Solutions of Introduction to Algorithms

Saman Saadi

Contents

Chapter 1

Dynamic Programming

1.1 Rod cutting

Proof of the number of cuts Imagine for the rod of size n we have n cuts of length 1. We assign numbers $1, 2, \ldots, n$ to these cuts of length 1. For $n = 3$ we can have

1:	1	*	2	*	3
2:	1	*	2		3
3:	1		2	*	3
4:	1		2		3

So for a rod of length n we can have at most $n - 1$ stars (cuts). Each of these stars can appear or disappear. So the number of ways to cut the rod is 2^{n-1}.

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 $O(n^2)$

```
1:  function F(p, n, c)
2:      let r[0..n] be a new array
3:      r[0] ← 0
4:      for j from 1 to n do
5:          q ← p[j]
6:          for i from 1 to j − 1 do
7:              q = max(q, r[j − i] + p[i] − c)
8:          end for
9:          r[j] = q
10:     end for
11:     return r[n]
12: 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 $i \le k < j$.
Usually $|V|$ determines space complexity and $|V| + |E|$ time complexity. we know for every subprobelm m[i, j], $j \ge i$. 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

```
5   for l = 2 to n                    // l is the chain length
6       for i = 1 to n − l + 1
7           j = i + l − 1
8           m[i, j] = ∞
9           for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
```

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

directed graph:

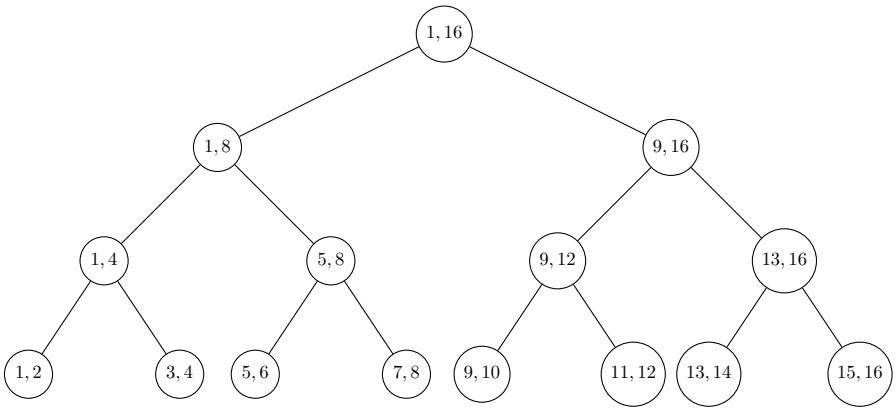
$$
\begin{aligned}
|E| &= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \sum_{k=i}^{i+l-2} 2 \\
&= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} 2(l-1) \\
&= 2\sum_{l=2}^{n} (n-l+1)(l-1) \\
&= 2\sum_{l=2}^{n} (n-(l-1))(l-1) \\
&= 2\sum_{l=1}^{n-1} (n-l)l \\
&= 2(\sum_{l=1}^{n-1} nl - \sum_{l=1}^{n-1} l^2) \\
&= 2(n\sum_{l=1}^{n-1}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

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

```
1: function DECREMENT(A)
2:     i = 0
3:     while i < A.length and A[i] == 0 do
4:         A[i] = 1
5:         i = i + 1
6:     end while
7:     if i < A.length then
8:         A[i] = 0
9:     end if
10: 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:

000
111
000
111

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

```
1:  function TransposeGraph(G)
2:      Let G' be a new graph
3:      G' ← G
4:      for all u ∈ V do
5:          for all v ∈ V do
6:              G'.A[v][u] = G.A[u][v]
7:          end for
8:      end for
9:      return G'
10: 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 $O(V^3)$. Note that having matrix-list named A:

$$A^2 = A_{|V|\times|V|} \times A_{|V|\times|V|}$$

Suppose $a_{ij} \in A$ and $a^2_{ij} \in A^2$ and $a_{ii} = 1$:

$$a^2_{ij} = \sum_{k=1}^{|V|} a_{ik} \times a_{kj}$$

So if $a_{ij}^2 > 0$ there is at least one path with at most 2 edges between i and j. So we can use Strassen multiplication algorithm in $O(n^{2.807})$. But here we use $O(V^3)$ straightforward algorithm.

For analyzing the running time of adjacency-list implementation we can use amortized analysis.

1. $\boldsymbol{in_u}$ **:** The number of edges that enter u
2. $\boldsymbol{out_u}$ **:** The number of edges that leave u
3. $\boldsymbol{(u,v)}$ **:** And edge from u to v $v \neq u$
4. $\boldsymbol{c_u}$ **:** The number of times we visit vertex u with at most one edge
5. $\boldsymbol{c_{(u,v)}}$ **:** The number of times we visit vertex v with exactly two edges (x_i, u) and (u, v) in which $x_i \in in_u$.
6. So we visit each vertex u no more than $c_u + \sum_{x \in in_u} c_{(x,u)}$.

We assign to each vertex u cost $c_u = 1 + in_u$. 1 is for line 4 and in_u is for line 6. For lines (7 - 9) we assign to each edge cost $c_{(u,v)} = in_u$. So the running time of algorithm, the total number of times we visit each vertex, is $\sum_{u \in V} c_u + \sum_{(u,v) \in E} c_{(u,v)}$. We calculate each of them separately. We know the following facts:

$$\begin{aligned}
\sum_{u \in V} in_u &= |E| \\
\sum_{u \in V} out_u &= |E| \\
\sum_{(u,v) \in E} in_u &= \sum_{u \in V} in_u \times out_u \\
\sum_{(u,v) \in E} in_u &\leq \sum_{(u,v) \in E} |V| - 1 \\
&= |E| \times (|V| - 1)
\end{aligned}$$

So we calculate each of the costs separately:

$$\begin{aligned}
\sum_{u \in V} c_u &= \sum_{u \in V} 1 + in_u \\
&= \sum_{u \in V} 1 + \sum_{u \in V} in_u \\
&= |V| + |E|
\end{aligned}$$

and for the last part:

$$\begin{aligned}\sum_{(u,v)\in E} c_{(u,v)} &= \sum_{(u,v)\in E} in_u \\ &\leq \sum_{(u,v)\in E} |V| \\ &= |V| \times |E|\end{aligned}$$

So the running time of algorithm is $O(|V| + |E| + |V| \times |E|) = O(|V| + |E| \times (1 + |V|)) = O(|V| + |V| \times |E|) = O(|V| \times (1 + |E|)) = O(|V| \times |E|)$.

Algorithm 3 Finding square graph using matrix-list

```
1:  function MakeSquareGraph(G)
2:      Let G' be a new Graph                          ▷ G.A[1..|V|, 1..|V|]
3:      for all u ∈ G.V do
4:          for all v ∈ G.V do
5:              G'.A[u][v] = G.A[u][v]                 ▷ 1-edge paths
6:              if G.A[u][v] = 1 then
7:                  for all k ∈ G.V do
8:                      G'.A[u][k] = G.A[v][k]         ▷ 2-edge paths
9:                  end for
10:             end if
11:         end for
12:     end for
13: end function
```

Algorithm 4 Finding square graph using adjacency-list

```
1:  function MakeSqureGraph(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[u].insert(v)                    ▷ 1-edge paths
7:              for all w ∈ G.Adj[v] do
8:                  if w ∉ G'.Adj[u] then
9:                      G'.Adj[u].insert(w)            ▷ 2-edge paths
10:                 end if
11:             end for
12:         end for
13:     end for
14: 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 $O(V) + O(2V) = O(V)$.

```
1:  function getUniversalSink(G)
2:      A = G.A                                          ▷ A[1..|V|, 1..|V|]
3:      u ← 1
4:      while u ≤ |V| do
5:          v ← u + 1
6:          sink ← u          ▷ Vertices from sink to |V| can be universal sink
7:          while v ≤ |V| ∧ A[u, v] = 0 do
8:              v ← v + 1                          ▷ v cannot be a universal sink
9:          end while
10:         u ← v                          ▷ u to v − 1 cannot be a universal sink
11:     end while
12:     for c from 1 to sink − 1 do
13:         if A[sink, c] ≠ 0 then
14:             return "No universal sink"
15:         end if
16:     end for
17:     for r ∈ V − {sink} do
18:         if A[r, sink] ≠ 1 then
19:             return "No universal sink"
20:         end if
21:     end for
22:     return sink
23: 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 $P_{|V|\times|V|} = B_{|V|\times|E|} \times B^T_{|E|\times|V|}$.

$$\begin{aligned} p_{ij} &= \sum_{e\in E} b_{ie} \times b^T_{ej} \\ &= \sum_{e\in E} b_{ie} \times b_{je} \end{aligned}$$

There are two cases:

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

$$b_{ie} \times b_{je} = \begin{cases} -1 \times 1 & e = (i,j) \in E \\ 1 \times -1 & e = (j,i) \in E \\ 0 & otherwise \end{cases}$$

So p_{ij} is the number of edges between i and j.

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

$$b_{ie} \times b_{je} = b_{ie} \times b_{ie} = \begin{cases} -1 \times -1 & e = (i,j) \in E \wedge j \neq i \\ 1 \times 1 & e = (j,i) \in E \wedge j \neq i \\ 0 \times 0 & otherwise \end{cases}$$

In this case p_{ii} is the sum of all edges that enter and leave the vertex i.

We can summarize the answer

$$p_{ij} = \begin{cases} -|S| \text{ such as } S = \{e \in E : e = (i,j) \vee e = (j,i)\} & i \neq j \\ indegree(i) + outdegree(i) & i = j \end{cases}$$

3.2 Breadth-first search

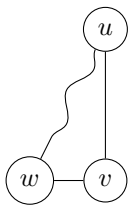
BFS algorithm For understanding the following algorithm see Lemma 22.3 from the textbook.

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



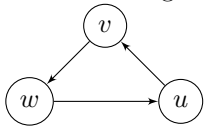
Algorithm 5 BFS algorithm

```
1: function BFS(G, s)
2:     for all u ∈ G.V − {s} do
3:         u.color = WHITE
4:         u.d = ∞
5:         u.π = NIL
6:     end for
7:     s.color = GRAY
8:     s.d = 0
9:     s.π = NIL
10:    Q = ∅
11:    ENQUEUE(Q, s)
12:    while Q ≠ ∅ do
13:        u = DEQUEUE(Q)
14:        for all v ∈ G.adj[u] do
15:            if v.color == WHITE then          ▷ edge (u, v) is a tree edge
16:                v.color = GRAY
17:                v.d = u.d + 1
18:                v.π = u
19:                ENQUEUE(Q, v)
20:            else if u.d == v.d + 1 then       ▷ back edge
21:                ASSERT(v.color == BLACK)
22:            else if v.d == u.d + 1 then       ▷ cross edge
23:                ASSERT(v.color == GRAY)
24:            else if u.d == v.d then           ▷ cross edge
25:                ASSERT(v.color == GRAY)
26:            end if
27:        end for
28:        u.color = BLACK
29:    end while
30: 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 $w.d = v.d + 1 \wedge u.d = w.d + 1$.

