Algorithms Example Sheet 4: Problems Solution Notes

In this handout, you will find some algorithmic graph theory problems with some solution notes. Some of these exercises would normally appear as part of the theory. Some of the solutions elaborate on the solution notes by the lecturer for Example Sheet 4.
These notes have not be fully proofread. So if you find any typos/mistakes or have any suggestions (even if very small), please do let me know.

Core graph theory

Exercise 4.P.1 [Trees]

(a) Define a *tree*.

(b) Prove that a tree with $V \geq 2$ has at least two leaves (i.e. a node with degree 1).

(c) Show that every tree with $V \geq 2$ vertices can be formed by a tree T' of $V-1$ vertices plus a vertex and an edge.

(d) Show that an undirected graph is a tree iff it is connected and $|E| = V - 1$.

(e) (Maybe only a few of these) Show that the following definitions are equivalent to the definition of a tree:

i. A tree is one component of a forest. (A forest is an acyclic graph.)

ii. A tree is a connected graph with at most $V-1$ edges.

iii. A tree is a minimally connected graph; removing any edge disconnects the graph.

iv. A tree is an acyclic graph with at least $V-1$ edges.

v. A tree is a maximally acyclic graph; adding an edge between any two vertices creates a cycle.

vi. A tree is a graph that contains a unique path between each pair of vertices.

(f) Is it true that any subgraph of a tree is a tree?

(a) A *tree* is an undirected acyclic graph that is connected.

(b) There are several ways of doing this.

We will first show that there is at least one leaf. Assume not. Then all vertices have degree ≥ 2 (otherwise the graph is not connected). Then starting from a certain node, follow an edge that we have not followed before. This is possible because every node has degree at least 2. Since the graph is finite after V steps we will have visited $V+1$ vertices. So some vertex has been visited at least twice and so there must be a cycle (contradiction).

So, there must be some vertex ℓ with degree < 2. The vertex cannot have degree 0 as otherwise there would not be a path from ℓ to another vertex (there exists one since $V \geq 2$) and so the graph would not be connected. So, ℓ has degree 1.

But then we can repeat the same procedure starting from ℓ. For the same reason at some point we should reach a vertex with only one connection. Hence, there is another leaf

(c) Consider a tree of degree V, then it must have a leaf as proven in part (b). By removing a leaf (and its adjacent edge), the graph remains connected because no path can go through a vertex with degree 1 (so all original connections (apart from the removed vertex) remain unaffected). Removing a vertex maintains the acyclic property. Hence, the graph is a undirected acyclic and has $V-1$ vertices.

Question: *Is this tree unique?*

(d) We prove this by induction over the vertices of the tree. For $V = 1$, it is true. Assume it is true for $V = k$. Consider a tree T of $V = k+1$ vertices. By part (c), we can form T from a tree T' with k vertices plus a leaf and an edge. Hence, $E' = E + 1 = (k-1) + 1 = k$. So, it follows by induction.

It is also possible to remove any edge and split the tree into two trees of $< V$ vertices (and use strong induction). However, you would need to prove that removing an edge from the tree results into two trees.

(e) i. A component of forest is a connected subgraph of a forest. Being a subgraph of an acyclic graph it will also be acyclic. Hence, it is a connected acyclic graph, hence a tree. The other direction follows since a tree is also a forest (and hence a component of one).

ii. We will show that a connected component of at most $V-1$ edges has exactly $V-1$ edges (so the equivalence follows from part (d)). Assume not. Then, there are at most $V-2$ edges and V vertices. Start a DFS from one of the vertices, then in each step we visit one new vertex and traverse one new edge. Hence, after $V-2$ traversals we will visit $V-2$ vertices plus the initial one. Hence, at most $V-1$ (not V). Contradiction.

iii. Removing an edge from a tree results in a graph with $V-2$ edges, so by the previous part, it must be disconnected.

Now, We will prove by strong induction that a minimally connected graph of V vertices is a tree. (the reverse direction has been proven above). For this, we will use the following observation:

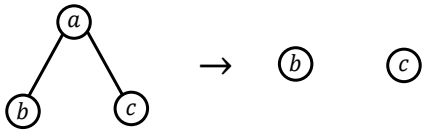
Observation: Removing any edge from a minimally connected component produces two minimally connected components.

(proof) Let's remove the edge (u, v) and let G' be the resulting graph. Let A_u be the set of vertices connected to u and A_v the set of vertices connected to v. By the definition of minimally connected, $A_u \cap A_v = \emptyset$ (otherwise u and v would be connected through the common vertices). Also $A_u \cup A_v = V$, as if there is an edge that is not reachable from u nor from v, then it would not be connected to either vertices in the original graph. Contradiction. Now removing any edge e from the subgraph of A_u should make it disjoint as otherwise the original graph would not be minimally connected.

Base case: $V = 1$, a single vertex is a tree.
Inductive step: Assume that it is true that any minimally connected graph of $\ell \leq k$ vertices is a tree. Consider a minimally connected graph G of $k+1$ vertices. Then removing any edge e creates two minimally connected components of size smaller than k. By the inductive hypothesis, these are both trees. Connecting two trees via an edge creates a tree (*can you explain why?*). So G is a tree.

iv. We will show that an acyclic graph of at least $V-1$ edges has exactly $V-1$ edges (so the equivalence follows from part (d)). Assume not. So it has at least V edges, then consider a DFS from any vertex. This will create the DFS tree that includes all vertices, so it has $V-1$ edges. When adding the V-th edge it connects two vertices that are already connected, and hence it forms a cycle.

(f) No it is not true. Consider the subgraph consisting of $\{b, c\}$ in the following tree which gives two disconnected vertices.



