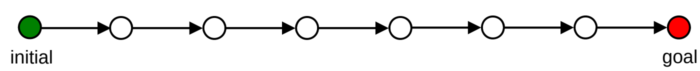
State-Space Search

Computer Science E-22
Harvard Extension School

David G. Sullivan, Ph.D.

Solving Problems by Searching

- A wide range of problems can be formulated as *searches*.
  - more precisely, as the process of searching for a *sequence of actions* that take you from an *initial state* to a *goal state*



- Examples:
  - n-queens
    - initial state: an empty n x n chessboard
    - actions (also called *operators*): place or remove a queen
    - goal state: n queens placed, with no two queens on the same row, column, or diagonal
  - map labeling, robot navigation, route finding, *many others*
- State space = all states reachable from the initial state by taking some sequence of actions.

The Eight Puzzle

- A 3 x 3 grid with 8 sliding tiles and one “blank”
- Goal state:

	1	2
3	4	5
6	7	8

- Initial state: some other configuration of the tiles
  - example:

3	1	2
4		5
6	7	8

- Slide tiles to reach the goal:

3	1	2
4		5
6	7	8

$\Rightarrow$

3	1	2
	4	5
6	7	8

$\Rightarrow$

	1	2
3	4	5
6	7	8

Formulating a Search Problem

- Need to specify:
  1. the *initial state*
  2. the *operators:* actions that take you from one state to another
  3. a *goal test:* determines if a state is a goal state
     - if only one goal state, see if the current state matches it
     - the test may also be more complex:
       – n-queens: do we have n queens on the board without any two queens on the same row, column, or diagonal?
  4. the *costs* associated with applying a given operator
     - allow us to differentiate between solutions
     - example: allow us to prefer 8-puzzle solutions that involve fewer steps
     - can be 0 if all solutions are equally preferable

Eight-Puzzle Formulation

- *initial state:* some configuration of the tiles
- *operators*: it's easier if we focus on the blank
  - get only four operators
    - move the blank up
    - move the blank down
    - move the blank left
    - move the blank right

3	1	2
4		5
6	7	8

move the blank up

3		2
4	1	5
6	7	8

- *goal test*: simple equality test, because there's only one goal

	1	2
3	4	5
6	7	8

- *costs:*
  - cost of each action = 1
  - cost of a sequence of actions = the number of actions

Performing State-Space Search

- Basic idea:

  If the initial state is a goal state, return it.

  If not, apply the operators to generate all states that are one step from the initial state (its *successors*).

3	1	2
4		5
6	7	8

initial state

3		2
4	1	5
6	7	8

3	1	2
4	7	5
6		8

3	1	2
	4	5
6	7	8

3	1	2
4	5	
6	7	8

its successors

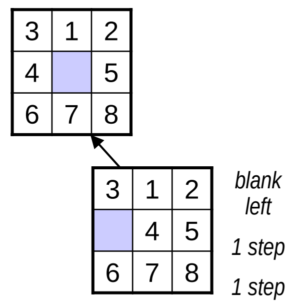
  Consider the successors (and their successors…) until you find a goal state.

- Different search strategies consider the states in different orders.
  - they may use different data structures to store the states that have yet to be considered

Search Nodes

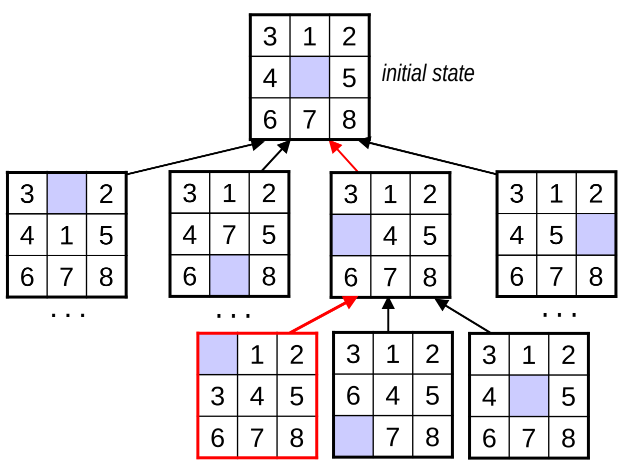
- When we generate a state, we create an object called a *search node* that contains the following:
  - a representation of the state
  - a reference to the node containing the *predecessor*
  - the operator (i.e., the action) that led from the predecessor to this state
  - the number of steps from the initial state to this state
  - the cost of getting from the initial state to this state
  - an estimate of the cost remaining to reach the goal

```
public class SearchNode {
    private Object state;
    private SearchNode predecessor;
    private String operator;
    private int numSteps;
    private double costFromStart;
    private double costToGoal;
    …
}
```



