CSCI E-119
Section Notes
Section 7 Solutions

1. Searching a list

Suppose we have a list of items *in sorted order*, in both the ArrayList implementation and the LLList implementation.

If we want to find a single item in the list, what can we do?
We can iteratively go through the entire list. Or we can use binary search, since we know that the list is being maintained in sorted order.

What is the efficiency if we use binary search on the array implementation and the linked list implementation?
For the array implementation, our efficiency will be O(log n). However, for the linked list implementation, our efficiency will be O(n log n) since we may need to traverse the list O(n) on each of the log n iterations of binary search.

What would be a better way to search if we knew we were using the linked list implementation?
A better way to search would be to just use linear search and traverse the linked list once. This yield O(n) efficiency.

But implementing a linear search with an LLList is not as straightforward as it might seem. Let's say that we are using an instance of our LLList class in which the items are in sorted order, and that our search() method is ***external*** to the LLList class. Let's also assume that the objects in the LLList implement the Comparable interface.

Here's one way that we could implement the search() method:

```
public static boolean search( Object item, LLList list ) {
        if ( item == null || list == null )
                return false;

        for ( int i = 0; i < list.length(); i++ ) {
                Comparable listItem = (Comparable)list.getItem(i);
                if ( item.equals(listItem) )
                        return true;
                if ( item.compareTo(listItem) < 0 )
                        return false;
        }

        return false;
}
```

What is the problem with this version of the search() method?
Every call to getItem() begins at the first node of the list and traverses the list until it reaches node i. As a result, this method has a time efficiency of O(n^2), rather than O(n).

How could we fix this problem?

Use an iterator, since then our search() method will be able to complete its task using only a single pass down the underlying linked list.

```
public static boolean search( Object item, LLList list ) {
        if ( item == null || list == null )
                return false;

        ListIterator iter = list.iterator();
        while ( iter.hasNext() ) {
                Comparable listItem = (Comparable)iter.next();
                if ( item.equals(listItem) )
                        return true;
                if ( item.compareTo(listItem) < 0 )
                        return false;
        }

        return false;
}
```

2. Breadth-first search
Consider the Eight Puzzle with the following initial state:

1	4	2
3	7	5
6		8

Trace the breadth-first search (BFS) through the **first three nodes that it tests for a goal state**. Show the evolution of the queue and the state-space search tree.

BFS begins by adding a node containing the initial state to the queue:

queue:

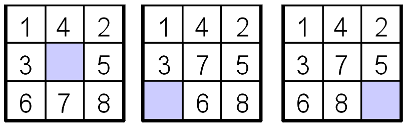
1	4	2
3	7	5
6		8

search tree:

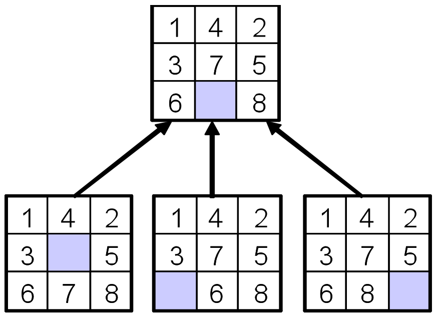
1	4	2
3	7	5
6		8

1) BFS removes the node at the front of the queue and tests to see if it contains a goal state. It doesn't, so it generates the successors of this node (i.e., all states that are one step away) and inserts them in the queue. The parent references in the successors add them to the search tree.

queue:



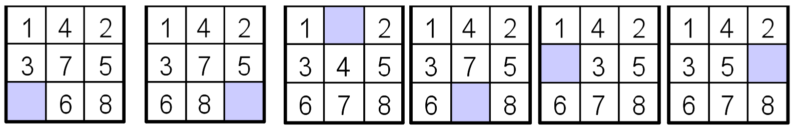
search tree:



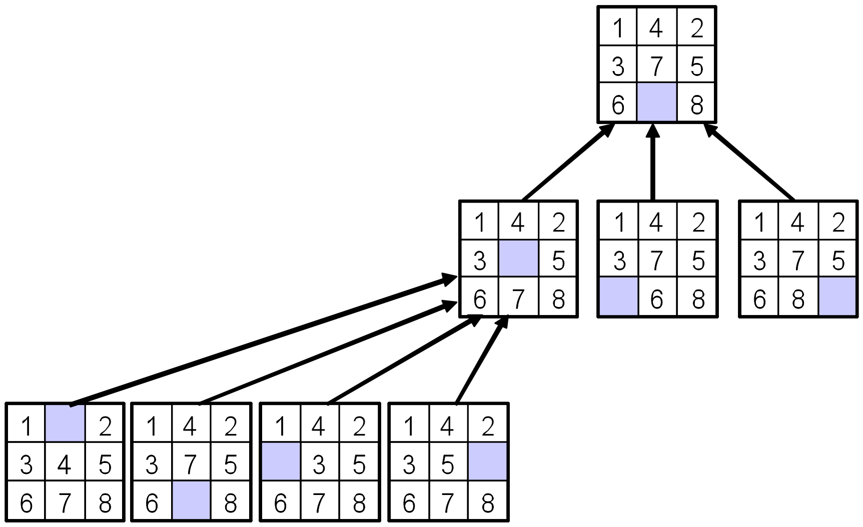
Note that the initial node is still in memory, because its successors have references to it.

2) BFS then removes the node at the front of the queue (the first successor of the initial node) and tests to see if it contains a goal state. It doesn't, so it generates its successors and inserts them in the queue – after the nodes currently in the queue. The parent references in the successors add them to the search tree.

queue:

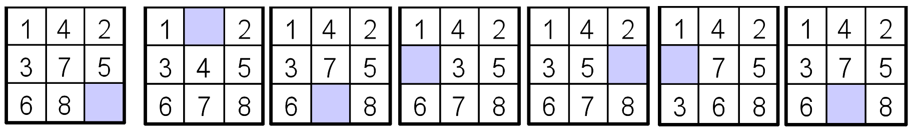


search tree:

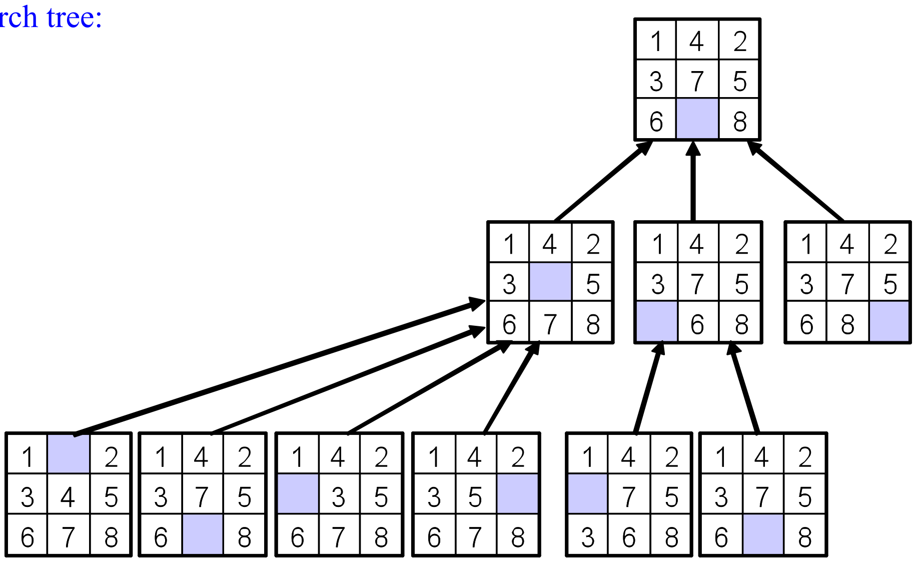


3) BFS then removes the node at the front of the queue (the second successor of the initial node) and tests to see if it contains a goal state. It doesn't, so it generates its successors and inserts them in the queue – after the nodes currently in the queue. The parent references in the successors add them to the search tree.

queue:



search tree:

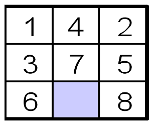


What will BFS do next?

