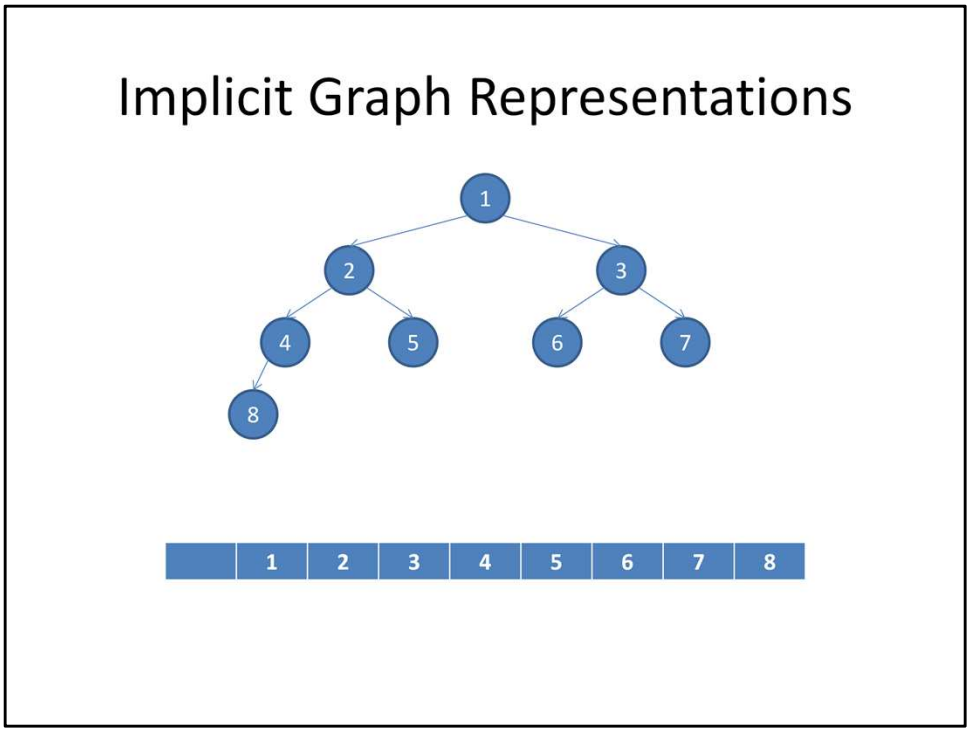
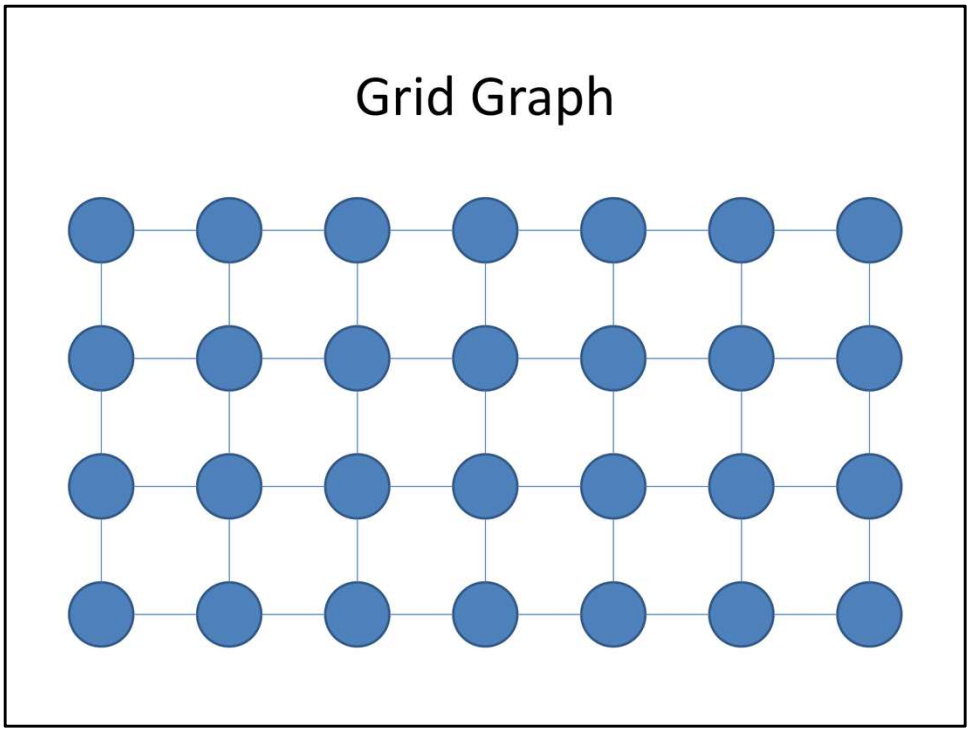
CS161 Programming Section

