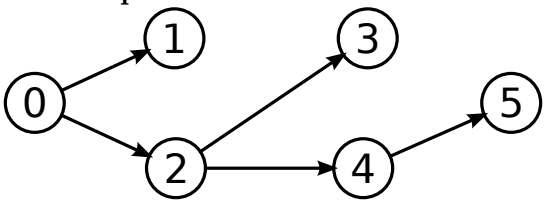
OA 4202, Homework 1 Solutions

Nedialko B. Dimitrov

1. In class, we ran BFS on the graph in Figure 1. When we had a choice of multiple "white" neighbors to push on to Q, in class, we always pushed the lowest numbered neighbor first. Re-do BFS on the same graph, but this time, always push the highest numbered neighbor first. Draw the resulting BFS search tree. Is the resulting search tree the same as the one we saw in class?

   An acceptable solution:

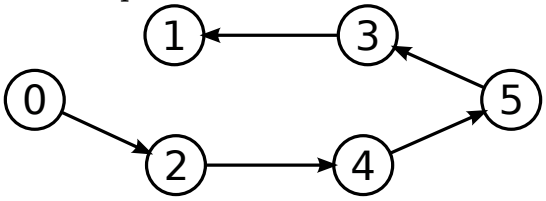
   

   The search trees are different. *Punchline:* BFS search trees are not unique.

---

2. In class, we ran DFS on the graph in Figure 1. When we had a choice of multiple "white" neighbors to push on to Q, in class, we always pushed the lowest numbered neighbor first. Re-do DFS on the same graph, but this time, always push the highest numbered neighbor first. Draw the resulting DFS search tree. Is the resulting search tree the same as the one we saw in class?

   An acceptable solution:

   The search trees are different. *Punchline:* DFS search trees are also not unique.

---

3. What is the running time of BFS if the input graph is stored using a dense adjacency matrix?

   An acceptable solution:
   $O(n^2)$. For each node u at the head of Q, we have to look at all possible $n-1$ neighbors to determine which are the successors, taking $O(n)$ time. We have to do this n times, once for each node.

---

4. You are dropped in the middle of a maze full of hallways and intersections. Everything looks the same to you. Every intersection looks exactly like the previous, every hallway looks exactly like the next. Luckily, you know there is an exit and you happen to have an infinite supply of pennies. How do you get out?

An acceptable solution:
We can run DFS. First, we define a graph where every intersection is a node, and two intersections share an edge if there is a hallway connecting them. We use our pennies to mark intersections as "white," "gray," or "black." An intersection with no penny is "white." An intersection with a penny turned to tails is "black." And, an intersection with the penny turned to "heads" is gray.

At every point in DFS, there is a trail of gray nodes leading to the starting node. At every point in our search, there is a trail of intersections marked with heads that can lead us back to the start. We can use this trail to back-track and continue running DFS until we reach the exit.

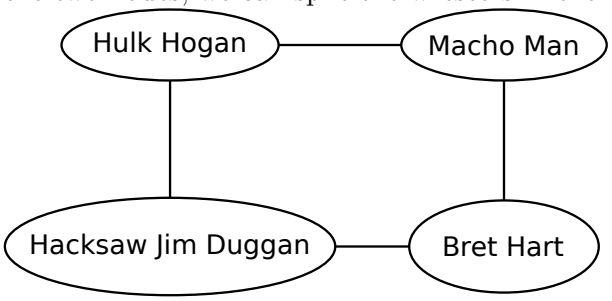
Running BFS would not work. If the maze is big, after a while, BFS would mark so many intersections "black," that we would simply be lost walking around similar looking intersections that have pennies turned to tails in them (black nodes).

The key here is that moving to the next node in Q in DFS is easy, it neighbors the current one. But in BFS, the next node in Q could be all the way across the graph.

