Design of Algorithms (L.EIC016)

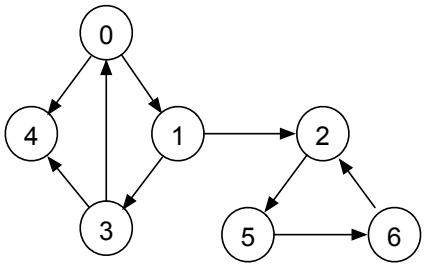
First Test

Spring 2023

Faculdade de Engenharia da Universidade do Porto (FEUP)
Departamento de Engenharia Informática (DEI)

March 31, 2023

Question 1 [2 points]
Consider the directed graph G below with 7 nodes and a representation that uses an adjacency list and where neighbors are list in ascending order of node numbering.

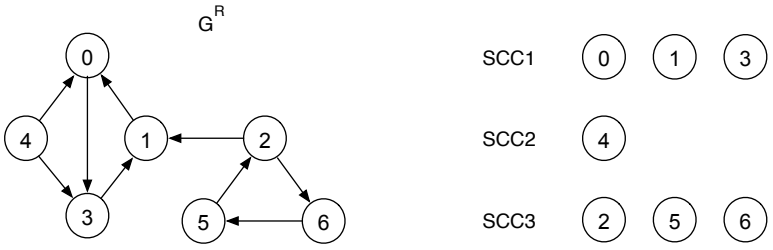


For this graph answer the following:

a) Show the reverse graph G^R and its strongly-connected components (SCCs)
b) Determine a possible topological sorting of G.
c) Discuss the time complexity of the topologic sorting in terms of the number of vertices (nodes) and number of edges.

Solution:

a)



b) The directed graph has cycles, so there is no possible topological sorting order.
c) Each edge is visited only once, and each node inserted in the queue only once, so the time complexity is O(E+V).

Question 2 [3 points]

Ralph wants to drive from Los Angeles to Chicago along route 66. His car's fuel tank, when full, holds enough fuel to travel n miles, and his map gives the distances between gas stations on this route. Ralph wishes to make as few stops for fuel as possible along the way. Give an efficient algorithm by which he can determine which gas stations he should stop at, and prove that your strategy yields an optimal solution.

Solution:

The optimal strategy is a greedy one. Starting will a full tank of gas, Ralph should go to the farthest gas station he can get to within n miles of Los Angeles. Fill up there. Then go to the farthest gas station he can get to within n miles of where he filled up, and fill up there, and so on. Looked at another way, at each gas station, Ralph should check whether he can make it to the next gas station without stopping at this one. If he can, skip this one. If he cannot, then fill up. Ralph does not need to know how much gas he has or how far the next station is to implement this approach, since at each fill-up, he can determine which is the next station at which he will need to stop.

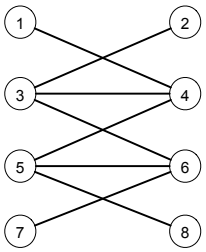
This problem has optimal substructure. Suppose there are m possible gas stations. Consider an optimal solution with s stations and whose first stop is at the k^{th} gas station. Then the rest of the optimal solution must be an optimal solution to the sub-problem of the remaining $(m-k)$ stations. Otherwise, if there were a better solution to the subproblem, i.e., one with fewer than $(s-1)$ stops, we could use it to come up with a solution with fewer than s stops for the full problem, contradicting our supposition of optimality.

This problem also has the greedy-choice property. Suppose there are k gas stations beyond the start that are within n miles of the start. The greedy solution chooses the k^{th} station as its first stop. No station beyond the k^{th} works as a first stop, since Ralph runs out of gas first. If a solution chooses a station $j < k$ as its first stop, then Ralph could choose the k^{th} station instead, having at least as much gas when he leaves the k^{th} station as if he would have chosen the j^{th} station. Therefore, he would get at least as far without filling up again if he had chosen the k^{th} station.

If there are m gas stations on the map, Ralph needs to inspect each one just once. The running time is O(m).

Question 3 [2 points]

Using the Brute-Force algorithmic approach devise an algorithm that determines a maximal matching for a generic undirected bipartite graph. In addition, develop a lazy strategy that can find a maximal matching as soon as possible. To clarify, we illustrate a sample instance of a bipartite undirected graph below. Your algorithm should be generic and work correctly for a generic bipartite undirected graph.



