*Introduction to Algorithms: 6.006*
Massachusetts Institute of Technology March 10, 2011
Professors Erik Demaine, Piotr Indyk, and Manolis Kellis Problem Set 4

# Problem Set 4

This problem set is divided into two parts: Part A problems are theory questions, and Part B problems are programming tasks.

**Both Part A and Part B questions** are due **Friday, March 18** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF format using LaTeX . Your solution to Part B should be a valid Python file which runs from the command line. 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 a 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. See the course website for our full grading policy.

## Part A:

**Problem 4-1.** [32 points] **Cycles**

This problem deals with finding cycles in a graph.

**(a)** [8 points] Given a directed graph $G = (V, E)$, suppose there is a vertex $v$ such that there exists a path from $v$ to all other vertices in the graph. Consider the following method to find cycles in the graph. We proceed with BFS starting with vertex $v$. If we ever hit a vertex twice, we claim that we have found a cycle. If we hit every vertex only once, we claim that there doesn't exist a cycle in the graph. Is this algorithm correct? If not, point out an error with our algorithm.

**(b)** [8 points] Now consider running the above algorithm on a connected undirected graph $G = (V, E)$. Would it work correctly?

**(c)** [8 points] Given a directed graph $G = (V, E)$, give an algorithm based on DFS that determines the existence of a cycle in $O(|V| + |E|)$ time.

**(d)** [8 points] Now, given an undirected graph $G = (V, E)$, propose an algorithm whose running time is $O(|V|)$ to determine whether the graph contains a cycle. Notice that the running time should not depend on $|E|$.

**Problem 4-2.** [18 points] **Bipartite**

You are setting up a friendly match of the hit new multiplayer video game, *Call 4 Duty: Counter Company 2*. You have $n$ people who want to play, and you've looked up which $m$ pairs of them are friends. To make the game competitive, you would like to divide the players into two teams, *red* and *blue*, such that every pair of friends end up on opposite teams. (The teams do not need to

be the same size.) Give an $O(n + m)$-time algorithm that determines whether such a division into teams is possible, and if so, finds one suitable labeling of players as red and blue.

## Part B:

**Problem 4-3.** [50 points] **Rubik's Cube**

In this problem, you will develop algorithms for solving the $2 \times 2 \times 2$ Rubik's Cube, known as the *Pocket Cube*. Call a configuration of the cube "$k$ levels from the solved position" if it can reach the solved configuration in exactly $k$ twists, but cannot reach the solved configuration in any fewer twists.

Download `ps4_rubik.zip` from the class website. We also provide a GUI representation of the Rubik's cube in `ps4_rubik_GUI.zip`, courtesy of two previous 6.006 students.

**(a)** **(20 points)** For this problem, we will use breadth-first search to recreate the column labeled $f$ in the chart seen at `http://en.wikipedia.org/wiki/Pocket_Cube`.

Write a function `positions_at_level` in `level.py` that takes a nonnegative integer argument `level`, and returns the number of configurations that are `level` levels from the solved configuration (`rubik.I`), using both quarter twists and half twists (twisting the cube by $90°$ or $180°$).

The code in `rubik.py` only defines the `rubik.quarter_twists` move set, so you should start by defining a new move set that includes half twists as well. Do not modify `rubik.quarter_twists` because you will need it for the next part of this problem.

Test your code using `test_level.py`, and submit it to the class website. Testcases above level 8 are commented out, since they may require more memory than many computers have.

**(b)** **(30 points)** Now we will solve the Rubik's Cube puzzle by finding the shortest path between two configurations (the start and goal). For this part of the problem, we will limit the move set to only allow quarter twists (half twists are not allowed).

Your code from part (a) could easily be modified to find shortest paths, but a BFS that goes as deep as 14 levels will take 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, and then check to see whether any of

the new nodes have been discovered in both searches. If there is such a node, we can just read off parent pointers (in the correct order) to return the shortest 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, after modifying it to use just the quarter twist move set.

Submit your code to the class website. No written part is required for any part of this problem, but you should make sure your code is adequately documented so that we can understand it.