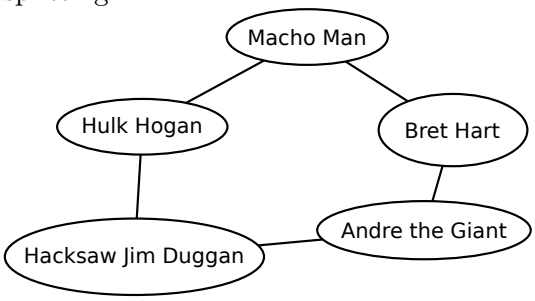
---

5. Between any pair of professional wrestlers, there may or may not be a rivalry. For marketing purposes, it is helpful to divide professional wrestlers in two separate groups: "good guys" and "bad guys." Suppose we have n professional wrestlers and a list of m rivalries between pairs of wrestlers. Describe an algorithm that runs in $O(n+m)$ time and determines whether it is possible to split the professional wrestlers into "good guys" and "bad guys," so that each rivalry is between a good guy and a bad guy. If such a labeling exists, your algorithm should produce it.

   For example, if each wrestler is a node and a rivalry between two wrestlers is an edge between the two nodes, we can split the wresters in the following graph



   by making Hulk Hogan and Bret Hart "good guys," and Macho Man and Hacksaw Jim Duggan "bad guys." On the other hand, in the following graph, it is impossible to do the splitting



   (adapted from CLRS)

   An acceptable solution:

We can use BFS to do the separation, or tell us if such a separation does not exist. First, we define a graph between the wrestlers. Each wrestler is a node, and two wrestlers share an edge if there is a rivalry between them.

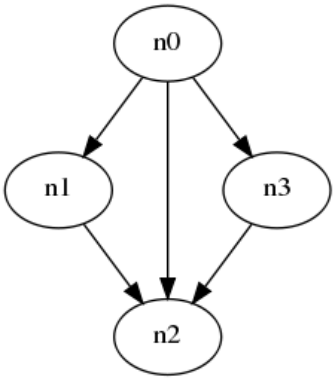
- Pick an arbitrary wrestler s, label him as a "good guy."
- Start the BFS on the s node.
- Whenever we add a wrestler to the BFS search tree, we label him the opposite of his parent.
- Without loss of generality, consider a wrestler w at the head of Q and suppose that his label is "good guy." To continue the search, we look through w's edges. At this time, we'll also check for neighboring "gray/black" nodes (labeled wrestlers) with the same label of "good guy". Finding a neighbor also labeled "good guy" produces a conflict, since we have a "good guy"-"good-guy" rivalry.

If we never find a conflict in our labeling, we have produced a proper labeling of the wrestlers. If we do find a conflict in our labeling, say between wrestlers w and u, then the odd-length cycle given by the search tree and the conflict edge (w, u) is a proof that no proper labeling of the wrestlers can exist.

Punchline: This problem is about identifying a *bipartite* graphs. For a bipartite graph, we can split the graph in such a way so that all the edges have one node on each side of the split. Bipartite graphs arise often in matching problems, and network algorithms are often easier and faster on bipartite graphs.

---

6. Give an $O(n + m)$ time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two nodes s and t, and returns the number of paths from s to t in G. For example, this graph



has three paths from 0 to 2 ([0,2], [0,1,2], and [0,3,2]). (Hint: Check your algorithm by making sure it works on a few small examples.) (adapted from CLRS).

An acceptable solution:
The following algorithm works:

- Run topological sort on the graph
- Process the nodes right to left in the sorted order:
  - if a node is right of t, label it with "0"
  - if a node is equal to t, label it with "1"

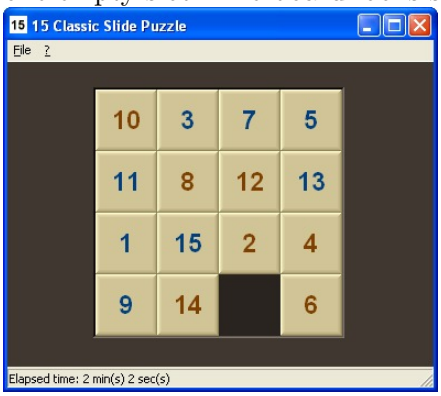
- if a node is left of t, label it with the sum of the labels on the nodes it points to.

