq k-1$ and $v_0.\pi = v_k.\pi = u$. Since the cycle is negative, $v_{k-1}.d + w(k-1, k) < v_k.d$. So we change the value of $v_k.\pi$ from $u$ to $v_{k-1}$. In other words, $v_i$ for $0 \leq i \leq k$ are unreachable from $u$ in shortest-path tree.

```
function BELLMAN-FORD-MODIFIED(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i = 1 to |G.V| − 1 do
        for all edge (u, v) ∈ G.E do
            RELAX(u, v, w)
        end for
    end for
    for all u ∈ G.V do
        u.mark = FALSE
    end for
    for each edge (u, v) ∈ G.E do
        if v.d > u.d + w(u, v) then
            w = v
            while w.mark == FALSE do
                w.mark = TRUE
                PRINT(w)
                w = w.π
            end while
        end if
    end for
end function
```

## 5.2 Dijkstra's algorithm

For an intuition of Dijkstra shortest path, see figure 5.2.

### 5.2.1 Exercises

**Exercise 6** Suppose the path $p = (v_0, v_1, \ldots, v_k)$ in which $v_0 = u$ and $v_k = v$ is one of the paths between $u$ and $v$. Since the probabilities are independent we want to find $\max(\prod_{i=0}^{k-1} r(v_i, v_{i+1}))$. We can reduce the problem to a shortest-path one by changing the weight function $w(u,v) = r(u,v)$ to $w'(u,v) = -\log r(u,v)$.

$$0 \leq r(u,v) \leq 1$$
$$\log 0 \leq \log r(u,v) \leq \log 1$$

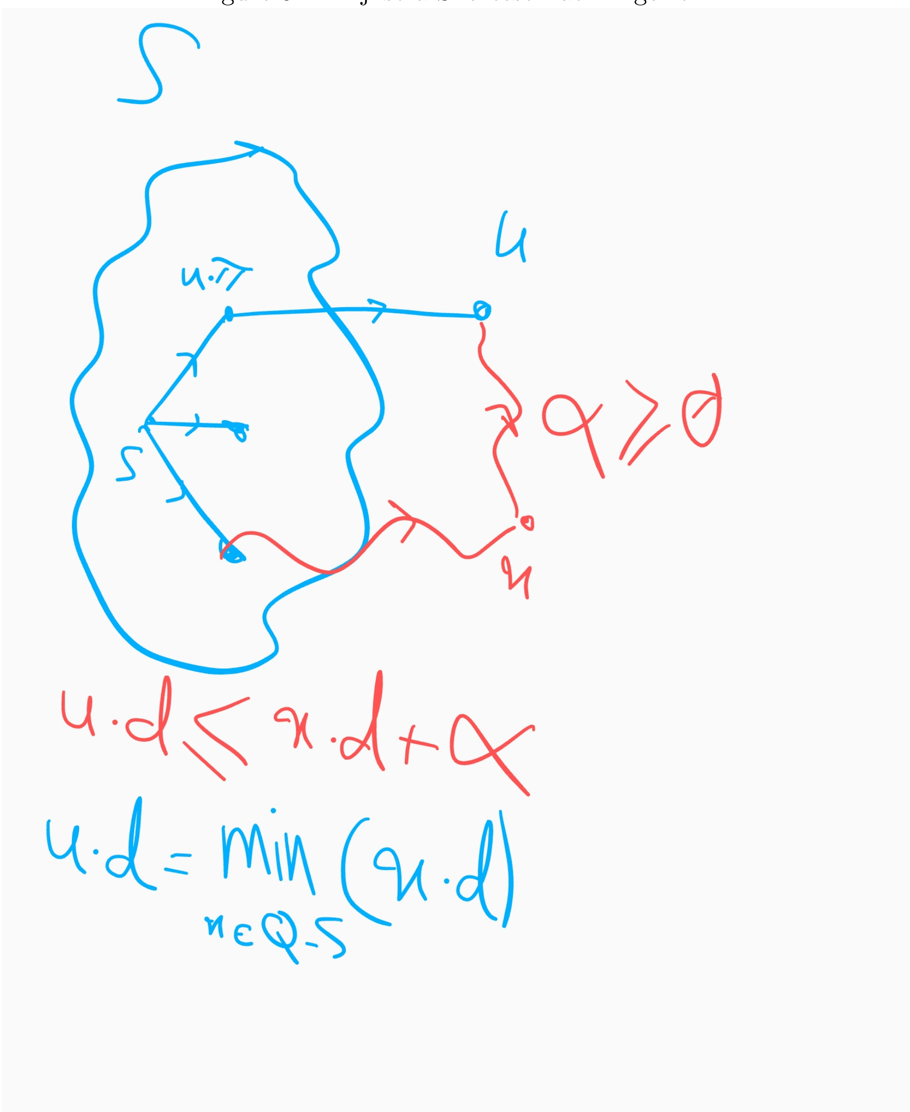
If we define $\log 0 = -\infty$, then $-\infty \leq \log r(u,v) \leq 0$ which is equivalent to $0 \leq -\log r(u,v) \leq \infty$.