Question: *What can you say about the subgraph?*

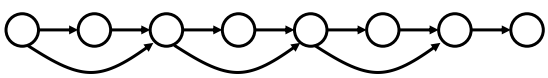
Exercise 4.P.2 Read the definitions of directed acyclic graph and tree from lecture notes. Draw an example and give its representation in adj. list format, of each of the following, or explain why no example exists:

(a) A directed acyclic graph with 8 vertices and 10 edges

(b) An undirected tree with 8 vertices and 10 edges

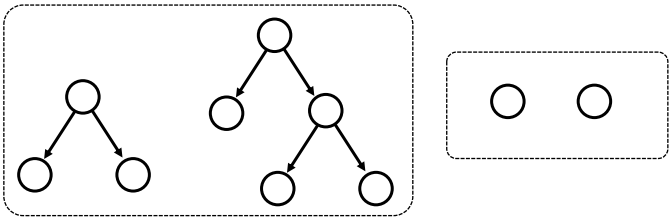
(c) A graph without cycles that is not a tree

(a) There are many examples. Here is one,



(b) A tree must have $E = V - 1$, but this is not the case.

(c) Any forest with more than one trees is valid here. The simplest being two nodes without edges (the graph on the right).



Exercise 4.P.3 In an undirected graph, the degree $d(u)$ of a vertex u is the number of neighbours u. In a directed graph, we distinguish between the *indegree* $d_{in}(u)$, which is the number of edges incoming to u, and the *outdegree* $d_{out}(u)$, the number of edges outgoing from u.

(a) Show that in an undirected graph, $\sum_{u \in V} d(u) = 2|E|$.

(b) Use part (a) to show that in an undirected graph, there must be an even number of vertices whose degree is odd.

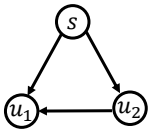
(c) Does a similar statement hold for the number of vertices with odd indegree in a directed graph?

(a) Consider the contribution of each edge (u, v) to the sum. It will be counted twice: once in $d(u)$ and once in $d(v)$. Since there are $|E|$ edges, and each one contributes 2, this gives a total of $2|E|$.

An alternative way to express this is using indicator functions, $\sum_{u \in V} d(u) = \sum_{u \in V} \sum_{v \in V} \mathbf{1}((u, v) \in E) = \sum_{e \in E} 2 = 2|E|$.

(b) Assume there was an odd number of vertices with odd degree, then their contribution to the sum would be an odd term (since sum of odd number of odds is odd). The contribution of the vertices with even degrees would be even, so the total would be odd. But this contradicts part (a).

(c) No. Consider the following graph, where there is only one vertex (namely u_2) with odd in-degree:



Exercise 4.P.4 [Bipartite Graphs] An undirected graph $G = (V, E)$ is *bipartite* if the vertices V can be partitioned into two subsets L and R, such that every edge has one vertex in L and the other in R.

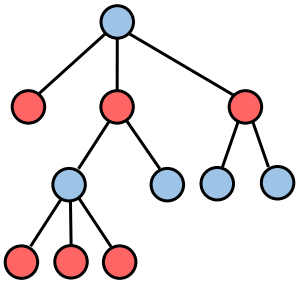
(a) Prove that every tree is a bipartite graph.

(b) Give an example of a graph that is non-bipartite.

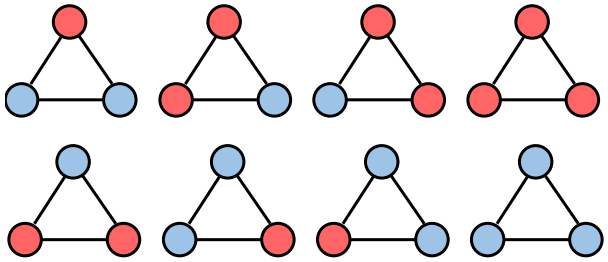
(c) Design an efficient algorithm that determines whether a given undirected graph is bipartite. Argue that your algorithm works. What is the time complexity of your algorithm?

(d) Prove that a graph G is bipartite if and only if every cycle in G has an even number of edges.

(a) Find the depth of each node. All nodes at an even depth assign to L. All nodes at an odd depth assign to R. There cannot be a edge between two nodes with depth of the same parity (because all edges are from depth i to depth $i + 1$).



(b) The cycle of length 3 is an example of non-bipartite graph. Using two colors to color three vertices means that two of the colors will be the same and hence they must be adjacent in the graph. There is also the brute force approach, where we show that any colouring :



(c) If the graph is not connected, then we can perform the check on all of its subcomponents. If any of these is non-bipartite, then the entire graph will not be. Otherwise, we take $L := L_1 \cup \ldots \cup L_k$ and $R := R_1 \cup \ldots \cup R_k$, where (L_i, R_i) are the partitions for the i-th component.

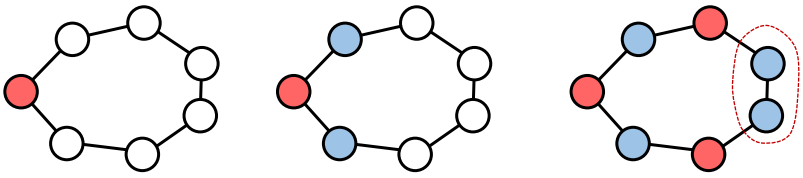
So we assume that the graph is connected. We start a DFS from any node s and add it to L (color it blue). Then this implies that all of its neighbours must be in R (color it red). Then proceed recursively for each of the neighbours. If at some point we attempt to colour a node that is already coloured with a different colour, then we have reached a contradiction (so the graph is non-bipartite). If the DFS terminates and there was no contradiction, then the graph is bipartite.

Each colouring that we made (except for that of node s) was obligatory, so this algorithm will determine if the graph is bipartite.

The time complexity is that of standard DFS, so $\mathcal{O}(V + E)$.

