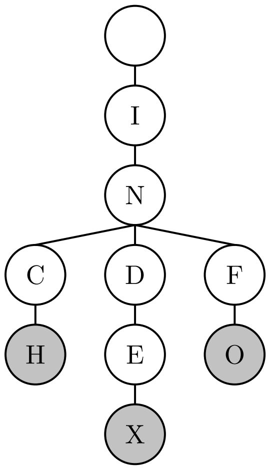
CS 61B
Spring 2021

Tries and More Graphs

Discussion 11: April 5th, 2021

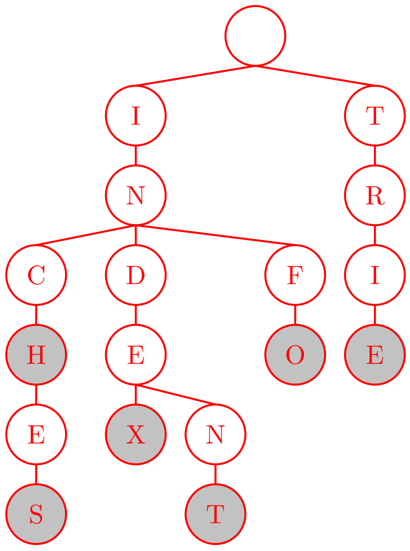
1 Trie Your Best

(a) What strings are stored in the trie below? Now insert the strings *indent, inches, and trie* into the trie.



The strings originally contained in the trie are *inch, index, and info.*

Below is the trie after inserting *indent, inches, and trie.*



(b) What is the runtime to find out if a given string is in the tree? What is the runtime to add a string to the tree? Describe your answers in terms of N, the number of words in the trie. You may assume the max length of any word in the trie is a constant.

To find out if a given string is in the tree, we simply need to walk the tree and step through each letter of the string we're looking for. Because the length of the string is a constant, this operation takes $\theta(1)$ time. To add a string to the tree, we again need to walk the tree and add new nodes as necessary, which will take as many operations as there are letters in the string. Because we view the length of the string as bounded to be constant, this operation also takes $\theta(1)$ time.

(c) *Extra:* How could you modify a trie so that you can efficiently determine the number of words with a specific prefix in the trie? Describe the runtime of your solution.

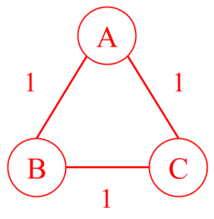
We can add a `numWordsBelow` variable to each of the nodes in our trie. When we insert we will increment this variable for all nodes on the path to insertion. In order to determine the number of words that start with a specific prefix, we can traverse the trie following the letters in the prefix. Once we reach the end of the prefix, we return `numWordsBelow` of the last character in the prefix, or 0 if the entrie prefix is not contained in the tree. If the length of the prefix is k then this code will run in $\Theta(k)$ in the worst case. If we have the case that the lengths of the strings will be assumed to be a constant, then this runtime of $\Theta(k)$ will actually be $\Theta(1)$ as we drop the constant coefficients.

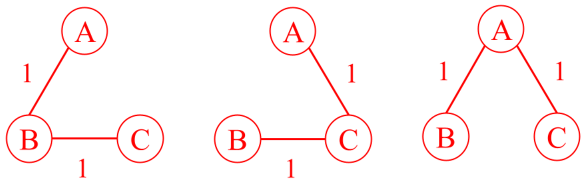
2 A Tree Takes on Graphs

Your friend at Stanford has made some statements about graphs, but you believe they are all false. Provide counterexamples to each of the statements below:

(a) ”Every graph has one unique MST.”

This false statement can be disproved with the below example. The graph given below has three valid MSTs, which are shown underneath it. In general, graph will only be guaranteed to have a unique MST if all of it’s edge weights are unique. If a graph has duplicate edge weights, there may or may not be a unique MST.

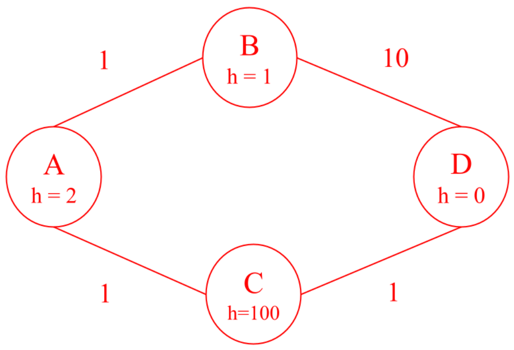




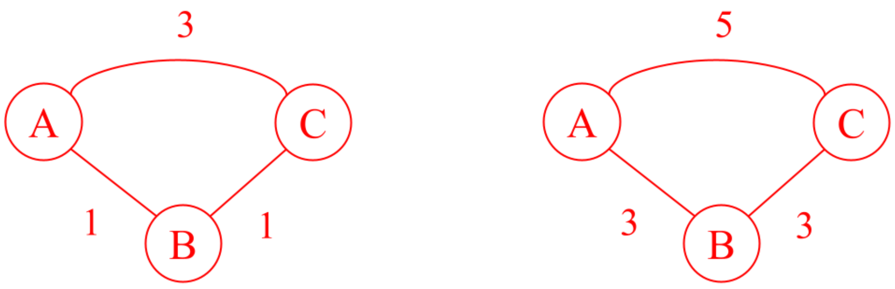
(b) ”No matter what heuristic you use, A* search will always find the correct shortest path.”