$$\begin{aligned} &\max(\prod_{i=0}^{k-1} r(v_i, v_{i+1})) \\ &\qquad \equiv \max(\sum_{i=0}^{k-1} \log r(v_i, v_{i+1})) \\ &\qquad \equiv \min(\sum_{i=0}^{k-1} -\log r(v_i, v_{i+1})) \end{aligned}$$

which is exactly the shortest path problem. Since $w'(u,v)$ is non-negative, we can use Dijkstra algorithm which its run-time can be $O(E \log V)$. We only need to change *RELAX* function.

```
function w′(u, v, w)
    if w(u, v) == 0 then
        return ∞
    else
        return − log w(u, v)
    end if
end function
```

Figure 5.2: Dijkstra Shortest Path Algorithm




---

**function** RELAX(u, v, w)
    **if** $v.d > u.d + w'(u, v, w)$ **then**
        $v.d = u.d + w'(u, v, w)$
        $v.\pi = u$
    **end if**
**end function**

---

# Chapter 6

# All-Pairs Shortest Path

## 6.1 The Floyd-Warshall algorithm

**Transitive closure of a directed graph**

1. Given a directed graph G. Design an algorithm to determine whether or not there is at least one cycle. See TopCoder SRM 705 DIV 2 question 500.

**Solution** We can use DFS and if we find a back edge then it's part of a cycle. We can use transitive closure to solve this problem. We assume we don't have loop edges.

```
function CYCLE(G)
    n = |G.V|
    let T = (t_ij) be a new n × n matrix
    for i = 1 to n do
        for j = 1 to n do
            if i ≠ j ∧ (i, j) ∈ G.E then
                t_ij = 1
            else
                t_ij = 0
            end if
        end for
    end for
    for k = 1 to n do
        for i = 1 to n do
            for j = 1 to n do
                t_ij = t_ij ∨ (t_ik ∧ t_kj)
            end for
        end for
    end for
    for i = 1 to n do
        if t_ii == TRUE then
            return TRUE
        end if
    end for
    return FALSE
end function
```

# Chapter 7

# Maximum Flow

# Chapter 8

# Flow networks

## 8.1 The Ford-Fulkerson method

### Dealing with antiparallel edges

**Forbidding antiparallel edges** This is the approach of the textbook. Suppose flow network $G = (V, E)$. If $(u, v) \in E$, then $(v, u) \notin E$. With this definition we always have $0 \le f(u, v) \le c(u, v)$ in which $f(u, v)$ is the flow between $(u, v)$ and $c(u, v)$ is its capacity. Based on this definition of flow network we have the following "residual capacities".