(d) There are two directions to this claim:

- (Bipartite $\Rightarrow$ every cycle is even) We will prove the contrapositive. Assume that there is a cycle with odd length. Then starting with a node s in the cycle, colouring it red and colouring its neighbours blue, and so on. Since the cycle has odd length, (and since we are colouring two nodes in every step except for the first one) in the last step there will be two vertices left and these would need to be coloured with the same colour. But since these are adjacent this will not lead to a valid colouring. Hence, the graph is not bipartite.



- (Every cycle is even $\Rightarrow$ bipartite) Again, we will prove the contrapositive. A graph is non-bipartite iff the above algorithm cannot find a colouring. This happens when we attempt to colour some node v with two different colours. This happens iff there was one path $p = s \rightsquigarrow v$ of odd length and one path $p' = s \rightsquigarrow v$ of even length. Merging these two paths (since they have the same endpoints), we get a cycle of odd length.

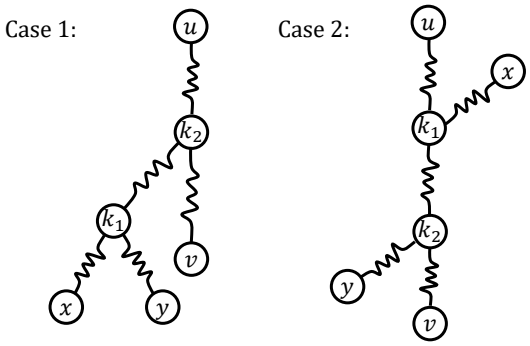
Exercise 4.P.5 [Longest path in a tree]

(a) Consider a graph without edge weights, and write $d(u, v)$ for the length of the shortest path from u to v. The diameter of the graph is defined to be $\max_{u,v \in V} d(u, v)$. Give an efficient algorithm to compute the diameter of an undirected tree, and analyse its running time. *Hint:* One way to solve this is by using breadth-first search, twice.

(b) Design a linear time algorithm for finding the diameter in a weighted tree. Argue that your algorithm is correct. *Hint:* Root the tree at some vertex r. For each vertex v, find the longest path in its subtree that ends at v. Why is it sufficient to consider only paths in its children?

(a) Pick an arbitrary vertex u, and run BFS starting at u. Pick some vertex v at maximum distance from u, and run BFS starting at v. The maximum distance from v is the diameter.

In order to prove this claim, consider the tree rooted at u and two arbitrary vertices x and y, and show that $d(v, x) \geq d(x, y)$. There are two possible cases with regards to the position of x and y relative to the path between $u \rightsquigarrow v$:



1. Either x and y meet (say at k_1) before joining to the $u \rightsquigarrow v$ path at k_2 (potentially $k_2 = u$).
   By construction, $d(u, v) \geq d(u, x) \Rightarrow d(u, k_2) + d(k_2, v) \geq d(u, k_2) + d(k_2, x) \Rightarrow d(k_2, v) \geq d(k_2, x)$.
   Hence, $d(x, y) = d(x, k_1) + d(k_1, y) \leq d(x, k_2) + d(k_2, y) \leq d(v, k_2) + d(k_2, y) \leq d(v, y)$.
   So taking $d(v, y)$ is at least as good as taking $d(x, y)$.
2. Or x and y join separately on the $u \rightsquigarrow v$ path, say at k_1 and k_2 respectively (without loss of generality we can assume that k_1 is closer to u).
   Proceeding in a similar way as before, $d(u, v) \geq d(u, y) \Rightarrow d(u, k_2) + d(k_2, y) \geq d(u, k_2) + d(k_2, v) \Rightarrow d(k_2, y) \geq d(k_2, v)$.
   Now considering $d(x, y) = d(x, k_1) + d(k_1, k_2) + d(k_2, y) \leq d(x, k_1) + d(k_1, k_2) + d(k_2, v) = d(x, v)$.
   So taking $d(x, v)$ is at least as good as taking $d(x, y)$.

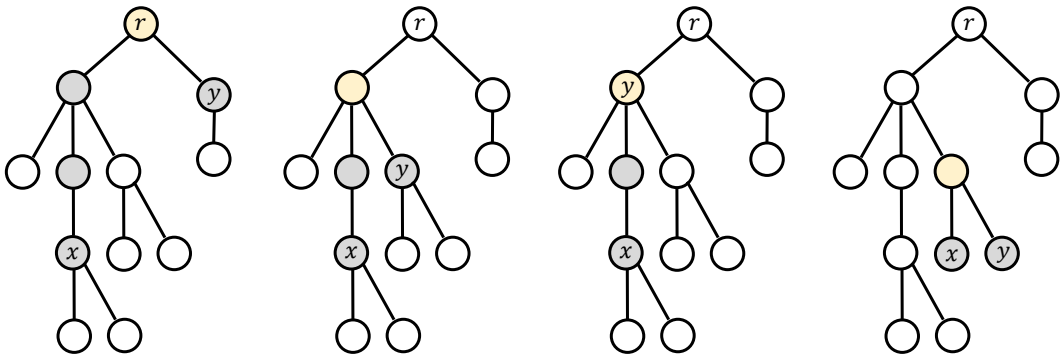
Hence, in either case, $d(v, *)$ is better than $d(x, y)$, so finding the maximum distance from v will give the diameter. The algorithm requires $\mathcal{O}(V)$ time.

(b) Root the tree at some arbitrary vertex r. Define for each vertex v, $d[v]$ the length of the longest path in the subtree of v ending at v. Then this can be recursively computed using the relation:

$$d[v] = \max(0, \max_{u \in \text{children}(v)} d[u])$$

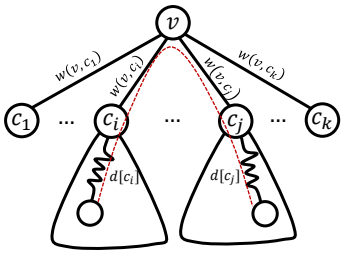
In order to find the maximum path in the tree, we make the following observation:

Observation: For every two nodes x and y, there is a unique k (call it the peak) in the path $x \rightsquigarrow y$ with minimum depth. Here are some example paths between x and y with the peak displayed in yellow:



For each vertex v, we will find the longest path that has v as its peak. The maximum over all vertices, will be the longest path in the tree. The longest path will be the sum of the two longest paths ending in one of its children,

$$\text{max\_peak}[v] = \max(d[v], \max_{u \in \text{children}(v)}(d[u]+\text{weight}(u \to v))+\text{second\_max}_{w \in \text{children}(v)}(d[w]+\text{weight}(w \to v))).$$



The reasoning for why it works is greedy. If it did not use one of the two maximum paths, then by swapping one of these with one of the maximal we would get a path that is at least as long.

