Homework 4

Adam Karl

September 15, 2020

1 Problem 13 (6 points)

1.1 a

Goal: My approach is to first mark every uncolored vertex in V as either red or blue in such a way that V will contain an optimal solution when we then iterate over every edge in E and add every edge that connects a red node to a blue node to V.

- go through each vertex in V, counting the number of uncolored nodes. Let this number be m
- create an array ARR with 2 rows and m columns
- arbitrarily choose a vertex to be the root node
- Using a depth first search, we are going to fill in the array such that:
  - ARR[1,i] has the minimum cost of solving the problem on the sub-tree with uncolored node i as the root, assuming that i is marked red
  - ARR[2,i] has the same solution, but assuming i is marked blue
  - note that neither solution gives any consideration to the parent of node i, only their children
- this array initially looks like this

	1	2	3	...	m
Assume vertex is red					
Assume vertex is blue					

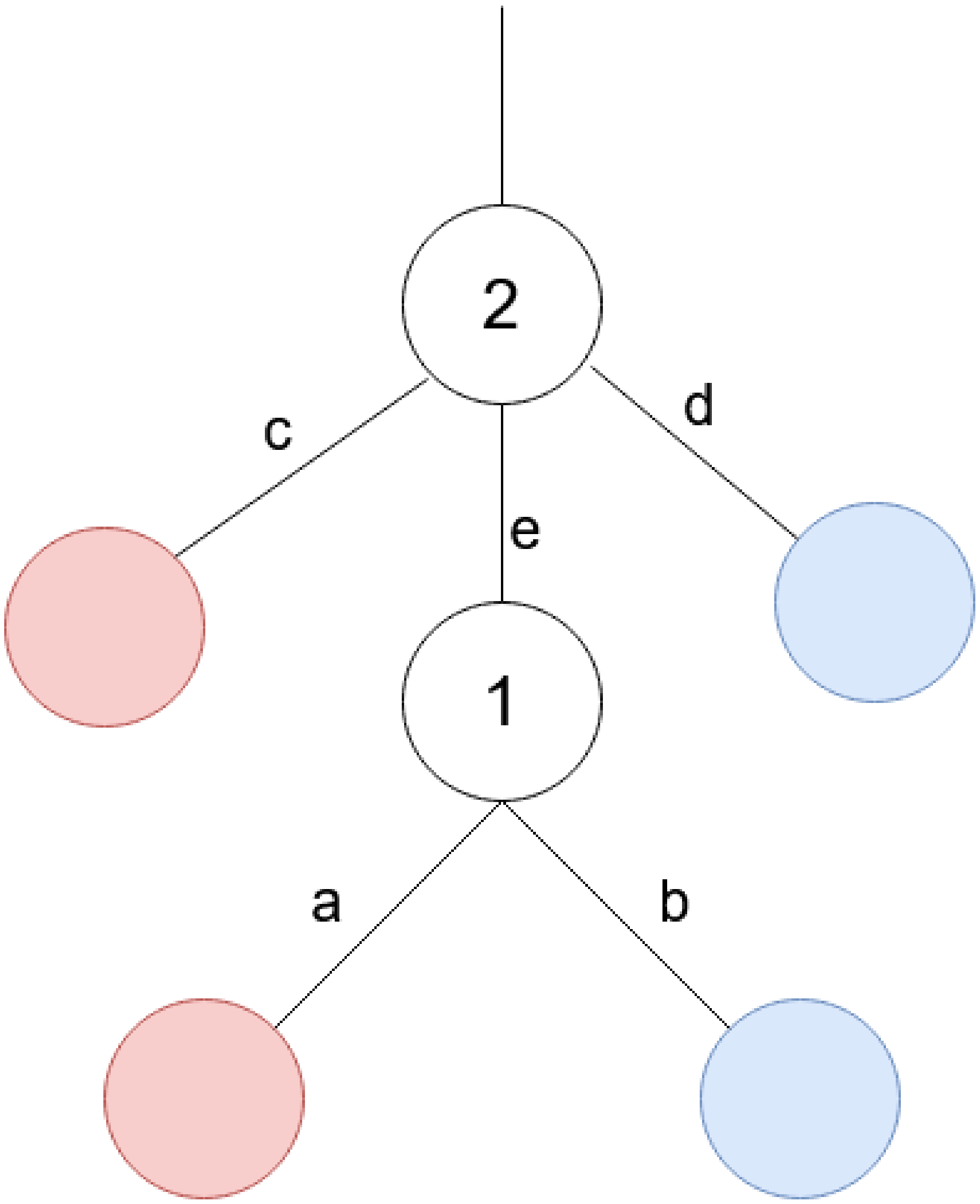
Starting with the root node, use a depth first search until an uncolored node is found. (Let the first uncolored node we find be "node 1" for the sake of simplicity, then "node 2", "node 3", all the way until the last uncolored node "node m")

