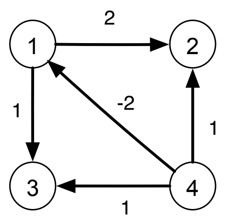
Design of Algorithms (L.EIC016)

Second Test with Solutions

June 16, 2023

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

Question 1 [4 points]. Consider the directed and weighted graph below.

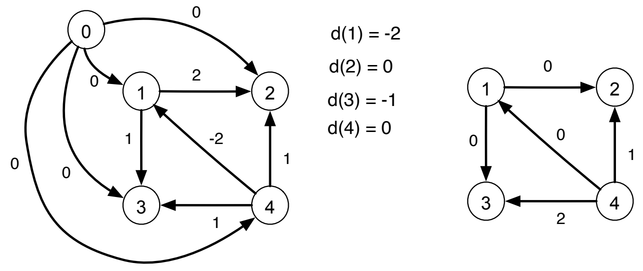


For this graph answer the following questions:

a) Can you apply Dijkstra single-source algorithm (in a pairwise fashion) to find all the shortest paths between all the nodes of this graph? Why or, why not?
b) Would your answer to the previous question change if there were other negative edges? Why or why not?
c) Can you use Bellman-Ford-Moore algorithm? Why, or why not?
d) If possible use Johnson's edge reweighting algorithm and determine the shortest path between all nodes of these graph and every other node. Show your work by depicting the augmented graph with the revised non-negative weights due to Johnson's algorithm.

Solution:

a) Although in general you cannot use Dijkstra shortest-path algorithm in the presence of edges with negative weights, for this particular graph you can.
b) In general, in the presence of negative edges, we cannot use Dijkstra's shortest-path algorithm.
c) Yes, although there are negative edges, there are no negative cycles.
d) It is possible and desirable to use edge reweighting so that we can use the more efficient O(n^3) algorithm rather than simple or straightforward application of the Bellman-Ford which would lead to an O(n^4).



Question 2 [3 points]
Develop an efficient (non-brute-force) algorithm that takes a sequence X of n characters $X =< x_1, \dots, x_n >$, and returns the length of the longest palindrome subsequence (not necessarily contiguous) in X. A subsequence is a palindrome if it is the same whether read left to right or right to left. For example, if $X = ABBABCCABBAB$ the longest palindrome subsequence in X is "ABBACCABBA" with length 10. Note that the subsequences "*ABBAABBA*", "*ABABABA*" and "*BBBBBB*" are also palindromes of X, but not the longest ones.

Formulate an optimal substructure for this problem and outline a dynamic-programming algorithm that returns the length of the longest palindrome of X, explicitly describing the used recurrence. Derive the time and space complexity of your solution, which should not use a brute-force approach.

Solution:
A brute-force solution to this problem is to generate all subsequences of the given X sequence and find the longest palindrome subsequence. However, such a solution has exponential time complexity, and we need to find an algorithm that responds to the problem more efficiently. Considering a given sequence $X[0 \dots n-1]$ of length n. Strings of length 1 (i.e., each single character) are palindromes of length 1. If the last and first characters of X are the same it means that 2 characters count towards the length of the palindrome subsequence and the initial problem reduces to an identical subproblem for the remaining subsequence, that is, find the length of the longest palindrome subsequence in the subsequence $X[1 \dots n-2]$. If the last and first characters of X are different the problem reduces to finding the length of the longest palindrome subsequence between the subsequences $X[1 \dots n-1]$ and $X[0 \dots n-2]$. Such a solution leads to the following recurrence function,

$$m[i,j] = \begin{cases} 1 & \text{if } i = j \\ m[i+1, j-1] + 2 & \text{if } X[i] = X[j] \\ \max\left(m[i+1, j], m[i, j-1]\right) & \text{otherwise} \end{cases}$$

where $m[0, n-1]$ represents the length value of the longest palindrome subsequence of X.

A recursive algorithm that follows the above recursion function solves many subproblems that are repeated many times. This implies that we can solve the problem more efficiently using dynamic programming method, to avoid these overlapping subproblems. For this we use a table matrix $m[][]$ that stores the results of these subproblems. To fill the table of n x n entries we need a space that is quadratic $O(n^2)$ and to compute each entry of the table a constant time, so overall the algorithm is $O(n^2)$ using a simple doubly-nested loop.

Question 3 [3 points]
In class we've seen the concepts of polynomial reduction from problem A into problem B, denoted by $A \leq_p B$, with which we could develop a polynomial algorithm for A having a polynomial algorithm for B. In the context of the study of greedy algorithms, we have used the algorithm of Edmonds-Karp (Max-Flow) to determinate the maximal matching in a bipartite graph.

Considering this reduction answer the following questions:

a) Is this a polynomial-time reduction? Why or why not?
b) What does this reduction tell you about the relative complexity of the Edmonds-Karp algorithm and the algorithm that uses it for solving the maximal bipartite matching? Can you say that the combined algorithm (reduction followed by Edmonds-Karp) is a polynomial-time algorithm for maximal matching on bipartite graphs on the size of the input graph? Explain.

