Algorithms Example Sheet 4: Core Questions
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.

BFS/DFS

Exercise 4.C.1 [Implementation aspects of DFS]

(a) Give pseudocode for a function `dfs_recurse_path(g, s, t)` based on `dfs_recurse`, that returns a path from s to t.

[Exercise 2 in Lecturer's handout]

(b) Modify your function from part (a) so that it does not visit any further vertices once it has reached the destination vertex t.

[Exercise 3 in Lecturer's handout]

(c) Do `dfs` and `dfs_recurse` (as given in lecture notes) always visit vertices in the same order? Either prove they do, or give an example of a graph where they do not. You may assume that there is an ordering on vertices, and that `v.neighbours` returns a sorted list of v's neighbouring vertices.

If they do not, then modify `dfs` so they do. Give pseudocode.

[Exercise 4 in Lecturer's handout]

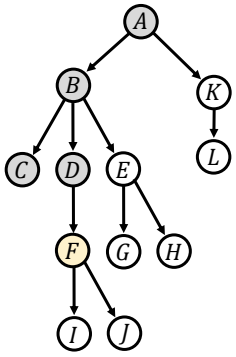
(a) Once the target is reached, construct the answer path and then let each ancestor on the stack append itself on the path. In the end the path is reversed. (If we use a linked list, then we can also prepend efficiently)

```
from algorithms.graph import *


def dfs_recurse_path(g, s, t):
    for v in g.vertices:
        v.visited = False
    return reversed(path(s, t))


def path(v, t):
    if v == t:
        return [v]
    v.visited = True
    p = None
    for w in v.neighbours:
        if not w.visited:
            ans = path(w, t)
            if ans is not None:
                ans += [v]
                p = ans
    return p
```

```
ret = dfs_recurse_path(simple_graph_3, simple_graph_3.vertices[2],
    simple_graph_3.vertices[7])
print(f"Path : ")
for x in ret:
    print(x)
```

Note: The solution using a `parent` or `next` array is essentially implementing the linked list from scratch.

(b) Note that a simple return statement when we find t is not sufficient. Consider the following example, where $s = A$ and $t = F$. When F is reached, we return and so I and J will not be visited. But, nodes E, G, H, K, L will still be visited, when the DFS resumes from a higher ancestor (in this case B).

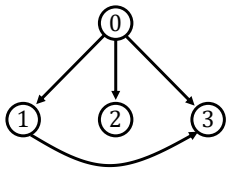


So, we need to find a way to terminate the search for all ancestors. We do this by breaking the neighbour for-loop once we find a path to t.

```
from algorithms.graph import *


def dfs_recurse_path(g, s, t):
    for v in g.vertices:
        v.visited = False
    return reversed(path(s, t))


def path(v, t):
    if v == t:
        return [v]  # This alone is not sufficient.
    v.visited = True
    p = None
    for w in v.neighbours:
        if not w.visited:
            p = path(w, t)
            if p is not None:
                # Found a path, so we don't need to explore
                # any other nodes.
                break
    if p is not None:
        p += [v]
    return p


ret = dfs_recurse_path(simple_graph_3, simple_graph_3.vertices[2],
    simple_graph_3.vertices[7])
print(f"Path : ")
for x in ret:
    print(x)
```

Question: *How does the output of this solution compare to the output of the previous solution?* If there are more than one paths from s to t, then the previous solution returns the one discovered last, while this returns the one discovered first.

(c) The two implementations may process the elements in different orders, even in simple graphs as the following (recursive implementation gives $0, 1, 3, 2$, while the iterative gives $0, 3, 2, 1$):



Failed approach: Reversing the order in which we explore the neighbours in the DFS does not fix the problem. Consider the above graph, then the iterative stack implementation with reversed neighbours gives $0, 1, 2, 3$.