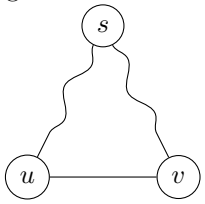
Figure 3.2: Back edge in BFS



- 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 can be gray. Unlike DFS in which the color of v is always 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 easier to use BFS for finding cycles in undirected graphs which takes $O(V + E)$. If we encounter a cross edge, we have a cycle. If we want a faster approach for undirected graph, we need to use DFS which takes $O(V)$. for more information see exercise 3 of Topological sort. For directed graphs, it's better to use DFS. If we have a back edge, it means we have a cycle. If you want to use DFS for undirected graph see **Exercise 10 of DFS**.

Exercise 7 For another good example see "Find the number of shortest path between" on this page 5.3.1. 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 not a tree edge (since the graph is undirected, it's a cross edge). An upper bound for $u.d$:

$$u.d \leq v.d + 1 \Rightarrow u.d - v.d \leq 1$$

On the other hand an upper bound for $v.d$ (note that the graph is undirected):

$$v.d \leq u.d + 1 \Rightarrow v.d - u.d \leq 1$$

So we can say:

$$|u.d - v.d| \leq 1 \Rightarrow -1 \leq u.d - v.d \leq 1$$

If $|u.d - v.d| = 1$, 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

```
1: function IsBipartiteGraph(G)
2:     for all u ∈ G.V do
3:         u.color = WHITE
4:         u.d = ∞
5:         u.π = NIL
6:     end for
7:     for all u ∈ G.V do
8:         if u.color == WHITE ∧ BFS(G, u) == FALSE then
9:             return FALSE
10:        end if
11:    end for
12:    return TRUE
13: end function
```

```
1: function BFS(G, s)
2:     s.color = GRAY
3:     s.d = 0
4:     Q = ∅
5:     Enqueue(Q, s)
6:     while Q ≠ ∅ do
7:         u = Dequeue(Q)
8:         for all v ∈ G.adj[u] do
9:             if v.color == WHITE then          ▷ edge (u, v) is a tree edge
10:                v.color = GRAY
11:                v.d = u.d + 1
12:                v.π = u
13:                Enqueue(Q, v)
14:            else if u.d == v.d then
15:                return FALSE
16:            end if
17:        end for
18:        u.color = BLACK
19:    end while
20:    return TRUE
21: 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. $\boldsymbol{s \in p}$

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 3.4: $s \in p$

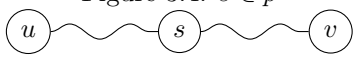
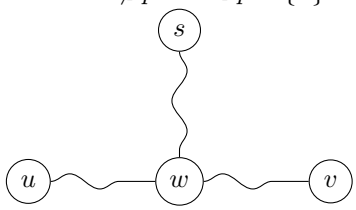


Figure 3.5: $s \notin p \wedge w \in p - \{u\} - \{v\}$



If we run BFS on s, then $\max_{x \in G.V}(x.d)$ belongs to either u or v. Otherwise the diameter is not between u and v. Without loss of generality suppose it is $u.d = \max_{x \in G.V}(x.d)$. 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 $O(V)$.

```
1: function FindDiameter(G)
2:     Let s be an arbitrary vertex such that s ∈ G.V
3:     InitBFS(G)
4:     u = BFS(G, s)
5:     InitBFS(G)
6:     v = BFS(G, u)
7:     return u, v, v.d
8: end function
```

```
1: function BFS(G, s)
2:     s.color = GRAY
3:     s.d = 0
4:     Q = ∅
5:     ENQUEUE(Q, s)
6:     max = −∞
7:     while Q ≠ ∅ do
8:         u = DEQUEUE(Q)
9:         for all v ∈ G.adj[u] do
10:            if v.color == WHITE then
11:                v.color = GRAY
12:                v.d = u.d + 1
13:                if v.d > max then
14:                    max = v.d
15:                    z = v
16:                end if
17:                v.π = u
18:                ENQUEUE(Q, v)
19:            end if
20:        end for
21:        u.color = BLACK
22:    end while
23:    return z
24: 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".

```
1: function DFS(G, u)
2:     u.color ← Gray
3:     paths ← φ
4:     for all v ∈ G.Adj[u] do
5:         if v.color = White then                          ▷ Tree edge
6:             paths ← {(u, v)} ∪ DFS(G, v) ∪ {(v, u)}
7:         else if v.color = Black then                     ▷ Forward edge
8:             paths ← paths ∪ {u, v} ∪ {v, u}
9:         end if
10:    end for
11:    u.color ← Black
12:    return paths
13: end function
```

3.3 Depth-first search

For another example of DFS, See exercise 8 in Floyd-Warshall shortest-path algorithm here 6.1.1.

Edge classification

Tree edges are edges in the depth-first forest G_π. 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.

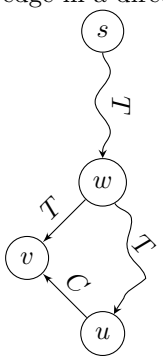
- **Tree edge**
  - Directed graph
    - DFS: We can have tree edges
    - BFS: We can have tree edges
  - Undirected graph
    - DFS: We can have tree edges
    - BFS: We can have tree edges
- **Forward edge**
  - Directed graph
    - DFS: We can have forward edges
    - BFS: We can't have forward edges. 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. See problem 22-1 from the textbook. If (u, v) is a back edge, then we have $u.d \geq v.d \geq 0$
  - Undirected graph
    - DFS: We can have back edges
    - BFS: We don't have back edges. Note that since it's undirected graph, if (u, v) is a tree edge, $(v, u) \in E$. So it has back edge properties but we consider it as tree edge.
- **Cross edge**
  - Directed graph
    - DFS: We can have cross edges. The cross edges can be in the same DFS tree or it can connect two different DFS trees.
    - BFS: We can have cross edges. refer to the beginning of chapter 3.2. For an example see figure 3.6. Suppose that (u, v) is a cross edge.
      We can only define an upper bound for $v.d$: $v.d \leq u.d + 1$. Because if $(u, v) \in E$ then it may be possible that $(v, u) \notin E$.

- Undirected graph
  - DFS: We can't have cross edges
  - BFS: We can have cross edges. See the beginning of chapter 3.2. Since $(u, v) \in E \wedge (v, u) \in E$, we can define upper bound for both $v.d$ and $u.d$:

$$\begin{aligned} v.d \leq u.d + 1 &\Rightarrow v.d - u.v \leq 1 \\ &\Rightarrow u.v - v.d \geq -1 \\ u.d \leq v.d + 1 &\Rightarrow u.d - v.d \leq 1 \end{aligned}$$

    So in undirected graphs for cross edge (u, v) we have $-1 \leq u.d - v.d \leq 1$

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 $v.d = u.d \wedge v.f = u.f$).

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

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

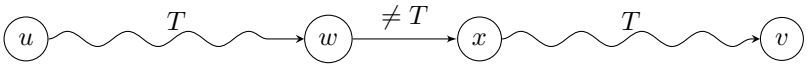
	white	gray	black
white	tree, back, forward, cross	back, cross	cross
gray	tree, forward	tree, back, forward	tree, forward, cross
black	impossible	back	tree, back, forward, cross

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

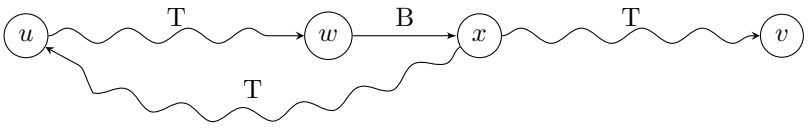


- **Forward Edge:** If (w, x) is forward ege, then w is an ancestor of x which leads to v be a descendant of u. So it cannot be a forward edge

- **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

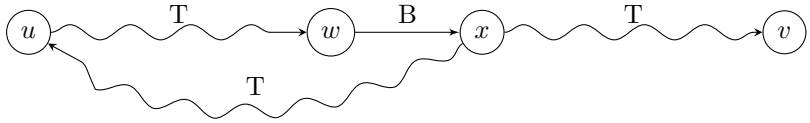


Excercise 9 Suppose we have edge (u, v) and we consider loops as back edges.

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: $v.d < v.f < u.d < u.f$

We want to find an example in which there is a path from u to v such that $u.f < v.d$. In the path from u to v there should be at least one edge (w, x) which is not a tree edge and $w.f < x.f$ so all ancestors of w, including u have a chance to finish before all descendants of x, including v. As you can see only back edge has such property. This is like previous example. We start DFS from x and then first visiting u and then finally v. So we have $x.d < u.d < w.d < w.f < u.f < v.d < v.f < x.f$.

