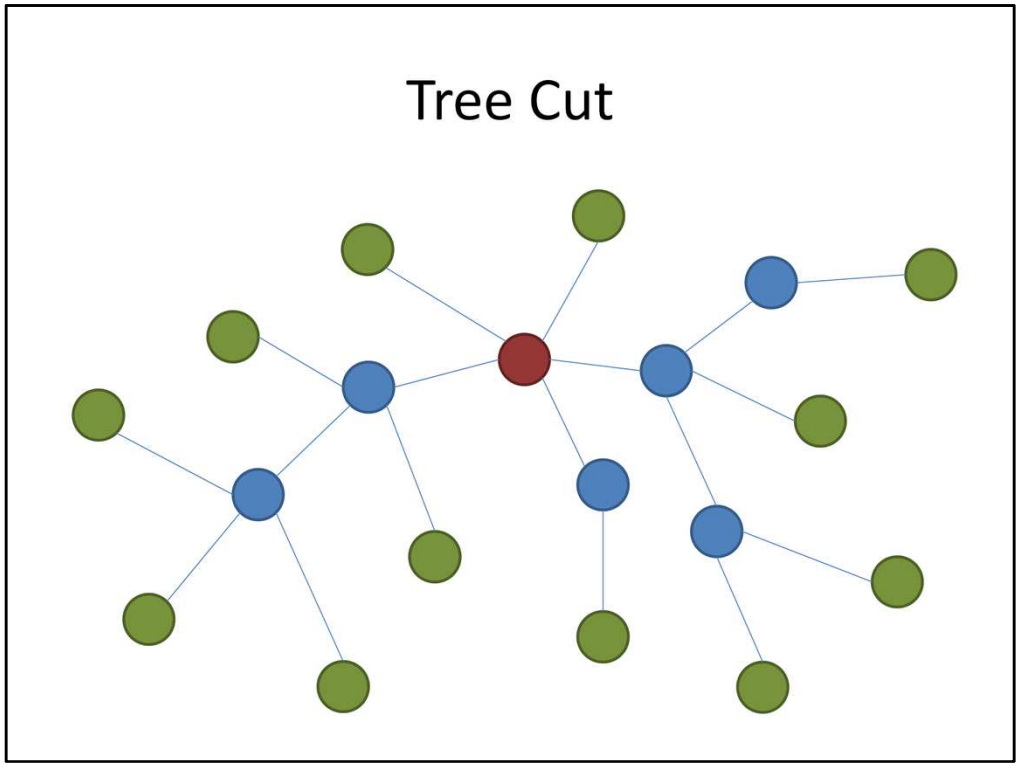
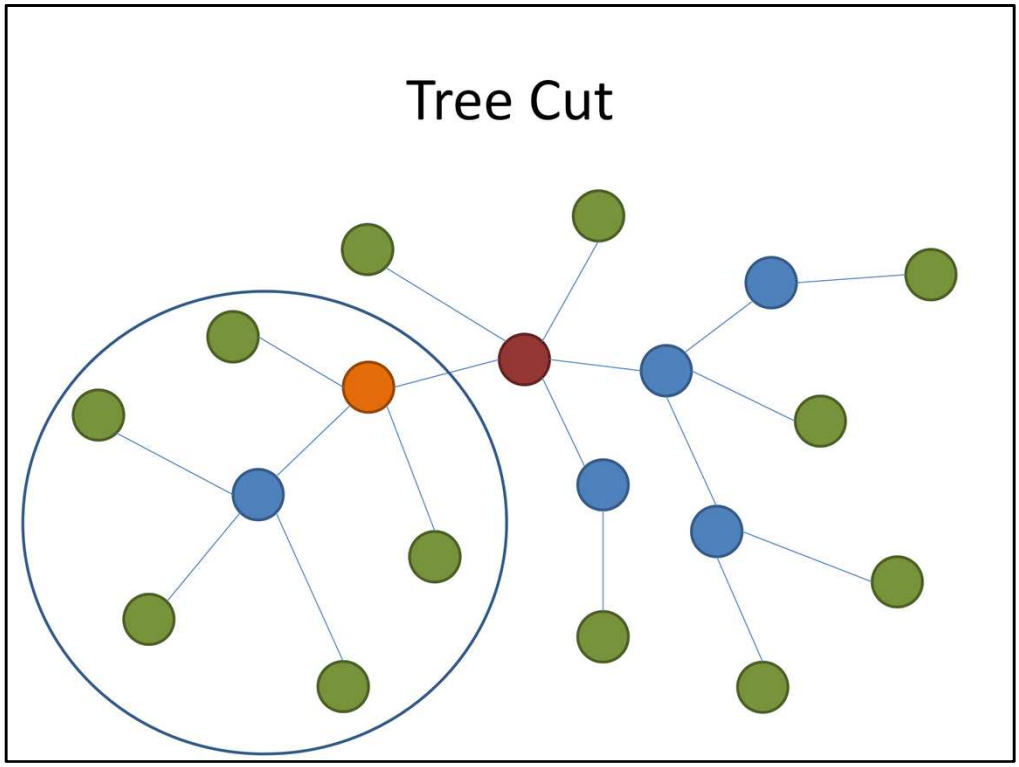
CS161 Programming Section