Solution: One solution is to simulate the call stack, i.e. we can store along with each node, the index in the neighbours' list (equivalent to storing the local variable in the recursive function). Each time we pop the top node, we will search for its next non-visited neighbour to explore and append it back to the stack with increased counter. The code below implements this approach:

```
from algorithms.graph import *


def dfs(g, s):
    for v in g.vertices:
        v.seen = False
    # The stack initially contains a single element with 0 counter.
    toexplore = [(s, 0)]
    s.seen = True

    while len(toexplore) > 0:
        v, cnt = toexplore.pop()  # Now visiting vertex v.
        if cnt == 0:
            print(f"Visiting: {v}")
        while cnt < len(v.neighbours):
            w = v.neighbours[cnt]
            cnt += 1
            if not w.seen:
                # Add the old node to the stack with increased counter.
                toexplore.append((v, cnt))
                # Add the new node to the stack with 0 counter.
                toexplore.append((w, 0))
                w.seen = True
                # Once we find the first node we should terminate.
                break


dfs(simple_graph, simple_graph.vertices[0])
```

Exercise 4.C.2 [Implementation aspects of BFS]

(a) Modify `bfs_path(g, s, t)` to find all shortest paths from s to t.

[Exercise 6 in Lecturer's handout]

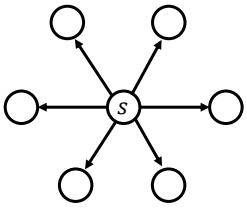
(b) The breadth-first search algorithm from lecture notes uses $\mathcal{O}(1)$ storage within each vertex object (to store the seen flag), plus extra memory for `toexplore`. What is the worst case memory requirement of `toexplore`? Give your answer using Ω notation, in terms of V and E. Modify the algorithm to use $\mathcal{O}(1)$ storage within each vertex object, plus $\mathcal{O}(1)$ extra memory.

(a) We want to store all equally-good parents in the `come_from` list. Here's how it might be done. Note that BFS visits vertices in order of distance from start, thus when we push a novel vertex w it must be at distance $+1$ compared to v.

```
from algorithms.graph import *
from queue import Queue


def bfs(g, s):
    for v in g.vertices:
        v.distance = float('inf')
        v.come_from = []
    toexplore = Queue()
    toexplore.put(s)
    s.distance = 0
    while not toexplore.empty():
        v = toexplore.get()
        for w in v.neighbours:
            if w.distance <= v.distance:
                continue
            w.come_from.append(v)  # w must be at dist. v.distance+1
            if w. distance == float('inf'):
                toexplore.put(w)
                w.distance = v.distance + 1


def paths_to(v):
    return [p+[v] for w in v.come_from for p in paths_to(w)] if v.come_from
        else [[v]]


bfs(simple_graph_3, simple_graph_3.vertices[0])
paths = paths_to(simple_graph_3.vertices[7])
for path in paths:
    print(f"Path : ")
    for x in path:
        print(x)
```

Additional questions by the lecturer:
Question 1: *The algorithm loops until `toexplore` is empty. Can you stop sooner?*
Question 2: *How would we achieve a similar effect in the Dijkstra setting, where edges have costs?*

(b) The worst-case is $\Omega(V)$ space and this happens, for example, in the star graph when starting the BFS at vertex s (in the first iteration we add all $n-1$ vertices). So the memory is $\Omega(V)$:




```
from algorithms.graph import *


def bfs(g, s):
    for v in g.vertices:
        v.seen = False
```

```
            v.left = None
        s.seen = True
        # The invariant is that
        # toexplore = rightmost -> rightmost.left -> rightmost.left.left -> .. ->
            leftmost
        leftmost = s
        rightmost = s
        while rightmost is not None:
            v = rightmost
            print(f"Visiting: {v}")
            for w in v. neighbours:
                if not w.seen:
                    # To add something to the queue, access through the leftmost
                        pointer.
                    leftmost.left, leftmost = w, w
                    w.seen = True
            rightmost.left, rightmost = None, v.left


bfs(simple_graph_3, simple_graph_3.vertices[0])
```

Note: According to the lecturer, this is useful "if you want to search an absolutely gigantic graph, [since] you have to find ways to stream it through memory, without keeping it all in memory".

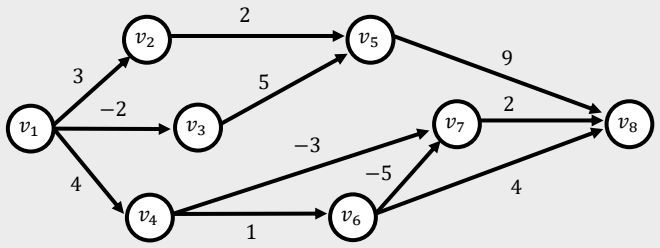
Directed Acyclic Graphs (DAGs)

Exercise 4.C.3

(a) Describe the topological sorting algorithm and argue why it works.

(b) What is its time complexity?

(c) Show its operation in the following DAG.



