Algorithm Theory - Theoretical exercise 5

Téo Bouvard

March 24, 2020

Problem 1

Since we deal with a directed acyclic graph, we can use the acyclic property to count the number of paths from a source node to a target node recursively. By using memoïzation, we can avoid solving overlapping subproblems. However, we can also solve this problem by using a bottom-up approach, using the same idea we used for computing the Fibonacci sequence in the previous assignment. We start at the target node with a number of paths equal to one, and add the node's counter to its parents while adding them to FIFO queue if they have not already been visited. If we do that until the queue is empty, we get the number of paths from every node in the graph to the target, while still having a linear complexity. An example implementation is given below. For simplicity, we assume that we have access to a FIFO data structure, and that the graph data structure can retrieve the predecessors of a node. Furthermore, we assume that nodes are objects with an *already_visited* attribute. If that was not the case, we could still maintain a separate list of already visited nodes.

```
N-PATHS(G, s, t)
    let n[0...length(G.V) − 1] be a zero-initialized array
    queue = FIFO()
    queue.put(t)
    n[t] = 1
    while queue.empty() = FALSE
        current = queue.get()
        pred = G.predecessors(current)
        for p ∈ pred
            if p.already_visited = FALSE
                queue.put(p)
                p.already_visited = TRUE
            n[p] += n[current]
    return n[s]
```

This algorithm has complexity $\mathcal{O}(V + E)$ because it considers each node and its predecessors at most once. In fact it is nearly identical to a BFS, except that its goal is to count paths rather than searching for a particular node.

Problem 2

a) Contrarily to what is written in the lecture notes, Prim's algorithm time complexity is mainly influenced by the data structure used to keep track of the edges weights. When using a simple adjacency matrix, linearly searching for the next lightest edge makes the algorithm run in $\mathcal{O}(V^2)$, but it can be greatly improved by using a binary heap, making it run in $\mathcal{O}(E \log V)$ or a Fibonacci heap, lowering down the time complexity to $\mathcal{O}(E + V \log V)$. On the other hand, Kruskal's time complexity is $\mathcal{O}(E \log E)$, using a disjoint-set data structure. For dense graphs, Prim is asymptotically faster than Kruskal. If time complexity is the only factor to consider, I would prefer to use Prim's algorithm when $E = \omega(V)$.

b) As Kruskal's algorithm incrementally adds edges in increasing weight order, we could add a virtual weight to all edges not connecting with a certain node as an incentive for the algorithm to make most connections through this particular node.

c) When using DFS from a single source node, we can list the cycles from this source node by displaying the current path when we re-visit this node. To find all cycles in a directed graph, we could repeat this process for each node of the graph, which would lead to a complexity of V times the complexity of DFS i.e. $\mathcal{O}(V(V+E))$.

Problem 3

We compute the distance and predecessors tables, according to Dijkstra's algorithm.

d	A	B	C	D	E
init	0	∞	∞	∞	∞
pick A	0	10	3	∞	∞
pick C	0	7	3	11	5
pick E	0	7	3	11	5
pick B	0	7	3	9	5
pick D	0	7	3	9	5

π	A	B	C	D	E
init	$\emptyset$	?	?	?	?
pick A	$\emptyset$	A	A	?	?
pick C	$\emptyset$	C	A	C	C
pick E	$\emptyset$	C	A	C	C
pick B	$\emptyset$	C	A	B	C
pick D	$\emptyset$	C	A	B	C

Problem 4

a) We use the Ford-Fulkerson method to find a full flow of maximum value in the graph.

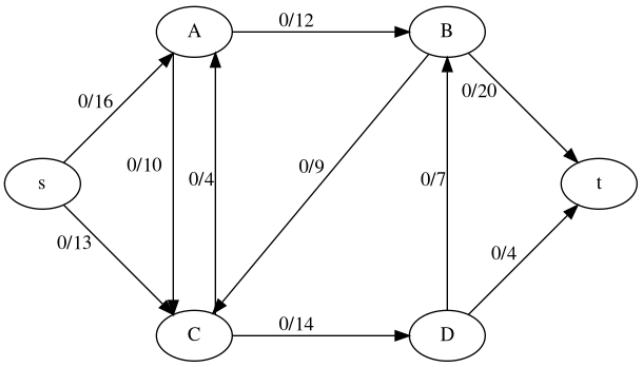

Figure 1: Initial state

The first augmenting path we use is s $\to$ A $\to$ B $\to$ t, which has a residual capacity of 12. This results in a total flow of 12.

