Blind Search

State Space Problem Definition

- One widely used way to describe problems is by listing or describing all possible states.
- Solving a problem means moving through the state space from a start state to a goal state.
- We need to devise a set of operators that move from one state to another.

State Space Problem Definition

- Set of possible states.
- Set of possible operations that change the state.
- Specification of a starting state(s).
- Specification of a goal state(s).

State Space

- It is usually not possible to *list* all possible states:
  - use abstractions to describe legal states.
  - It may be easier to describe the illegal states.
  - Sometimes it is useful to provide a general description of the state space and a set of constraints.

Operations

- The problem solving system moves from one state to another according to well defined operations.
- Typically these operations are described as *rules*.
- A control system decides which rules are applicable at any state, and resolves conflicts and/or ambiguities.

Example: Water Jug Problem

Unlimited Water

Goal: 2 Gallons in the 4 Gal. Jug

Water Jug State Space

- The state can be represented by 2 integers, x and y:
- x = gallons in the 4 gallon jug
- y = gallons in the 3 gallon jug

State Space = (x,y)

such that $x \in \{0,1,2,3,4\}$, $y \in \{0,1,2,3\}$

Water Jug Start and Goal States

- The start state is when both jugs are empty:

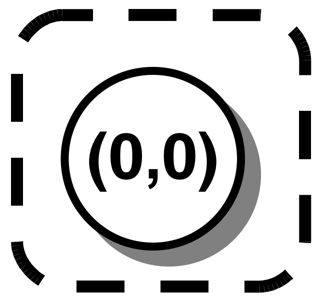
  (0,0)

- The goal state is any state that has 2 gallons in the 4 gallon jug:

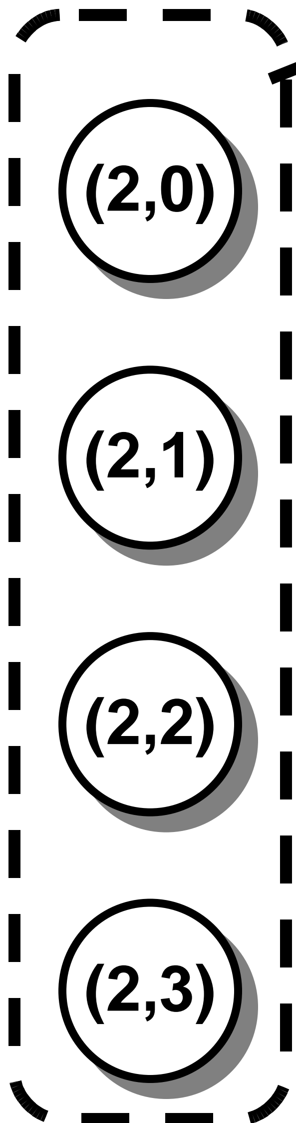
  (2,n) for any n

Water Jug States

Start State



Goal States



(1,0) (3,0) (4,0)

(0,1) (1,1) (3,1) (4,1)