(d) Give a few examples of DAGs and the interpretation of the topological ordering (e.g. build systems, neural networks).

Exercise 4.C.4 Given a DAG with weights,

(a) Design an algorithm that finds the shortest path from s to t in time $\mathcal{O}(V+E)$.

(b) Design an algorithm that finds the longest path from s to t in time $\mathcal{O}(V+E)$.

(c) Design an algorithm that counts the number of paths from s to t in time $\mathcal{O}(V+E)$.

(d) (optional ++) Design an algorithm that finds the path of maximum average length (i.e. the sum of the weights in the path normalised by the number of edges in the path) from s to t in time $\mathcal{O}(V+E)$.

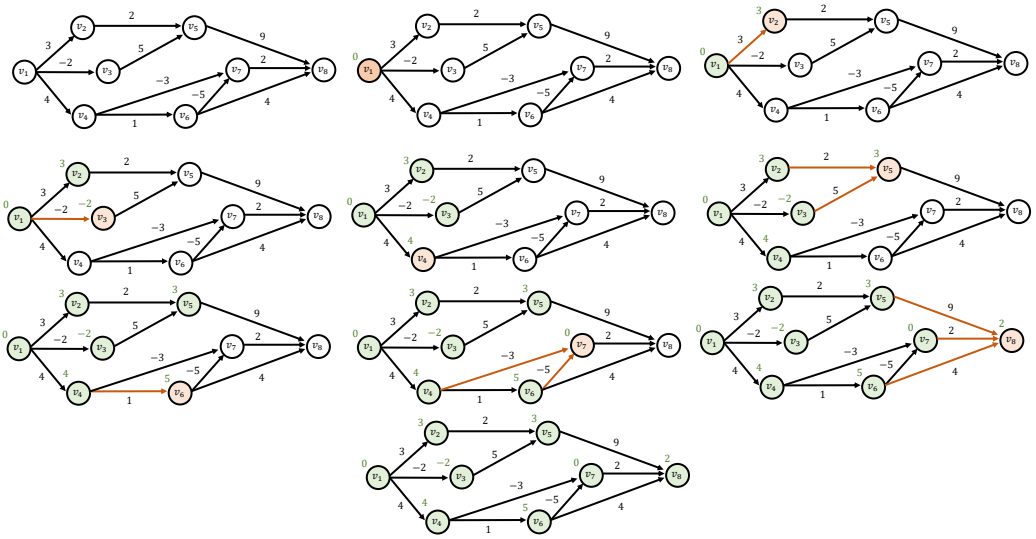
(a) We can find the longest path by first finding the topological order of the nodes in the DAG and then

looping over the vertices in topological order, we compute the following recursive equation:

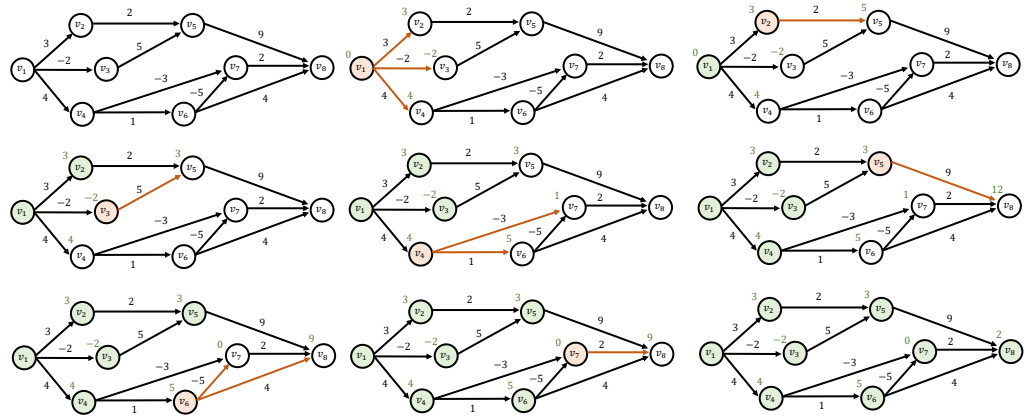
$$d[v] = \begin{cases} \min_{u \in \text{incoming}(v)} d[u] + \text{weight}(u \to v) & \text{if incoming}(v) \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

where $d[s] =$. The invariant is that when we are at vertex v, we have computed the correct shortest path for all previous nodes in the topological order (and hence all incoming vertices for v). So, the update for v will be correct.