July 5, 2013

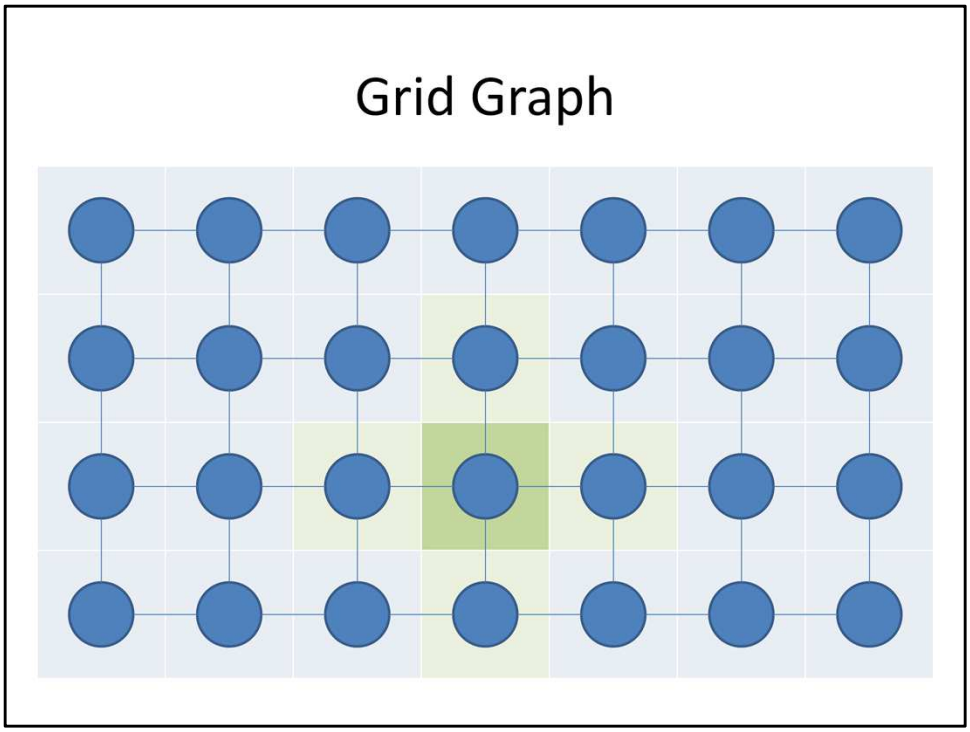
Hi everyone. I hope everyone had a good Fourth of July. Today we're going to be covering graph search. Now, whenever we bring up graph algorithms, we have to talk about the way in which we represent the graphs. In lecture, we've seen the adjacency matrix and the adjacency list representations. These are what we'll call explicit graph representations.



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. Many of you have already seen the binary heap, which is a great example of this. A binary heap is a binary tree filled in layers from left to right. When we use a heap, we often store it as a 1-indexed array. We can then find the children of a node by multiplying its index by 2, and the parent of a node by dividing its index by 2. In this way, we can implicitly represent the graph without having to store any edges. Today we're going to code up our graph search algorithms on graphs that admit implicit representations.



The first graph type 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.

