State-Space Search (cont.)

Computer Science 111
Boston University

David G. Sullivan, Ph.D.

Recall: The Eight Puzzle

3	1	2
4		5
6	7	8

one possible initial state

	1	2
3	4	5
6	7	8

goal state

- Want to find a sequence of moves that will take us from the *initial state* to the *goal state*.
  - think in terms of what happens to the blank cell:

3	1	2
4		5
6	7	8

move the blank left →

3	1	2
	4	5
6	7	8

move the blank up →

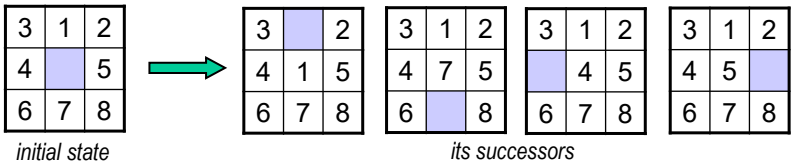
	1	2
3	4	5
6	7	8

Recall: Performing State-Space Search

- Basic idea:

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

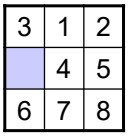
  If not, generate its successors.



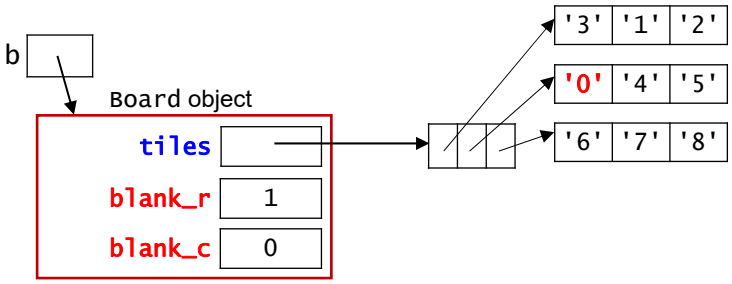
  Consider the successors (and their successors…) until you find a goal state.

- Different search algorithms consider the states in different orders.

Part I: `Board` Objects

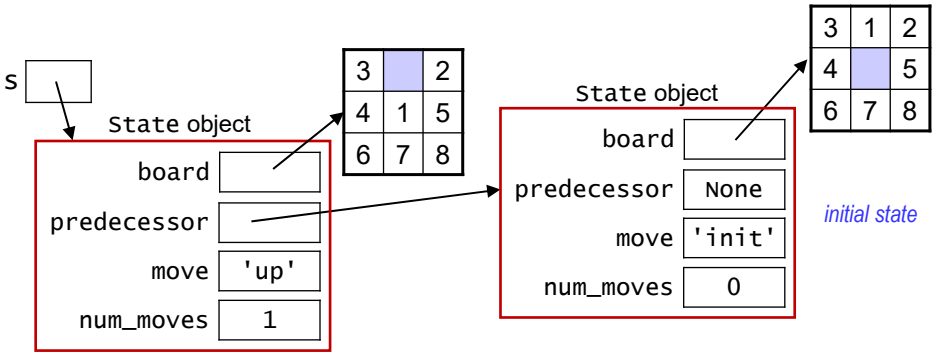


```
b = Board('312045678')
```

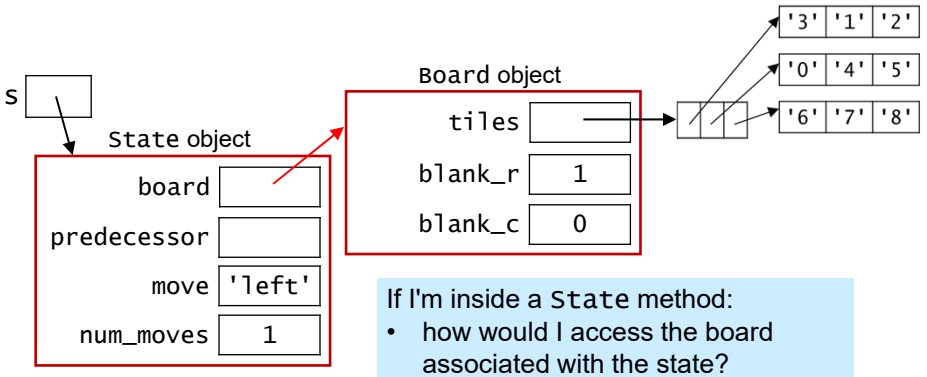


Part II: `State` Objects

- Each state is represented by an object that contains:
  - the corresponding `Board` object
  - a reference to a `State` object for the *predecessor* state
  - the move that led from the predecessor to this state
  - the number of moves from the initial state to this state