August 8, 2013

Hi everyone. Today we'll be discussing how to implement dynamic programming solutions. In particular, we're going to focus on when the ordering of the subproblems is somewhat unusual. Many of the examples you've seen in class involve just filling a table from left to right, or row-by-row, which is something I'm confident you'd be able to figure out how to code on your own. Dynamic programming problems share a lot in common with greedy problems in that figuring out the algorithm and proving it's correct is often much harder than actually coding it up. The problems we'll look at today, though, will focus a bit more on implementation issues than on the theory side.



The first problem we'll look at today is finding a tree cut. Here we have a tree with undirected edges where we've had a root node marked for us in red. The edges in this tree are all weighted, and you want to find the minimum-weight cut that separates the root from all the leaves.



So what's the optimal substructure that we can use for this problem? Well, let's look at one subtree of the root, which we've circled, and whose root we've marked in orange. Now, in order to make sure that we've cut all the leaves from this subtree off from the root, we can either cut the orange-red edge, or we can find the minimum-weight cut that separates the orange node from all the leaves in the circled subtree. This gives us one subproblem per node, where each subproblem is the minimum-weight cut for the subtree rooted at that node.

Tree Cut

- Ordering: bottom-up from leaves to root
- Use either DFS or BFS from root to get ordering

Now, what's the ordering in which we need to solve the subproblems? Well, the subproblems of size one are the leaves, so those are the ones we want to solve first. The root is the one with the most dependencies, so that comes last. In general, as long as all the children of a node are solved before that node, we're fine. One way we can ensure this is just to implement our solution recursively. In fact, we don't even need memoization, since the fact that we're using a tree means that each subproblem is used exactly once. However, if we really want to try a bottom-up approach, we can use a graph search to label the nodes in such a way as to guarantee that the children come before the parent. Hearing this should make you immediately think of topological sort, and indeed that's the right way to order the subproblems whenever you define a DP on a DAG. Our case is more special than that, though, so we can get away with other traversals, including say a breadth first search.

Tree Cut

- Input
  - A weighted undirected tree with n nodes rooted at r
  - $1 \le n \le 1000$
  - all weights between 0 and 1000
  - nodes are 1-indexed, edges given as pairs (u, v) with weight w
- Input Format Per Case
  - Line 1: 2 integers n r
  - Lines 2 to n: 3 integers u_i v_i w_i
- Sentinel: n = r = 0
- Sample

```
15 15
1 2 1
2 3 2
2 5 3
5 6 7
4 6 5
6 7 4
```

- Sample (cont)

```
5 15 6
15 10 11
10 13 5
13 14 4
12 13 3
9 10 8
8 9 2
9 11 3
0 0
```

- Output
  - The weight of the minimum root-leaf cut (0 if no leaves)
- Output Format Per Case
  - Line 1: 1 integer
- Sample

```
16
```

Here's the specification slide for this problem. There are a couple things I'd like to point out. First is that we've restricted our graphs to be fairly small; this is intentional so that you can choose whichever graph representation you find most convenient to work with. While you should convince yourself that this problem can be done in linear time, you might find it more convenient to pay the extra factor of n to use an adjacency matrix instead of an adjacency list. Next, if you decide to use topological sort to order the nodes, keep in mind that topsort, strictly speaking, does not work on undirected graphs. However, each edge has an implicit direction in that it points away from the root, so you can make topological sort work by only considering the direction that points from visited nodes to unvisited nodes.

Matrix Permanent

$$perm(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)}$$

$$det(A) = \sum_{\sigma \in S_n} sgn(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)}$$

For the second problem for today, we're going to go over how to compute the permanent of a matrix. The formal definition of the permanent of a matrix A is given here. This definition may look familiar to some of you, because it's very similar to the definition of a determinant, which is much more widely known. Basically, the determinant of a matrix is the signed sum of the product of permutations of entries in the matrix, while the permanent is the unsigned sum. It's interesting to note that we know how to compute a determinant in polynomial time using row reduction, but computing the permanent is known to be NP-hard. It turns out that those alternating signs have a huge impact on the tractability of the problem.