Question 1: *Is the* $\max(0, \ldots)$ *needed in the definition of* $d[v]$*?*
Question 2: *How would you prove formally the existence of peak for every path in the tree?*

BFS/DFS

Exercise 4.P.6 [Jar Problem] You are given 3 jars with capacities c_1, c_2 and c_3. Initially all jars are empty. In each step, you have the following options:

- Fill in the i-th jar.
- Empty the i-th jar.
- Move the contents of the i-th jar to that of the j-th jar.

Construct an algorithm that given the jar capacities and a target value t, finds the fewest number of moves to make the target capacity t. For example, if the capacities are $10, 7, 4$ and the target is 2, then the following moves reach the target: `(0, 0, 0) -> (10, 0, 0) -> (6, 0, 4) -> (6, 0, 0) -> (2, 0, 4)`.

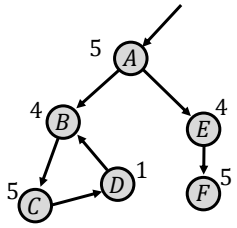
The following code solves the problem for general n jars using BFS. It should be noted that in this case the graph is not given, but it is implicit (read the comments for differences with classic BFS).

```
from queue import Queue


visited = set()
come_from = {}
```

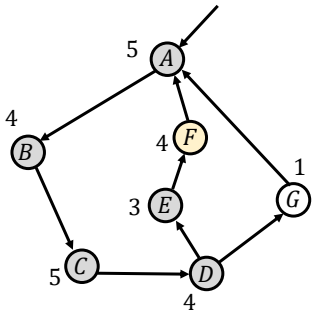
```
capacities = [10, 7, 4]


def get_neighbours(v):
    neighbours = []
    for i in range(len(v)):
        # Try to empty the jar.
        new_state = list(v)
        new_state[i] = 0
        neighbours.append(tuple(new_state))

        # Try to fill the jar.
        new_state = list(v)
        new_state[i] = capacities[i]
        neighbours.append(tuple(new_state))

        # Try to move from jar i to jar j.
        for j in range(len(v)):
            if i == j: continue
            new_state = list(v)
            new_state[j] = min(capacities[j], v[j] + v[i])
            new_state[i] = v[i] + v[j] - new_state[j]
            neighbours.append(tuple(new_state))
    return neighbours


def bfs_path(s, target):
    visited.add(tuple(s))
    toexplore = Queue()
    toexplore.put(tuple(s))

    # Traverse the graph, visiting everything reachable from s
    target_config = None
    while not toexplore.empty():
        v = toexplore.get()
        # The terminating condition is a bit different because we are searching for
        # any state that has the target jar load.
        if target in v:
            target_config = v
            break
        # Generating the neighbours is different, because we do not want to
        # construct the entire graph.
        neighbours = get_neighbours(v)
        for w in neighbours:
            if w not in visited:
                toexplore.put(w)
                visited.add(w)
                come_from[w] = v

    # Reconstruct the full path from s to t, working backwards
    if target_config is None:
        return None
    path = [target_config]
    while path[-1] in come_from:
        path.append(come_from[path[-1]])
    path.reverse()
    return path


path = bfs_path([0, 0, 0], 2)
for p in path:
    print(f"{p} -> ", end='')
print("DONE")
```

Exercise 4.P.7 Consider a directed graph in which every vertex v is labelled with a real number x_v. For each vertex v, define m_v to be the minimum value of x_u among all vertices u such that either $u = v$ or u is reachable via some path from v. Give an $\mathcal{O}(E + V \log V)$-time algorithm that computes m_v for all vertices v.
Extra credit: After you have read the section on further topics in DFS, try to find an $\mathcal{O}(E + V)$ solution.

Failed approach 1: one approach is to start a DFS and keep the minimum label m of the nodes visited. Then update all the nodes visited in the current DFS to that value m. This algorithm is not correct, since it might be that you are setting m to a node that cannot visit m. For example, in the following diagram starting a DFS from A would set the minimum value of 1 to both F and G, but they cannot reach it.



Failed approach 2: another approach is to run a DFS and each time we encounter a node of minimum value, then consider this as candidate for all nodes currently in the stack. While it is true, that all elements in the stack can reach the currently examined value, it is not true that these are the only ones. For example, when finishing the DFS for F we do not find the minimum reachable node which is G.



Lecturer's solution:

Algorithm 1: Finding the minimum value reachable for each node.

Function `FindMn`(n)**:**
- V = `SortVerticesByX`(x); ▷ `Takes` $\mathcal{O}(V \log V)$.
- G = `ReverseEdges`(G); ▷ `Takes` $\mathcal{O}(V + E)$ `with adj. list.`
- m = `InitialiseToNone`$(|V|)$; ▷ `Takes` $\mathcal{O}(V)$ `time.`
- **for** u *in* V **do**
  - **if** m_u *is None* **then**
    - `visit`(v, G); ▷ `Also sets` $m_k = x_u$ `on every vertex` k `visited along the way`
- **return** m

The check for whether m_u is `none` ensures that each vertex is visited exactly once. So, DFS takes $\mathcal{O}(V + E)$ time and the overall algorithm has time complexity $\mathcal{O}(E + V \log V)$.
(optional) Efficient solution: There exists an efficient solution based on SCCs components. The SCC graph forms a DAG.
Observation 1: If the input graph is a DAG, then the problem can be solved as follows:

- Find the topological ordering of the nodes.
- Denote by $mn[v]$ the minimum value reachable by v.
- Go through the elements in the topological order and set:

$$mn[v] = \min(x_v, \min_{u \in out(v)} mn[u]).$$

Observation 2: Any node in a SCC S can reach any node S, so it can reach the minimum of the SCC.
Observation 3: Using Observations 1 and 2, we can:

- Construct the SCC DAG and set $x_S := \min_{v \in S} x_v$ for each SCC S.
- Solve this problem for the DAG using topological ordering (Observation 1).
- Set $m_u = m_{S_u}$, where S_u is the SCC corresponding to u (Observation 2).