It will consider the last successor of the initial node, and then move on to the nodes that are two steps away from the initial node. All nodes at a given depth in the search tree are visited before it moves onto nodes that are deeper in the tree.

3. Depth-first search

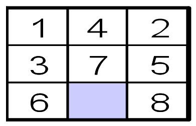
Consider the Eight Puzzle with the same initial state as in 1:



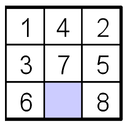
Trace the depth-first search (DFS) through the **first three nodes that it tests for a goal state**. Show the evolution of the stack and the state-space search tree.

DFS begins by pushing a node containing the initial state onto the stack:

stack:

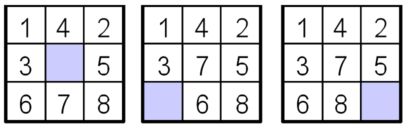


search tree:

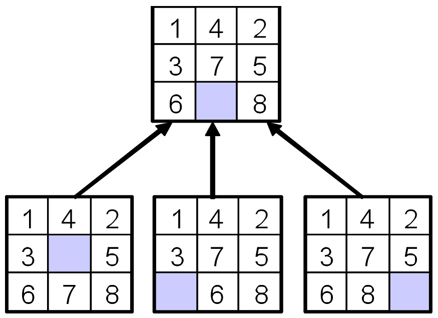


1) DFS pops the node on the top of the stack and tests to see if it contains a goal state. It doesn't, so it generates the successors of this node (i.e., all states that are one step away) and pushes them onto the stack. The parent references in the successors add them to the search tree.

stack:



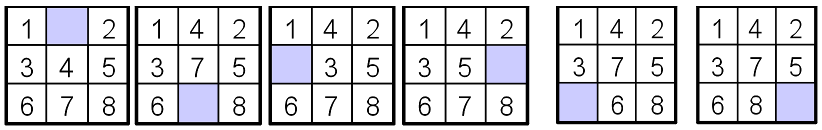
search tree:



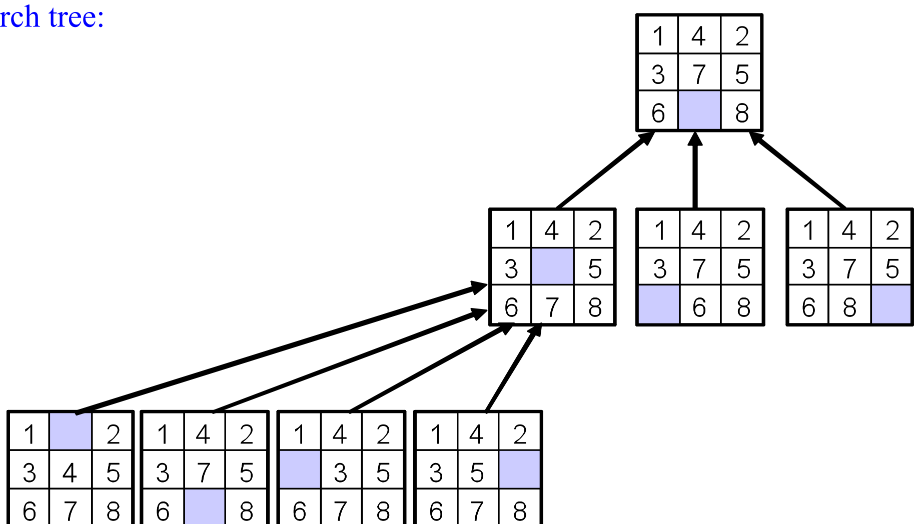
Here again, the initial node is still in memory, because its successors have references to it.