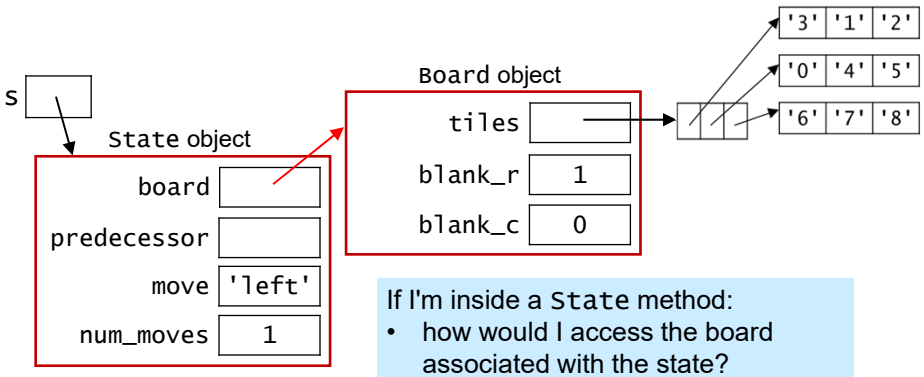
Advanced Dot Notation!



If I'm inside a `State` method:

- how would I access the board associated with the state?
- how would I access the tiles list of that board?

Advanced Dot Notation!

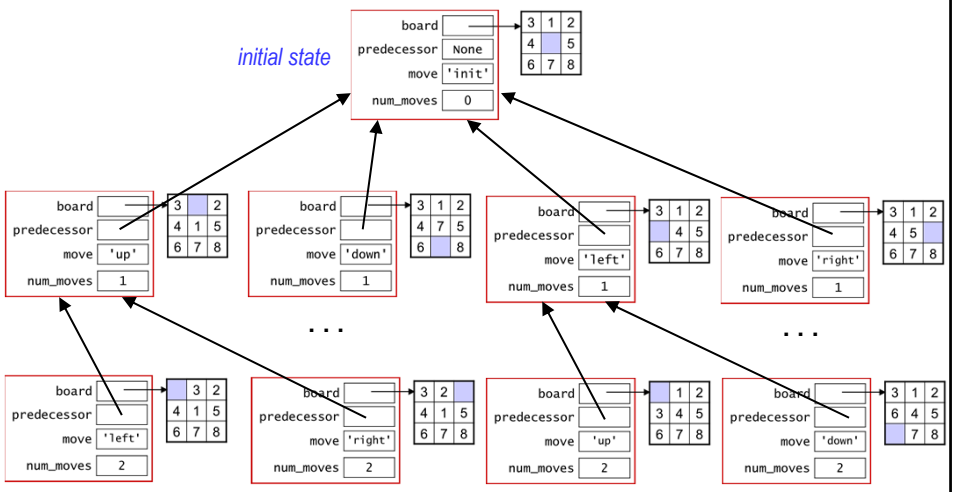


If I'm inside a `State` method:

- how would I access the board associated with the state?
  `self.board`
- how would I access the tiles list of that board?
  `self.board.tiles`

State-Space Search Tree

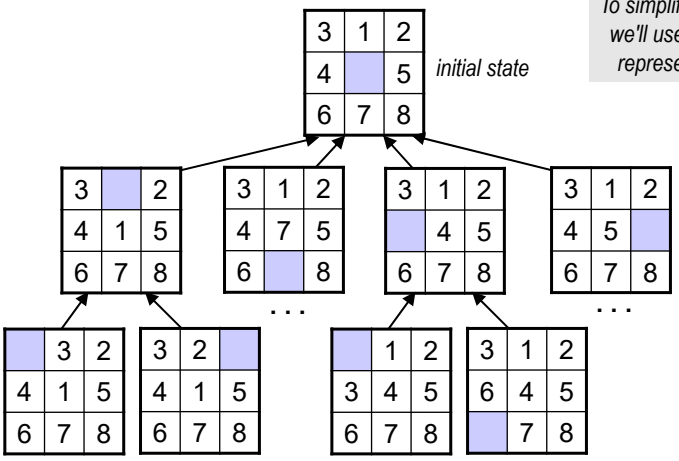
- The predecessor references connect the `State` objects, creating a structure known as a *tree*.