Solution:

a) Yes, in fact, the only things this transformation needs to do is to assign unit weights to all the edges, in a number that is polynomial on the number of nodes (in this case in fact linearly) and add two more nodes and L+R where L and R are the number of nodes on the left and right side of the bipartite respectively. As such this is clearly a polynomial transformation. As seen in class, the value of the Max-Flow is by construction the value (not necessarily unique) of the maximal matching.

b) The general complexity of Edmonds-Karp is O(E |f*|) where f* is the resulting maximal flow. Note that in many cases, where irrational number are at play, the algorithm might not converge at all or converge to the incorrect numeric value. In the worst-case scenario, and when it converges to the correct result, the Edmonds-Karp algorithm is pseudo-polynomial. The reduction, therefore does not guarantee that the resulting combined algorithm is polynomial. However, the transformation creates a flow-graph with a particular structure there at most there are max(L,R) augmenting paths all of length 3. Furthermore, each augmenting path increases the flow by 1, so the resulting max-flow algorithm is O(E+V) and hence the combined algorithm is indeed polynomial.

Question 4 [4 points]
In class we've studied an approximation algorithm for the Vertex-Cover, that greedily choses an edge that connects two uncovered vertices until all the vertices are covered by at least one edge. As we've seen in class, this approach leads to a 2-approximation algorithm, i.e., one that is guaranteed not to be worse that a factor of 2 off of the optimal vertex cover.

In this problem, you are asked to:

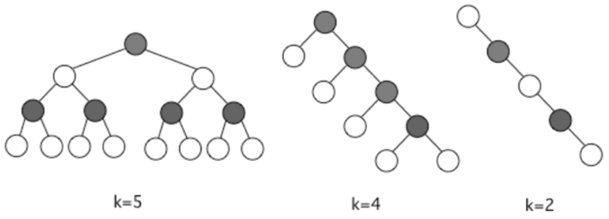
a) Show that there are tree graph instances (i.e., instances of graphs that are trees) for which the vertex-cover application algorithm described in class never finds the optimal vertex cover.
b) Develop an efficient (and ideally optimal) vertex cover algorithm of your own for the restricted case of tree graphs. You do not need to argue that is optimal, but exemplify its operation on a small complete binary tree and on a linear tree (where each node, except the "bottom" node) has a single descendant node.

Solution:

a) Consider a "star" graph with N nodes and N-1 edges where a central node is connected to all the remainder N-1 node with a single edge. For this graph, the algorithm described in class will pick an edge (any edges) and yields a vertex cover with two nodes. However, the optimal vertex code only has a single node, the central node, so this approximation algorithm always produces a solution that is a factor of 2 off of the optimal solution.

b) A simple and polynomial-time algorithm will perform a depth-first-traversal and cover all edges on its "way up" in the traversal by including in the cover a node if any of its edges (connecting to the children) is not already covered. Ignoring the leaves, this strategy covers all the edges in a single traversal. The pseudo-code below illustrates this algorithm and the figures below depict its resulting covers and the cover value.

```
void TREE-VERTEX-COVER(node n){
  if(color[n] == white){
    color[n] = gray;
     for all children c of n do {
       TREE-VERTEX-COVER(c);
     }
     if(exists uncovered edges to children){
        cover += {n};
        cover all edges to children of n;
     }
     color[n] = black;
  }
}
```



This is a slight modification of the depth-first traversal of a graph the only modification is that for each node examines its adjacent edges twice attaining still a complexity of O(V+E) i.e., linear in the size of the input graph specification.

Question 5 [3 points]
Consider the following linear problem:

$$\max\ 3x_1 - 2x_2$$

subject to:

$$\begin{array}{rrcr} x_1 & +x_2 & \leq & 2 \\ -2x_1 & -2x_2 & \leq & -10 \end{array}$$

$$x_1, x_2 \geq 0$$

For this Linear Programming problem answer the following:

a) Convert it to Slack form.
b) Determine graphically if this problem is feasible, infeasible, bounded or unbounded. Explain.

Solution:
a) We convert the problem into the slack form as shown below.