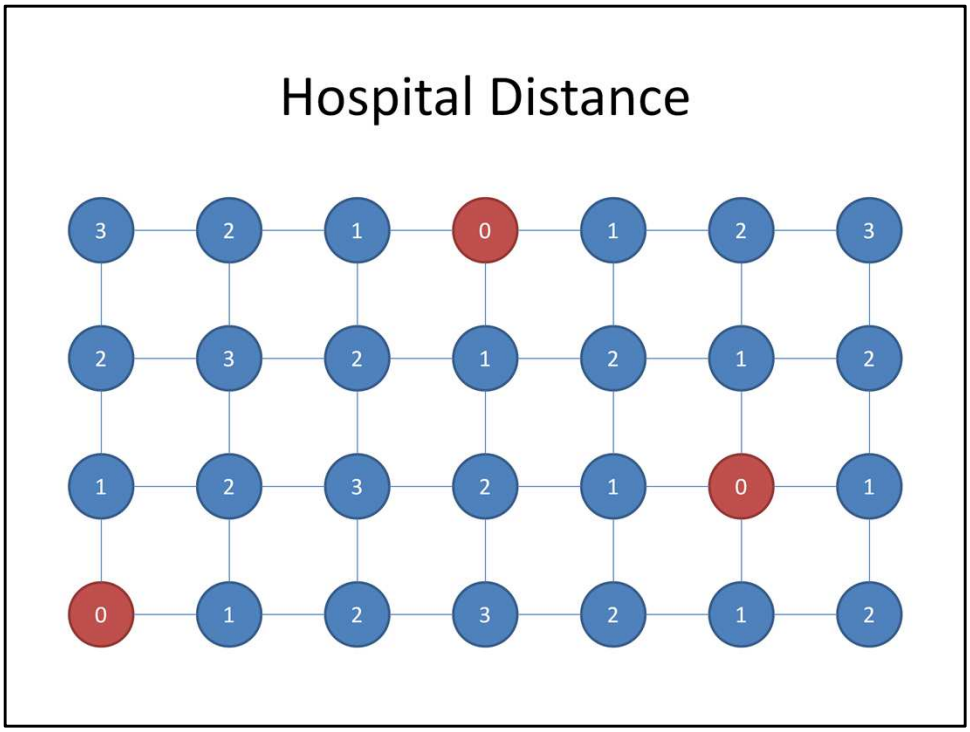


These grid graphs have a very natural representation; we can align the grid with a 2-dimensional array, and arrange the nodes in that array. Then the edges incident to each node can be obtained simply by adding to or subtracting from the coordinates or indices of the node.

Delta Arrays

```
int[] dr = new int[]{1, 0, -1, 0};
int[] dc = new int[]{0, 1, 0, -1};
for (int i = 0; i < 4; i++) {
  int nr = r + dr[i];
  int nc = c + dc[i];
  if (nr >= 0 && nr < R &&
      nc >= 0 && nc < C)
    doStuff(nr, nc);
}
```

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.



So, we're going to practice working with grid graphs by implementing a solution to the hospital distance problem that was on the last problem set. As a reminder, we have a graph where some of the nodes are hospitals, and we want to find the distance from each node to the closest hospital. For today, we're going to solve this problem in the special case where the graph is a square grid graph.

Hospital Distance

- Input
  - An r-by-c rectangular grid
  - k grid cells where hospitals are located
  - r, c, k all between 1 and 1000
- Input Format
  - Line 1: 3 integers r, c, k
  - Lines 2 to k+1: 2 integers r_i, c_i (0-indexed)
- Sample

```
4 7 3
3 0
0 3
2 5
```

- Output
  - The maximum distance of any cell to its closest hospital
- Output Format
  - Line 1: 1 integer
- Sample

```
3
```

As usual, we have our problem specification slide right here. Notice that instead of asking you to print out all of the distances on the entire grid, we're only asking you to print out the maximum distance. We're going to do things like this pretty often in order to keep file size down. Your program should still compute all the distances, though, like your algorithm did for the problem set.

Multiple Source BFS

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

Now remember that when we set up the theoretical justification for this algorithm, we used a virtual source that we connected to all of the hospitals, and ran breadth first search from there. Let's stop and think about what happens when we do this, though. 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.

Structs

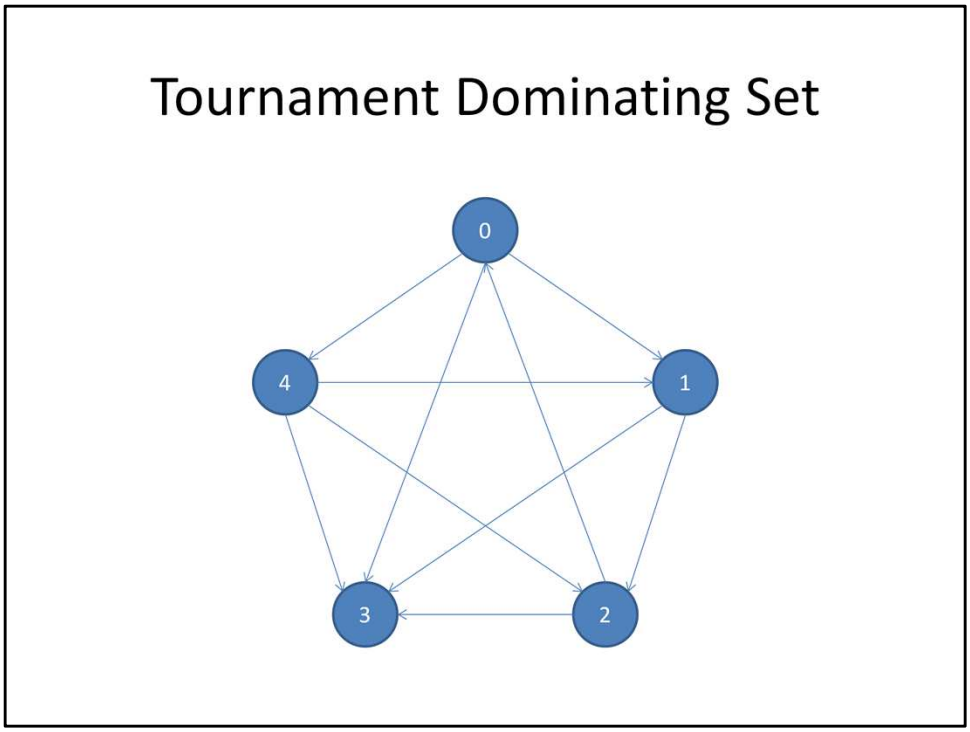
```
struct cell {
  int r, c;
  cell(int r, int c) : r(r), c(c) {}
};
-----------------------------------------------------
static class Cell {
  int r, c;
  public Cell(int r, int c) {
    this.r = r;
    this.c = c;
  }
}
```

