50:198:213 Data Structures
Rutgers University–Camden
Professor Desmond Lun

March 10, 2021
Handout 8

Problem Set 4

This problem set is due **Monday, March 29** at **11:55PM**.
Solutions should be turned in through the course Sakai site in PDF form or scanned handwritten solutions.
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.

---

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.

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.

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

   (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.

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 Sakai site.

(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 Sakai site. Test-cases 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).