Solution:

A simple brute-force approach would call for the enumeration of all possible E edge subsets and then test each of them to see which one of them has at least one node with more than one incident edge as that would disqualify the selection. The time and space complexity of such an algorithm would be $2^E \times V \times E$ where E is the cardinality of the set of edges of the graph and V its number of nodes. The pseudo-code below depicts such an approach where the counting of edges can be done by associating a Boolean value to each edge in the graph and then repeatedly clearing and setting the edge Boolean variable to determine the number of incident edges of each node v in the graph for each tentative maximal edge set.

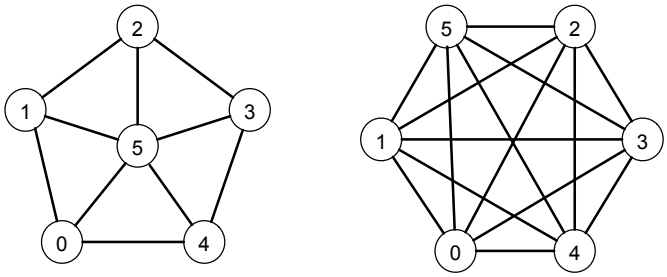
```
match maximalBipartiteMatching(graph G){
  integer K = 0;
   matchSet = genAllPossibleEdgeSubSets(E(G));
   for all set s in matchSet do
     if(checkValidMatching(G,v) == true)
       K = max(K,cardinality(s));
     end if
   end for
   return K
}
```

```
boolean checkValidMatching(graph G, edgeSet s)
  for all v in V(G) do
    for all node w in neighbors(v) do
      c = count edges (w,v) in s
      if (c > 1) then
        return false;
      end for
    end for
  return true
```

One can "statistically" improve the algorithm's running time by lazily generating the edge subsets and start by the larger sets first, i.e., begin with a subset that includes all edges and then progressively moving to lower cardinality tentative edge subsets. Also, and given that the maximal number of matches is given by max(L,R) where L and R are the cardinality of the set of nodes on the left and on the right, respectively, one can also begin with a lower cardinality subset. Moreover, we generate and test the edge subsets lazily. Since we begin with the highest value first, if successful, and once we find a maximal match the algorithm can immediately terminate. Note that, although the short-cut can lead to an expected fast execution time, the algorithm still exhibits the same exponential worst-case time complexity as the original, "simple" approach. Nevertheless, the lazy version now has much lower space complexity as one needs not to store the entire set of edge subsets, but only the one currently being considered, so a worst-case space complexity of E.

Question 4 [2 points]

Consider the two weighted undirected graphs below as an illustrative example of a class of undirected and connected graphs with N nodes where all edges have a unit weight. The specific graphs below are particular instances of this class of graphs for N = 6 and for a particular arrangement of nodes.



For this class of graphs adapt Prim's algorithm, to find a minimum-cost spanning tree possibly simplifying it and arriving at a more efficient (from a time complexity standpoint) algorithmic variant. Describe your algorithm using a "reasonable" pseudo-code and argue for its improved time complexity.

Solution:

If all the edges are of the same weight, the more expensive portion of Prim's algorithm, the sorting of the edges, is unnecessary. As such, the algorithm just needs to scan all the edges and check for cycles. A single, "visited" Boolean value associated with each node (initialized to false) is sufficient to check if the node has been visited and as such as simple DFS traversal is sufficient until N-1 edges are added. The algorithm complexity is the complexity of a DFS traversal, i.e., O(E+V) and the MST cost is N-1 as each edge has unit cost.

Question 5 [2 points]

Consider the recursive function G depicted below with one integer argument and other arguments omitted here for simplicity. The G function makes use of another function, F which does some work that is either polynomial, logarithmic or even constant time with respect to the input instance's size, n.

```
public int G (int n, …) {
   if (n < 1) {
     return 1;
   }

   F(n, …);

   int x = G(n / 9, ...);
   int y = G(n / 9, ...);
   int z = G(n / 9, ...);

   return x + y * z;
}
```

Derive a recurrence that models the execution time complexity of G and derive its asymptotic time complexity using the Master Theorem as discussed in class. Clearly, you need to examine various complexity scenarios regarding F itself. Discuss the overall complexity when F(n, …) is both linear and quadratic with respect to n. Would it make sense to invest in a more efficient implementation of F to be, say, logarithmic with respect to n?. Why or why not?

