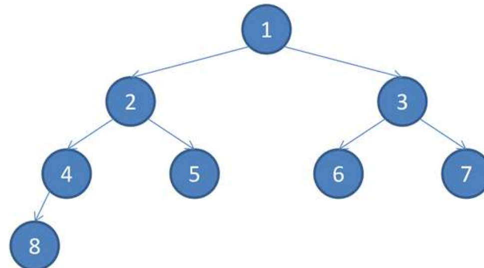
CS161L: Implementation of Algorithms

Friday, May 23, 2014

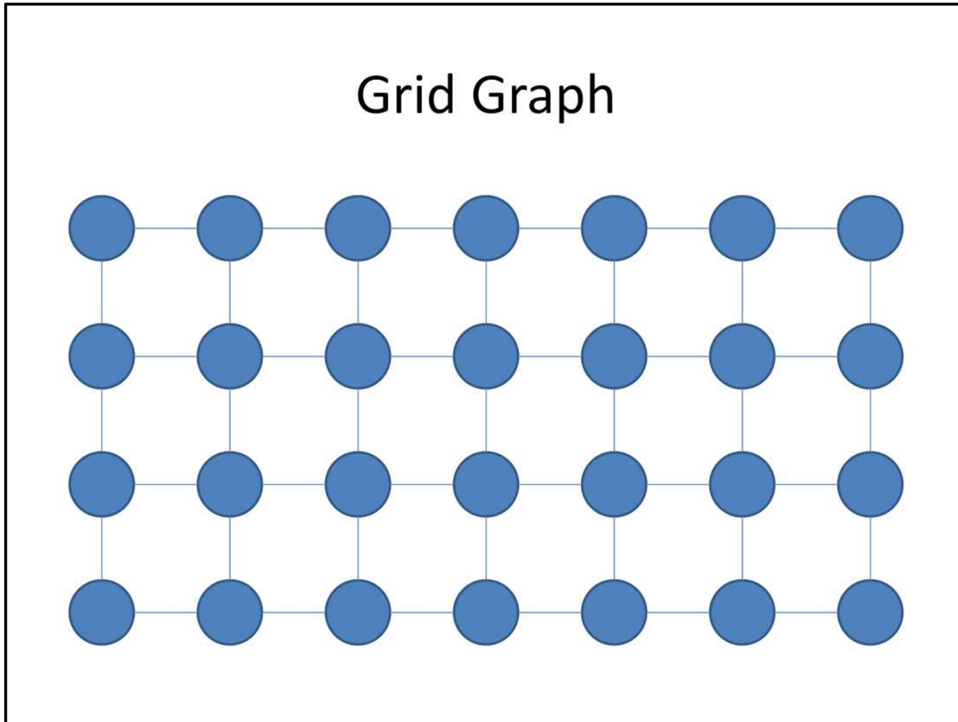
Logistics

- Monday is a holiday
  - Office hours moved to Tuesday from 3 to 5 pm
  - Week 8 problem due Tuesday 11:59pm
- All remaining problems named