State-Space Search Tree

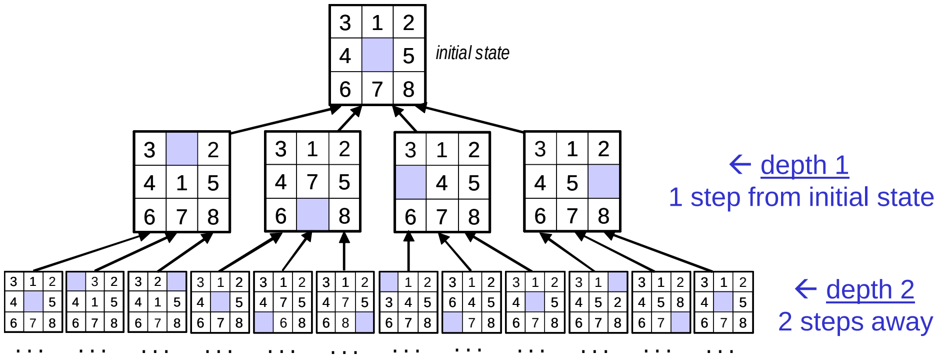
- The predecessor references connect the search nodes, creating a data structure known as a *tree*.



- When we reach a goal, we trace up the tree to get the solution – i.e., the sequence of actions from the initial state to the goal.

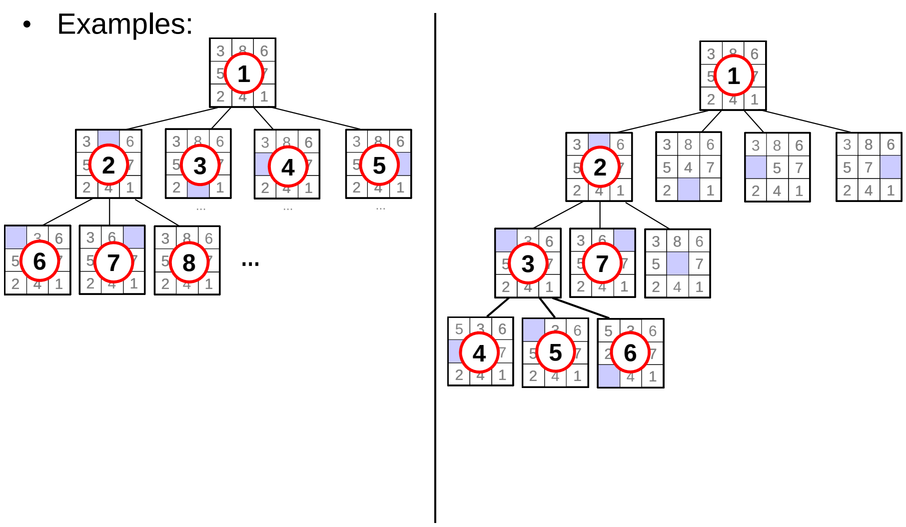
State-Space Search Tree (cont.)

- The top node is called the *root*. It holds the initial state.
- The predecessor references are the *edges* of the tree.
- *depth* of a node *N* = # of edges on the path from *N* to the root
- All nodes at a depth i contain states that are i steps from the initial state:



State-Space Search Tree (cont.)