Solution:

The recurrence implied by this code is $T(n) = 3T(n/9) + f(n)$ ignoring the simple constants implied by the arithmetic operation in the return statement of the function and the complexity of the base case. For this case, a = 3, b = 9 and so, $\log_b(a) = 0.5$. So, if F is $\Theta(n^1)$, or even worse $\Theta(n^2)$, we get $T(n) = \Theta(n^1)$, and $T(n) = \Theta(n^2)$, respectively. That is the homogeneous solution dominates. If F is $\Theta(\sqrt{n})$, We have the second case and get $T(n) = \Theta(\sqrt{n} \log n)$. If F were, say O(log n), if would also be polynomially lower than $\sqrt{n}$, and so the overall asymptotical complexity of $T(n)$ would still be $\Theta(\sqrt{n})$. So, there is indeed an advantage to try to get F to be logarithmic with respect to n.

Question 6 [1 point]

In class we examined the 0/1 Knapsack problem and its fractional variant where fractions of selected items can be included in the Knapsack with maximum capacity W. We also discussed a greedy algorithm where items were inserted in the Knapsack by descending order of their ratio of value per weight, *i.e.,* (v/w). We showed that this greedy approach was indeed optimal.

In this problem we explore another variant of the fractional Knapsack problem in which all the items have the same value, but possibly different weights. Note that you can include a fraction of any item in the Knapsack.

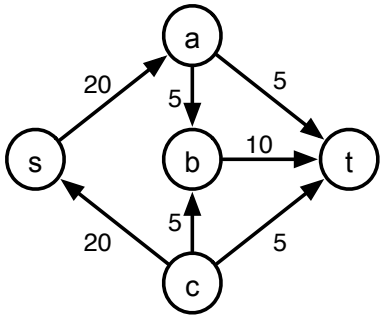
For this particular class of problem instances, derive and discuss the complexity of an efficient algorithm that maximizes the value of the items in a Knapsack of capacity W and argue for the optimality of your solution.

Solution:

Given that all items have the same value, say v, to maximize the value of the knapsack, we need to put as many items in the knapsack as possible. As such, a greedy and optimal approach will be to include the items in the knapsack by increasing order of weight, i.e., start by including the lightest items first before considering the heavier items. A simple algorithmic implementation will sort all the items by ascending weight value. Whenever the knapsack is approaching the weight limit, we insert a fractional portion of the last item and the algorithm terminates. The overall complexity is dominated by the sorting step, and so it O(n log n).

One can "reuse" the same optimality proof used for the original unconstrained fractional Knapsack problem by observing that ordering the items by decreasing value of v/w is the same as to order them by increasing value of w for the case where all values of v for all the items are identical ($v_i/w_i > v_{i+1}/w_{i+1} \Rightarrow w_i < w_{i+1}$). Alternatively, you could argue that if you were to replace an item of a higher weight with another one of lower weight, given that the value of the knapsack would not increase (as they all have the same value) you could build reserve capacity for either another item and thus increase the value of the knapsack with even a fraction of that additional item.

Question 7 [4 points]
Consider the network flow below and nodes s and t as source and sink, respectively.



For this network, derive:

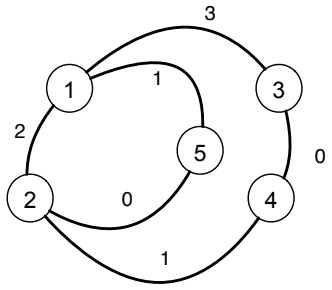
a) A minimal cut indicating the corresponding capacity.
b) List the augmenting paths using Edmonds-Karp algorithm strategy, indicating the order in which the Edmonds-Karp would explore them.
c) The maximal flow between s and t.
d) Identify the edges whose capacity can be reduced, and to which value, while not reducing the overall maximal flow between s and t. Explain your choice.

Solution:

a) A minimal-cut of cut capacity of 10 is S:{s,a} and T:{c,b,t}.
b) path1= {s-a-t} with capacity 5; path2 = {s-a-b-t} with capacity 5; No more augmenting paths afterwards.
c) The maximal flow between s and t is 10.
d) The edge {b-t} can be reduced to 5 and {s-a} to 10. All other edges are at full capacity.