- The label on s is the number of paths from s to t.

Running topological sort takes $O(n+m)$ time. Placing the labels on the nodes takes $O(n+m)$ time, since we look at each node and each edge at most once. So, the algorithm takes $O(n+m)$ time total.

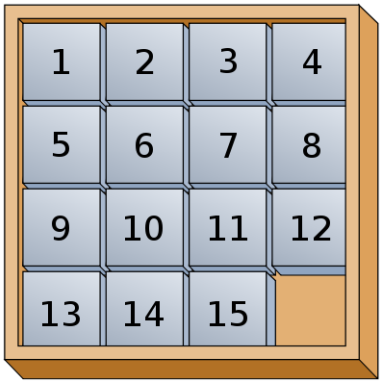
---

7. Recall the classic "15 puzzle" game. In this game, we are given a board with 15 pieces and one empty slot. The board looks something like this:



We are only allowed to slide tiles into the empty slot, leaving a new empty slot where the tile used to be. For example, in the above picture, we can slide the "2" down into the empty slot, leaving a new empty slot on row 2, column 2. Similarly, we can slide the "6" left into the slot or the "14" right into the slot. Those three moves are the only moves allowed from that position of the board.

Our goal is to get the board to look like this:



Suppose we are given an arbitrary board position and are asked to solve the puzzle. First, formulate a graph reachability problem that tells us whether the puzzle is solvable. Second, if the puzzle is solvable, how could we use an algorithm we learned in class to solve the puzzle with a minimum number of moves? (adapted from AMO)

An acceptable solution:
We can create a graph from the possible board positions. Each arrangement of the 15 tiles and the empty slot are a node. There are about 16! board positions... over a billion. Two nodes, s and t, have an edge between them if there is a valid move from the board position

of s to the board position of t. Each node has at most 4 neighbors, since there are at most 4 valid moves from any position.

If we are given a board position corresponding to some node s, we can run a search algorithm on the described graph to solve the puzzle. We start at s and end if we find the node corresponding to the winning position. Lets call the winning node w. The path from s to w gives us a sequence of moves for solving the puzzle. If the algorithm ends before finding w, the puzzle is unsolvable.

To find the solution with a minimum number of moves, we can choose BFS as our search algorithm. BFS's shortest path property ensures that we find the shortest path between s and w.

---

8. Suppose you only have acces to BFS, and cannot change the way the algorithm works, only its input. You are given a graph with positive integer edge lengths. How can you find the shortest path between two nodes, s and t, using BFS.

   An acceptable solution:

   If edge (i, j) has length k, we can add $k - 1$ "fake nodes" between node i and node j. Call the original graph G and the graph with all the fake nodes G'.

   We can run BFS starting at node s in G'. That will produce a shortest path between s and t in terms of the total number of edges in G'.

   The number of edges on a path in G' corresponds to the length of a path in G. Removing all the fake nodes from the path we get at the end of the BFS in G', gives us a minimum length path in G.

   Punchline: This is a perfectly fine, and very slow, way of solving shortest paths. For a graph with only two nodes and a single edge between them, if the edge has length 1000, the algorithm would take 1000 iterations to find the shortest path. Dijkstra's algorithm is significantly faster.

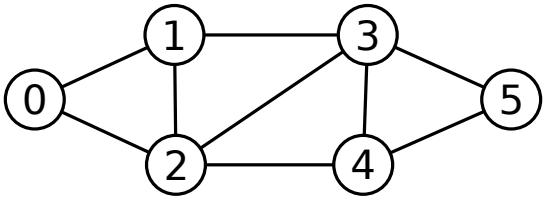


Figure 1: The example graph from class, on which we ran BFS and DFS.