This false statement can be disproved with the below example. Here, A* would incorrectly return A - B - D as the shortest path from A to D. Starting at A, we would add B to the queue with priority 1+1 (the known distance to B, as well as our estimated distance from B to the goal), and we would add C to the queue with priority 1+100 (the known distance to C, as well as our estimated distance from C to the goal). We then pop B off the queue, and add D to the queue with priority 11+0 (the known distance to D, as well as the estimated distance from D to the goal). Our queue now contains C with priority 101, and D with priority 11, so we pop D off the queue and complete our search, returning A - B - D as the shortest path instead of the correct answer: A - C - D.

In general, A* is only guaranteed to be correct if the heuristic is good–specifically, it should be both admissible and consistent (note that applying admissibility and consistency are out of scope for this class, you only need to know the definition). In the example given, our heuristic is neither admissible nor consistent.

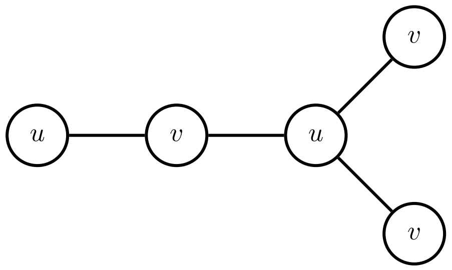


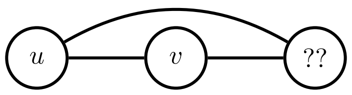
(c) "If you add a constant factor to each edge in a graph, Dijkstra's algorithm will return the same shortest paths tree."
This false statement can be disproved with the example below, where we add a constant $c = 2$ to every edge. Adding a constant factor per edge will disadvantage paths with more edges. In our example, though A - B - C had more edges, in our original graph it still has shorter total path cost, at $1 + 1 = 2$. On the other hand, A - C had fewer edges but larger total path cost, 3. After adding a constant factor, however, the A - B - C path cost was $(1 + 2) + (1 + 2) = 6$ and the A - C had a path cost of $(2+3) = 5$. So before the addition, Dijkstra's shortest path tree would have said the shortest path from A to C was A - B - C, but afterwards it would say A - C.



3 Graph Algorithm Design

(a) An undirected graph is said to be bipartite if all of its vertices can be divided into two disjoint sets U and V such that every edge connects an item in U to an item in V. For example below, the graph on the left is bipartite, whereas on the graph on the right is not. Provide an algorithm which determines whether or not a graph is bipartite. What is the runtime of your algorithm?
Hint: Can you modify an algorithm we already know?





To solve this problem, we run a special version of a traversal from any vertex. This can be implemented using either DFS and BFS as the underlying traversal that we will modify. Our special version marks the start vertex with a u, then each of its neighbors with a v, and each of their neighbors with a u, and so forth. If at any point in the traversal we want to mark a node with u but it is already marked with a v (or vice versa), then the graph is not bipartite.

If the graph is not connected, we repeat this process for each connected component.

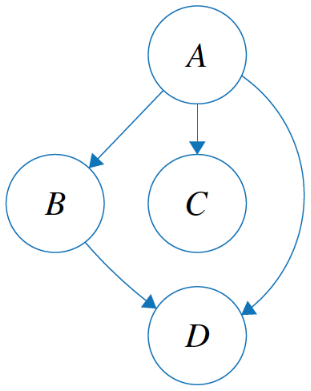
If the algorithm completes, successfully marking every vertex in the graph, then it is bipartite.

The runtime of the algorithm is the same whether you use BFS or DFS: $\Theta(E+V)$.

(b) Consider the following implementation of DFS, which contains a crucial error:

```
create the fringe, which is an empty Stack
push the start vertex onto the fringe and mark it
while the fringe is not empty:
    pop a vertex off the fringe and visit it
    for each neighbor of the vertex:
        if neighbor not marked:
            push neighbor onto the fringe
            mark neighbor
```

First, identify the bug in this implementation. Then, give an example of a graph where this algorithm may not traverse in DFS order.
Hint: When should we be marking vertices?



For the graph above, it's possible to visit in the order $A - B - C - D$ (which is not depth-first) because D won't be put into the fringe after visiting B, since it's already been marked after visiting A. One should only mark nodes when they have actually been visited, but in this buggy implementation, we mistakenly mark them before we visit them, as we're putting them into the fringe.

(c) *Extra:* Provide an algorithm that finds the shortest cycle (in terms of the number of edges used) in a directed graph in $O(EV)$ time and $O(E)$ space, assuming $E > V$.

The key realization here is that the shortest directed cycle involving a particular source vertex s is just the shortest path to a vertex v that has an edge to s, along with that edge. Using this knowledge, we create a shortestCycleFromSource(s) subroutine. This subroutine runs BFS on s to find the shortest path to every vertex in the graph. Afterwards, it iterates through all the vertices to find the shortest cycle involving s: if a vertex v has an edge back to s, the length of the cycle involving s and v is one plus distTo(v) (which was computed by BFS).

An alternative approach to the above subroutine (that is slightly more optimized) actually modifies BFS to short circuit if it's visiting a node v and sees it has an edge v -¿ s. Because BFS visits in order of distance from S, we can know that the first vertex v we see with an edge back to s will be the shortest cycle.

Regardless of which approach you take, asymptotically our subroutine takes $O(E+V)$ time because it uses BFS and a linear pass through the vertices. To find the shortest cycle in an entire graph, we simply call the subroutine on each vertex, resulting in an $V \cdot O(E+V) = O(EV + V^2)$ runtime. Since $E > V$, this is still $O(EV)$, since $O(EV + V^2) \in O(EV + EV) \in O(EV)$.