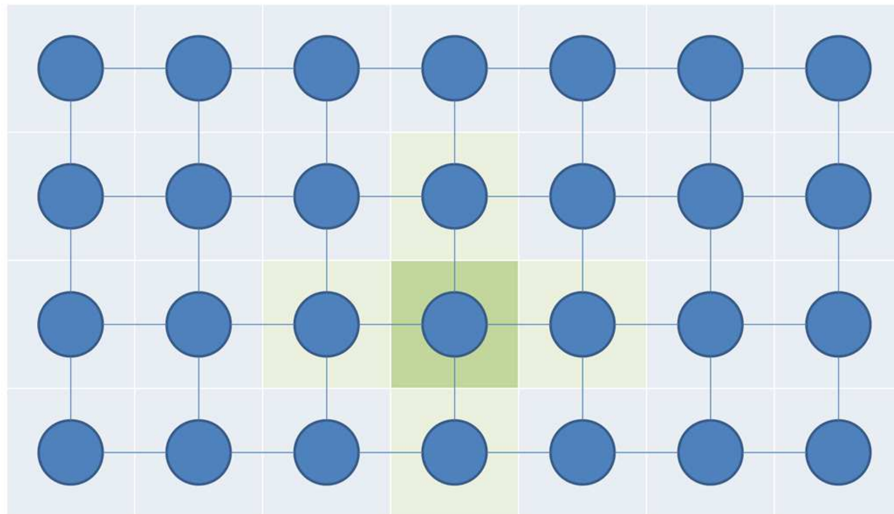
Implicit Graph Representations



On Wednesday in 161 lecture we went over the adjacency matrix and adjacency list representations of graphs. Now, these representations do come up a lot both in theory and in practice, as they are great general-purpose representations. Sometimes, though, we have graphs with a lot of structure to them. In those cases, instead of representing the graph explicitly, we can map the nodes of the graph to an array or similar structure, and then use special rules to figure out what the edges are. We'll call this an implicit graph representation. We've already seen an implicit graph this quarter, namely, when we were working with binary heaps. Today we're going to code up graph search algorithms on implicit graphs.



The first thing we're going to cover is the grid graph. In the grid graph, the nodes are arranged in a lattice or grid, and the edges form a regular pattern. What I have displayed here is a square grid, but you can imagine triangular and hexagonal grid graphs as well.

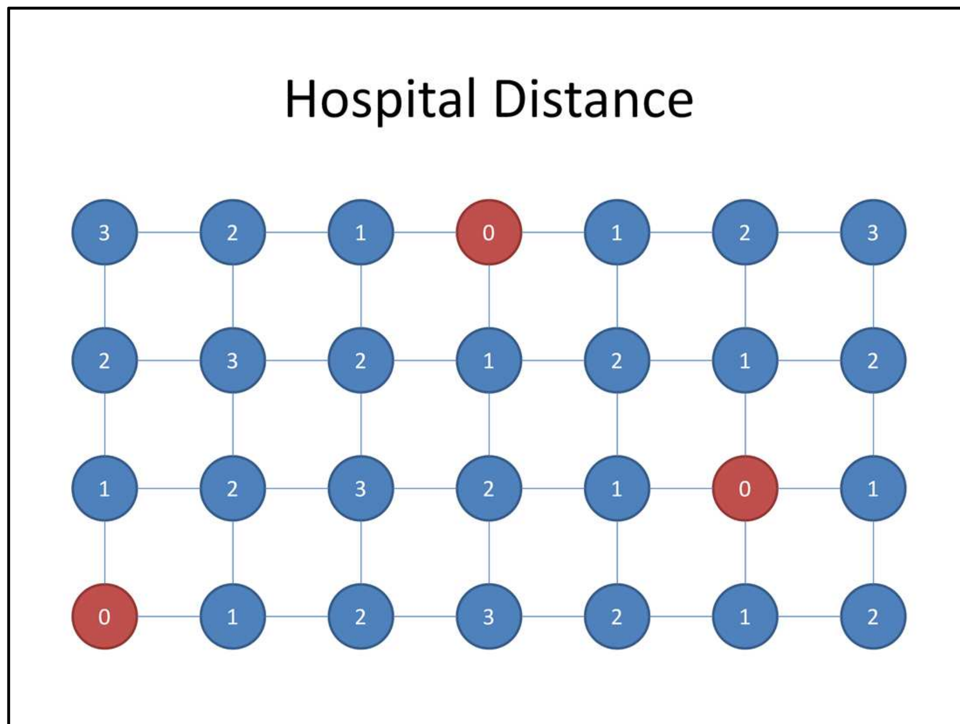


These grid graphs have a very natural representation; we can align the grid with a 2-dimensional array, and arrange the nodes in that array. Then the edges incident to each node can be obtained simply by adding to or subtracting from the coordinates or indices of the node.