Figure 3.9: Counterexample using back edge



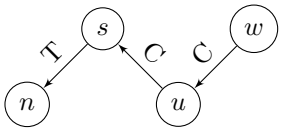
Exercise 10 The algorithm for directed graph:

```
1: function DFS-VISIT(G, u)
2:     time = time + 1
3:     u.d = time
4:     u.color = GRAY
5:     for all v ∈ G.adj[u] do
6:         if v.color == WHITE then                ▷ edge (u, v) is a tree edge
7:             PRINT-EDGE(u, v, TREE)
8:             v.π = u
9:             DFS-VISIT(G, v)
10:        else if v.color == GRAY then            ▷ edge u, v is a back edge
11:            PRINT-EDGE(u, v, BACK)
12:        else if u.d < v.d then
13:            PRINT-EDGE(u, v, FORWARD)
14:        else
15:            PRINT-EDGE(u, v, CROSS)
16:        end if
17:    end for
18:    u.color = BLACK
19:    time = time + 1
20:    u.f = time
21: end function
```

In an undirected graph forward edges are equivalent to back edges and cross edges are equivalent to tree edges. In other words we only have tree and back edges. Another tricky thing that we need to handle: if $(u, v) \in E$ then $(v, u) \in E$. So if we consider (u, v) as a tree edge then (v, u) is a back edge. In most cases this is not desirable. So we assume if (u, v) is a tree edge, then (v, u) is also a tree edge. Another tricky thing is back edges. If (u, v) is a back edge, then (v, u) is also a back edge. When we are visiting edge (u, v) the color of v is GRAY. On the other side when we are visiting edge (v, u) (which is after visiting edge (u, v)), the color of u is BLACK.

```
1: function DFS-VISIT(G, u)
2:     time = time + 1
3:     u.d = time
4:     u.color = GRAY
5:     for all v ∈ G.adj[u] do
6:         if v.color == WHITE then                ▷ edge (u, v) is a tree edge
7:             PRINT-EDGE(u, v, TREE)
8:             v.π = u
9:             DFS-VISIT(G, v)
10:        else if v.color == GRAY ∧ v.π == u then    ▷ edge (u, v) is a tree edge
11:            PRINT-EDGE(u, v, TREE)
12:        else if v.color == GRAY ∧ v.π ≠ u then     ▷ edge (u, v) is a back edge
13:            PRINT-EDGE(u, v, BACK)
14:        else                                    ▷ v.COLOR is BLACK
15:            PRINT-EDGE(u, v, BACK)
16:        end if
17:    end for
18:    u.color = BLACK
19:    time = time + 1
20:    u.f = time
21: end function
```

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.



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 3.10. 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

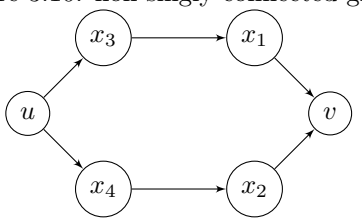
```
1: function DFS(G)
2:     for all u ∈ G.V do
3:         u.color = WHITE
4:         u.π = NIL
5:     end for
6:     time = 0
7:     ccn = 0                    ▷ ccn is the number of connected components
8:     for all u ∈ G.V do
9:         if u.color == WHITE then
10:            ccn = ccn + 1
11:            DFS-VISIT(G, u)
12:        end if
13:    end for
14: end function
```

```
1: function DFS-VISIT(G, u)
2:     time = time + 1
3:     u.d = time
4:     u.cc = ccn
5:     u.color = GRAY
6:     for all v ∈ G.adj[u] do
7:         if v.color == WHITE then
8:             v.π = u
9:             DFS-VISIT(G, v)
10:        end if
11:    end for
12:    u.color = BLACK
13:    time = time + 1
14:    u.f = time
15: end function
```

Exercise 13 It is obvious that we should only have tree and back edges and possibly some cross edges. If we have a cross edge in the same DFS tree it's not a singly connected graph. But If we have cross edges between different DFS trees, it's possible that it's not a single connected graph (See figure 3.11). It is important to start from the right vertex. Consider figure 3.10. 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 1 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.

Figure 3.10: non singly connected graph

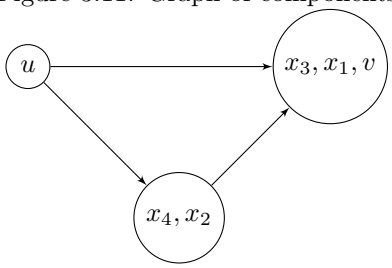


Wrong idea 2

component: All undiscovered vertices which will be discover in one DFS run. For example if we run DFS from x_3 in figure 3.10, x_3, x_1 and u are belong to the same component. In other words a component is a DFS tree

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 3.10 we run DFS first on x_3, then x_4 and finally u. You can see the result in figure 3.11. This algorithm is not always correct. Suppose we run DFS again on new graph (figure 3.11) first on node $\{x_3, x_1, v\}$, then $\{x_4, x_2\}$ and finally $\{u\}$. As you can see we don't have any tree edges but only cross edges. In this scenario we cannot combine some nodes to get a new graph with smaller number of nodes.

Figure 3.11: Graph of components



Wrong Idea 3 If you google for solution you may find a solution which suggest to run topological sort. This algorithm assumes that we have at least one vertex with indegree zero which is not a correct assumption for singly connected graph.

Correct Answer Since we don't know on which vertex we must start the DFS algorithm, we run DFS on every vertex $v \in V$. In every run we initialize all vertices attributes (.e.g $v.color = WHITE$). If we encounter forward or cross edges (note that we reset attributes in each run, so the cross edges are only in the same DFS tree and we don't encounter cross edges between two different DFS trees), we know that it's not singly connected. In each iteration we visit vertices at most $|E| + 1$ times. so the total runtime of this algorithm is $O(V \times (E + 1)) = O(V + VE)$.

3.4 Topological Sort

Exercise 2 Note that since it's a acyclic graph, we cannot have back edges. We add an attribute name *path* to each vertex which holds the number of simple paths from that vertex to t. The running time of this algorithm is $O(V + E)$.

```
1: function DFS(G, s, t)
2:     for all u ∈ G.V do
3:         u.color = WHITE
4:         u.π = NIL
5:         u.path = 0          ▷ u.path is the number of simple paths from u to t
6:     end for
7:     time = 0
8:     DFS-VISIT(G, s, t)
9: end function
10: function DFS-VISIT(G, u, t)
11:     time = time + 1
12:     u.d = time
13:     u.color = GRAY
14:     if u == t then
15:         u.path = 1
16:         u.color = BLACK
17:         return
18:     end if
19:     for all v ∈ G.adj[u] do
20:         if v.color == WHITE then                ▷ edge (u, v) is a tree edge
21:             v.π = u
22:             DFS-VISIT(G, v, t)
23:             u.path = u.path + v.path
24:         else if u.d < v.d then                  ▷ edge (u, v) is a forward edge
25:             u.path = u.path + v.path
26:         else                                    ▷ edge (u, v) is a cross edge
27:             u.path = u.path + v.path
28:         end if
29:     end for
30:     u.color = BLACK
31:     time = time + 1
32:     u.f = time
33: end function
```

Exercise 3 Since the graph is undirected, we encounter only tree and back edges. Note that for all tree edges (u, v), edge (v, u) is a back edge. In this case we consider it as tree edge. In other words, we assume that if edge (u, v) is a back edge, then v is not the parent of u. We start DFS on the graph If we encounter a back edge, we terminate the algorithm immediately. We visit every vertex only once. We also visit tree edges only once. If the DFS tree has at least one back edge, we only visit one of those back edges. So the running time of algorithm is the number of vertices plus the number of tree edges plus one back edge: $O(V + V - 1 + 1) = O(V)$.

```
1:  function DFS-VISIT(G, u)
2:      time = time + 1
3:      u.d = time
4:      u.color = GRAY
5:      for all v ∈ G.adj[u] do
6:          if v.color == WHITE then                          ▷ tree edge
7:              v.π = u
8:              if DFS-VISIT(G, v) then
9:                  return TRUE
10:             end if
11:         else if v.color == GRAY ∧ v.π == u then           ▷ tree edge
12:             continue
13:         else if v.color == GRAY ∧ v.π ≠ u then            ▷ back edge
14:             return TRUE
15:         else                                    ▷ v.COLOR is BLACK
16:             continue                              ▷ we never reach this
17:         end if
18:     end for
19:     u.color = BLACK
20:     time = time + 1
21:     u.f = time
22:     return FALSE
23: end function
```

3.5 Strongly connected components

The algorithm for SCC:

1. call $DFS(G)$ to compute finishing time $u.f$ for all vertices $u \in G.V$
2. Compute G^T. Remember $(u, v) \in G^T.V$ if $(v, u) \in G.V$
3. call $DFS(G^T)$, but in the main loop of DFS (the loop in which you run DFS-VISIT for white vertices), consider the vertices in order of decreasing $u.f$ (as computed in line 1). In other words run $DFS(G^T)$ in topological sort order of $DFS(G)$ which is computed in line 1

4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component. We are, in essence, visiting the vertices of the component graph (each of which corresponding to a strongly connected component of G) in topologically sorted order.

To understand the algorithm consider following paragraphs from textbook:

Lemma 22.13 Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that G contains a path $u \rightsquigarrow u'$. Then G cannot also contain a path $v' \rightsquigarrow v$.

Definitions If $U \subseteq V$, then we define $d(U) = \min_{u \in U}\{u.d\}$ and $f(U) = \max_{u \in U}\{u.f\}$. That is, $d(U)$ and $f(U)$ are the earliest discovery time and latest finishing time, respectively, of any vertex in U.