2) DFS then pops the node on the top of the stack (the first successor of the initial node) and tests to see if it contains a goal state. It doesn't, so it generates its successors and pushes them onto the stack – *in front of* the nodes currently on the stack.

stack:

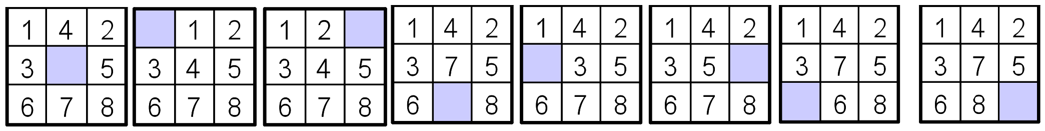


search tree:

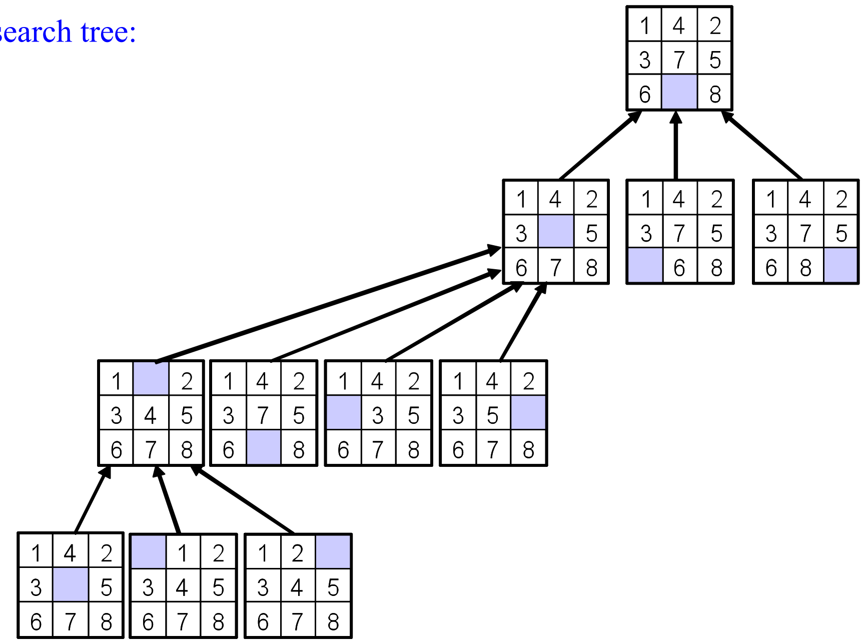


3) DFS then pops the node on the top of the stack (the first successor of the node considered in the previous step – which is two steps away from the initial node) and tests to see if it contains a goal state. It doesn't, so it generates its successors and pushes them onto the stack.

stack:



search tree:

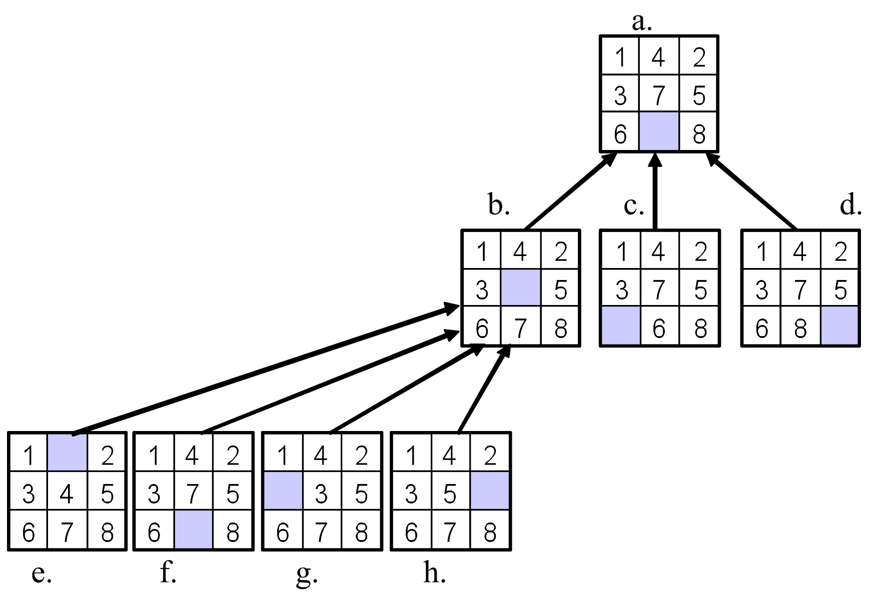


What will DFS do next?