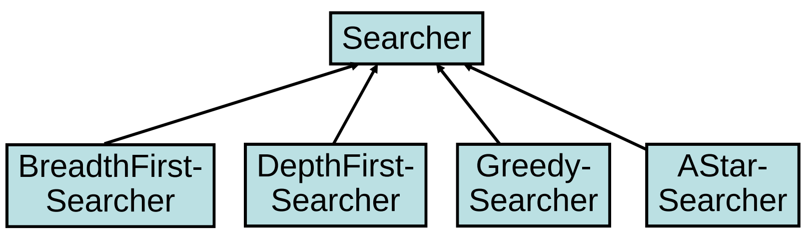
- Different search strategies correspond to different ways of considering the nodes in the search tree.
- Examples:



Representing a Search Strategy

- We'll use a *searcher* object.
- The searcher maintains a data structure containing the search nodes that we have yet to consider.
- Different search strategies have different searcher objects, which consider the search nodes in different orders.
- A searcher object may also have a *depth limit*, indicating that it will not consider search nodes beyond some depth.
- Every searcher must be able to do the following:
  - add a single node (or a list of nodes) to the collection of yet-to-be-considered nodes
  - indicate whether there are more nodes to be considered
  - return the next node to be considered
  - determine if a given node is at or beyond its depth limit

A Hierarchy of Searcher Classes



- `Searcher` is an *abstract* superclass.
  - defines instance variables and methods used by all search algorithms
  - includes one or more *abstract methods* – i.e., the method header is specified, but not the method definition
    - these methods are defined in the subclasses
  - it *cannot* be instantiated
- Implement each search algorithm as a subclass of `Searcher`.

An Abstract Class for Searchers

```
public abstract class Searcher {
    private int depthLimit;

    public abstract void addNode(SearchNode node);
    public abstract void addNodes(List nodes);
    public abstract boolean hasMoreNodes();
    public abstract SearchNode nextNode();
    …
    public void setDepthLimit(int limit) {
        depthLimit = limit;
    }
    public boolean depthLimitReached(SearchNode node) {
        …
    }
    …
}
```

- Classes for specific search strategies will extend this class and implement the abstract methods.
- We use an abstract class instead of an interface, because an abstract class allows us to include instance variables and method definitions that are inherited by classes that extend it.

Using Polymorphism

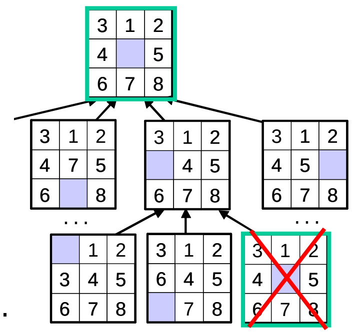
```
SearchNode findSolution(Searcher searcher, …) {
    numNodesVisited = 0;
    maxDepthReached = 0;

    searcher.addNode(makeFirstNode());
    ...
```

- The method used to find a solution takes a parameter of type `Searcher`.
- Because of polymorphism, we can pass in an object of *any* subclass of `Searcher`.
- Method calls made using the variable `searcher` will invoke the version of the method that is defined in the subclass to which the object belongs.
  - what is this called?

Pseudocode for Finding a Solution

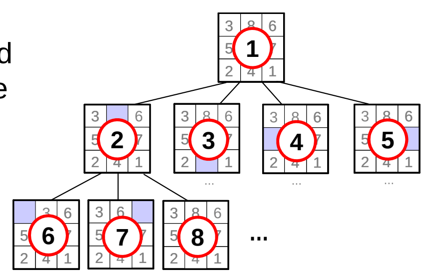
```
searcher.addNode(initial node);

while (searcher.hasMoreNodes()) {
    N = searcher.nextNode();
    if (N is the goal)
        return N;
    if (!searcher.depthLimitReached(N))
        searcher.addNodes(list of N's successors);
}
```

- Note that we don't generate a node's successors if the node is at or beyond the searcher's depth limit.
- Also, when generating successors, we usually don't include states that we've already seen in the current path from the initial state (ex. at right).