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & (u, v) \in E \\ f(v, u) & (v, u) \in E \\ 0 & \text{otherwise} \end{cases} \tag{8.1}$$

Since antiparallel edges are not allowed, it is impossible that we have $(u, v) \in E \wedge (v, u) \in E$. So the definition of residual capacities is well defined. Given a flow network $G = (V, E)$ and a flow $f$, the **residula network** of $G$ induced by $f$ is $G_f = (V, E_f)$, where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$.
Now we need to deal with the flow of edges. If $f$ is a flow in $G$ and $f'$ is a flow in the corresponding residual network $G_f$, we define $f \uparrow f'$, the **augmentation** of flow $f$ by $f'$, to be a function from $V \times V$ to $\mathbb{R}$, defined by

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & (u, v) \in E \\ 0 & \text{otherwise} \end{cases} \tag{8.2}$$

If $f$ is a flow in $G$, the **value** of $f$ is defined as $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$ in which $s$ is the source. In the textbook it was proven that $|f \uparrow f'| = |f| + |f'|$. Consider augmenting path $p$ from source to sink. We define **residual capacity** of $p$ as $c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is on } p\}$. In the textbook it was proven that the following function $f_p : V \times V \to \mathbb{R}$ is a flow in $G_f$ with value $|f_p| = c_f(p) > 0$.

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p \\ 0 & \text{otherwise} \end{cases}$$

So we can use $|f \uparrow f_p| = |f| + |f_p| > |f|$ to increase the flow until we reach maximum flow.

**Allowing antiparallel edges** Suppose in flow network $G = (V, E)$ it is possible $(u,v) \in E \wedge (v,u) \in E$. We need to redefine some definitions. Consider Ford-Fulkerson algorithm. In each step we increase flow $f$ until we reach maximum flow. Suppose the flow of $G$ is $f_i$ in i$^{\text{th}}$ step. So we have $f_1, f_2, \ldots, f_m$. It is obvious $|f_1| = 0$ and $|f_m|$ is the value of maximum flow. Suppose we are in the i$^{\text{th}}$ step and $p$ is an augmenting path from source to sink. The **residual capacity** in the i$^{\text{th}}$ step is given by

$$c_{f_i}(u,v) = \begin{cases} c_{f_{i-1}}(u,v) - c_{f_i}(p) & (u,v) \text{ is on } p \\ c_{f_{i-1}}(u,v) + c_{f_i}(p) & (v,u) \text{ is on } p \\ c_{f_{i-1}}(u,v) & \text{otherwise} \end{cases} \tag{8.3}$$

For the base case we have $c_{f_1} = C$ in which $C$ is the capacities for all edges. Note that equation 8.3 not only is the general definition of equation 8.1, but also easier to implement. Unlike equation 8.1 which required the amount of edge flow, equation 8.3 only relies on previous residual capacities.
Now we need to redefine equation 8.2. Suppose $p$ is an augmenting path in the i$^{\text{th}}$ step.

$$f_i(u,v) = (f_{i-1} \uparrow f_p)(u,v) = c(u,v) - c_{f_i}(u,v) \tag{8.4}$$

By this definition it is possible we have negative flow. If $f(u,v) < 0$, it means the actual flow is from $v$ to $u$. In other words, $f(v,u) > 0$. More precisely, if $f(u,v) = k$ which $k > 0$, then $f(v,u) = -k$. We can prove it through mathematical induction. At the start $|f| = 0$ which holds our assumption and we use it as our base case. Suppose in the $(i-1)^{\text{th}}$ step, $f(u,v) = k_{i-1}$ and $f(v,u) = -k_{i-1}$. We need to prove this condition holds in the step $i$. Without loss of generality suppose $(u,v)$ is on augmenting path $p$:

$$\begin{aligned} f_i(u,v) &= c(u,v) - c_{f_i}(u,v) \\ &= c(u,v) - [c_{f_{i-1}}(u,v) - c_{f_i}(p)] \\ &= k \end{aligned}$$

$$\begin{aligned} f_i(v,u) &= c(v,u) - c_{f_i}(v,u) \\ &= c(v,u) - [c_{f_{i-1}}(v,u) + c_{f_i}(p)] \\ &= -k \end{aligned}$$