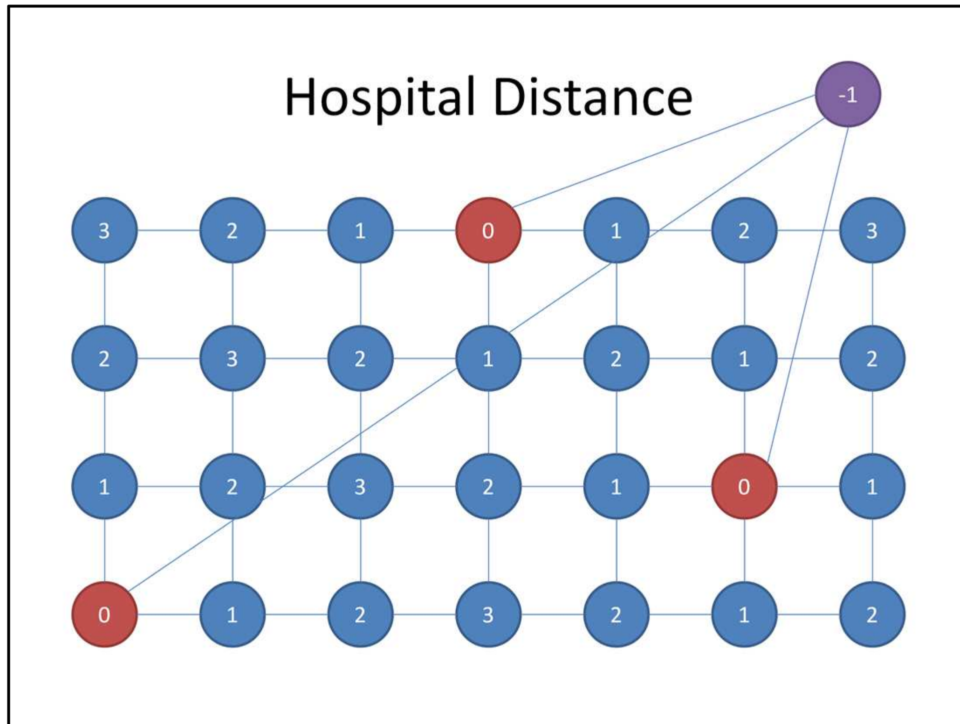
Delta Arrays

```
int[] dx = new int[]{1, 0, -1, 0};
int[] dy = new int[]{0, 1, 0, -1};
for (int i = 0; i < 4; i++) {
  int nx = x + dx[i];
  int ny = y + dy[i];
  if (nx >= 1 && nx <= W &&
      ny >= 1 && ny <= H)
    doStuff(nx, ny);
}
```

A convenient way to iterate over all the neighbors of a node in a grid graph are to use what are known as delta arrays. These are a pair of arrays that contains deltas to the coordinates of our current node that we apply in turn. Here, for example, you can see that the first pair corresponds to the southern neighbor, the second to the east, the third north, and the fourth west. Note that when we use these delta arrays we have to make sure that we remain within the bounds of our grid. Also notice that we can easily change this to accommodate diagonal neighbors as well; we just have to add 4 more entries to each of these delta arrays.



The first problem for this week is about hospital distance. We have a city were the street corners are arranged in a square grid. Some of the street corners have hospitals, and we want to find the distance from each corner to the nearest hospital. We can solve this problem using breadth first search. One idea is to run a BFS from each hospital outward, and then for each node, take the minimum over its distance to all the different hospitals. Unfortunately, this can be inefficient if we have a lot of hospitals. The question is, can we get away with doing only one BFS overall instead of one BFS per hospital?