Lemma 22.14 Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

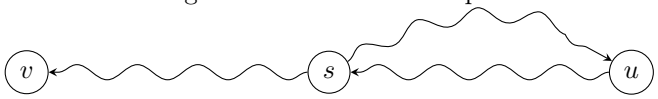
Corollary 22.15 Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$. In other words each edge in G^T that goes between different strongly connected components goes from a component with an earlier finishing time (in the first-depth search) to a component with a later finishing time.

Understanding strongly connected component algorithm Let us examine what happens when we perform the second DFS, which is on G^T. We start with strongly connected component C whose finishing time $f(C)$ is maximum. The search starts from some vertex $x \in C$, and it visits all vertices in C. By Corollary 22.15, G^T contains no edge from C to any other strongly connected component, and so the search from x will not visit vertices in any other component. Thus, the tree rooted at x contains exactly the vertices of C. Having completed visiting all vertices in C, the search in line 3 (SCC algorithm) selects as a root a vertex from some other strongly connect component C' whose finishing time $f(C')$ is maximum over all components other than C. Again, the search will visit all vertices in C', but by Corollary 22.15, the only edges in G^T from C' to any other component must be to C, which we have already visited. In general, when the DFS of G^T in line 3 visits any strongly connected component, any edges out of that component must be to components that the search already visited.Each depth-first tree, therefore, will be exactly one strongly connected component.

Tricky point You may think that we can run the second DFS on G instead of G^T and we need to sort vertices in decreasing order of their finishing time. It's not always correct. See exercise 3.

Exercise 3 Not always. See the following graph. Suppose in the first DFS we start from s, then visit u and finally we visit v. In other words, $s.d < u.d < u.f < v.d < v.f < s.f$. So in the second DFS, we start from u and both s and v are reachable from u. It means that all u and s and v are in the same component which is wrong.

Figure 3.12: Counterexample

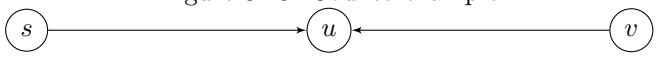


Note in SCC algorithm we say $f(C) > f(C')$ if there is an edge (u, v) in which $u \in C \wedge v \in C'$. $f(C)$ means $max_{u \in C}\{f_u\}$. Let say C has a vertex x in which its f_x is minimum among all the vertices in Graph. It doesn't guarantee that $f(C)$ is maximum among all SCCs. Because it's possible that C is discovered before C' so there is a vertex $x \in C$ that $f_x \leq f_y$ in which $y \in C \cup C\prime$. So if we run DFS on x for sure we will visit all vertices in C' so we consider both C and $C\prime$ as one SCC which is wrong. On the other hand, if we first discover C' then this algorithm works correctly.

Exercise 7

Wrong idea Convert the directed graph to an undirected graph and then run a DFS to see if all vertices are reachable from the root of DFS tree. This algorithm doesn't work. This algorithm consider the following graph as a semi-connected graph which is wrong:

Figure 3.13: Counterexample



Answer

1. Call strongly connected components algorithm on G.
2. Form the component graph (each vertex represent a strongly connected component and each edge connect two different strongly connected components).
3. Since component graph is a directed acyclic graph, you can run topological sort on it. Suppose we stored the sorted vertices in the $L = \{v_1, v_2, \ldots, v_n\}$.
4. Verify there are edges (v_i, v_{i+1}) for $i = 1$ to $i = n - 1$.

Chapter 4

Minimum Spanning Trees

For questions similar to Prim's minimum spanning tree see "Variants" in chapter 5.3.1.

4.1 The algorithms of Kruskal and Prim

Suppose $L = \{(u_1, v_1), (u_2, v_2), \ldots, (u_{|E|}, v_{|E|})\}$ such that $(u_i, v_i) \in E \wedge (u_{i+1}, v_{i+1}) \in E \wedge w(u_i, v_i) \leq w(u_{i+1}, v_{i+1})$ for all $1 \leq i \leq |E| - 1$.

Algorithm 8 MST-Kruskal. Depending on data structure can be $O(E \log V)$

```
function MST-PRIM(G, w)
    A = ∅
    for all v ∈ G.V do
        MAKE-SET(v)
    end for
    L = CREATE-LIST-L(G, w)        ▷ O(E log E) = O(E log V^2) = O(E log V)
    for all (u, v) ∈ L do
        if FIND-SET(u) ≠ FIND-SET(v) then
            A = A ∪ {(u, v)}
            UNION(u, v)
        end if
    end for
end function
```

Algorithm 9 MST-Prim. $O(V \log V + E \log V) = O(E \log V)$

```
function MST-PRIM(G, w, r)
    for all u ∈ G.V do
        u.key = ∞
        u.π = NIL
    end for
    r.key = 0
    Q = G.V
    while Q ≠ ∅ do
        u =EXTRACT-MIN(Q)                          ▷ O(V log V)
        for all v ∈ G.adj[u] do                    ▷ O(E log V)
            if v ∈ Q ∧ w(u, v) < v.key then
                v.π = u
                v.key = w(u, v)
            end if
        end for
    end while
end function
```

For a simple $O(V^2)$ implementation of Prim algorithm see exercise 2. Also see Dijkstra shortest path algorithm.

Exercise 2 We add a new variable named $u.MST$ to every vertex of graph. It's a boolean variable. It determines whether or not the vertex belongs to minimum spanning tree. As you can see I didn't use an adjacency matrix. You can easily replace it with adjacency matrix. The running time of this algorithm is $O(V^2)$. To be more specific, finding the minimum safe edge (lines 10-15) takes $O(V^2)$. Updating the keys (lines 17-22) takes no more than the sum of out-degree of each vertex. As we know

$$\sum_{u=\in V} indegree(u) + outdegree(v) = 2 \times |E| \leq 2 \times |V|^2$$

So the total running time of this algorithm is $O(V^2 + 2 \times V^2) = O(V^2)$.

Algorithm 10 MST-Prim in $O(V^2)$

```
1: function MST-PRIM(G, w, r)
2:     for all u ∈ G.V do
3:         u.key = ∞
4:         u.π = NIL
5:         u.MST = FALSE
6:     end for
7:     r.key = 0
8:     for c = 1 TO c = |G.V| do
9:         min = ∞
10:        for all w ∈ G.V do
11:            if w.MST == FALSE ∧ w.key < min then
12:                min = w.key
13:                u = w
14:            end if
15:        end for
16:        u.MST = TRUE
17:        for all v ∈ G.adj[u] do
18:            if v.MST == FALSE ∧ w(u, v) < v.key then
19:                v.π = u
20:                v.key = w(u, v)
21:            end if
22:        end for
23:    end for
24: end function
```

4.2 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 $O(E)$.

```
1: function VALID-BST-VALUE(G, b)
2:     G' = REMOVE-EDGES(G, b)
3:     InitDFS(G')
4:     c = 0                                  ▷ The number of connected components
5:     for all u ∈ G'.V do
6:         if u.color == WHITE then
7:             c = c + 1
8:             DFS(G', u)
9:         end if
10:    end for
11:    return (c == 1)
12: end function
```

```
1: function REMOVE-EDGES(G, b)
2:     G_b.V = G.V
3:     G_b.E = ∅
4:     for all (u, v) ∈ G.E do
5:         if (u, v).c ≤ b then                          ▷ (u, v).c ≡ w(u, v)
6:             G_b.E = G_b.E ∪ {(u, v)}
7:         end if
8:     end for
9:     return G_b
10: 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 $b_m = \min_{b \in G.E \wedge |G'_b.V|=1}(b)$. 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 $(u, v).c = w(u, v)$.

```
1: function BST(G)
2:     if |G.E| == 1 then
3:         return G.E
4:     end if
5:     m = FIND-MEDIAN(G.E)                                  ▷ O(E)
6:     if VALID-BST-VALUE(G, m) then
7:         G_b = REMOVE-EDGES(G, m)                          ▷ O(E)
8:         R = BST(G_b)
9:     else
10:        G'_b = MAKE-G'_b(G)                               ▷ O(E)
11:        R = BST(G'_b)
12:    end if
13:    return R
14: end function
```

```
1: function MAKE-G'_b(G)
2:     C, comp = CC(G)
3:     G'_b.V = C
4:     for all (u, v) ∈ G.E do
5:         if comp[u] ≠ comp[v] then
6:             if (comp[u], comp[v]) ∉ G'_b.E then
7:                 (comp[u], comp[v]).c = (u, v).c
8:                 G'_b.E = G'_b.E ∪ {(comp[u], comp[v])}
9:             else if (u, v).c < (comp[u], comp[v]).c then
10:                (comp[u], comp[v]).c = (u, v).c
11:            end if
12:        end if
13:    end for
14:    return G'_b
15: end function
```

```
1: function CC(G)
2:     Let comp[1..G.V] be a new array
3:     C = ∅                                  ▷ The set of connected components
4:     c = 0                                  ▷ The number of connected components
5:     INIT-DFS(G)
6:     for all u ∈ G.V do
7:         if u.color == WHITE then
8:             c = c + 1
9:             C = C ∪ {c}
10:            DFS(G, u, c, comp)
11:        end if
12:    end for
13:    return C, comp
14: end function
```

```
1: function DFS(G, u, c, comp)
2:     u.color = GRAY
3:     comp[u] = c
4:     for all v ∈ G.adj[u] do
5:         if v.color == WHITE then
6:             DFS(G, v, c, comp)
7:         end if
8:     end for
9: 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

For the following algorithms we are using these functions:

```
1: function INITIALIZE-SINGLE-SOURCE(G, s)
2:     for all v ∈ G.V do
3:         v.d = ∞
4:         v.π = NIL
5:     end for
6:     s.d = 0
7: end function
1: function RELAX(u, v, w)
2:     relaxed = FALSE
3:     if v.d > u.d + w(u, v) then
4:         v.d = u.d + w(u, v)
5:         v.π = u
6:         relaxed = TRUE
7:     end if
8:     return relaxed
9: end function
```

5.1 The Bellman-Ford algorithm

There is also similarity between Bellman-Ford and Floyd-Warshall algorithm.