Below is an example run on the DAG of the previous exercise.



You can also run the dynamic programming in the forward direction, so that once you have found the shortest path to v, you relax the outgoing edges.



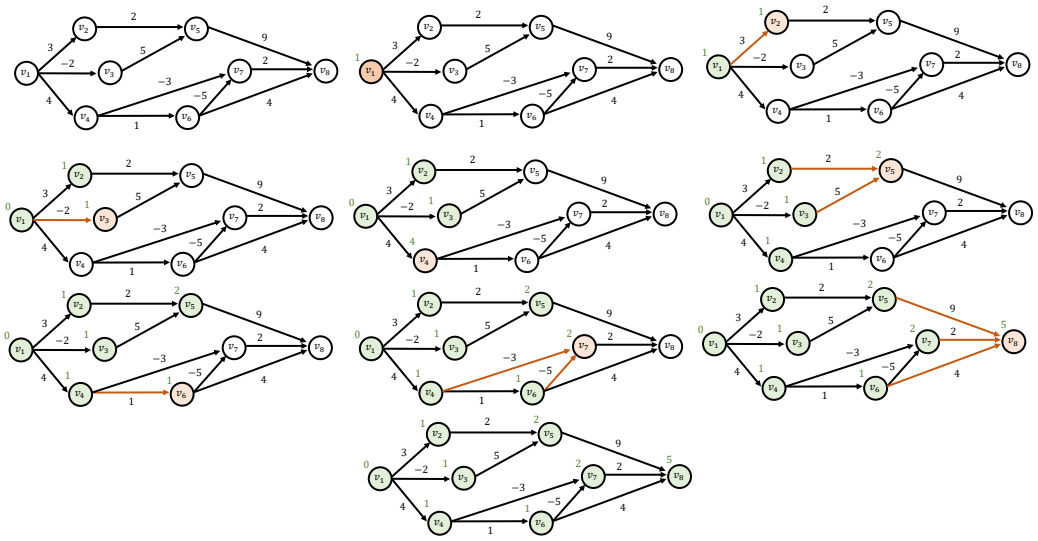
(b) As in the previous question, but we swap the min operator with max.

$$d[v] = \begin{cases} \max_{u \in \text{incoming}(v)} d[u] + \text{weight}(u \to v) & \text{if incoming}(v) \neq \emptyset \\ -\infty & \text{otherwise} \end{cases}$$

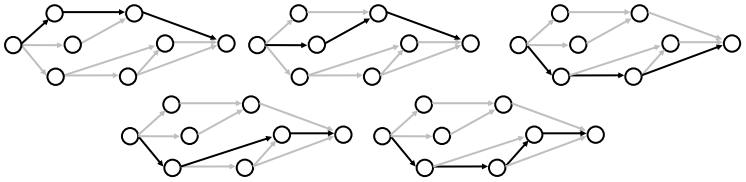
(c) As in the previous question, but we swap the max operator with $+$, weights are set to 0 and $d[s] = 1$ (since there is one path from s to s):

$$d[v] = \begin{cases} \sum_{u \in \text{incoming}(v)} d[u] & \text{if incoming}(v) \neq \emptyset \\ -\infty & \text{otherwise} \end{cases}$$

Below, an example run of the algorithm on the previous graph:



And here are the 5 possible paths:



Note: All these operations can be seen as the same algorithm but under a different semiring.

Exercise 4.C.5 Explain how to model a dynamic programming recurrence relation using a graph. Draw this graph for the Longest Common Subsequence (LCS) problem with $n = 5$ and $m = 3$.

Dijkstra's algorithm

Exercise 4.C.6 In a directed graph with edge weights, give a formal proof of the triangle inequality

$$d(u, v) \leq d(u, w) + c(w \to v) \text{ for all vertices } u, v, w \text{ with } w \to v$$

where $d(u, v)$ is the minimum weight of all paths from u to v (or ∞ if there are no such paths) and $c(w \to v)$ is the weight of edge $w \to v$. Make sure your proof covers the cases where no path exists.