Now a brief reminder about some language tools that will come in handy in coding up BFS. You're gonna need a struct to represent the nodes or cells that you place into the queue. As usual, C++ is on top, and Java as below. And this time I actually remembered to put a line between the two languages...

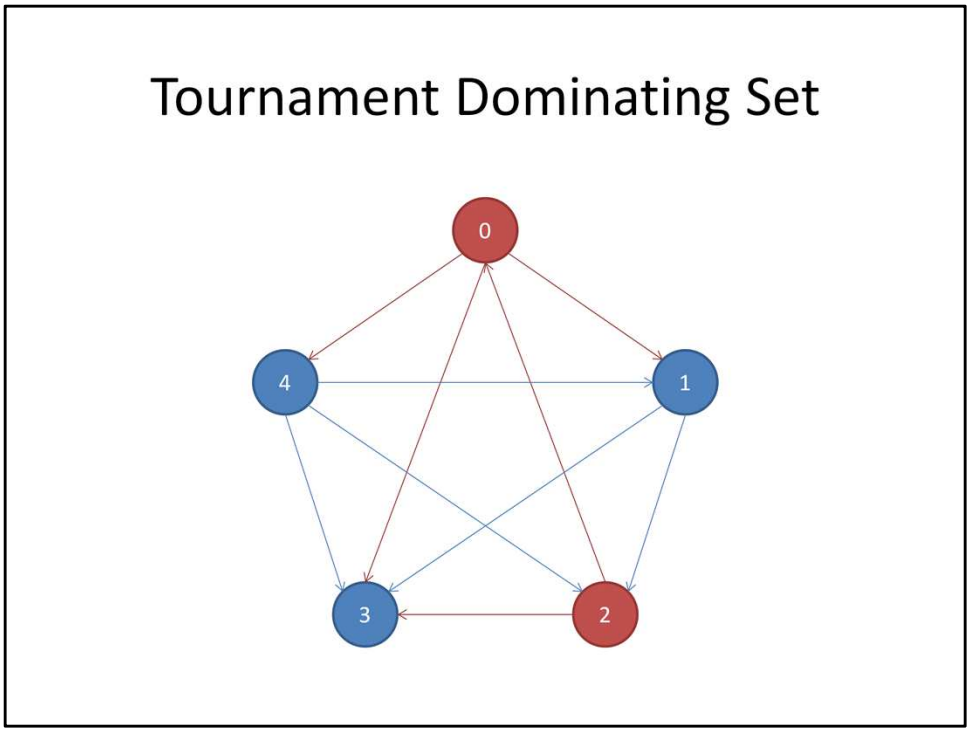
Queues

```
#include <queue>
queue<cell> q;
q.push(cell(r, c));
cell cur = q.front();
q.pop();
-----------------------------------------------------
import java.util.*;
Queue<Cell> q = new LinkedList<Cell>();
q.add(new Cell(r, c));
Cell cur = q.remove();
```

To actually make the queue that you'll use for BFS, you'll use the standard libraries of your language. Here's a brief overview of how to make a queue, enqueue something into it, and dequeue something from it. All right, I'll let you loose on this problem; we'll aim to reconvene around 4:30 to move on to the next program.