In Bellman-Ford algorithm we know that a shortest path between u and v has at most $|V| - 1$ edges. So we relax all edges at most $|V| - 1$ times. On the other hand, in Floyd-Warshall algorithm we suppose we know the shortest path for all $u, v \in V$ using vertices $\{1, 2, \ldots, k-1\}$ and want to see if the shortest path from u to k and k to v give us a better one.

For some good example of this algorithm see **exercise 3** and examples 4 and 4a.

Algorithm 11 Bellman-Ford algorithm which runs in $O(VE)$

```
1: function BELLMAN-FORD(G, w, s)
2:     INITIALIZE-SINGLE-SOURCE(G,s)
3:     for i = 1 TO |G.V| − 1 do
4:         relaxed = FALSE
5:         for all (u, v) ∈ G.E do
6:             relaxed = RELAX(u, v, w) ∨ relaxed
7:         end for
8:         if relaxed == FALSE then
9:             break
10:        end if
11:    end for
12:    for all (u, v) ∈ G.E do
13:        if v.d > u.d + w(u, v) then
14:            return FALSE
15:        end if
16:    end for
17:    return TRUE
18: end function
```

5.1.1 Exercises

Exercise 3 See examples 4 and 4a for the application of this modified algorithm.

Suppose $\{v_{i_1}, v_{i_2}, \ldots, v_{i_m}, v_{i_{m+1}}\}$ is a shortest path between $u = v_{i_1}$ and $v = v_{i_{m+1}}$. If the order of edges in $G.E$ are in such away that edge $(v_{i_k}, v_{i_{k+1}})$ appears before edge $(v_{i_{k+1}, v_{i_{k+2}}})$ for all $1 \leq i \leq m-1$, then in normal Bellman-Ford algorithm we discover the shortest path from u to v in the first iteration of outer loop ($m = 1$). To avoid that and make sure the algorithm find the shortest path between u and v after m iterations, we define a new attribute named $v.d'$ which is a copy of $v.d$ for all $v \in V$. Then during relaxation we update $v.d'$, instead of $v.d$. After we relaxed all edges, then we sync $v.d$ with $v.d'$.

For iteration $m = k$, at the beginning for all $v \in G.V$ $v.d = v.d'$ which is the shortest path from s to v with edges less than k edges. When we relaxed all edges in step $m = k$, The only difference between $v.d$ and $v.d'$ is that $v.d'$ has shortest paths with k edges. At the end of iteration we update $v.d$ to has shortest paths for k edges. If you are only interested in to store shortest paths with only a specific amount of edges keep reading.

```
1: function BELLMAN-FORD2(G, w, s)
2:     INITIALIZE-SINGLE-SOURCE(G, s)
3:     for m = 1 to |G.V| − 1 do        ▷ shortest path with exactly m edges
4:         updated = FALSE
5:         for all edge (u, v) ∈ G.E do
6:             if RELAX(u, v, w) == TRUE then
7:                 updated = TRUE
8:             end if
9:         end for
10:        if updated == FALSE then
11:            break
12:        end if
13:        for all v ∈ G.V do
14:            v.d = v.d′
15:        end for
16:    end for
17:    return m
18: end function
```

```
1: function INITIALIZE-SINGLE-SOURCE(G, s)
2:     for all v ∈ G.V do
3:         v.d = ∞
4:         v.d′ = ∞
5:         v.π = NIL
6:     end for
7:     s.d = 0
8:     s.d′ = 0
9: end function
```

```
1: function RELAX(u, v, w)
2:     if v.d′ > u.d + w(u, v) then
3:         v.d′ = u.d + w(u, v)
4:         v.π = u
5:         return TRUE
6:     else
7:         return FALSE
8:     end if
9: end function
```

Note that if we want to know the shortest paths from s to other vertices with exactly $1 \leq k \leq m$ edges, instead of $v.d'$ we can use $v.d[0..|G.V| - 1]$. So $v.d[i]$ tell us the shortest path from s to v with exactly i edges:

```
1: function BELLMAN-FORD3(G, w, s)
2:     INITIALIZE-SINGLE-SOURCE(G, s)
3:     for m = 1 to |G.V| − 1 do        ▷ shortest path with exactly m edges
4:         updated = FALSE
5:         for all edge (u, v) ∈ G.E do
6:             if RELAX(u, v, w, m) == TRUE then
7:                 updated = TRUE
8:             end if
9:         end for
10:        if updated == FALSE then
11:            break
12:        end if
13:        for all v ∈ G.V do
14:            v.d = v.d′
15:        end for
16:    end for
17:    return m
18: end function
1: function INITIALIZE-SINGLE-SOURCE(G, s)
2:     for all v ∈ G.V do
3:         for all i = 0 to |G.V| − 1 do
4:             v.d[i] = ∞
5:             v.π[i] = NIL
6:         end for
7:     end for
8:     s.d[0] = 0
9: end function
1: function RELAX(u, v, w, m)
2:     if v.d[m] > u.d[m − 1] + w(u, v) then
3:         v.d[m] = u.d[m − 1] + w(u, v)
4:         v.π[m] = u
5:         return TRUE
6:     else
7:         return FALSE
8:     end if
9: end function
```

If we are only interested in shortest paths from s to other vertices with exactly y edges, instead of $d[1..|V|][0..y]$, we can have $d'[1..|V|][1..2]$. For simplicity I use $v.d$ and $v.d'$ like $BELLMAN-FORD2$ to see with a little modification of the solution we can achieve it. Suppose we want to know the shortest path from src to dst with exactly y edges:

```
1: function BELLMAN-FORD4(G, w, src, dst, y)
2:     INITIALIZE-SINGLE-SOURCE(G, src)
3:     for m = 1 to y do                    ▷ shortest path with exactly m edges
4:         updated = FALSE
5:         for all v ∈ G.V do
6:             v.d' = ∞
7:         end for
8:         for all edge (u, v) ∈ G.E do
9:             if RELAX(u, v, w) == TRUE then
10:                updated = TRUE
11:            end if
12:        end for
13:        if updated == FALSE then
14:            break
15:        end if
16:        for all v ∈ G.V do
17:            v.d = v.d'
18:        end for
19:    end for
20:    if updated == FALSE then
21:        return 0
22:    else
23:        return dst.d
24:    end if
25: end function
1: function INITIALIZE-SINGLE-SOURCE(G, s)
2:     for all v ∈ G.V do
3:         v.d = ∞
4:         v.π = NIL
5:     end for
6:     s.d = 0
7: end function
1: function RELAX(u, v, w)
2:     if v.d' > u.d + w(u, v) then
3:         v.d' = u.d + w(u, v)
4:         v.π = u
5:         return TRUE
6:     else
7:         return FALSE
8:     end if
9: 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 $-\infty$.

```
1: function BELLMAN-FORD-MODIFIED(G, w, s)
2:     INITIALIZE-SINGLE-SOURCE(G, s)
3:     for i = 1  to  |G.V| − 1 do
4:         for all edge (u, v) ∈ G.E do
5:             RELAX(u, v, w)
6:         end for
7:     end for
8:     INITIALIZE-DFS(G)
9:     for each edge (u, v) ∈ G.E do
10:        if v.d > u.d + w(u, v) then
11:            v.d = −∞
12:            DFS(G, v)
13:        end if
14:    end for
15: end function
```

```
1: function DFS(G, u)
2:     u.color = GREY
3:     for all v ∈ G.adj[u] do
4:         if v.color == WHITE then
5:             v.d = −∞
6:             v.color = GREY
7:             DFS(G, v)
8:         end if
9:     end for
10: end function
```

Exercise 5 We need to modify *Relax* function (note that we can have edges with negative weights):

```
1: function RELAX(G, w, u, v)
2:     if v.d > min(w(u, v), u.d + w(u, v)) then
3:         v.d = min(w(u, v), u.d + w(u, v))
4:         v.π = u
5:     end if
6: end function
1: function BELLMAN-FORD(G, w, s)
2:     INITIALIZE-SINGLE-SOURCE(G,s)
3:     for i = 1 TO |G.V| − 1 do
4:         relaxed = FALSE
5:         for all (u, v) ∈ G.E do
6:             relaxed = RELAX(u, v, w) ∨ relaxed
7:         end for
8:         if relaxed == FALSE then
9:             break
10:        end if
11:    end for
12:    for all (u, v) ∈ G.E do
13:        if v.d > u.d + w(u, v) then
14:            return FALSE
15:        end if
16:    end for
17:    return TRUE
18: 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.

```
1: function BELLMAN-FORD-MODIFIED(G, w, s)
2:     INITIALIZE-SINGLE-SOURCE(G, s)
3:     for i = 1  to  |G.V| − 1 do
4:         for all edge (u, v) ∈ G.E do
5:             RELAX(u, v, w)
6:         end for
7:     end for
8:     for all u ∈ G.V do
9:         u.mark = FALSE
10:    end for
11:    for each edge (u, v) ∈ G.E do
12:        if v.d > u.d + w(u, v) then
13:            w = v
14:            while w.mark == FALSE do
15:                w.mark = TRUE
16:                PRINT(w)
17:                w = w.π
18:            end while
19:        end if
20:    end for
21: end function
```

5.2 Single-source shortest paths in directed acyclic graphs

Note that by running Strongly connected components on a normal graph we can convert it to a DAG.

Since there is not cycle, we are not worried about negative cycles but it's ok if we have at least one edge with negative weight.