Each step takes $\mathcal{O}(V + E)$ time, hence the entire algorithm takes $\mathcal{O}(V + E)$ time.

Exercise 4.P.8 [Connection to DFAs/NFAs]

(a) Given a DFA D and a string s explain how you would check if the string is accepted by the DFA (i.e. $s \in L(D)$). What is the time complexity of your algorithm?

(b) Given a DFA D, how can you check if it accepts any string (or equivalently if $L(D) = \emptyset$)? How can you check if $L(D) = \Sigma^*$?

(c) Given an NFA N and a string s explain how you would check if the string is accepted by the NFA. What is the time complexity of your algorithm?

(d) Given a DFA D, how can you efficiently construct a DFA D' that accepts the prefixes of the strings in $L(D)$? For example, if $L(D) = \{\text{abc}, yx\}$, then $L(D') = \{\text{a}, \text{ab}, \text{abc}, \text{y}, \text{yx}\}$.

(a) Start from the starting node s of D and then follow the edge that corresponds to $s[0]$, then the edge corresponding to $s[1]$, and so on until the entire string has been consumed. The string is in the language iff the final state is accepting.

In order to traverse the edges efficiently, we assume that the alphabet symbols are in the range $0, \ldots, |\Sigma|-1$. Then for each node n we maintain an array $a[n]$, such that $a[n][i] = \delta(n, i)$. Hence, each traversal takes constant time and determining if a string is in the dictionary takes $\mathcal{O}(|s|)$ time.

(b) A DFA accepts a string iff there is a path from the starting to an accepting state. Hence, we only need to check if there is an accepting state reachable from the starting node. This can be done using either DFS or BFS in time $\mathcal{O}(V + E) = \mathcal{O}(|Q| \cdot \Sigma)$.

(c) We start by finding the set $A[0]$ of nodes reachable from the starting node using ϵ-transitions. Then, we find the set $A[1]$ of nodes reachable from $A[0]$ by following $s[0]$ (and potentially some ϵ-transitions, and so on. Hence, $A[i]$ contains all possible nodes that we could have reached when reading $s[0 \ldots i]$. In the end, the string s will be in $L(N)$ iff there is a state in $A[|s| - 1]$ that is accepting.

Each $A[i]$ could have $\mathcal{O}(|Q|)$ entries, so we can implement it in $\mathcal{O}(|s| \cdot Q)$ time.

Note that this mimics the powerset construction of the DFAs.

(d) The key observation is that we want to accept a string u iff we can append v such that uv is accepted. Let k be the state we reach once we consume u. This condition is equivalent to checking if there is a path from s to some accepting accepting state. This can be done naively using Q DFSs (in total $\mathcal{O}\left(|Q|^2 \cdot \Sigma\right)$ time) or in $\mathcal{O}(|Q| \cdot \Sigma)$ time using one DFS from the starting node (and returns whether an accepting node was encountered when checking its connections).

Dijkstra's algorithm

Exercise 4.P.9 [Bounded edge capacities]

(a) Design an algorithm for finding the shortest path in a graph where the edges have weight 0 or 1. (Your algorithm should be more efficient than Dijkstra's)

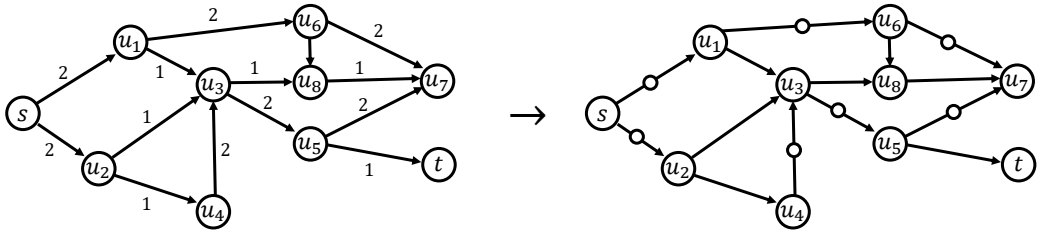
(b) Design an algorithm for finding the shortest path in a graph where the edges have weight 1 or 2. (Your algorithm should be more efficient than Dijkstra's)

(c) Design an algorithm for finding the shortest path in a graph where all edges have integer weight in $[1, L]$. What is the time complexity of your algorithm?

(d) (optional) Read about Dial's algorithm (or see problems 24.3.8 and 24.3.9 in CLRS).

(a) If we used Dijkstra's algorithm, then it would requite $\mathcal{O}(E + V \log V)$ time. We can do it more efficiently. Let's run a BFS starting at s. Then when we visit a vertex v, there will some outgoing edges with weight 0 and some with weight 1. If $w(v \to u) = 0$ and u has not be seen, then $d(s, u) = d(s, v)$. So we add it to the front of the queue to be processed as soon as possible. Vertices with $w(v \to u) = 1$ are placed as usual to the back of the queue.