All right, this next problem might be a little ambitious to fit into the time we have left, but it's a great motivator for a bunch of different ideas, so let me work through these slides, and if there's not enough time to code it up fully afterwards, you can finish it up on your own time. For this next problem we're going to be looking at tournament graphs. A tournament graph is a directed graph where each pair of nodes has an arc going in exactly one of the two possible directions. We call it a tournament graph because you can think of a round robin tournament, where every pair of nodes faces off in a game, and each game has exactly one winner. NOTE: this is NOT the same as the graph of a tournament as given in Problem Set 2 Problem 4; that graph is not necessarily complete.



In this problem we want to find a dominating set of a tournament graph. A dominating set is a subset of the nodes of the graph such that EVERY node in the graph is either in the subset itself or a child of one of the nodes in the subset. Now, obviously we could choose all the nodes as our subset to get a dominating set. So what we're actually interested in is a dominating set that is as small as possible. For this graph, you can see that the red nodes form a dominating set of size 2, and you can also see that there is no dominating set of size 1. An interesting fact about tournament graphs is they always have a dominating set of size log n, where n is the number of nodes. The explanation of why this is the case is a little long, so I won't present it here, but if you're interested, come talk to me after section.

Tournament Dominating Set

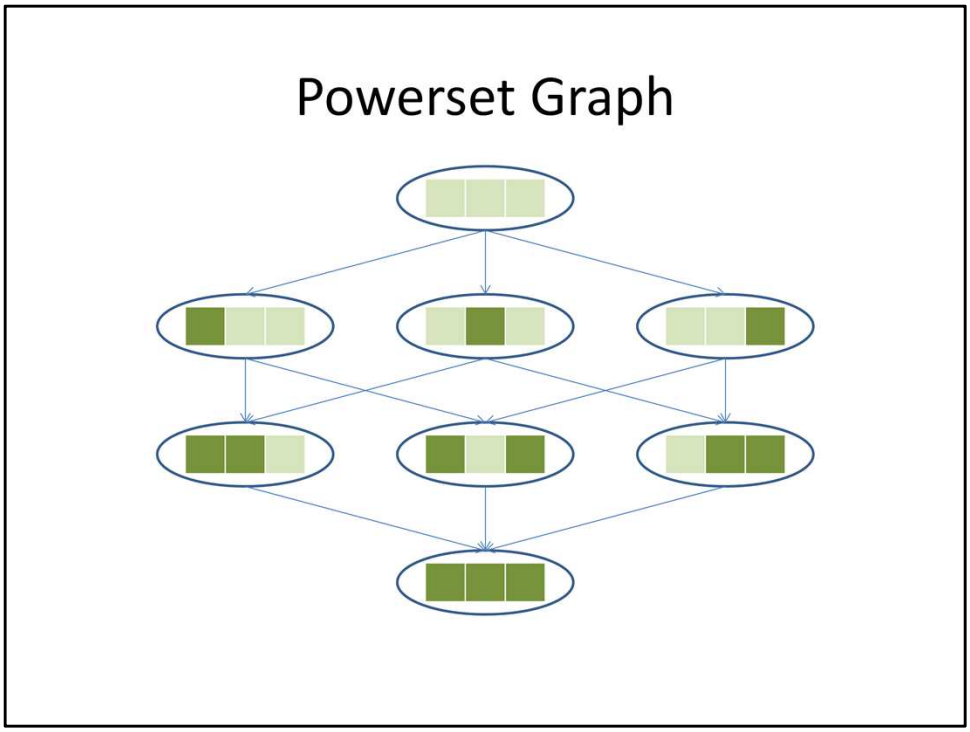
- Input
  - A tournament graph of size n, given as an adjacency matrix
  - n between 1 and 60
- Input Format
  - Line 1: 1 integer n
  - Lines 2 to n+1: n integers
    - The jth number on the ith line is a 1 if there is an arc from i to j, 0 otherwise
- Sample

```
5
0 1 0 1 1
0 0 1 1 0
1 0 0 1 0
0 0 0 0 0
0 1 1 1 0
```

- Output
  - The size of the smallest dominating set
- Output Format
  - Line 1: 1 integer
- Sample

```
2
```