First we sort all vertices in topological sort order. So if there is a path between s (the root of shortest-path tree) to u, We visit s before u. So we start from the first vertex in the topological sorted order and relax all its children. We do the same for the next vertex. The running time of this algorithm is $O(V + E)$.

```
1: function INITIALIZE-SINGLE-SOURCE(G, s)
2:     for all v ∈ G.V do
3:         v.d = ∞
4:         v.π = NIL
5:     end for
6:     s.d = 0
7: end function
1: function RELAX(u, v, w)
2:     relaxed = FALSE
3:     if v.d > u.d + w(u, v) then
4:         v.d = u.d + w(u, v)
5:         v.π = u
6:         relaxed = TRUE
7:     end if
8:     return relaxed
9: end function
1: function DAG-SHORTEST-PATHS(G, w, s)
2:     Topological sort the vertices in G
3:     INITIALIZE-SINGLE-SOURCE(G,s)
4:     for all vertex u taken in topologically sorted order do
5:         for all v ∈ G.adj[u] do
6:             RELAX(u, v, w)
7:         end for
8:     end for
9: end function
```

5.3 Dijkstra's algorithm

```
1: function INITIALIZE-SINGLE-SOURCE(G, s)
2:     for all v ∈ G.V do
3:         v.d = ∞
4:         v.π = NIL
5:     end for
6:     s.d = 0
7: end function
1: function RELAX(u, v, w)
2:     relaxed = FALSE
3:     if v.d > u.d + w(u, v) then
4:         v.d = u.d + w(u, v)
5:         v.π = u
6:         relaxed = TRUE
7:     end if
8:     return relaxed
9: end function
1: function DIJKSTRA(G, w, s)
2:     INITIALIZE-SINGLE-SOURCE(G,s)
3:     S = ∅
4:     Q = G.V                                          ▷ heapify: O(V)
5:     while Q ≠ ∅ do                                   ▷ O(V)
6:         u = EXTRACT-MIN(Q)                           ▷ O(log V) or O(V)
7:         S = S ∪ {u}
8:         for all v ∈ G.adv[u] do                      ▷ Σ_{v∈V} outdegree(v) = O(E)
9:             if RELAX(u, v, w) == TRUE then
10:                DECREASE-KEY(Q, v, v.d)              ▷ O(log V) or O(1)
11:            end if
12:        end for
13:    end while
14: end function
```

As you can see if we implement Dijkstra algorithm using binary heap, converting the array to a heap (heapify) takes $O(V)$. At the beginning the size of Q is $|V|$ and in each iteration we extracting the minimum. So it takes $O(V \log V)$. Decreasing the key (up-heapifying) is done at most E times ($\sum_{v \in V} outdegree(v) = E$). So it takes $O(E \log V)$. So the total running time is $O(V \log V + E \log V)$.

If we use a heap like Fibonacci or Brodal heap which takes $O(1)$ for decreasing the key, the total running time is $O(V \log V) + O(E)$. For more information about heaps see this page.

Instead of heap we can use a normal array. So it takes $O(V)$ to populate the array. Finding the minimum takes $O(V)$. So EXTRACT-MIN takes $O(V^2)$. DECREASE-KEY takes $O(1)$. So the total running time of this implementation is $O(V^2) + O(E)$. If $E = O(V^2)$ then this implementation has the same performance as Fibonacci implementation:

Algorithm 12 $O(V^2) + O(E)$ Dijkstra shortest path

```
1: function DIJKSTRA(G, w, s)
2:     INITIALIZE-SINGLE-SOURCE(G, s)
3:     for i = 1 to |G.V| do                                   ▷ O(V)
4:         min = ∞
5:         for all w ∈ G.V do                                  ▷ O(V)
6:             if w.mark == FALSE ∧ w.d < min then
7:                 min = w.d
8:                 u = w
9:             end if
10:        end for
11:        u.mark = TRUE
12:        for all v ∈ G.adj[u] do          ▷ Σ_{v∈V} outdegree(v) = O(E)
13:            if v.mark == FALSE then
14:                RELAX(u, v, w)
15:            end if
16:        end for
17:    end for
18: end function
1: function INITIALIZE-SINGLE-SOURCE(G, s)
2:     for all v ∈ G.V do
3:         v.d = ∞
4:         v.π = NIL
5:     end for
6:     s.d = 0
7: end function
1: function RELAX(u, v, w)
2:     relaxed = FALSE
3:     if v.d > u.d + w(u, v) then
4:         v.d = u.d + w(u, v)
5:         v.π = u
6:         relaxed = TRUE
7:     end if
8:     return relaxed
9: end function
```

In most languages like std::priority_queue in C++, we don't have operation Decrease-Key. We can insert an edge instead of a vertex. More precisely $(u, v).d = v.d = u.d + w(u, v0)$. We define function $INSERT(PQ, v, (u, v).d)$ that insert edge (u, v) into the priority queue. The key in priority queue is $(u, v).d$, so $Extract - Min(PQ)$ returns v.

In this case extracting the minimum takes $O(E \log E)$. Inserting into priority queue which is equivalent to Decrease-Key takes also $O(E \log E)$. So the total running time of this implementation is $O(E \log E)$. So the total running time

of this algorithm is $O(E \log E) = O(E \log V^2) = O(2E \log V) = O(E \log V)$. If $E = O(V^2)$ this implementation is worse than the previous one. But for maximum flow implementation this implementation has better running time.

Algorithm 13 $O(E \log V)$ Dijkstra shortest path

```
 1: function DIJKSTRA(G, w, s)
 2:     INITIALIZE-SINGLE-SOURCE(G,s)
 3:     S = ∅
 4:     INSERT(PQ, s, 0)
 5:     visited = 0
 6:     while PQ ≠ ∅ ∧ visited < |G.V| do                          ▷ O(E)
 7:         u = EXTRACT-MIN(PQ)                                    ▷ O(log E)
 8:         if u ∈ S then
 9:             continue
10:         end if
11:         visited = visited + 1
12:         S = S ∪ {u}
13:         for all v ∈ G.adv[u] do          ▷ Σ_{v∈V} outdegree(v) = O(E)
14:             if v ∉ S ∧ RELAX(u, v, w) then
15:                 INSERT(Q, v, v.d)                              ▷ O(log E)
16:             end if
17:         end for
18:     end while
19: end function
 1: function INITIALIZE-SINGLE-SOURCE(G, s)
 2:     for all v ∈ G.V do
 3:         v.d = ∞
 4:         v.π = NIL
 5:     end for
 6: end function
 1: function RELAX(u, v, w)
 2:     relaxed = FALSE
 3:     if v.d > u.d + w(u, v) then
 4:         v.d = u.d + w(u, v)
 5:         v.π = u
 6:         relaxed = TRUE
 7:     end if
 8:     return relaxed
 9: end function
```

5.3.1 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 indepen-

dent 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 $0 \leq -\log r(u,v) \leq \infty$.

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

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.

```
1: function w'(u, v, w)
2:     if w(u,v) == 0 then
3:         return ∞
4:     else
5:         return −log w(u,v)
6:     end if
7: end function
```

```
1: function RELAX(u, v, w)
2:     if v.d > u.d + w'(u,v,w) then
3:         v.d = u.d + w'(u,v,w)
4:         v.π = u
5:     end if
6: end function
```

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 For another example about cross edges in BFS see BFS exercise 7 (It's on this page 3.2). 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 cross edges (since it's an undirected graph if $(u, v) \in E$, then $(v, u) \in E$; note that if (u, v) is a tree edge, we need to consider (v, u) also as tree edge). It is possible a tree edge be another shortest path which we need to consider it. Suppose edge (u, v) is a cross edge. According to d property (note that d is distance not discovery time) we know that $u.d+1 \leq v.d$ and $v.d+1 \leq u.d$. So we have $-1 \leq u.d-v.d \leq 1$. So $u.v-v.d$ can have three values -1, 0, 1. Only if $u.d-v.d = 1$ then $u.d+1$ is another shortest-path for v.

```
1:  function FIND-SHORTEST-PATH-COUNT(G, s, t)
2:      INIT-BFS(G)
3:      s.d = 0
4:      s.num = 1
5:      s.color = GREY
6:      Q = ∅
7:      ENQUEUE(Q, s)
8:      while Q ≠ ∅ do
9:          u = DEQUEUE(Q)
10:         for all v ∈ G.adj[u] do
11:             if u.color == WHITE then          ▷ (u, v) is a tree edge
12:                 v.color = GREY
13:                 v.d = u.d + 1
14:                 v.num = u.num
15:                 ENQUEUE(G, v)
16:             else if u.d + 1 == v.d then
17:                 v.num = v.num + u.num
18:             end if
19:         end for
20:     end while
21:     return t.num
22: 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 $m = \min_{(u,v)\in p}(w(u,v))$ and $M = \max_{(u,v)\in p}(w(u,v))$.

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. Hint: extract the maximum element from Q in each iteration.

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 or Floyd-Warshall 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 or Floyd-Warshall 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 $a_m = \max_{(u,v)\in p}(a(u,v))$. 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 $O(E \log V)$.