It will consider the first successor of the node considered in step 3 (the leftmost node at the bottom of the search tree, which is three steps away from the initial state), and then move on to that node's first successor (which is four steps away), etc. It continues as far as it can on a given path before going back to consider siblings of the nodes on the original path.

4. Iterative Deepening Search (IDS)

Consider the state-space search tree shown below, with the initial state at the root of the tree. Give the labels of the first 10 states to which IDS applies the goal test. You can assume that the nodes in a given level of the tree are tested in alphabetical order.



The first 10 states that it tests are: a, a, b, c, d, a, b, e, f, g

IDS begins with a depth limit of 0, which allows it to test only the initial state (state a).

It then performs DFS with a depth limit of 1. It again tests state a. Since it isn't the goal state, it generates a's successors (b, c, and d) and pushes them on the stack. It then pops and tests state b: it's not the goal state, but it's at the depth limit, so it doesn't generate b's successors. Similarly, it tests states c and d but doesn't generate their successors.

It then performs DFS with a depth limit of 2. It again tests state a and generates its successors (b, c, and d). It then pops and tests state b: it's not the goal, and it's not at the depth limit, so it generates its successors (e, f, g, h) and pushes them on the stack. It then pops and tests state e: it's not the goal, but it's at the depth limit, so it doesn't generate e's successors. Similarly, it tests states f, g, and h, but doesn't generate their successors. This brings us past the first 10 states tested.

By trying all the depths one by one, IDS resolves the issue of selecting the correct depth limit so that DFS goes deep enough to find the solution, yet does not get stuck going down a long path that doesn't lead to a goal state. In this way IDS combines the benefits of DFS and BFS – linear space complexity and optimality and completeness. However, IDS seems wasteful because as it deepens the search, many of the nodes that were tested in the previous iteration get tested again. Looking at the above example, node a is tested at every iteration as IDS increases the depth.

Let's try to get a feel for how wasteful IDS really is.

First consider using depth-limited DFS with no iterative deepening. If each node has B successors in the worst case and D is the depth limit, what is the maximum number of nodes that will be tested?

$1 + B + B^2 + \ldots + B^{D-1} + B^D$

In the worst case we would consider every node at every level once. With B successors for each node, there are B^N nodes at depth N (N goes between 0 and D). Summing this up over the depths 0 though D gives the above result.

Now consider IDS which we run up to (and including) depth D. Again assuming each node has B successors in the worst case, what is the maximum number of nodes that will be tested?

$(D+1)1 + (D)B + (D-1)\, B^2 + \ldots + 2B^{D-1} + 1B^D$

In the worst case, we would test each node at level N (N between 0 and D) once for every level starting with N and increasing to D – that is D-N+1 times. Since there are B^N nodes at level N we get the above sum.

If we take B = 10 and D = 6 what is the maximum number of nodes tested by the depth-limited DFS?

$1 + B + B^2 + \ldots + B^{D-1} + B^D = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 + 1{,}000{,}000 = 1{,}111{,}111$

And testing up to and including depth D = 6, assuming B = 10 how many nodes are tested by IDS in the worst case?

$(D+1)1 + (D)B + (D-1)\, B^2 + \ldots + 2B^{D-1} + 1B^D =$
$7*1 + 6*10 + 5*100 + 4*1{,}000 + 3*10{,}000 + 2*100{,}000 + 1*1{,}000{,}000 = 1{,}234{,}567$

With B = 10, in testing to depth D, IDS tests only about 11% more nodes than BFS that tests up to level D and DFS that is depth-limited with depth D.

The intuition behind the above calculation is that in a tree in which each node has B successors, most of the nodes are at the greatest depth. Thus the majority of the nodes only get tested once. For example, with B = 10, the number of nodes at the greatest depth level represents about 90% of the total number of nodes in the tree.