Question 8 [2 points]

Consider the undirected and weighted graph depicted below.



For this graph consider the following:

a) Using Dijkstra's greedy shortest-path algorithm determine the values of the shortest-distance between node 1 and the other nodes. Should the values this algorithm produces after one iteration after the initialization phase.
b) List the order in which the nodes are examined and considered as part of the resulting shortest-paths tree.
c) What is the resulting MST using Prim's algorithm rooted at node 1 and why is this tree the same as the tree of shortest paths resulting from Dijkstra's algorithm? Discuss when are they different?

Solution:

a)

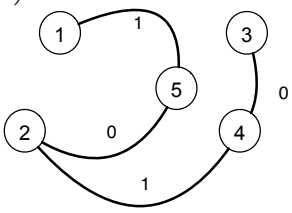
	1	2	3	4	5	
d	0	2	3	∞	1	initial

	1	2	3	4	5	
d	0	1	3	3	1	after 1 iteration

	1	2	3	4	5	
d	0	1	2	2	1	final

b) The list of node insertion in the shortest-paths tree is: 1, 5, 2, 4, 3

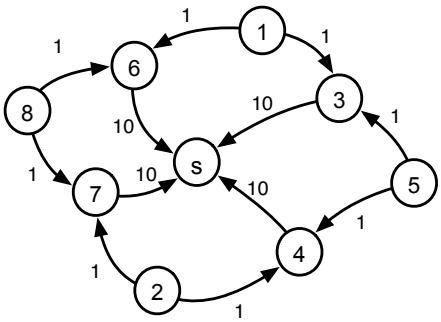
c)



In this case there is a single shortest path from 1 to each of the nodes, so the MST found using Prim's algorithm and Dijkstra are necessarily identical.

Question 9 [2 points]

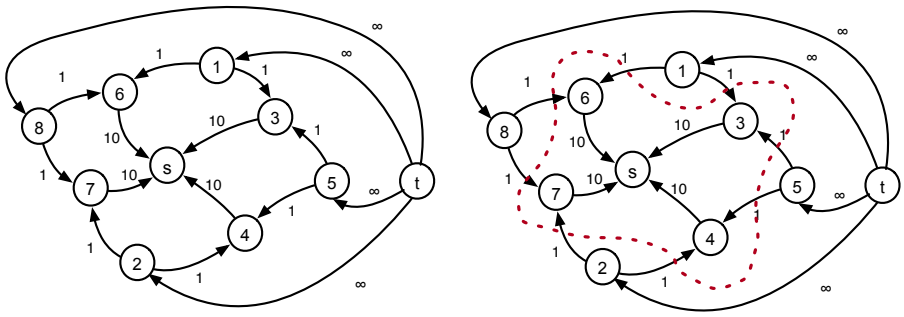
You are asked to develop an efficient algorithm that determines the maximum flow to a given node of a network flow graph, say node `s`, from all the other graph nodes. A simple algorithm could consider adding the capacity of all the edges directly connected to `s`, but as the sample network flow graph below shows, this can be a gross over estimation of this maximum flow.



You may describe your solution using pseudo-code, if you would like, but a description of your approach alongside a simple, but not simplistic example with a network flow graph example, would suffice. Also, related the complexity of your implementation with the complexity of the implementation of the max-flow algorithms discussed in class.

Solution:

One way to approach this problem is to consider that only nodes of the network flow connected only to one or two nodes are considered to be the outermost (or peripheric) nodes of the network which in this case would consist of nodes 1, 2, 5 and 8. Augmenting the network flow with a node t that would be the source of the node and with infinite capacity edges connecting just to these nodes would now define the maximum flow that could be injected into the network having as s as the sink. The figure below depicts this augmented network flow graph resulting in a more realistic flow of 8 into node s.



Possibly a better structured or an algorithm with a better rationale would consider identifying the periphery of the graph by carrying out a breath-first-transversal (following the reversed edges) and identifying all the leaf nodes of the resulting traversal. Then, using the newly added source t connected to the leaf nodes of the BFS tree would define the "injection" point of the network. Carrying out a DFS is a simple "linear" step (on the number of edges and nodes of the graph) and so the resulting algorithm's complexity is the same complexity as the Edmonds-Karp maximum flow algorithm.