```
1: function DESERT(G, w, C, s, t)
2:     INITIALIZE-SINGLE-SOURCE(G, s)
3:     S = ∅
4:     Q = G.V
5:     while Q ≠ ∅ do
6:         u = EXTRACT-MIN(G)
7:         S = S ∪ {u}
8:         for all v ∈ G.Adj[u] do
9:             RELAX(u, v, w, C)
10:        end for
11:    end while
12:    if t.d < ∞ then
13:        return t.d
14:    else
15:        "IMPOSSIBLE"
16:    end if
17: end function
```

```
1: function RELAX(u, v, w, C)
2:     m = max(w(u, v), u.a_m)
3:     food = u.d + w(u, v)
4:     if (food + m) ≤ C ∧ food < v.d then
5:         v.d = food
6:         v.a_m = m
7:         v.π = u
8:     end if
9: 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
```

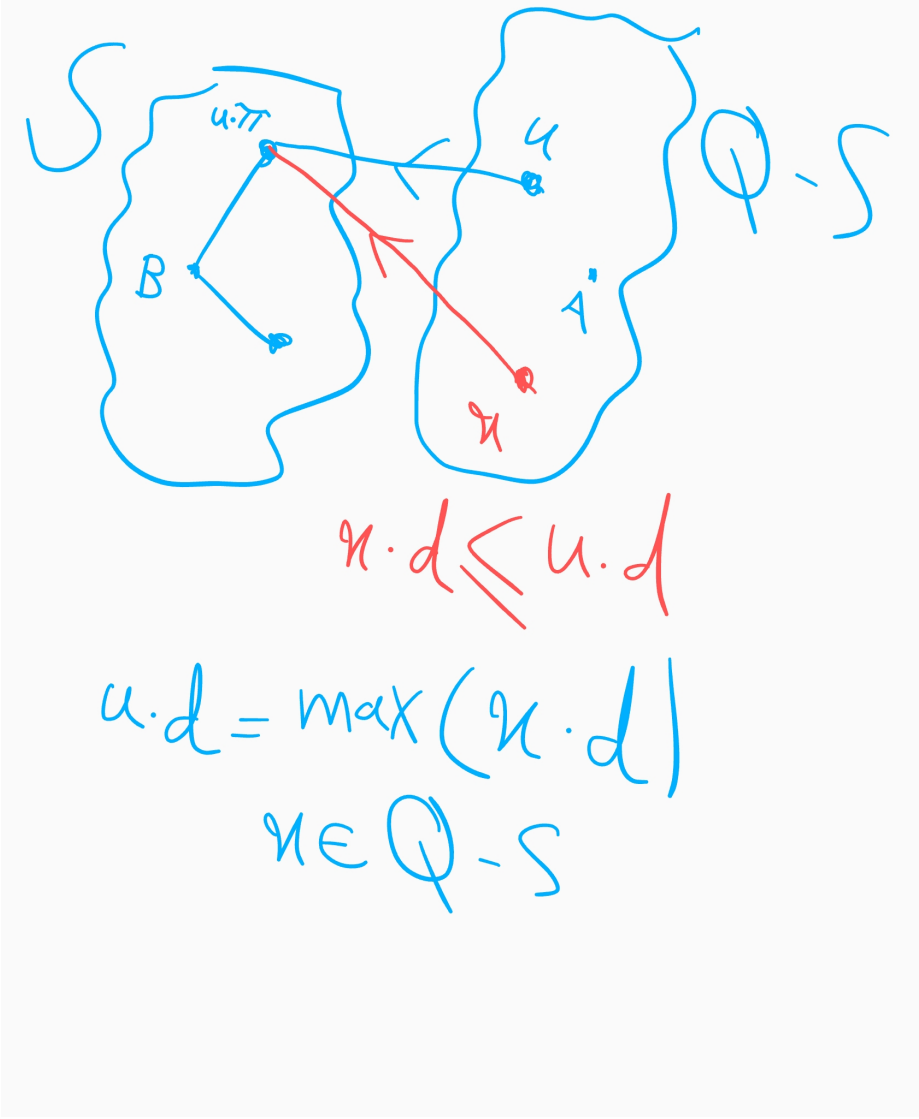
2. **Single-destination shortest-path problem:**

   (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 $(\max_{(B,u)\in C}(d_{uB}))$. 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.

Figure 5.1: Algorithm correctness



```
1: function SCHEDULING(G, A, B, d, a, t)
2:     G' = REVERSE-GRAPH-EDGES(G)
3:     INITIALIZE-SINGLE-SOURCE(G', s)
4:     B.d = t + 1                ▷ To make sure we arrive at B no more than t
5:     S = ∅
6:     Q = G'.V
7:     while Q ≠ ∅ do
8:         u = EXTRACT-MAX(G')
9:         S = S ∪ {u}
10:        for all v ∈ G'.Adj[u] do
11:            RELAX(u, v, d, a)
12:        end for
13:    end while
14:    if A.d > −∞ then
15:        u = A
16:        while u ≠ B do
17:            PRINT(u, u.π)
18:            u = u.π
19:        end while
20:    else
21:        PRINT("IMPOSSIBLE")
22:    end if
23: end function
```

```
1: function RELAX(u, v, d, a)
2:     if (a_vu + 1) ≤ u.d ∧ d_vu > v.d then
3:         v.d = d_vu
4:         v.π = u
5:     end if
6: end function
```

```
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:**

   (a) Count all possible walks from a source to a destination with exactly e edges

Solution 1: We can use an algorithm similar to Bellman-Ford shortest-path. The running time of this algorithm is $O(VE)$. We define $cnt[i][j]$ as the number of paths from s to i with exactly j edges. As you can see we can improve space complexity by using $cnt'[1..|V|][1..2]$, instead of $cnt[1..|V|][0..e]$. But for simplicity we use cnt.

```
1: function INITIALIZE-SINGLE-SOURCE(G, s, d, e)
2:     for all v = 1 to |G.V| do
3:         for all i = 1 to e do
4:             cnt[v][i] = 0
5:         end for
6:     end for
7:     cnt[s][0] = 1
8: end function
1: function RELAX(u, v, d, i)
2:     cnt[v][e] += cnt[u][e − 1]
3: end function
1: function BELLMAN-FORD(G, s, d, e)
2:     Let cnt[1..|V|][0..e] be a new array
3:     INITIALIZE-SINGLE-SOURCE(G,s, d, e)
4:     for i = 1 TO e do          ▷ Finding shortest paths with exactly i edges
5:         for all (u, v) ∈ G.E do
6:             RELAX(u, v, d, i)
7:         end for
8:     end for
9:     return cnt[s][d][k]
10: end function
```

Solution 2: You can use DFS but without marking the previously visited vertices. The running time of this algorithm is $O(V^k)$. For more information visit this website.

There is a better solution using an algorithm similar to Floyd-Warshall shortest past algorithm. We define $cnt[i][j][v]$ as the number of all paths from i to j which have exactly v vertices as intermediate. In other words a path like $i, v_{i_1}, v_{i_2}, \ldots, v_{i_v}, j$. So if we want to find the number of paths between s and d with exactly e edges, we need to find $cnt[s][d][e-1]$. So we define adjacency matrix $A = (a_{ij})$ as:

$$a_{ij} = \begin{cases} 0 & i = j \vee (i,j) \notin E \\ 1 & i \neq j \wedge (i,j) \in E \end{cases}$$

So we have:

```
1:
2: function COUNT-PATHS(A, s, d, e)
3:     n = A.rows
4:     let cnt[1..n][1..n][e − 1] be a new array
5:     for all i = 1 to n do
6:         for all j = 1 to n do
7:             cnt[i][j][0] = A[i][j]
8:         end for
9:     end for
10:    for all v = 1 to e − 1 do
11:        for all i = 1 to n do
12:            for all j = 1 to n do
13:                cnt[i][j][v] = 0
14:                for all k = 1 to n do
15:                    if A[i][k] ≠ 0 ∧ k ≠ j then          ▷ We know A[i][i] = 0
16:                        cnt[i][j][v]+ = cnt[k][j][v − 1]
17:                    end if
18:                end for
19:            end for
20:        end for
21:    end for
22:    return cnt[s][d][e − 1]
23: end function
```

(b) Shortest path with exactly k edges in a directed and weighted graph

Solution 1: For more information visit this page.
We can use an algorithm similar to Bellman-Ford shortest path. We define $d[i][j]$ as the shortest path from s to i with exactly j edges. The running time of this algorithm is $O(VE)$. As you can see for saving more space we can use $d'[1..|V|][1..2]$, instead of $d[1..|V|][0..k]$ but for simplicity we use the spacious one.

```
1: function INITIALIZE-SINGLE-SOURCE(G, s, d, k)
2:     for all v = 1 to |G.V| do
3:         for all i = 1 to k do
4:             d[v][i] = ∞
5:         end for
6:     end for
7:     d[s][0] = 0
8: end function
1: function RELAX(u, v, w, d, e)
2:     if d[v][e] > d[u][e − 1] + w(u, v) then
3:         d[v][e] = d[u][e − 1] + w(u, v)
4:     end if
5: end function
1: function BELLMAN-FORD(G, w, s, d, k)
2:     Let d[1..|V|][0..k] be a new array
3:     INITIALIZE-SINGLE-SOURCE(G,s, d, k)
4:     for e = 1 TO k do          ▷ Finding shortest paths with exactly e edges
5:         for all (u, v) ∈ G.E do
6:             RELAX(u, v, w, d, e)
7:         end for
8:     end for
9:     return d[s][d][k]
10: end function
```

Solution 2: For more information visit this website.
We can use an algorithm similar to Floyd-Warshall shortest path. We define $d[i][j][m]$ as the shortest path from i to j with exactly m intermediate vertices. In other words, a path like $i, v_{i_1}, v_{i_2}, \ldots, v_{i_m}, j$. So the shortest path between s and d with exactly k edges is $d[s][d][k-1]$. We define adjacency matrix $W = (w_{ij})$ like this:

$$w_{ij} = \begin{cases} 0 & i = j \\ w(i,j) & i \neq j \wedge (i,j) \in E \\ \infty & i \neq j \wedge (i,j) \notin E \end{cases}$$

See the following implementation for more details. The running time of this algorithm is $O(V^3 \times k)$

```
1:
2: function SHORTEST-PATHS(W, s, d, e)
3:     n = W.rows
4:     let d[1..n][1..n][e − 1] be a new array
5:     for all i = 1 to n do
6:         for all j = 1 to n do
7:             d[i][j][0] = W[i][j]
8:         end for
9:     end for
10:    for all v = 1 to e − 1 do
11:        for all i = 1 to n do
12:            for all j = 1 to n do
13:                cnt[i][j][v] = ∞
14:                for all k = 1 to n do
15:                    if A[i][k] ≠ ∞ ∧ k ≠ i ∧ k ≠ j ∧ d[k][j][v − 1] ≠ ∞ then
16:                        d[i][j][v] = min (d[i][j][v], A[i][k] + d[k][j][v − 1])
17:                    end if
18:                end for
19:            end for
20:        end for
21:    end for
22:    return cnt[s][d][e − 1]
23: end function
```

Chapter 6

All-Pairs Shortest Path

For an alternative solution see exercise 5 of Bellman-Ford algorithm.