Breadth-First Search (BFS)

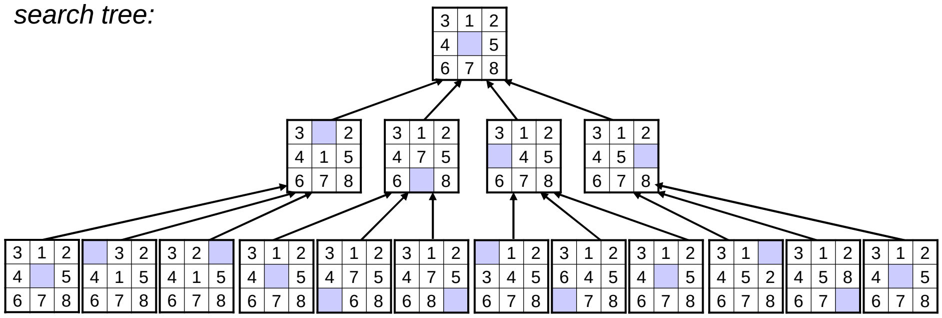
- When choosing a node from the collection of yet-to-be-considered nodes, always choose one of the shallowest ones.
  - consider all nodes at depth 0
  - consider all nodes at depth 1
  - …
- The searcher for this strategy uses a *queue*.



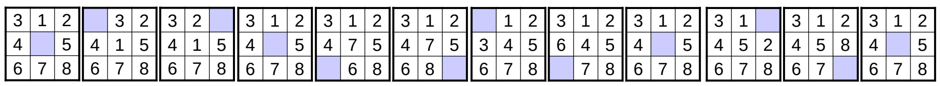
```
public class BreadthFirstSearcher extends Searcher {
    private Queue<SearchNode> nodeQueue;

    public void addNode(SearchNode node) {
        nodeQueue.insert(node);
    }
    public SearchNode nextNode() {
        return nodeQueue.remove();
    }
    …
}
```

Tracing Breadth-First Search

search tree:



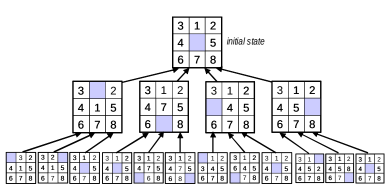
queue:



After considering all nodes at a depth of 1, BFS would move next to nodes with a depth of 2. *All previously considered nodes remain in the tree, because they have yet-to-be-considered successors.*

Features of Breadth-First Search

- It is *complete:* if there is a solution, BFS will find it.
- For problems like the eight puzzle in which each operator has the same cost, BFS is *optimal*: it will find a minimal-cost solution.
  - it may *not* be optimal if different operators have different costs
- Time and space complexity:
  - assume each node has b successors in the worst case
  - finding a solution that is at a depth d in the search tree has a time *and* space complexity = ?



← 1 node at depth 0

← $O(b)$ nodes at depth 1

← $O(b^2)$ nodes at depth 2

  - nodes considered (and *stored*) = $1 + b + b^2 + \ldots + b^d$ = ?

Features of Breadth-First Search (cont.)

- Exponential space complexity turns out to be a bigger problem than exponential time complexity.
- Time and memory usage when b = 10:

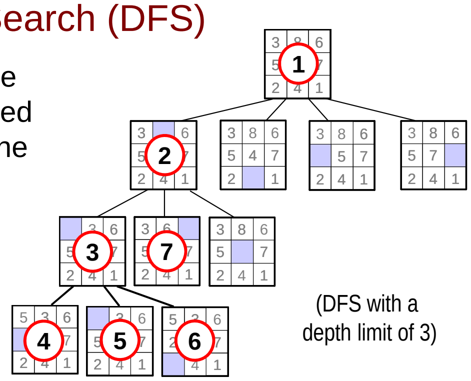
solution depth	nodes considered	time	memory
0	1	1 millisecond	100 bytes
4	11,111	11 seconds	1 megabyte
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	***1 terabyte***
12	10^{12}	35 years	111 terabytes

- Try running our 8-puzzle solver on the initial state shown at right!

	8	7
6	5	4
3	2	1

Depth-First Search (DFS)

- When choosing a node from the collection of yet-to-be-considered nodes, always choose one of the deepest ones.
  - keep going down a given path in the tree until you're stuck, and then backtrack
- What data structure should this searcher use?