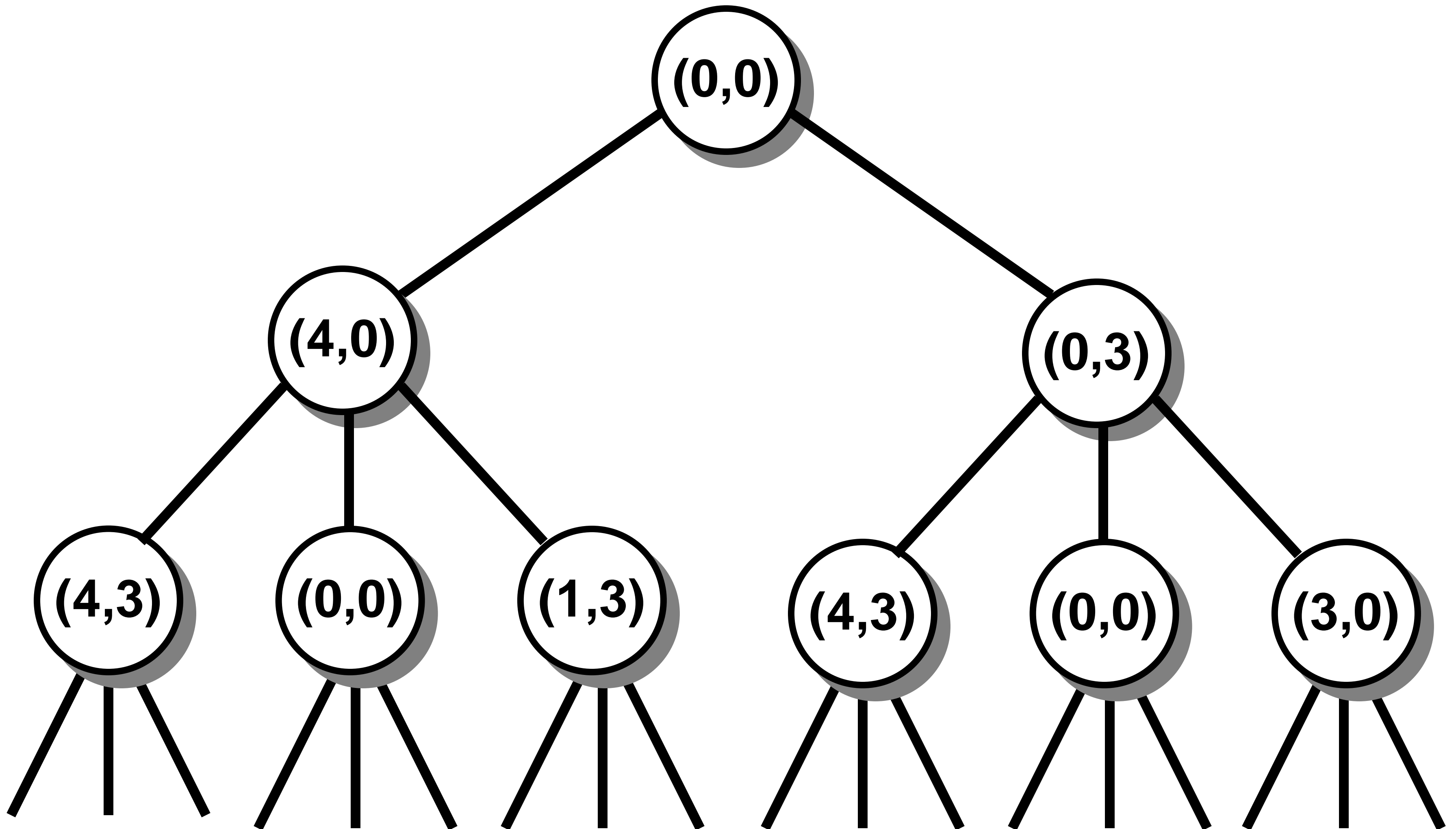
(0,2) (1,2) (3,2) (4,2)

(0,3) (1,3) (3,3) (4,3)

Water Jug Operations

- Fill 3 gal. jug from pump (x,y) -> (x,3)
- Fill 4 gal. jug from pump (x,y) -> (4,y)
- Empty 3 gal. jug (x,y) -> (x,0)
- Empty 4 gal. jug (x,y) -> (0,y)
- Pour contents of 3 gal. jug into 4 gal. Jug (x,y) -> (x+y,0) or (4,x+y-4)
- ...

Water Jug Tree (partial)



Search Trees

- The search for a solution can be described by a tree - each node represents one state.
- The path from a parent node to a child node represents an operation.
- Search Trees provide a convenient description of the search space, they are not a data structure stored in memory!!!

Search Trees

- Each node is a search tree has child nodes that represent each of the states reachable by the parent state.
- Another way of looking at it: child nodes represent all available options once the search procedure reaches the parent node.
- There are a number of strategies for traversing a search tree.

Breadth-First-Search

- Breadth-First Search visits all nodes at depth n before visiting any nodes at depth $n+1$.
- General Algorithm for BFS:

create *nodelist* (a queue) and initialize to the start state.

repeat until a goal state is found or *nodelist* = {}

remove the first element from nodelist:

apply all possible rules and add resulting states to nodelist.

Breadth-First Search Characteristics

- If there is a solution, BFS will find it.
- BFS will find the minimal solution (shortest path length to the solution).
- BFS will not get caught in state space cycles.
- Requires space available to store the nodelist queue. This can be very large!!!