(b) If we used Dijkstra's algorithm, then it would requite $\mathcal{O}(E + V \log V)$ time. However, there is a more efficient way. We can split each edge of weight 2 into two edges of cost 1 with a "dummy" vertex in between. Then finding the shortest path can be done using BFS. See the transformed graph below,



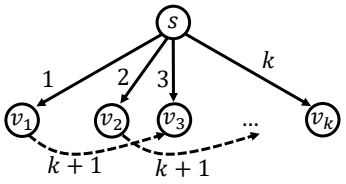
(c) The same technique generalises to edges of length L. Each edge will be transformed to $L - 1$ edges in the worst case, so the running time of BFS will be $\Theta(V + EL)$.

(d) Dial's algorithm can improve this to $\mathcal{O}(E + VL)$ by using a table like counting sort of size VL. The key property needed is that at any point in the priority queue, the difference between the maximum and the minimum is at most VL (because the candidate paths can differ by at most V edges, so at most VL cost). You can also improve it to $\mathcal{O}((V + E) \log L)$. See the referenced problems for more details.

Exercise 4.P.10 Suppose we have implemented Dijkstra's algorithm using a priority queue for which `push` and `decreaskey` have running time $\Theta(1)$, and `popmin` has running time $\Theta(\log n)$ where n is the number of items in the queue. Construct a sequence of graphs indexed by k, where the kth graph has $|V| = k$ and $|E| = \Theta(k^\alpha)$, such that Dijkstra's algorithm has running time $\Omega(k^\alpha + k \log k)$. Here α is a constant, $1 \leq \alpha \leq 2$.

We construct the following graph for $|V| = k + 1$, with the following properties:

- Choose one of the vertices to be s.
- Connect s with v_i with weight i.
- For the remaining $|V|^\alpha$ edges, add connections between any two vertices (it does not matter which) with weight $k + 1$.



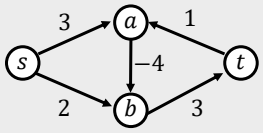
Then Dijkstra started from vertex s will perform the following steps:

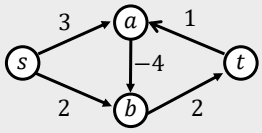
- For each i, insert v_i with weight i. So, there are k values in the priority queue.
- Process vertex v_1. All of its outgoing edges have weight $k + 1$, so it will not decrease the shortest path to any vertex. (So there will be $k - 1$ items in the priority queue).
- Process vertex v_2. All of its outgoing edges have weight $k + 1$, so it will not decrease the shortest path to any vertex. (So there will be $k - 2$ items in the priority queue).
- $\vdots$

Hence, processing s requires k insertions to the queue (where each one takes $\Theta 1$ time). Popping the i-th vertex requires $\Theta(\log(k - i + 1))$ time. Aggregating these, we get $\Theta(\log(k) + \log(k - 1) + \ldots + \log(1)) = \Theta(k \log k)$ as we showed in the first supervision. The subsequent steps simply iterate through the edges, without performing any operation on the priority queue. Hence, this gives a running time of $\Omega(k^\alpha + k \log k)$.
Note: The way the question is currently phrased (i.e. with α constant), for $\alpha > 1$ this is trivial because k^α is the dominating term).

Exercise 4.P.11 [Negative edges] *In the following exercises, run dijkstra as described in lectures, which loops until* `toexplore` *is empty. Some textbooks use different versions of Dijkstra's algorithm, that terminate once a destination vertex has been popped, or that don't allow popped vertices to re-enter* `toexplore`*.*

(a) By hand, run both Dijkstra's algorithm and the Bellman-Ford algorithm on each of the graphs below, starting from the vertex s. The labels indicate edge costs, and one is negative. Does Dijkstra's algorithm correctly compute minimum weights?





(b) Prove that Dijkstra's algorithm will produce the correct distance if it terminates.

(c) Prove that Dijkstra's algorithm will always terminate if the input graph has no negative-weight cycles.

(d) (+) Give a DAG, where Dijkstra's algorithm takes an exponential number of steps to compute the answer.

(e) If re-insertions are not allowed, provide an instance where the algorithm gives the wrong output. In this case could the algorithm loop forever?

(f) Given a directed graph with a single negative edge, give an efficient algorithm to find the shortest path from s to v assuming no negative cycles.

(a) Running Dijkstra's algorithm on the first example, we get the following:

Priority Queue (before)	Vertex	Priority Queue (after)	$d(s,v)$
$(s,0)$	s	$(a,3),(b,2)$	$(s,0),(a,3),(b,2),(t,\infty)$
$(a,3),(b,2)$	b	$(a,3),(t,5)$	$(s,0),(a,3),(b,2),(t,5)$
$(a,3),(t,5)$	a	$(b,-1),(t,5)$	$(s,0),(a,3),(b,-1),(t,5)$
$(b,-1),(t,5)$	b	$(t,2)$	$(s,0),(a,3),(b,-1),(t,2)$
$(t,2)$	t	$\emptyset$	$(s,0),(a,3),(b,-1),(t,2)$

Running on the second example, the negative cycle will lead the algorithm into an infinite loop:

Priority Queue (before)	Vertex	Priority Queue (after)	$d(s,v)$
$(s,0)$	s	$(a,3),(b,2)$	$(s,0),(a,3),(b,2),(t,\infty)$
$(a,3),(b,2)$	b	$(a,3),(t,4)$	$(s,0),(a,3),(b,2),(t,4)$
$(a,3),(t,4)$	a	$(b,-1),(t,4)$	$(s,0),(a,3),(b,-1),(t,4)$
$(b,-1),(t,4)$	b	$(t,1)$	$(s,0),(a,3),(b,-1),(t,1)$
$(t,1)$	t	$(a,2)$	$(s,0),(a,2),(b,-1),(t,1)$
$(a,2)$	a	$(b,-2)$	$(s,0),(a,2),(b,-2),(t,1)$
$(b,-2)$	b	$(t,0)$	$(s,0),(a,2),(b,-2),(t,0)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

The execution of the Bellman-Ford algorithm depends on the way we order the edges. For any ordering, the algorithm terminates after $|V|$ iterations.