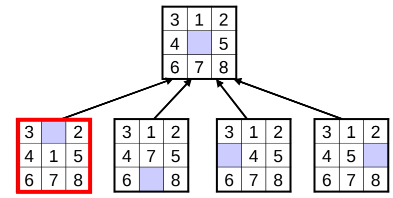


(DFS with a depth limit of 3)

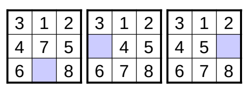
```
public class DepthFirstSearcher extends Searcher {

    public void addNode(SearchNode node) {

    }
    public SearchNode nextNode() {

    }
    …
}
```

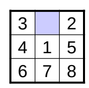
Tracing Depth-First Search (depth limit = 2)

search tree:



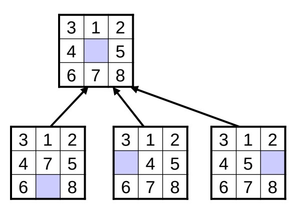
stack:



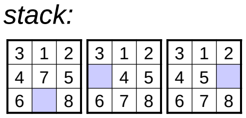
Once all of the successors of  have been considered, there are no remaining references to it. Thus, the memory for this node will also be reclaimed.

Tracing Depth-First Search (depth limit = 2)

search tree:



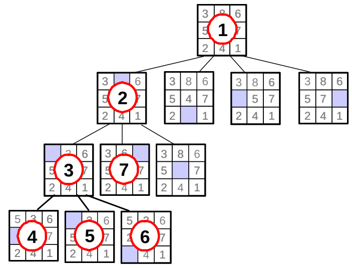
stack:



DFS would next consider paths that pass through its second successor. *At any point in time, only the nodes from a single path (along with their "siblings") are stored in memory.*

Features of Depth-First Search

- Much better space complexity:
  - let m be the maximum depth of a node in the search tree
  - DFS only stores a single path in the tree at a given time – along with the “siblings” of each node on the path
  - space complexity = $O(b*m)$

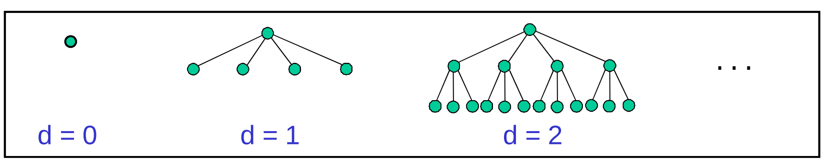


- Time complexity: if there are many solutions, DFS can often find one quickly. However, worst-case time complexity = $O(b^m)$.
- Problem – it can get stuck going down the wrong path.
  - ➔ thus, it is neither complete nor optimal.
- Adding a depth limit helps to deal with long or even infinite paths, but how do you know what depth limit to use?

Iterative Deepening Search (IDS)

- Eliminates the need to choose a depth limit
- Basic idea:

```
d = 0;
while (true) {
    perform DFS with a depth limit of d;
    d++;
}
```



- Combines the best of DFS and BFS:
  - at any point in time, we’re performing DFS, so the space complexity is linear
  - we end up considering all nodes at each depth limit, so IDS is complete like BFS (and optimal when BFS is)

Can't We Do Better?

- Yes!
- BFS, DFS, and IDS are all examples of *uninformed* search algorithms – they always consider the states in a certain order, without regard to how close a given state is to the goal.
- There exist other *informed* search algorithms that consider (estimates of) how close a state is from the goal when deciding which state to consider next.
- We'll come back to this topic once we've considered more data structures!
- For more on using state-space search to solve problems:

  Artificial Intelligence: A Modern Approach.
  Stuart Russell and Peter Norvig (Prentice-Hall).

- The code for the Eight-Puzzle solver is in code for this unit.
- To run the Eight-Puzzle solver:

```
javac EightPuzzle.java
java EightPuzzle
```

  When it asks for the initial board, enter a string specifying the positions of the tiles, with 0 representing the blank.

  example: for

	8	7
6	5	4
3	2	1

  you would enter 087654321

- To get a valid initial state, take a known configuration of the tiles and swap *two pairs* of tiles. Example:

3	1	2
4		5
6	7	8

⇨

3	**7**	2
4		5
6	**1**	8

⇨

3	7	**4**
2		5
6	1	8

  (you can also "move the blank" as you ordinarily would)