BFS Time and Space Analysis

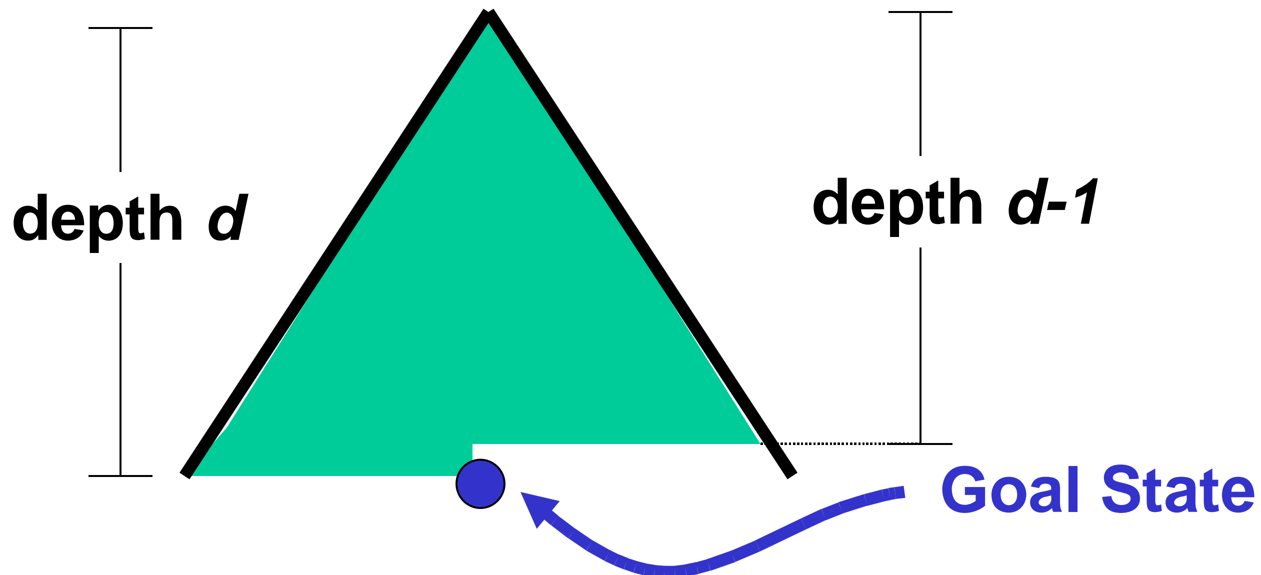
- Assumptions:
  - There is a single solution (goal state) in the tree.
  - The search tree is “regular” and has branching factor b.
  - The goal state is at depth d in the tree.
  - The goal state is in the middle of the tree (average case analysis).

BFS Space Requirements

- The queue initially has 1 state.
- After the first step - the queue will contain b states.
- Processing each of the b states at level 1 results in adding b more states to the queue.
- After processing all states at depth n, the queue will hold b^{n-1} states.

BFS Space Requirements

- Since we assume the goal state is in the middle of level *d*:
  - the queue will hold $b^{d-1}/2$ states.



BFS Time Analysis

- Measure time in terms of the number of states visited.
  - Assume that we spend the same time processing each node (state).

Time = # level 1 nodes + #level 2 nodes + …
+ # level *d-1* nodes + (# level *d* nodes/2).

$$= 1 + b + b^2 + b^3 + \ldots + b^{d-1} + b^d/2$$

$$O(b^d)$$

Depth-First Search

Depth-First processes all children (choices) of a node before considering any siblings (at the same depth).

Depth-First Search is similar to BFS, but instead of a queue we use a stack.

Depth-First Search can easily be described recursively.

DFS recursive definition

While applicable rules exist:

- apply the next rule and generate a new state.
- If new state is the goal - all done,
- Otherwise - do a depth-first search on the new state.

DFS

- General Algorithm for DFS:
  create *nodelist* (a stack) and initialize to the start state.
  repeat until a goal state is found or *nodelist* = {}
    pop the first element from nodelist:
      apply all possible rules and add (push) resulting states to nodelist.