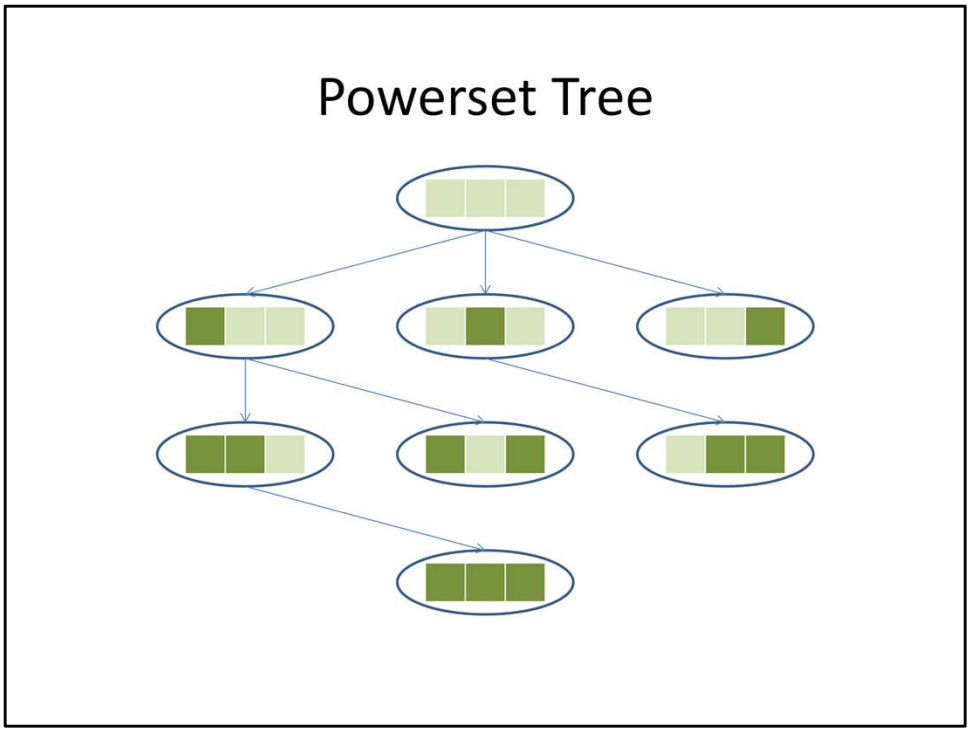
Here's the problem spec slide. Just like in the first problem we did today, we're only asking for one number in the answer. This time, it's the size of the smallest dominating set.



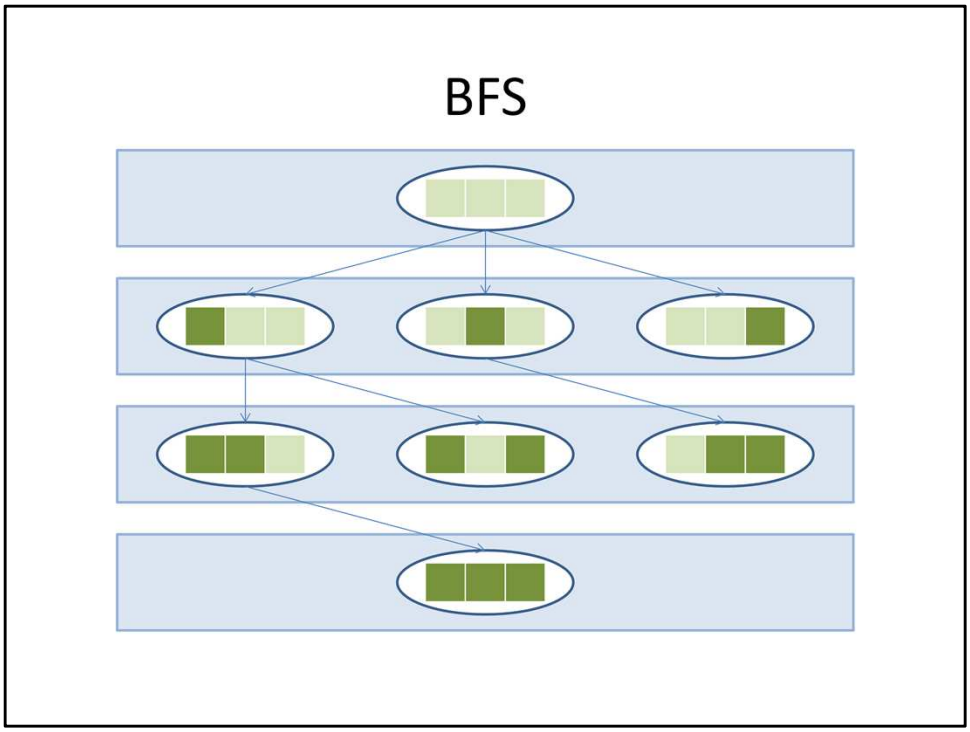
Our general approach to this problem is going to be pretty straightforward. We want to try all subsets of size 1, then all subsets of size 2, then all subsets of size 3, and so on, until we find a dominating set. The interesting thing is we can actually look at this as a graph search problem on the graph of all possible subsets. Some of you may have heard of this thing called the powerset of a set. This is the set of all subsets of a set. So when we say we're trying all the subsets, we're going through the powerset of the set of nodes of our tournament graph. We can then construct a graph on the powerset, by treating each element of the powerset (or each subset of the original set of nodes) as its own node, and drawing an arc from one element to another if we can add one original node to the first subset to get the second. Then trying all subsets of size 1, 2, 3, and so on in that order is the same as running BFS on the powerset graph. Why is this useful? Well, when we're running BFS, we can store the nodes that we cover with our current subset, and we can use that as a starting point for finding the nodes we cover when we add one more node. This is better than having to start from scratch to compute the cover of each and every subset we test.