[Exercise 8 in Lecturer's handout]

There are two cases to consider depending on whether there exists a path from u to w:

- If there is no path, then $d(u, w) = \infty$ and so the RHS of the inequality is ∞, so it will be at least as large as the LHS.
- If there is a path $p = u \rightsquigarrow w$, appending the edge $w \to v$ creates a path p' from u to v (namely $u \rightsquigarrow w \to v$) with weight $c(p') = d(u, w) + c(w \to v)$. Because $d(u, v)$ is the weight of the shortest path from u to v, it must $c(p') \geq d(u, v) \Rightarrow d(u, w) + c(w \to v) \geq d(u, v)$.

Exercise 4.C.7 [Proving shortest path properties] Read section 24.5 in CLRS and provide proofs for some of the following:
(a) Upper-bound property

(b) No-path property

(c) Convergence property

(d) Convergence property

(e) Path relaxation property

(f) Predecessor-subgraph property

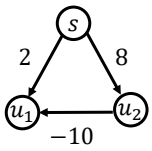
The proofs are in section 24.5 of CLRS.

Bellman-Ford algorithm

Exercise 4.C.8 In the course of running the Bellman-Ford algorithm, is the following assertion true? "Pick some vertex v, and consider the first time at which the algorithm reaches line 7 with `v.minweight` correct i.e. equal to the true minimum weight from the start vertex to v. After one subsequent pass of relaxing all the edges, `u.minweight` is correct for all $u \in$ neigbours(v)." If it is true, prove it. If not, provide a counterexample.

[Exercise 14 in Lecturer's handout]

This is not true. Consider the following graph:

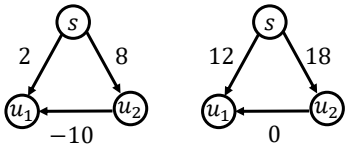


In the first iteration, we are relaxing s to its correct value `s.minweight` (which is 0) and setting `u1.minweight = 2` and to `u2.minweight = 8`. But, the distance to u_1 is not the optimal which is -2.
Lecturer's comment: The point of this exercise is to show that mimicking the proof technique in Dijkstra's does not work for the Bellman-Ford. We need to consider the last correct node in the optimal path for each vertex.

Exercise 4.C.9 An engineer friend tells you there is a simpler way to reweight edges than the method used in Johnson's algorithm. Let w^* be the minimum weight of all edges in the graph, and just define $w'(u \to v) = w(u \to v) - w^*$ for all edges $u \to v$. What is wrong with your friend's idea?

[Exercise 25.3-4 in CLRS]

This does not work. Consider the following graph and its transformation:



The shortest path from s to u_1 in the original graph is $s \to u_2 \to u_1$ and in the second it is $s \to u_1$.
Lecturer's comment: PLEASE tell your students that to answer a question like this they must produce a concrete example and not some waffly text.

Exercise 4.C.10 [Floyd-Warshall algorithm] We are given a directed graph where each edge is labelled with a weight, and where the vertices are numbered $1, \ldots, n$. Assume it contains no negative weight cycles. Define $F_{ij}(k)$ to be the minimum weight path from i to j, such that every intermediate vertex is in the set $\{1, \ldots, k\}$. Give a dynamic programming equation for $F_{ij}(k)$, and a suitable definition for $F_{ij}(0)$.

Since we are given that the graph does not contain any negative cycles, it means that in the optimal path from some starting vertex s to some target vertex t (if it exists), will contain a vertex at most once.
Floyd-Warshall algorithm proceeds by determining $F_{ij}(k)$ if there is a path from i to j using only vertices $1, \ldots, k$. The base case is when $k = 0$,

$$F_{ij}(0) = \begin{cases} \text{weight}(i \to j) & \text{if } i \to j \\ \infty & \text{otherwise.} \end{cases}$$

For the general case, there will be a path $i \rightsquigarrow j$ using vertices $1, \ldots, k$ in one of the following two cases:

- if there was a path $i \rightsquigarrow j$ using vertices $1, \ldots, k-1$.
- if there is a path $i \rightsquigarrow k$ (using vertices $1, \ldots, k-1$) and there is a path $k \rightsquigarrow j$ (using vertices $1, \ldots, k-1$)

Hence,

$$F_{ij}(k) = \min(F_{ij}(k-1), F_{ik}(k-1) + F_{kj}(k-1)).$$

Question: *When will the algorithm find the optimal path between i and j?* If the z is the largest index of the vertex used in the optimal path, then when computing $F_{ij}(z)$, the algorithm will have already found the path $i \rightsquigarrow z$ (since it consists of smaller indices) so $F_{ij}(z-1)$ contains the correct value and similarly for $z \rightsquigarrow j$.