- if the vertex's parent is uncolored:
  - for ARR[i, 1] we assume i is colored red. Start with ARR[i, 1] = 0
  - for each child of i (if any):
    - if child is red, do nothing
    - if child is blue, add the cost of the edge from i to the child to ARR[i, 1]
    - if child is uncolored, add the MIN(cost of assuming child is red, cost of assuming child is blue + cost of removing edge from i to child) to ARR[i, 1]
  - for ARR[i, 1] we assume i is colored blue. Start with ARR[i, 2] = 0
  - for each child of i:
    - if child is red, add the cost of the edge from i to the child to ARR[i, 2]
    - if child is blue, do nothing

    * if child is uncolored, add the MIN(cost of assuming child is red, cost of assuming child is blue + cost of removing edge from i to child) to ARR[i, 1]

  – NOTE: since we do a depth first search, we can be sure all of i's uncolored children already have values in the array

  – here is what an example of this might look like at the start: (note that while 2 only has a single red, blue, and uncolored child, an uncolored node may have any number of red, blue, and uncolored children)



	1	2	3	...	m
Assume vertex is red	cost to cut edges to blue children	cost to cut edges to blue children + min(cost if child is red, cost if child is blue + cost to cut edge to that child)			
Assume vertex is blue	cost to cut edges to red children	cost to cut edges to red children + min(cost if child is red + cost to cut edge to that child, cost if child is blue)			

- else: (the vertex's parent is colored)

	1	2	3	...	m
Assume vertex is red	b	d + min(ARR[1,1], ARR[2,1] + e)			
Assume vertex is blue	a	e + min(ARR[1,1] + e, ARR[2,1])			

- in this situation, we are able to definitively mark the current vertex i as red or blue, then follow the algorithm backwards to color any uncolored descendants of i
- first calculate ARR[1,i] and ARR[2,1] the same as before
- now treat i's parent node as though it was a child of i, adding the cost of the edge from i to i's parent to ARR[1,i] if i's parent is blue, or to ARR[2,i] if i's parent is red
- if ARR[1,i] $>$ ARR[2,i] mark node i red. If ARR[1,i] $<$ ARR[2,i] mark node i blue. If ARR[1,i] $=$ ARR[2,i] node i may be arbitrarily either red or blue.
- if i was marked red, do the same MIN calculations used in ARR[1,i]. For each MIN calculation, the minimum indicates the optimal coloring of the child node. For example, if ARR[1,j] was chosen in the minimum, this indicates node j should be colored red (if ARR[2,j] was chosen in the minimum, j should be colored blue). Repeat this process to color the children of j now that j's color is known.
- likewise, if i was marked blue do the same MIN calculations used in ARR[2,i] and mark and repeat the process on to color each of i's children

When the depth first search is complete:

- all nodes in the tree are either indicated red or blue
- start with V as the empty set
- for each edge in E, add the edge to V if and only if the edge connects a red node and a blue node
- after iterating through every edge, V now contains the subset of E that minimizes the aggregate cost while ensuring there is no path from any red node to any blue node
- The initial calculation of m was O(E), and a generic depth first search of the tree is O(2E) (since our implementation adds an additional backtracking possibility, it's technically O(3E)), and the final traversal of all edges was O(E). These steps happened in succession, so they all add together, maintaining a linear runtime.

1.2 Implementation Notes

In order to focus on the core of the algorithm, I chose to skip over some of the finer details that ensure the algorithm runs in linear time.

Before the depth first search, make a pass through all edges in E and add them to an adjacency list (O(E). This ensures that during a generic depth first search, the total runtime is 2E since each edge is added to the adjacency list twice (e.g. A-B and B-A).

In order to make things as clear as possible, I renamed each uncolored node 1 through m so that the array would only have information about uncolored nodes. If this approach is used, we may need to create a hash table that hashes uncolored node number to original node number in constant time. Then, each time an uncolored node was considered we would first hash back to its original node number before checking the adjacency list.

Alternatively, the array may contain columns for ALL nodes in V (not just the uncolored ones), but we would only bother to ever fill in the values ARR[1,i] and ARR[2,i] when i was an uncolored node. Then anytime we looked at a node, first check if it's red/blue or uncolored since the cost to any known color node can be immediately evaluated, so we only need to check the column of ARR when i is uncolored. At the end of the algorithm, ARR would be functionally identical to the ARR with only m columns, but with the columns shuffled around a bit and blank columns added.