Note that in Floyd-Warshall shortest-path algorithm, we can have negative weights but negative cycles are not allowed.

Weight definition We assume that the vertices are numbered $1, 2, \ldots, |V|$, so that the input is an $n \times n$ matrix W representing the edge weights of an n-vertex directed graph $G = (V, E)$. That is, $W = (w_{ij})$. Suppose $w(i, j)$ is the weight of directed edge (i, j) in which $(i, j) \in E$. We define w_{ij} as:

$$w_{ij} = \begin{cases} 0 & i = j \\ w(i,j) & i \neq j \wedge (i,j) \in E \\ \infty & i \neq j \wedge (i,j) \notin E \end{cases}$$

6.1 The Floyd-Warshall algorithm

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & k \geq 1 \end{cases}$$

Because for any path, all intermediate vertices are in the set $\{1, 2, \ldots, n\}$, the matrix $D^{(n)} = (d_{ij}^{(n)})$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

```
1: function FLOYD-WARSHALL(W)
2:     n = W.rows
3:     D^(0) = W
4:     for k = 1 TO n do
5:         let D^(k) = (d_ij^(k)) be a new n × n matrix
6:         for i = 1 To n do
7:             for j = 1 TO n do
8:                 d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
9:             end for
10:        end for
11:    end for
12: end function
```

For computing predecessor matrix:

$$\pi_{ij}^{(0)} = \begin{cases} NIL & i = j \vee w_{ij} = \infty \\ i & i \neq j \wedge w_{ij} < \infty \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj} & d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

As you can see $d_{ij}^{(k)}$ is a 3-dimensional array. We can use a 2D array. We assumed D_{ij} has the correct answer for intermediate vertices $1, 2, \ldots, k-1$. Now we want to add vertex k as a new intermediate vertex:

```
1: function FLOYD-WARSHALL(W)
2:     n = W.rows
3:     D = W
4:     for k = 1 TO n do
5:         for i = 1 To n do
6:             for j = 1 TO n do
7:                 if D_ik + D_kj < D_ij then
8:                     D_ij = D_ik + D_kj
9:                 end if
10:            end for
11:        end for
12:    end for
13: end function
```

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
```

6.1.1 Exercises

Exercise 5 It's correct because the only difference between this and the previous one is when we have two shortest path from i to j, One choose the one which passes k and the other don't.

Exercise 6 If there is a negative cycle, we have at least one $d_{ii}^{(n)} < n$ for $i \in V$. Assuming $n = |V|$.

Exercise 8 We know that for each vertex $v \in V$, if we run a DFS, we will visit all reachable vertices from v. This will take at most $|E| + 1$ times. We need that to do it for all vertices. So the total running time of algorithm is

$O(VE)$. After running the following algorithm $T[1..V][1..V]$ has the solution for transitive closure.

```
1: function DFS-VISIT(G, T, s, u)
2:     T[s][u] = TRUE
3:     for all v ∈ G.adj[u] do
4:         if T[u][v] == FALSE then
5:             DFS-VISIT(G, T, s, v)
6:         end if
7:     end for
8: end function
1: function TRANSITIVE-CLOSURE(G)
2:     let T[1..|V|][1..|V|] be a new array with False initialization
3:     for all v ∈ G.V do
4:         DFS-VISIT(G, T, v, v)
5:     end for
6: 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 $0 \leq f(u, v) \leq c(u, v)$ 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 **residual 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^{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 the network has $|f_{i-1}|$ and p is an augmenting path from source to sink. We define **residual capacity** of p as $c_{f_i}(p) = \min\{c_{f_{i-1}}(u,v) : (u,v) \text{ is on } p\}$.

Suppose we are in the i^{th} step and p is an augmenting path from source to sink. The **residual capacity** in the i^{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.

Now consider the Ford-Fulkerson Algorithm. For $(u,v) \in G.E$, $(u,v).c$ is the capacity of that edge. We don't change it during the iteration. $(u,v).c_f$ is residual capacity of edge (u,v) which changes during the execution. $(u,v).f$ is the current flow in (u,v). If it's negative it means the actual flow is from v to u. So $(u,v).f = -(v,u).f$. Also we assume if $(u,v) \in G.E$ then $(v,u) \in E$. $v.minCap$ is the minimum c_f in the chosen path from s to t.

```
1: function FORD-FULKERSON(G, s, t)
2:     for all (u, v) ∈ G.E do
3:         (u, v).c_f = (u, v).c
4:         (u, v).f = 0
5:     end for
6:     while TRUE do
7:         minCap = FIND-AUGMENTING-PATH(G, s, t)
8:         if minCap = 0 then
9:             break
10:        end if
11:        v = t
12:        while v ≠ NIL do
13:            u = v.π
14:            (u, v).c_f = (u, v).c_f − minCap
15:            (u, v).f = (u, v).f + minCap
16:            (v, u).c_f = (v, u).c_f + minCap
17:            (v, u).f = (v, u).f − minCap
18:            v = u
19:        end while
20:    end while
21: end function
```

Implementation function FIND-AUGMENTING-PATH can be trick. If we use BFS, the algorithm is called Edmonds-Karp. The running time of that algorithm is $O(VE^2)$. If we choose the path poorly (for example using DFS instead of BFS) the algorithm many never exit.

Algorithm 14 Edmonds-Karp algorithm

```
1: function INITIALIZE-SINGLE-SOURCE(G, s)
2:     for all v ∈ G.V do
3:         v.color = WHITE
4:         v.d = ∞
5:         v.π = NIL
6:         v.minCap = ∞
7:     end for
8:     s.d = 0
9: end function
1: function FIND-AUGMENTING-PATH(G, s, t)
2:     INITIALIZE-SINGLE-SOURCE(G, s)
3:     s.color = GRAY
4:     s.π = NIL
5:     Q = ∅
6:     ENQUEUE(Q, s)
7:     found = FALSE
8:     while Q ≠ ∅ do
9:         u = DEQUEUE(Q)
10:        if u == t then
11:            found = TRUE
12:            break
13:        end if
14:        for all v ∈ G.adj[u] do
15:            if v.color == WHITE ∧ (u, v).c_f > 0 then
16:                v.color = GRAY
17:                v.π = u
18:                v.minCap = min(v.minCap, (u, v).c_f)
19:                ENQUEUE(Q, v)
20:            end if
21:        end for
22:        u.color = BLACK
23:    end while
24:    if found = TRUE then
25:        return t.minCap
26:    else
27:        return 0
28:    end if
29: end function
```

Instead of queue, we can use a priority queue. This implementation is similar to Dijkstra shortest path algorithm. Function INSERT(Q, vertex, key) insert $(vertex, key)$ into priority queue. Function Extract-Max(Q) extracts the vertex in which key is maximum. We use $(u, v).c_f$ as key. Note that if indegree vertex

v is more than 1, then it's possible we inserted v more than once. In other words the maximum size of priority queue is $|G.E|$. Finding an augmenting path using the following implementation takes $O(E \log E) = O(E \log V)$.

If we want to use an implementation similar to $O(V^2+E)$ in Dijkstra shortest path, then finding an augmenting path takes $O(E^2 + E)$.

```
1: function INITIALIZE-SINGLE-SOURCE(G, s)
2:     for all v ∈ G.V do
3:         v.π = NIL
4:         v.minCap = ∞
5:     end for
6:     s.d = 0
7: end function
1: function FIND-AUGMENTING-PATH(G, s, t)
2:     INITIALIZE-SINGLE-SOURCE(G,s)
3:     S = ∅
4:     INSERT(Q, s, ∞)
5:     found = FALSE
6:     visited = 0
7:     while Q ≠ ∅ ∧ visited < |G.V| do
8:         u = EXTRACT-MAX(Q)
9:         if u ∈ S then
10:            continue
11:        end if
12:        visited = visited + 1
13:        S = S ∪ {u}
14:        if u == t then
15:            found = TRUE
16:            break
17:        end if
18:        for all v ∈ G.adv[u] do
19:            if (u, v).c_f > 0 ∧ v ∉ S then
20:                v.π = u
21:                v.minCap = min(v.minCap, (u, v).c_f)
22:                INSERT(Q, v, (u, v).c_f)
23:            end if
24:        end for
25:    end while
26:    if found == TRUE then
27:        return t.minCap
28:    else
29:        return 0
30:    end if
31: end function
```

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^{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 $(i-1)^{\mathrm{th}}$ 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 A minimum cut of a network is a cut whose capacity is minimum over all cuts of the network. Since we must respect the capacity of each edge when we calculating maximum flow, intuitively we calculated minimum cut.

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.

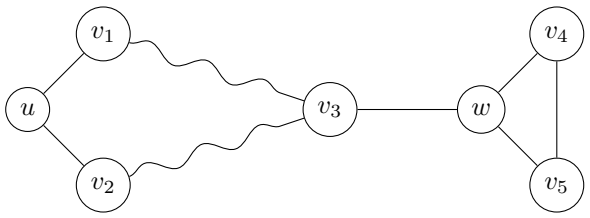
    - We convert graph G to a flow network. We use the weight of edges as their capacities. So $c(u,v) = w(u,v)$. Run max flow algorithm on the flow network until we don't have any augmenting path from s to t. Put all augmenting reachable vertices from s into set S and the others in T to get a min cut. All 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.