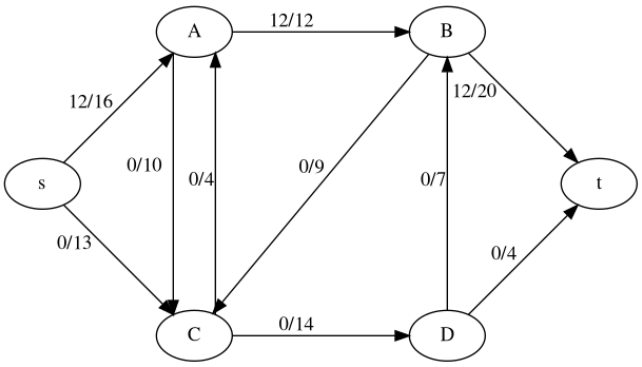

Figure 2: Augmenting via s $\to$ A $\to$ B $\to$ t

The second augmenting path we use is s $\to$ C $\to$ D $\to$ t, which has a residual capacity of 4. This results in a total flow of 16.

The third augmenting path we use is s $\to$ C $\to$ D $\to$ B $\to$ t, which has a residual capacity of 7. This results in a total flow of 23.

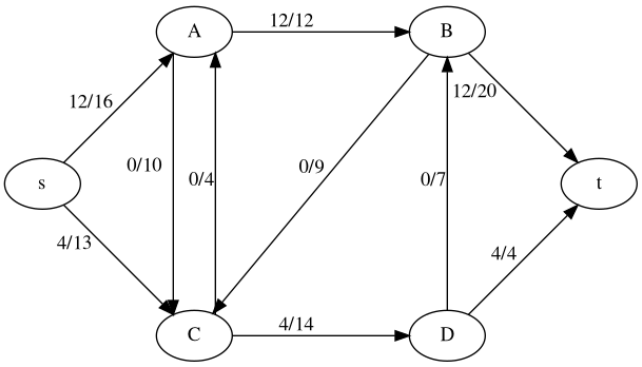

Figure 3: Augmenting via s → C → D → t

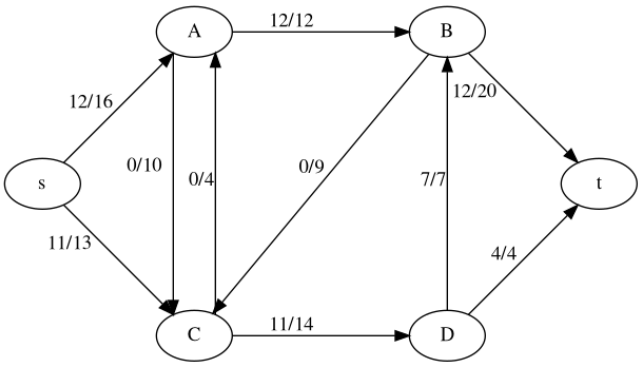

Figure 4: Augmenting via s → C → D → B → t

At this point, we can see that the if an augmenting path existed, it would have to end with B → t as D → t is already at its maximum capacity. However, the two edges through which B could increase its flow (A → B and D → B) are already at maximum capacity. Therefore, no augmenting path exists. Since we can not find another augmenting path, the maximum flow through this network is 23.

b) In the above algorithm, we picked augmenting paths at random. This procedure can take a lot of time to converge to a maximum flow, depending on the network structure. The Edmonds–Karp algorithm solves this problem by imposing an order on which the augmenting paths must be chosen. By picking the shortest of the existing augmenting paths, we can prove that at least one edge on the network becomes saturated at each iteration. This bounds the time complexity of the algorithm by $\mathcal{O}(EV^2)$.