DFS Space Analysis

- After the first step the stack will contain b nodes.
- After the second step the stack will contain $(b - 1) + b$ nodes.
- After the 3rd step the stack will contain $(b-1) + (b-1) + b$ nodes.
- After the first d steps the stack will hold $(b-1)*d + 1$ nodes (this is the maximum).

DFS Time Analysis

- In the best case - the goal state will be the first state examined at depth d - this will require looking at $d+1$ nodes.
- In the worst case - the goal state will be the last state examined at depth d - this will require looking at all the nodes:

$$1+b+b^2+b^3+...+b^d = (b^{d+1}-1) / (b-1)$$

Depth-First Search Characteristics

- Don’t need to keep track of a large list of states.
- May find a solution very fast (and might not).
- “Pruning” is possible
  - example: branch-and-bound
- Can easily get caught in loops.

Water Jug Problem Revisited

Rules

```
(x,y), y<3        ->   (x,3)
(x,y), x<4        ->   (4,y)
(x,y), y≠0        ->   (x,0)
(x,y), x≠0        ->   (0,y)
(0,y), y≠0        ->   (y,0)
(x,y), x+y≤4      ->   (x+y,0)
(x,y), x+y>4      ->   (4,x+y-4)
(x,y), x+y≤3      ->   (0,x+y)
(x,y), x+y>3      ->   (x+y-3,3)
```

Exercises:

1. Draw the BFS search
2. Draw the DFS search

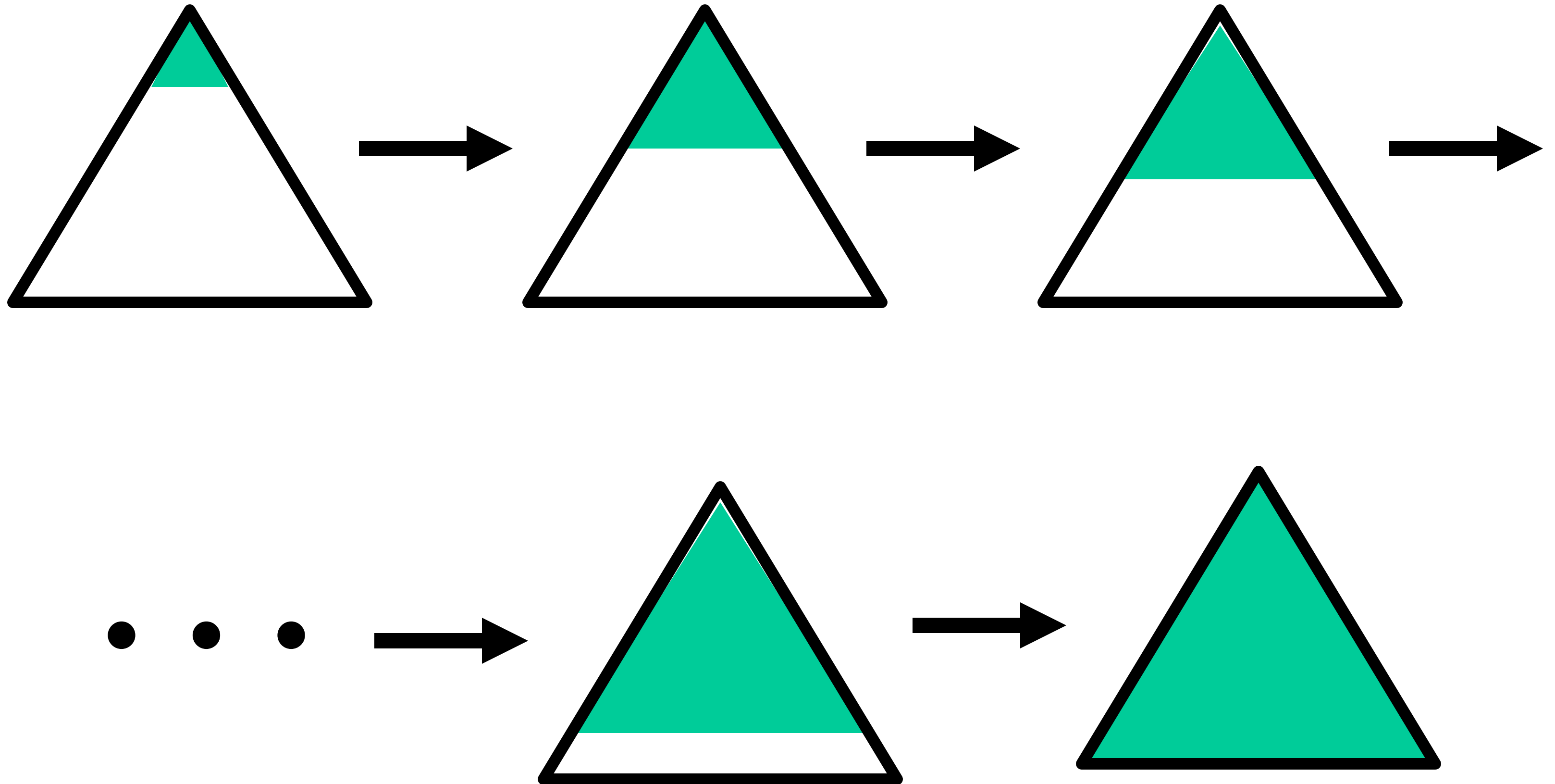
Iterated Deepening

- Attempt to combine BFS and DFS to get a search with the following desirable characteristics:
  - Find the minimal solution (shortest path length to the solution).
  - Minimal (reasonable) space requirements
  - *Similar* time requirements.

Iterated Deepening

- Iterated Deepening uses a depth cutoff that starts at depth 1.
- For each depth cutoff *1..d* do:
  - apply DFS stopping at the cutoff depth.
  - If goal state is not found increase the depth cutoff.

Iterated Deepening Steps



Iterated Deepening Analysis

- Maximum space requirements is the same as DFS: $O(bd)$
- Worst case/Average case time analysis:

$$O(1 + b + b^2 + b^3 + ... + b^d)$$

$$O(b^d)$$

Iterated Broadening

- Limit on the breadth of a search at each node (number of child nodes processed).
- Gradually increase the breadth limit until a goal is found.

- Iterated broadening is most useful when there are many goals in the search tree.

Iterated Broadening Time Analysis

- At most we must run *b* searches (each with different breadth cutoff).
- First search includes 1 node.
- Second search looks at 2^d
- Third looks at 3^d
- …

$$1+2^d+3^d+\ldots+b^d \Rightarrow O(b^d)$$

Loops

- DFS (and iterated deepening) can suffer if there are loops in the state space.
- In problems that can generate loops we need to treat the search space as a graph.
- There are a number of modifications that are possible

Searching Graphs

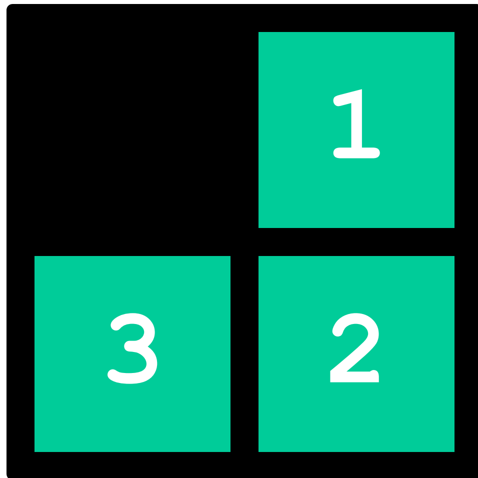
- The general idea is to keep track of all nodes that have already been searched.
- Each time a new node is put on the stack/queue we first make sure it has not already been processed.
- The list of nodes already processed is called the “closed list” and the stack/queue is called the “open list”.

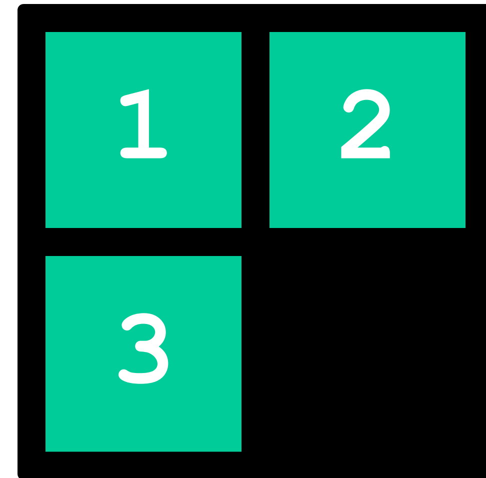
Problems with using Closed List

- The closed list can (and will) become too large.
- It takes time O(|*Closed List*|) to make sure each new node has not already been processed.
- Possible solutions:
  - Use hash tables
  - Limit the size of the Closed List?