It means:

$$\begin{aligned} c(u,v) - c_{f_{i-1}}(u,v) + c_{f_i}(p) &= -c(v,u) + c_{f_{i-1}}(v,u) + c_{f_i}(p) \\ \Rightarrow c(u,v) - c_{f_{i-1}}(u,v) &= -c(v,u) + c_{f_{i-1}}(v,u) \\ \Rightarrow f_{i-1}(u,v) &= -f_{i-1}(v,u) \end{aligned}$$

So with this definition we have

$$f_i(u,v) = -f_i(v,u) \tag{8.5}$$

**Minimum cut properties** Suppose we have a flow network with maximum flow. Consider the final residual network $G_f$. Obviously we shouldn't have any augmenting path from $s$ to $t$.

> **augmenting reachable vertex u:** If there is an augmenting path from $s$ to $u$ in the final residual network $G_f$, we call $u$ augmenting reachable from $s$. Obviously $t$ is not augmenting reachable in the final residual network. So we call it augmenting unreachable

Consider $cut(S,T)$ in which $S$ is the set of all augmenting reachable vertices from $s$ and $T = V - S$ has all augmenting unreachable vertices from $s$. Obviously $t \in T$. Otherwise we have an augmenting path from $s$ to $t$. We have $\forall (u,v) \in \{(x,y) \in E : x \in S \wedge y \in T\}$:

$$\begin{aligned} flow(u,v) &= c(u,v) \\ c_f(u,v) &= 0 \end{aligned}$$

and $\forall (a,b) \in \{(x,y) \in E : a \in T \wedge b \in S\}$ we have:

$$\begin{aligned} flow(a,b) &= 0 \\ c_f(a,b) &= c(a,b) \end{aligned}$$

$c_f(u,v)$ is residual capacity of each edge which is defined as:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & (u,v) \in E \\ f(v,u) & (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

Note that if $flow(a,b) > 0$, then $c_f(b,a) = flow(a,b)$ which means $a$ is augmenting reachable from $s$ which is a contradiction. Since $flow(u,v) = c(u,v)$, this is a **"minimum cut"**. This property holds weather or not anti-parallel edges are allowed. Suppose $(u,v) \in E$ and $(v,u) \in E$ and $u \in S$ and $v \in T$. Then we have:

$$\begin{aligned} flow(u,v) &= c(u,v) \\ flow(v,u) &= 0 \\ c_f(u,v) &= 0 \\ c_f(v,u) &= c(v,u) \end{aligned}$$

Note that if $c_f(u,v) > 0$ then $v$ is augmenting reachable from $s$ which is a contradiction. Hence $c_f(u,v) = 0$ which means $flow(u,v) = c(u,v)$ and $flow(v,u) = 0$. For more information you can see "TopCoder's Maximum Flow: Section 1" and "TopCoder's Maximum Flow: Section 2".

1. Given a weighted directed graph $G$, remove a minimum-set of edges in such a way that a given node is unreachable from another given node.

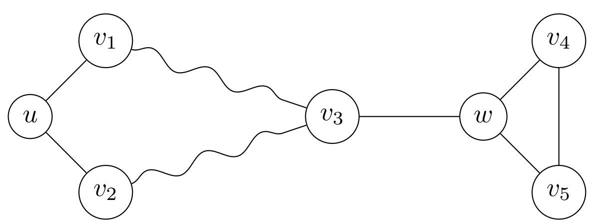
   - We convert graph $G$ to a flow network. We use the weight of edges as their capacities. So $c(u, v) = w(u, v)$. Run max flow algorithm on the flow network until we don't have any augmenting path from $s$ to $t$. Put all augmenting reachable vertices from $s$ into set $S$ and the others in $T$ to get a min cut. All edges $(u, v) \in E$ that are from $S$ to $T$ have flow $f(u, v) = c(u, v)$ and all edges $(x, y) \in E$ that are from $T$ to $S$ have flow $f(x, y) = 0$. In other words the capacity of min cut is $\sum_{u \in S} \sum_{v \in T} w(u, v)$. So by removing the crossing edges of min cut we solve the problem optimally.

