Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — March 29, 2008
Professors Srini Devadas and Erik Demaine — Handout 8

Problem Set 4

This problem set is due **Thursday April 10** at **11:59PM**.
Solutions should be turned in through the course website in PDF form using LaTeX or scanned handwritten solutions.
A template for writing up solutions in LaTeX is available on the course website.
Remember, your goal is to communicate. Full credit will be given only to the correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Exercises are for extra practice and should not be turned in.
Exercises:

- CLRS 22.2-3 (page 539)
- CLRS 22.2-8 (page 539)
- CLRS 22.3-9 (page 548)
- CLRS 22.3-10 (page 549)

---

1. **(7 points)** Connected Components

   An undirected graph can be separated into connected components. A *connected component* is a set of vertices for which (1) every two vertices in the set are connected by a path, and (2) no edge connects a vertex inside the set to a vertex outside the set.

   Give an $O(V + E)$-time algorithm for dividing an undirected graph into connected components, that is, for finding all of the connected components in the graph.

   Solution:

   Simply do a DFS on the graph, and each time you have to start a new DFS-visit, you are working on a new connected component. For each DFS-visit, store a set of all nodes seen.

   BFS works just as well, though the way it was presented in class it would have to be modified to start again at an unseen node when it runs out of nodes like DFS did.

2. **(28 points)** Cycle Detection

   A cycle is a path of edges from a node to itself.

(a) **(7 points)** You are given a directed graph $G = (V, E)$, and a special vertex v. Give an algorithm based on BFS that determines in $O(V + E)$ time whether v is part of a cycle.

Solution:

Do a BFS from v. If you ever reach an edge (u, v) pointing back to v again during this BFS, then there is a cycle containing v that starts with the path used to get from v to u, and ends with the edge (u, v).

Checking for if a node is v doesn't add any complexity to BFS, so the algorithm still runs in $O(V + E)$ time.

(b) **(14 points)** You are given a graph $G = (V, E)$, and you are told that every vertex is reachable from vertex s. You want to determine whether the graph has any cycles. Ben Bitdiddle proposes the following algorithm. Perform a BFS from s. If, during the search, you ever reach a node that you have already seen before, then declare that G has a cycle. If you never reach the same node twice, declare that there is no cycle.

i. Show that Ben's algorithm works for undirected graphs.

Solution:
If you see a node u twice during a BFS of an undirected graph, then there are two paths from s to u. Simply paste those paths together to make the cycle. It's important that the graph is undirected, because pasting the two paths together to make a cycle requires reversing one of the paths.

ii. Show that Ben's algorithm does not work for directed graphs.

Solution:
Take the following graph with 3 nodes and 3 edges:
(s, t), (s, v), (v, t). A BFS will discover t twice, because there are two paths to t, but there is no cycle, because the paths only go in one direction (towards t).

(c) **(7 points)** You are given a directed graph $G = (V, E)$. Give an algorithm based on DFS that determines in $O(V + E)$ time whether there is a cycle in G.

Solution:

Run a DFS on G. If you discover a backedge, declare that there is a cycle. Otherwise, there is no cycle.

If there is a backedge, then it is part of a cycle: Simply take the path currently on the stack for the DFS, and then follow the backedge, and we have a cycle.

If there is a cycle, then there is a backedge. At some point during the DFS, we will visit for the first time a node in the cycle, v. Before we finish with v, we will visit every other node in the cycle, because those nodes are all reachable from v, and none of them have been visited before. In particular, we will visit the node u which precedes v in the cycle, and u will have a backedge to v.

We can check for backedges without affecting the asymptotic running time of DFS, simply by keeping a set of vertices currently on the stack.

3. **(25 points)** $2 \times 2 \times 2$ Rubik's Cube

   We say that a configuration of the cube is k levels from the solved position if you can reach the position in exactly k twists (quarter-turns, either clockwise or counterclockwise), but you cannot reach the position in fewer twists.

   Download `ps4_rubik.zip` from the class website.

   (a) **(10 points)** Use breadth-first search to recreate the chart seen on `http://en.wikipedia.org/wiki/Pocket_Cube`. Write a function `positions_at_level` in `level.py` that takes an argument `level`, a nonnegative integer. Your function should return the number of configurations of the cube `level` levels from the solved position (`rubik.I`), using the `rubik.quarter_twists` move set.

   Test your code using `test_level.py`, and submit it to the class website. Testcases above level 10 are commented out, because they may require at least 1GB of RAM. Level 10 should take no more than a couple minutes, even with 512MB of RAM.

   (b) **(15 points)** Now you will actually solve a given configuration of the cube, by finding the shortest path between two configurations of the cube (the start and the goal).

   Your code from part (a) could easily be modified to find shortest paths, but a BFS that goes as deep as 11 levels takes a few minutes (not to mention the memory needed). A few minutes might be fine for creating a Wikipedia page, but we want to solve the cube fast!

   Instead, we will take advantage of a property of the graph that we can see in the chart. In particular, the number of nodes at level 7 (half the diameter) is much smaller than half the total number of nodes.

   With this in mind, we can instead do a two-way BFS, starting from each end at the same time, and meeting in the middle. At each step, expand one level from the start position, and one level from the end position, always checking to see whether any new nodes have been discovered in both searches. When you find such a node, you just have to read off parent pointers to return the correct path.

   Write a function `shortest_path` in `solver.py` that takes two positions, and returns a list of moves that is a shortest path between the two positions.

   Test your code using `test_solver.py`. Check that your code runs at close to the same speed as level 7 from part (a) in the worst case.

   (c) **(Optional)** Test your code using `test_human_solver.py`, which will ask you to input the current configuration of a real Rubik's cube, and then tell you the shortest path in human-readable symbols (read `rubik.py` to understand these symbols).