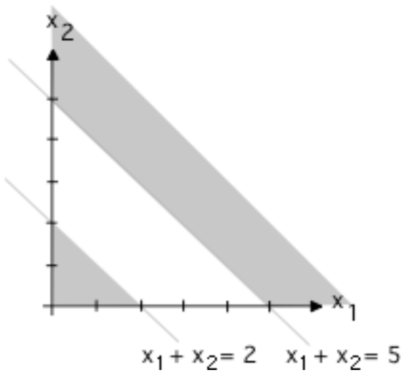
$$\max\ 3x_1 - 2x_2$$

subject to:

$$\begin{array}{rrrr} x_3 = & 2 & -x_1 & -x_2 \\ x_4 = & -10 & +2x_1 & +2x_2 \end{array}$$

$$x_1, x_2, x_3, x_4, \geq 0$$

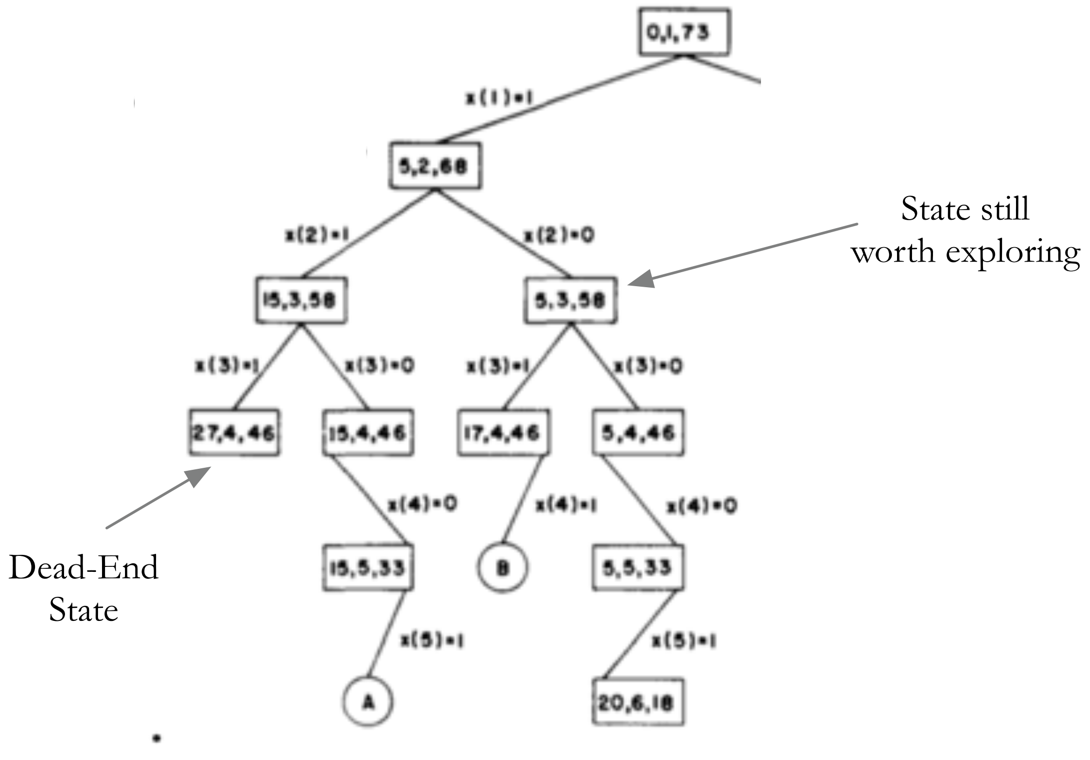
b) In the geometric approach, which is simple given that we are dealing with two variables only we can see that the domain of feasibility is empty. The first constraint implies that all feasible points are below the line $x_1 + x_2 = 2$ whereas the second constraint imposes that the feasible points are above the line $x_1 + x_2 = 5$. The intersection of the two regions is thus empty as the figure below illustrates.



Question 6 [3 points]
In class we've see an approach to bound the exhaustive exploration of state spaces in problems such as the sum of subsets of integers. This bounding made use of bounding functions, that used the sorted list of the numbers in the set to determine if from a given state being explored, is feasible or not.

In this context consider the problem instance with set S = {5, 10, 12, 13, 15, 18} and sum value M = 30. For this problem, can the potential solution state where the first three elements of the set are picked (corresponding to selection $x_1 = 1$, $x_2 = 1$ and $x_3 = 1$ and thus with a summation of 27), be pruned? Why or why not? What about for the state corresponding to selection $x_1 = 1$, $x_2 = 0$ (only two choices) and thus with a summation of 5?

Solution:
The idea of the bounding function described in class is to evaluate for a given state, that corresponds to a set of choices, if the set of subsequent choices can lead to a feasible state. In the case of the first state described here with a summation of 27 and given that the next item is 13 (and is the smallest of the remaining items that we can make a choice), the lowest next value is 27+13 = 40 which is already larger than the target value of 30. As such, there is no need to continue and the search can be terminated at this node. However, if the search is at the second state referred here, with a summation of 5, given that there are still values that can make the bound, the algorithm should pursue the exploration further. In fact, the state corresponding to x = (1,0,1,1,0,0) is a feasible or success state. The partial state diagram shown below illustrates this state search pruning.


0,1,73
x(1)=1
5,2,68
x(2)=1
x(2)=0
15,3,58
5,3,58
x(3)=1
x(3)=0
27,4,46
15,4,46
17,4,46
5,4,46
x(4)=0
x(4)=1
15,5,33
B
5,5,33
x(5)=1
A
20,6,18
State still worth exploring
Dead-End State