(b) Assume that the output is wrong. Consider an optimal path p from s to t. Since, it did not give the correct answer, it means that at least one of the edges (let (u,v)) is not relaxed. But since `v.minweight` never increases, it means that (u,v) was not relaxed after u was popped the last time (contradicting the operation of Dijkstra). Hence, there cannot be an unrelaxed edge and Dijkstra having terminated.

(c) We will use the following two observations:

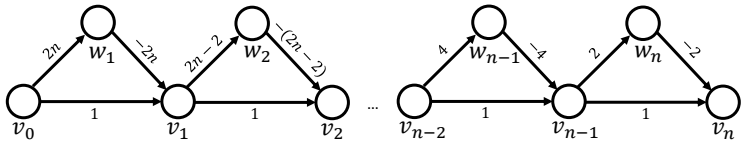
Observation 1: During Dijkstra's execution `u.minweight` always corresponds to the weight of a valid path.

Observation 2: The path corresponding to `u.minweight` will never have a non-negative weight loop.

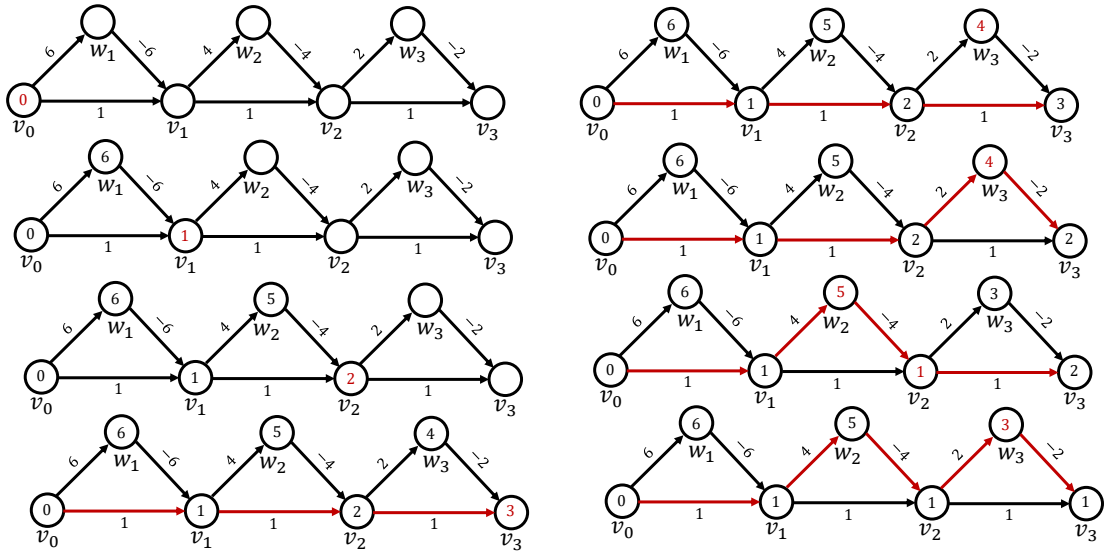
Assume that it did. Then, let (v, w) be the last edge before the cycle forms. For the cycle to form it means that w.minweight + cycle weight $<$ w.minweight, but this cannot be the case if cycle weight ≥ 0. Hence, there cannot be a cycle.

Hence, `u.minweight` corresponds to a simple path. But the number of simple paths in the graph is upper bounded by $2^{|V|}$ (a node will or will not be present). Each time we re-insert a node n to the priority queue it means that `u.minweight` changed. Since there are $2^{|V|}$ possible weights for a path, the vertex n cannot be inserted more than this number of times. So, the algorithm should terminate after $\mathcal{O}\left(2^{|V|} \cdot |V|\right)$ steps.

(d) In the following construction, every time $\mathtt{v}_i\mathtt{.minweight}$ is reduced, the change cascades along to every vertex to the right of v_i; and w_i is only touched after the cascade from v_i has completed. The total running time for computing minweights from v_0 is $\Theta(2^n)$.



The following figure shows the first few steps of the execution for $n = 3$:



(e) It is guaranteed to terminate since each vertex can be popped at most once from the priority queue. However, it is not guaranteed to find the shortest path. One such example is the graph in the previous question, where Dijkstra would stop after identifying the path of length $n - 1$.

(f) Since there are no negative cycles each edge can be used at most once. So, the negative edge (n, m) can also be used at most once. We construct a new graph where each vertex u of the original graph appears twice, as a_u and as b_u. We connect all vertices between a using the edges of the original graph (except for the negative edge) and similarly for b. Then we add $a_n \to b_m$ with weight 0. this means that the negative edge can be traversed at most once. Then run Dijkstra's algorithm on the modified graph, so the shortest path from s to u is $\min(d(a_s, a_u), d(a_s, b_u + w(n \to m))$.

Can you argue formally that this find the shortest path?

Question 1: *How can you detect a negative weight cycle?*
Question 2: *How could you solve this problem if the graph was undirected?*

Exercise 4.P.12 [Shortest-paths modelling problems] Solve the following problem by converting the input to a suitable graph and finding the shortest path there:

(a) Given an unweighted directed graph, find the minimum number of edges that need to be reversed in order to find a path from s to t. **[CodeChef REVERSE]**

(b) Given a graph where each edge has a colour, find the path that minimises the number of colour changes. **[CodeChef YASPP]**

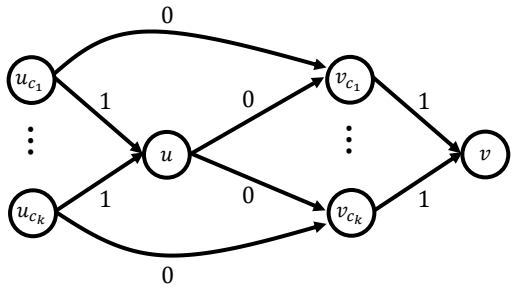
(c) Given a weighted graph G, find the shortest path from s to t given the option to zero out one of the edges.