2. Exercise 11 in 8.1.1

### 8.1.1 Exercises

**Exercise 11**

**Wrong greedy appraoch** Remove all edges that are incident to a vertex with minimum degree. We need to prove this greedy choice lead to a optimal solution in general. Suppose $u$ has the minimum degree $k$. We have an optimal solution $s$ in which at least one of the edges $(u, v)$ for $v \in V - \{u\}$ is not removed. We need to prove we can convert $s$ to $s'$ which all incident edges to $u$ is removed and is optimal as $s$. Without loss of generality suppose $k = 2$ and $w$ is the isolated vertex in $s$. As you can see neither $w$ nor $v_3$ have minimum degree and




with removing only one edge we can make the graph disconnected.

**Correct max flow min cut approach** We choose an arbitrary vertex $s$ as source and each $t \in V - \{s\}$ as sink. We create a new directed graph $G' = (V, E')$. $\forall (u, v) \in E\ (u, v) \in E' \wedge (v, u) \in E'$. It is obvious $|E'| = 2|E|$. Then we can make a flow network form $G'$. Note that we violate the assumption if $(u, v) \in E'$, then $(v, u) \notin E'$ for flow networks. But the algorithm still works and it's not a big deal. Also $s$ can have incoming edges and $t$ can have outgoing edges which is not violating any assumptions. We assign capacity 1 to each edge.
So we have $|V| - 1$ flow networks each of them has $s$ as its source and $t \in V - \{s\}$ as its sink. We need to find max flow in each of them and choose the

minimum of them as the result. Suppose $S$ and $T = V - S$ is a min cut and $E_c = \{(u, v) \in E' : u \in S, v \in T\}$. Since each edge has capacity 1, $\forall (u, v) \in E_c$, $flow(u, v) = 1 \wedge flow(v, u) = 0$, max flow is the number of edges in the min cut of that flow network. So by removing those edges in $G$ we solve the problem optimally.
Generally for min cut $S$ and $T = V - S$ and $\forall (u, v) \in \{(x, y) \in E : x \in V \wedge y \in T\}$ we have:

$$\begin{aligned} flow(u, v) &= c(u, v) \\ flow(v, u) &= 0 \\ c_f(u, v) &= 0 \\ c_f(v, u) &= c(u, v) \end{aligned}$$

$c_f(u, v)$ is the capacity of each edge in residual network. For more information you can see "TopCoder's Maximum Flow: Section 1" and "TopCoder's Maximum Flow: Section 2".

**Exercise 13** Suppose $S$ is a cut of $V$ which $s \in S$ and $t \in V - S$. We call $T = V - S$. We define the capacity of that cut $c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$. If we increase the capacity of each edge in $E$ by 1, we have $c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v) + 1 = \sum_{u \in S} \sum_{v \in T} c(u, v) + k$ which $k$ is the number of edges that cross the cut. But it's not enough. It is possible we have a cut which its capacity is not minimum but it has fewer edges than min cut. So by increasing the capacities, it'll become the new min cut. We know that $k \leq E$. Hence we can define $T = E + 1$ and change the capacities as following:

$$\begin{aligned} c(S, T) &= \sum_{u \in S} \sum_{v \in T} T \times c(u, v) + 1 \\ &= T \sum_{u \in S} \sum_{v \in T} c(u, v) + k \\ &= Tq + k \end{aligned}$$

So even if min cut has $E$ edges, increasing its edges by 1 is less than the other cuts which are not minimum. For more information you can see TopCoder's Maximum Flow: Section 2.