The problem with this is that this powerset graph is really really big. If we have n nodes in our tournament graph, how many nodes do we have in our powerset graph? (2^n). We said that we were going to go up to 60 nodes, so 2^60 is way too big to fit into memory, so we can't actually construct the full graph. Not only that, but each layer of our graph can be big. What's the biggest layer in this graph, and how many nodes does it have? (n/2, n choose n/2) Now, we know that we're only going to look

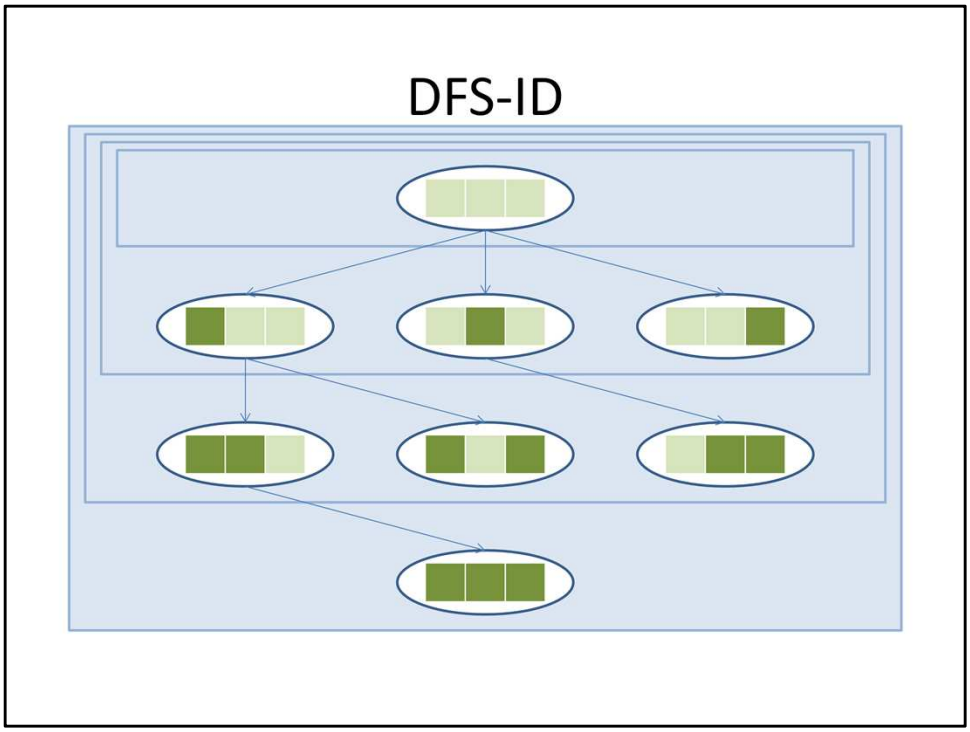
at the first log n layers of this graph, but even then, the last layer we look at is going to have a lot of nodes. On the other hand, this graph doesn't get very deep. Remember that the queue we use in BFS will grow about as large as the width of this graph, while if we were to use DFS for some reason, the stack would only grow as large as the depth of this graph. But also remember that DFS isn't guaranteed in general to give us the shortest path to a node. However, in our case it IS guaranteed to work, since EVERY path to a node has exactly the same length. There are still a few problems with this. The first problem we need to tackle is the fact that we have to mark the nodes as we go along, which uses up a lot of space. Remember that marking is necessary in order to prevent us from rexploring a node we've already explored.



