CS 5150/6150: Assignment 3
Due: Sep 23, 2011

This assignment has 5 questions, for a total of 100 points and 0 bonus points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question 1: So you think you can dance? . . . . . **[20]**

Solve question 3 from `http://compgeom.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/05-dynprog.pdf`

Question 2: MIS on planar graphs . . . . . **[20]**

We've seen how dynamic programming can be used to solve NP-hard problems on trees. It can also be used to solve NP-hard problems on graphs that aren't trees, as long as they're sufficiently ``treelike''. One of the projects explores this in more detail, but for now, let's consider planar graphs (i.e graphs that can be drawn in the plane without crossings), which are not at all tree-like (think of a grid)

There's a well known fact about planar graphs, that in one form, looks like this:

Theorem (Planar Separators)**.** *Given any planar graph $G = (V, E)$ where $n = |V|$ there exists a subset $S \subset V$ of size $\sqrt{n}$ such that if we delete S and all its edges from G, we get two* disconnected *components G_1, G_2 that have at most $n/2$ vertices each.*

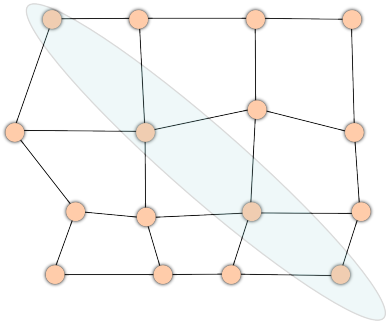

Figure 1: A planar graph with the separator marked

While there's a ``trivial'' algorithm that runs in time $O(n2^n)$ to find the maximum independent set (MIS) of a graph, use the above fact to obtain an algorithm that run in time $O(p(n)2^{\sqrt{n}})$ to find an MIS in a planar graph. Here $p(n)$ is some polynomial in n (i.e I don't care what you end up with for that term).

HINT: Re-express the MIS algorithm in trees in a form where for each root v, we store both the size of the independent set of the subtree rooted at v that contains v and the size of the independent set of subtree rooted at v that *does not* contain v. Express the recursion using only a node and its immediate children. Now use that as a template.

Note: this question is a little tricky.

Question 3: Card games . . . . . **[20]**

Solve question 4 from `http://compgeom.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/05-dynprog.pdf`

Question 4: Coloring intervals . . . . . **[20]**

Solve question 5 from `http://compgeom.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/07-greedy.pdf`

Question 5: Exploring dynamic programs . . . . . **[20]**

We've seen many examples of how dynamic programming, by letting us ``think top down, and solve bottom up'', helps us solve problems for which naive recursion would not do particularly well. But dynamic programming itself can be a bit of a memory and performance hog. In this question, we're going to explore various implementation strategies that improve the performance of a typical dynamic program both asymptotically and empirically.

The vehicle for this study will be the edit distance problem (see the lecture notes on dynamic programming):

Problem. *Given two strings s, s' over an alphabet Σ, determine the minimum number of insertions, deletions and substitutions that will transform s into s'.*

Your code will take as input (from standard input) two strings (one on each line) and will output a score for the edit distance between them, as well as the resulting match. Consider the example in the notes of matching ``ALGORITHM'' to ``ALTRUISTIC'', and suppose your output match is the one at the top of page 8 in the notes. Then your output will be

```
6
ALGOR.I.THM
AL.TRUISTIC
```

Note that the ``.'' denotes an insertion or deletion, and substitutions are implied by having unequal letters one above the other.

(a) We'll start with different recursive approaches to solving the edit distance problem.

i. [**2**] Implement a *recursive* algorithm for computing the edit distance **without** using memoization or storage of subproblems.

ii. [**3**] Implement a *memoized* algorithm for computing the edit distance. In such a method, you run the algorithm recursively as before, but every time you encounter a subproblem that you haven't solved before, you store it in an array so that you can look it up later.

iii. [**5**] Now implement a true dynamic programing solution, where you unroll the recursion ``in your head'', yielding a fully-iterative non-recursive algorithm for computing the edit distance.

Plot on a single chart the running time versus total length of inputs for the three methods. Also do a separate plot for the latter two algorithms. If you're given a budget of 60 seconds, what's the longest input size you can handle in the three algorithms ?

(b) We'll now look at ways of speeding up the dynamic programming procedure.

i. [**5**] As we noted in class it's often possible to implement a dynamic program with linear storage if you only wish to extract the solution value. However, extracting the solution itself is a lot trickier. Please read the ``Hirschberg trick'' described in the first section of `http://compgeom.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/06-sparsedynprog.pdf` and implement the outlined solution. Clearly, this will improve the memory usage of your method, but how does it affect the time ? Do a plot as before comparing this to the straight DP you just implemented, and again report the 60 second benchmark.

ii. [**5**] A DP can return multiple solutions of the same cost. But suppose I don't like insertions and deletions that much. I could address this issue by giving them more cost, but the DP will still explore the entire space of solutions in the table before returning one that I like. What I'd like is to see solutions progressively, where I'm willing to wait longer for a solution that is cheaper but might involve more insertions/deletions. Can you devise a solution ? (HINT: if you limit yourself to only a fixed budget of insertions and deletions, which parts of the DP will you not fill). Describe your solution in writing and implement it. Compare as before to the previous approaches.