(d) (Not quite modelling) Find the shortest path whose edge weights form a bitonic (increasing and then decreasing) sequence.

(e) Given a weighted graph G, an agent is placed at vertex s and their target is to reach t. Some of the vertices have a reward of y_v and the agent can get at most one of these. If the agent takes d time steps to reach the t and y_x is the reward they fetch (if any), their total reward is $d - y_x$. Design an algorithm to find the path that minimises this. **[Usaco DINING]**

(a) Model this problem as a shortest path problem: for each edge add the reversed edge with a weight of 1 and assign a weight of 0 to the normal edge. Since the weights are 0 or 1 we can find the shortest path in $\mathcal{O}(V + E)$ time (see Exercise 9).

(b) For each vertex u, we create vertex $u_{c_1}, \ldots, u_{c_k}$, one node for each colour of an edge outgoing u. These nodes represent having colour c_i in the edge entering u. We link u to all u_{c_i} with cost 1, to represent changing color (i.e. being able to use any coloured edge with 0 cost). Also, for each edge (u, v, c), we insert edge u_c to v_c with cost 0 and edge (u, v) with cost 1. Then, we can use the modified BFS of Exercise 9 to find the shortest path from s to t, which corresponds to the path of fewest colour changes, in $\mathcal{O}(V + E)$ time.



(c) Keep states $(n, 0)$ and $(n, 1)$. For each (u, v) connect $(u, 0)$ to $(v, 1)$ with weight 0, $(u, 0)$ to $(v, 0)$ with weight $w(u, v)$ and $(u, 1)$ to $(v, 1)$ with weight $w(u, v)$.

(d) Sort the edges by their weight. Relax the edges once in increasing order and once in decreasing order. This means that a shortest path can only consist of (possibly) some increasing edges followed by (possibly) some decreasing edges. Hence, it will be bitonic. Note that also any bitonic path should have been relaxed.

Exercise 4.P.13 A contractor has written a program that she claims solves the shortest path problem, on directed graphs with edge weights ≥ 0. The program produces `v.distance` and `v.come_from` for every vertex v in a graph, reporting distances and paths from a given start vertex s. Give a $\mathcal{O}(V + E)$-time algorithm to test whether or not the output of the contractor's program is correct.
As a test of your algorithm, what will its output be for the graph above, when the contractor's program produces `s.come_from=None`, `a.come_from=b`, `b.come_from=a`, `s.distance=0`, `a.distance=1`, `b.distance=1`?

[Exercise 24.3-4 in CLRS]

We have to ensure that both the `distance` and the `come_from` array is valid. For the `distance` values, it suffices to check that all edges are relaxed, i.e.

$$\texttt{v.distance} = \min_{w:w\to v} \texttt{w.distance} + \text{cost}(w \to s) \text{for } s \neq v,$$

and that the optimal value is attained by the claimed edge `(v.come_from, v)`. For the starting vertex s, we should have `s.distance`. These checks can be done in $\mathcal{O}(V+E)$ time.
For the `come_from` array, there is the special case where there might be cycle of zeros. So, we must verify that the `come_from` array represents a tree rooted at s. We can do this using a DFS.
The proof of correctness is a bit tedious.

Bellman-Ford algorithm

Exercise 4.P.14

(a) Explain how the Bellman-Ford algorithm can detect if there is a negative-weight cycle in the graph.

(b) Modify the code given in the lectures to find a negative-weight cycle in the graph.

(c) (+) Let N be the number of currencies in the world. Let $e_{ij} \in \mathbb{R}$ be the exchange rate between currency i and currency j. If the exchange rate from US dollars and UK pounds is 1.05 and the exchange rate between UK pounds and US dollars is 0.98, then if you convert 100\$ you get 105£ and then if you convert back to US dollars you get 102.9\$ (so a profit of 2.9\$). You can generalise this to more than 2 currencies. Design an algorithm that given the table e determines if there is such an opportunity.

(d) (+) Design an algorithm that given a set of inequalities of the form $x_i \leq x_j + a_{ij}$, determines if there exists an assignment to x_is, such that all of these are satisfied.

(a) Run the Bellman-Ford algorithm for $|V|$ steps. If there were any changes in the last step, then it means that there is an optimal path that has length at least $|V|$. In this path, there must be some vertex appearing twice. The path from the first occurrence of the vertex to the next must have negative weight (otherwise it wouldn't have been included).

(b) Find the repeating vertex in the optimal path and find the path between the two occurrences of the graph.

(c) We are searching for a path p with $e_{p_1p_2} \cdot e_{p_2p_3} \cdot \ldots \cdot e_{p_{k-1}p_k} > 1$ (where k is the length of the path). The products are a bit hard to handle, so we can take the logarithm of the costs to transform the problem into a problem involving sums,

$$\begin{aligned}
\exists p. e_{p_1p_2} \cdot e_{p_2p_3} \cdot \ldots \cdot e_{p_{k-1}p_k} > 1 \text{ iff (since log is increasing)} \\
\exists p. \log(e_{p_1p_2} \cdot e_{p_2p_3} \cdot \ldots \cdot e_{p_{k-1}p_k}) > \log(1) \text{ iff} \\
\exists p. \log(e_{p_1p_2}) + \log(e_{p_2p_3}) + \ldots + \log(e_{p_{k-1}p_k}) > 0 \text{ iff} \\
\exists p. 0 > -\log(e_{p_1p_2}) - \log(e_{p_2p_3}) - \ldots - \log(e_{p_{k-1}p_k}) \text{ iff}
\end{aligned}$$

which means that we can run the negative-weight cycle detection algorithm on the graph where the weights have been transformed by $w'(u \to v) = -\log(w(u \to v))$.

(d) Read section 24.4 in CLRS for a detailed explanation.