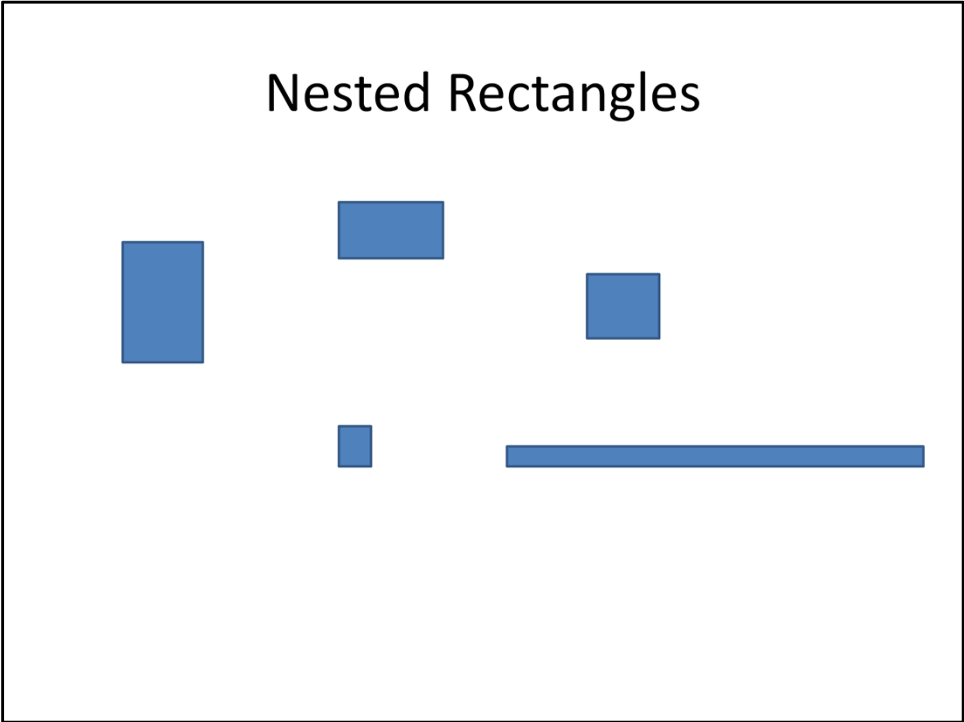


Well, what we can do is add one extra node to our graph, and link it up to all the hospitals. We can see that if we were to run BFS starting at this special node, we would get the distance from this node to every other node in the graph, which is just one larger than the distance from every node to its nearest hospital.

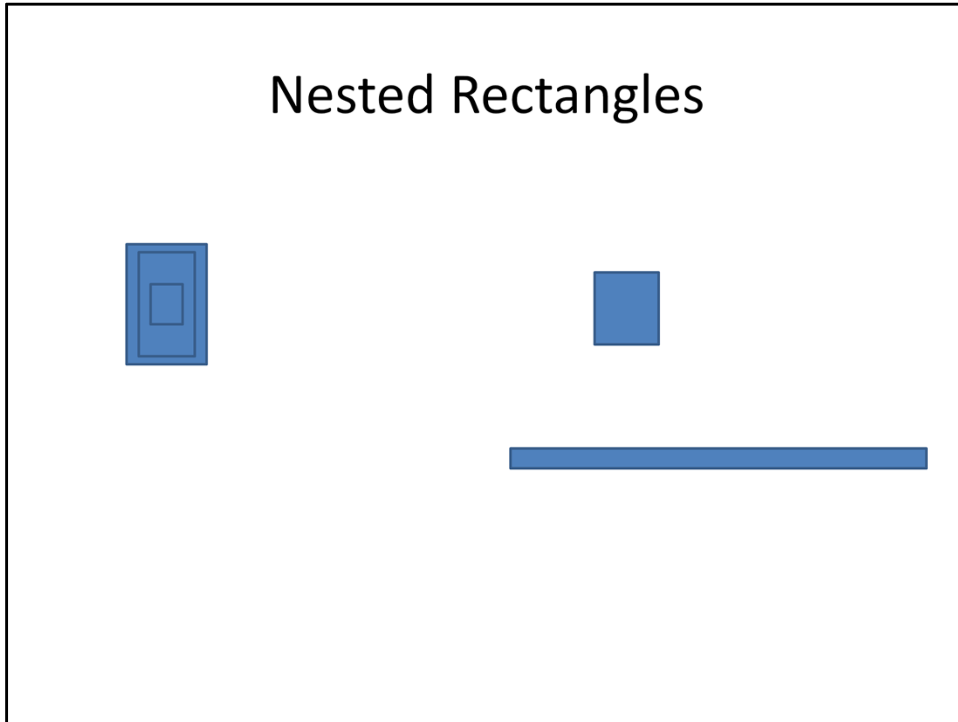
Multiple Source BFS

- Add virtual source
- After 1st iteration: Queue contains all original sources!
- Alternative: Skip virtual source, just put all original sources in directly

Before we go and actually add an extra node when we code this up, let's stop and think about what happens when we do this. We start by enqueueing our virtual source. We then dequeue the virtual source and enqueue all its neighbors, which are exactly the hospital locations. So when we code up this algorithm, we can skip the virtual source entirely and just seed the queue with all the hospital locations to begin with.

