Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Professors Sivan Toledo and Alan Edelman

March 17, 2009
Handout 6

Problem Set 4

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

Part A questions are due **Tuesday, April 7th** at **11:59PM**.

Part B questions are due **Thursday, April 9th** 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)

Part A: Due Tuesday, April 7th

1. **(50 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 configuration in exactly k twists, but cannot reach the it 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 6.006 students last year.

   (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 degrees).

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 quater 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.

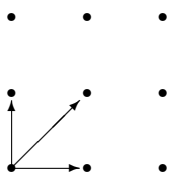
(c) **(Optional)** Go out and impress your friends with new 2x2x2 Rubik's Cube solver you just created! You can test your code using `rubik_solver_GUI.py`, which will ask you to input the starting configuration and show you the shortest path solution. You will need to copy your `solver.py` file into the directory where the GUI is located.

If you have any feedback, bug reports, or suggestions for improvement on the GUI, please send it to the TAs.

Part B: Due Thursday, April 9th

1. **(16 points)** Eliminating Cycles by Removing One Edge

   For each of the following statements, prove the statement or give a *small* counter example to show that it is false. You may use LaTeX to draw counter-example graphs if necessary (the solution template contains a drawing of the following graph to get you started).



   (a) **(8 points)** If DFS on a graph G produces exactly one back edge, then it is possible to remove an edge from G to make the graph acyclic. (Please recall that self-loops are back edges.)

   (b) **(8 points)** If G is cyclic but can be made acyclic by removing one edge, then DFS will encounter exactly one back edge.

2. **(8 points)** Bipartite graphs

   An undirected graph is called *bipartite* if its nodes can each be assigned a color, either red or blue, such that no red node is adjacent to another red node, and no blue node is adjacent to another blue node. Give an efficient algorithm to determine if a graph is bipartite. What is its running time?

3. **(26 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) **(12 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.