But we have a workaround for that. Remember that in a tree, there's a unique path to every node, so we don't have to mark the nodes we've already visited; we can just loop over all the children. How do we reduce our powerset graph to a powerset tree? Well, we could imagine removing all but the first parent that each node has. It turns out that there's a nice rule for generating these edges: the children of a subset are those subsets that can be obtained by adding exactly one more node that comes strictly after the last node in our current subset. So now, if we were to run DFS, we would have a memory overhead proportional to the depth of the graph as opposed to the width. Unfortunately, there's a bigger problem with using DFS instead of BFS. What is it? (pause) Remember that we wanted to find the smallest dominating set, so we used BFS to ensure that we would visit the layers of this graph from top to bottom, since we knew that we would only need to look at log n layers. DFS does NOT have this property; instead, it will go straight to some very large sets, and then discover, uninterestingly, that taking everything as our subset gives us a dominating set. Does this mean that we have to use BFS and abandon DFS entirely?



It turns out the answer is no; there's a technique we can use called DFS with iterative deepening, or DFS-ID for short. Whenever we have a graph where DFS is guaranteed to report the correct distance, we can use the following trick to get DFS's memory bounds but perform a BFS-like search. Remember that BFS scans the graph one layer at a time.



What we can do instead is we can cut off our graph at depth 1, then depth 2, then depth 3, and so on, and then run DFS on each of these graphs. What this does is it guarantees that we'll look at all of the nodes at depth k before we start looking for nodes at depth k + 1, so we will find the minimum depth node that has the property we want. Notice that we do end up exploring the top layers multiple times, since we're running DFS multiple times. What this does is it gives us a time-space tradeoff: When we run DFS-ID, we'll take more time than BFS because we repeat work, but we'll use less space as long as the graph is wide and shallow. In our case, each layer in the top log n layers grows by at least a factor of two from the layer above it as long as n is bigger than like 4, so we actually end up exploring at most twice as many nodes as BFS, even with the double counting.

Recursive DFS

```
bool search(node current, misc &metadata) {
  if (atGoal(current, metadata)) return true;
  for each child of current {
    updateMetadata(metadata, child);
    if (search(child, metadata)) return true;
    restoreMetadata(metadata, child);
  }
  return false;
}

search(root, metadata);
```

So what does DFS-ID look like in code? Well, let's start with basic DFS. We're going to look at the recursive formulation, where we use the program stack implicitly instead of declaring an explicit stack structure to store our state. Notice that just like before, all we really need to do with our graph is figure out who our children are, so we can implicitly traverse our powerset tree instead of storing the whole thing.

Recursive DFS-ID

```
bool search(node current, misc &metadata, int depth) {
  if (atGoal(current, metadata)) return true;
  if (depth == maxDepth) return false;
  for each child of current {
    updateMetadata(metadata, child);
    if (search(child, metadata, depth + 1)) return true;
    restoreMetadata(metadata, child);
  }
  return false;
}

maxDepth = 0;
while (!search(root, metadata, 0)) maxDepth++;
```

Turing this DFS code into DFS-ID code just takes a couple edits. We just need to track how far down we've gone, and run our search from our root multiple times, bumping up our maximum depth until we finally succeed. All right, we still have some time left in section for you guys to get started on this problem. I'll stick around for as long as you guys are still coding here. Some time tomorrow I'll upload reference solutions to our AFS folder; you can check that out after you've finished to see how your solution compares. Even if your solution is correct, you should check out the reference solution to see if you can make your code shorter or more elegant.

Bit Packing

- boolean array of size 64
  - Java `boolean[64]`
  - C++ `bool[64]`
- set
  - `a[i] = true`
  - `a[i] = false`
- get
  - `a[i]`
- union
  - `for (i = 0 to 63)`
    `c[i] = a[i] || b[i]`
- intersection
  - `for (i = 0 to 63)`
    `c[i] = a[i] && b[i]`

- 64-bit integer
  - Java `long`
  - C++ `long long`
- set
  - `a |= (1L << i)`
  - `a &= (-1L ^ (1L << i))`
- get
  - `(a & (1L << i)) > 0`
- union
  - `c = a | b`
- intersection
  - `c = a & b`

(Additional slide) While coding up the last problem, you might have noticed that one of the bigger pains is keeping track of which nodes have been covered by the subset so far. Chances are you used a boolean array to keep track of coverage. It might interest you to learn that a boolean array isn't actually very memory efficient. The thing is that for ease of use, languages will use far more than 1 bit to store a boolean. In addition, you'll need to loop over the whole array to do what are intuitively very basic operations, such as checking to see whether you've covered everything. What you can do instead is you can use a trick known as bit packing. Remember that an integer is stored in binary, which means it's equivalent to an array of bits. This slide shows you how you can use a 64-bit integer instead of a 64-boolean array and still do the exact same things as before. Notice that while getting and setting individual bits is more annoying, you can use much simpler operations to perform batch updates of your array. The reference solution uses bit packing to handle subset coverage; look at the difference it makes in code length!