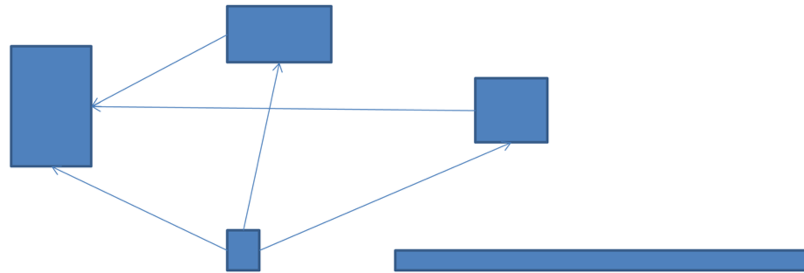


Now onto the second problem. We have a collection of rectangles of different sizes,



and we want to find the longest sequence of rectangles that fit together one inside the next. Notice that we're allowed to rotate the rectangles by 90 degrees to make them fit.

Nested Rectangles



Arrows point in direction of increasing area!

How is this a graph problem? Well, we can imagine a graph where we draw a directed edge from each rectangle to every rectangle that it can fit inside. Notice that if we do this, every edge points from a smaller rectangle to a larger rectangle, which means we have a directed acyclic graph. Whenever we have a DAG and we want to do something with subsets of the nodes, we should see whether we stand to gain anything from a topological ordering of the nodes.

"2-color" DFS

```
void DFS() {
  for (Node n : nodes)
    if (!n.visited)
      DoDFS(n);
}
```

```
void DoDFS(Node n) {
  n.visited = true;
  DoStuff(n);
  for (Node c : n.kids)
    if (!c.visited)
      DoDFS(c);
}
```

Before we do that, though, let's talk about how to do DFS without resorting to 3 different colors as we did in lecture. If you look at the pseudocode carefully, you'll notice that we write three different node colors, but we only check to see whether or not a node is white. That means we never treat gray and black as different colors, so there's no reason to use both those colors in our implementation. Instead, we can just use white and black, which is equivalent to using a boolean value telling us whether we've visited the node. You can do this with BFS as well.

"2-color" Topological Sort

```
void TopSort() {
  for (Node n : nodes)
    if (!n.visited)
      DoDFS(n);
}

Node[] order;
int idx = 0;
```

```
void DoDFS(Node n) {
  n.visited = true;
  DoStuff(n);
  for (Node c : n.kids)
    if (!c.visited)
      DoDFS(c);
  order[idx++] = n;
}
```

Now from this, we can derive a 2-color topological sort. If you trace through this code, you'll see that this puts each Node after all of its descendants. If you want each Node to appear before all of its descendants, you can reverse the order at the end.

For this problem in particular, we don't actually need to maintain the list of children for each node. Since it takes us constant time to figure out whether one rectangle can fit inside the other, and since in the worst case there are order n^2 edges anyway, we can skip storing the edges altogether, and whenever we need to do something for all the children of a rectangle, we can just loop over all the rectangles and check which ones actually fit. This will still take use quadratic time, but we'll only need linear space, as opposed to the quadratic space we would need with either the adjacency matrix or the adjacency list representation.

Now that we have this ordering, how can we solve the nested rectangles problem? Well, we can take a dynamic programming approach. For each rectangle, we're going to define a subproblem where we're looking for the longest sequence of nested rectangles such that the biggest one in the sequence is the one we're looking at. If we topologically sort the rectangles so that the rectangles that can fit inside us all come before us, then we can start from the beginning of that ordering, and build our way up.

Now, if you've been paying very close attention, you should be questioning whether we even need topological sort for this particular problem. (We don't.) What could we do instead? (...) I still want you folks to implement topological sort here, as it's a good exercise. If you didn't have provided sample output, though, this would be one way to generate a ground truth to compare your topsort implementation against.