State-Space Search Tree

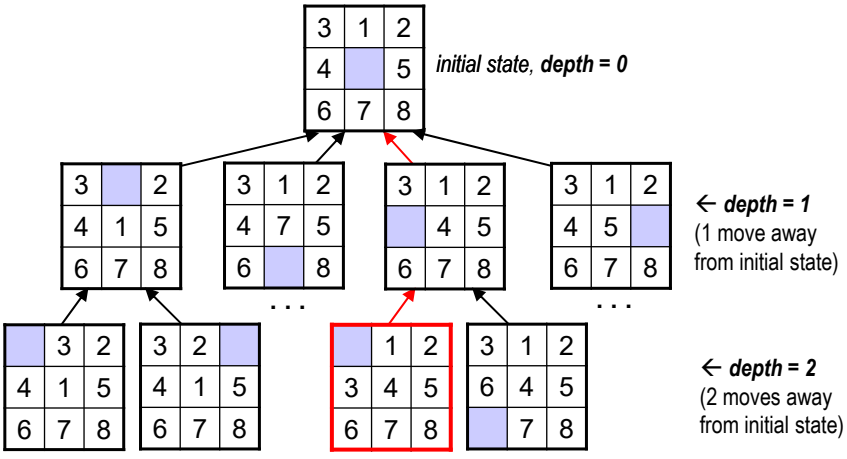
- The predecessor references connect the `State` objects, creating a structure known as a *tree*.

To simplify the diagrams, we'll use the boards to represent the states.



State-Space Search Tree

- The predecessor references connect the `State` objects, creating a structure known as a *tree*.



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

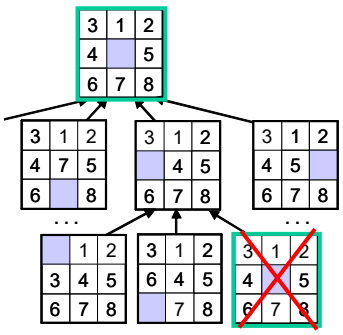
Part III: Initial `Searcher` Class

- To implement each search algorithm, we use a *searcher* object.
  - maintains a list of yet-to-be-tested states
- It also determines the order in which the states are considered.
- Searcher methods include:
  - `add_state(new_state)` - add a single state to the searcher's list of states
  - `add_states(new_states)` – add the states in the list `new_states` to the searcher's list of states
  - `next_state()` – get and return the next state that should be considered (removing it from the searcher's list of states)
    - in Part III – pick a state at random!
  - `find_solution(init_state)` – perform a search starting at `init_state`, and return the goal state when found

Pseudocode for `find_solution()`

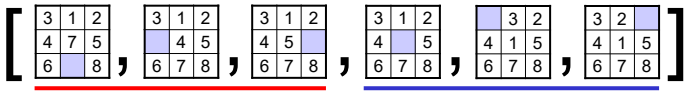
```
def find_solution(self, init_state):
   add init_state to the searcher's list of states
   while the searcher has more states:
       s = self.next_state()
       if s is the goal:
           return s
       else:
           self.add_states(list of successors of s)
   return None   # failure
```

- `add_states()` shouldn't add a state that forms a *cycle* – i.e., one that is already on the current path from the initial state (example at right)
  - we've given you a method that checks for this

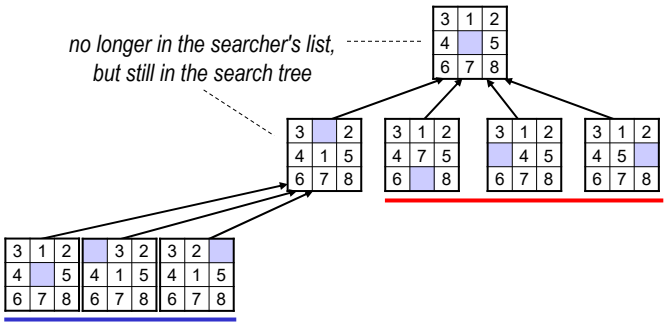


The Searcher's List and the Search Tree

- The searcher maintains a list of yet-to-be-tested states:



- The search tree includes:
  - all states in the searcher's list
  - all predecessors of those states



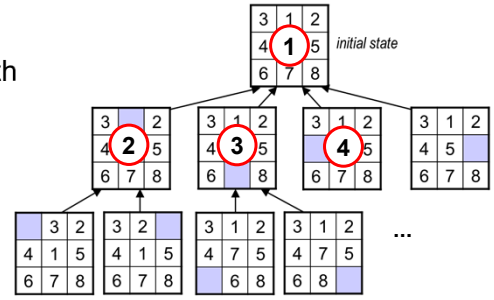
Part IV: Subclasses for Other Search Algorithms

- Each algorithm will have its own type of searcher object.
  - with its own version of at least one of the key methods
  - take advantage of inheritance!

Breadth-First Search (BFS)

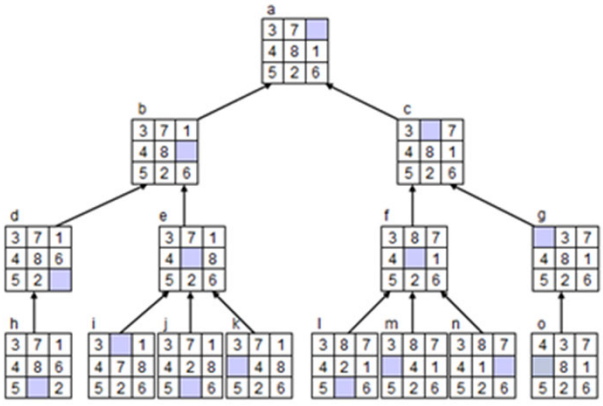
- When choosing from the list of yet-to-be-tested states, choose one of the states with the ***smallest depth***.
- Thus, BFS considers:
  - all states at depth 0
  - all states at depth 1
  - all states at depth 2

    …



- The breadth-first searcher should follow FIFO ("first in, first out").
  - its `next_state()` should remove the state that has been in the list the longest

What are the first 4 states BFS would consider? (break ties alphabetically)



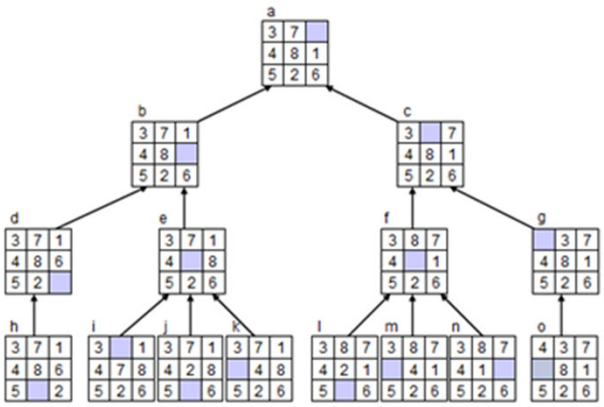
A. a, b, c, d

B. a, b, d, e

C. a, b, d, h

D. none of these

What are the first 4 states BFS would consider? (break ties alphabetically)



A. **a, b, c, d**

B. a, b, d, e

C. a, b, d, h

D. none of these

Features of Breadth-First Search

- It is *complete:* if there is a solution, BFS will find it.
- If each move has the same cost, BFS is *optimal*—it will find a minimal-cost solution.
- Key problems:
  - It can require too much *time.*
    - if the optimal solution is deep in the tree, it can take a long time for BFS to find it
  - It can require too much *memory.*
    - *all* previously tested states must be kept in memory
      - they have successors in the searcher's list of states
      - thus, they may be part of the eventual solution

Schedule Items

- No labs this week.
- Office hours today but not for the rest of this week.
- Final Project:
  - parts I and II due on Thursday, 12/2
  - full project due Thursday, 12/9
- Also happening after break:
  - a bit of CS theory
  - PS 10 (50 points) due on Sunday, 12/5
- ***Happy Thanksgiving!***