Matrix Permanent

$\sigma = (1,4,2,3)$

To make sure we understand exactly what the permanent is, let's look at an example. Remember that we said that we're summing over all permutations of entries. What does that mean? Well, let's look at one permutation, say 1, 4, 2, 3. What this means is we go through the rows one-by-one, and we take the first entry, then the fourth, then the second, and finally the third, and we multiply all those entries together. This gives us one product per permutation. Then we sum over these n factorial permutations to get our answer. If we were taking the determinant, we would multiply each product with the sign of the permutation before summing, but in the permanent, we sum them all directly.

Matrix Permanent

$$perm(A) = \sum_{i=1}^{n} A_{1,i} perm(M_{1,i})$$

So we could solve this problem in factorial time by enumerating all the permutations. But can we do better? It turns out we can, using a trick that's very similar to the traveling salesman trick we saw yesterday in class. First, let's write our permanent in terms of permanents of smaller matrices. What we can do is take the expression we had before and group it into n parts, the first being all the permutations that start with 1, the next being all the ones that start with 2, and so on. If we do this, then we see that each part contributes an amount equal to its corresponding entry in the first row, times the permanent of the submatrix you get if you delete the first row and the corresponding column. For example, the part that corresponds to permutations that start with 2 here equals the red entry times the permanent of the purple submatrix. Now, if we were just to use this recurrence directly, we'd still have to do factorial work. However, we can notice that submatrices can show up multiple times. For example, the submatrix that's left over for all permutations that start with (1, 2) is the same as the submatrix that's left over for all the ones that start with (2, 1). In general, to figure out what submatrix we want if we've fixed the first k entries of our permutation, we don't actually care about the order in which those entries were fixed. All we care about is which entries were fixed, because they correspond to the columns we have to delete. Remember that because of the way we're doing this expansion, we're always deleting the first k rows. This means that we actually only have 2^n subproblems, one for each possible submatrix that we create in this manner.

Matrix Permanent

- Use a bit mask to denote active columns
  - 0b111...1 = (1 << n) – 1 = 2^n – 1 = full permanent
  - 1 << i = 2^i = column i
  - column i is active iff (mask & (1 << i)) > 0
  - to deactivate column i, simply subtract (1 << i) from mask
- if k columns are active, so are bottom k rows

So, how do we organize these subproblems? Well, we can use a bit mask where the 1s indicate the columns that we still have, and the 0s are columns that we've deleted. This means that our full permanent corresponds to the bit mask of all 1s, and to find subproblems, we just subtract those 1s out. These numbers are then the numbers that we use to index into our table. You can see that they range from 1 to 2^n – 1, which corresponds to all nonempty subsets of n elements.

Matrix Permanent

- nothing wrong with filling the table recursively (memoization)
- for fun: iterating over all bit sets of size n with k 1s

```
int cur = (1 << k) - 1;
int last = cur << (n - k);
while (cur <= last) {
  ...
  int tmp = (cur | (cur - 1)) + 1;
  cur = tmp | ((((tmp & ~tmp) / (cur & ~cur)) >> 1) - 1);
}
```

(http://graphics.stanford.edu/~seander/bithacks.html#NextBitPermutation)

Now how do we fill this table? In the DP that Keith presented yesterday, we saw that we need to solve the subproblems in increasing order of subset size. If you really want, you can make this ordering explicit by looping over the size of the subsets, and then enumerating all subsets of a given size. I've provided what basically amounts to magic code that does this for you. But I'd also like to point out that there is nothing wrong with implementing a recursive solution that uses memoization. In fact, the solution I wrote up does exactly that. The important thing is to take advantage of the mapping from subsets to integers to be able to create an efficient lookup table without having to resort to hashing tricks.

Matrix Permanent

- Input
  - An n-by-n matrix
  - $1 \le n \le 20$
  - all weights between 1 and 100
- Input Format Per Case
  - Line 1: 1 integer n
  - Lines 2 to n+1: n integers
- Sentinel: n = 0
- Sample

```
3
1 2 3
4 5 6
7 8 9
1
1
0
```

- Output
  - The last 9 digits of the permanent of the matrix
- Output Format Per Case
  - Line 1: 1 integer
- Sample

```
450
1
```

Here's the specification slide for this problem. Notice that you should only print out the last 9 digits of the permanent of the matrix. This is because the permanent can get really big, so we'd like to avoid overflow problems. Instead, use 64-bit integers to store your intermediate values, and make sure to mod by a billion after every arithmetic operation you make.