Example Problem - 3 Puzzle

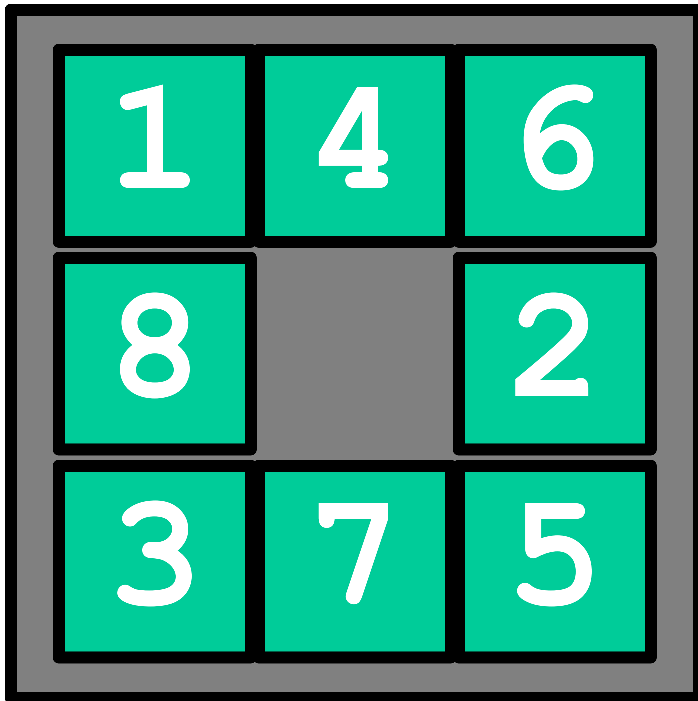
- Sliding Tile game with 3 tiles:

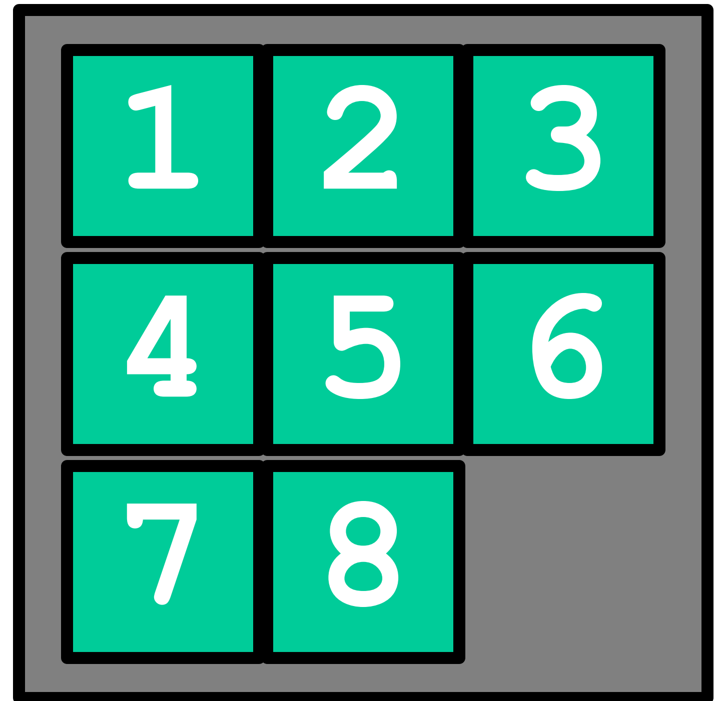




start state **goal state**

8 Puzzle





Missionaries and Cannibals

- 3 missionaries
- 3 cannibals
- 1 boat that holds (at most) 2 people.
- Want to get everyone across a river
- If cannibals outnumber the missionaries on either side of the river - missionaries