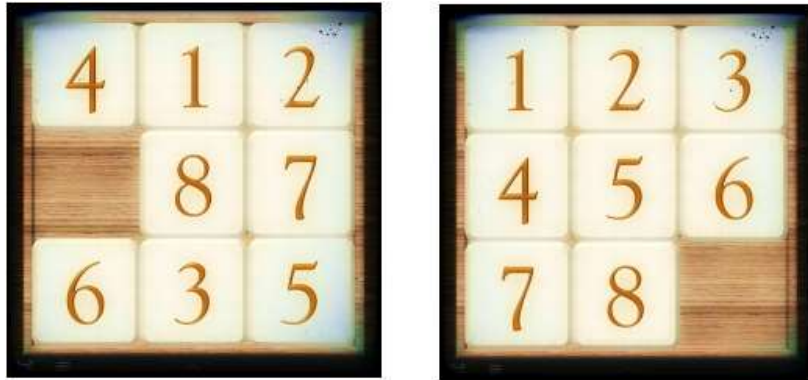


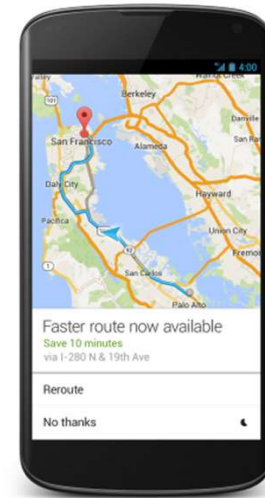
# Artificial Intelligence: State Space Search

- Many slides from:  
[robotics.stanford.edu/~latombe/cs121/2003/home.htm](http://robotics.stanford.edu/~latombe/cs121/2003/home.htm)

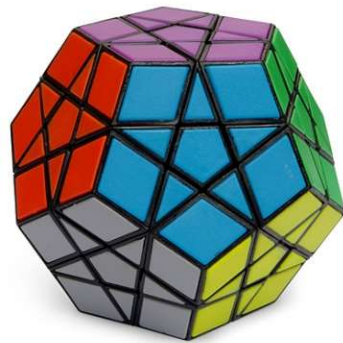
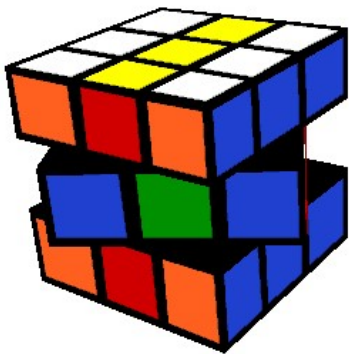
# Motivation



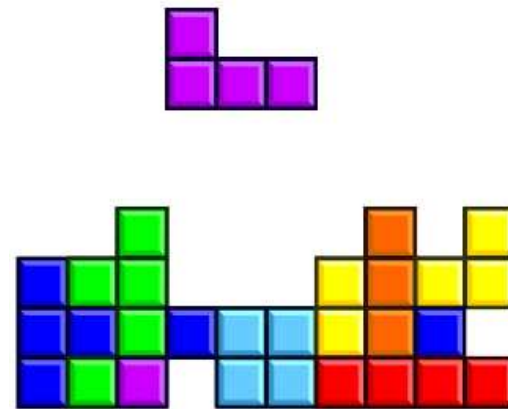
8-puzzle



Google itinerary

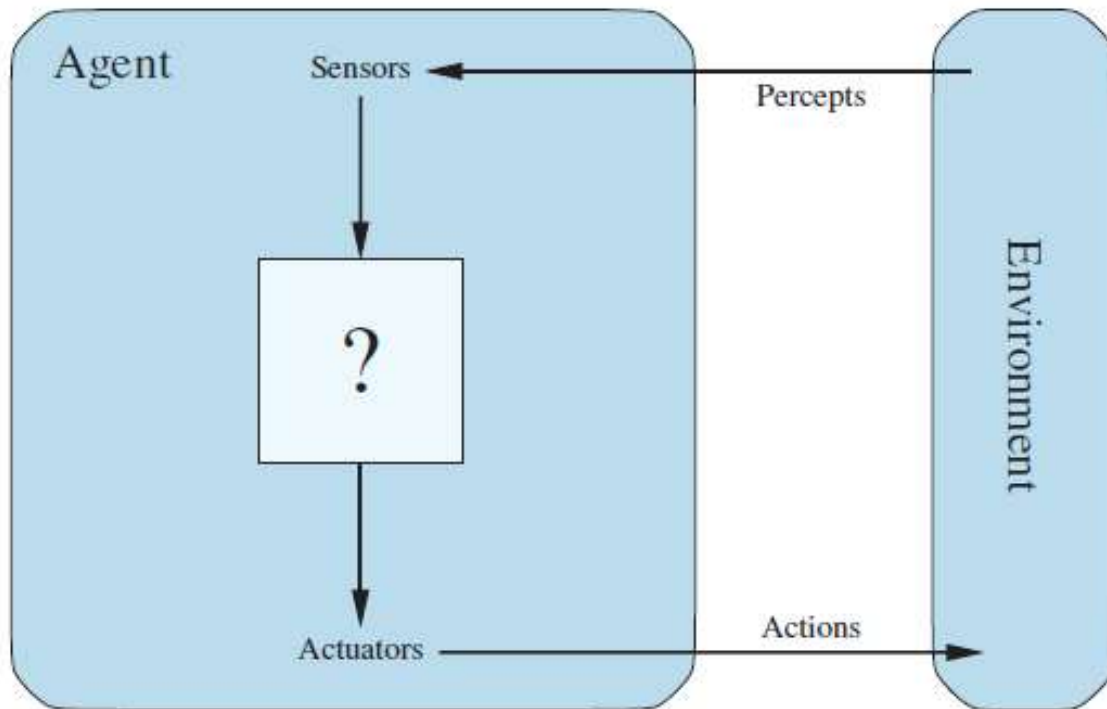


Rubik's cube



Tetris

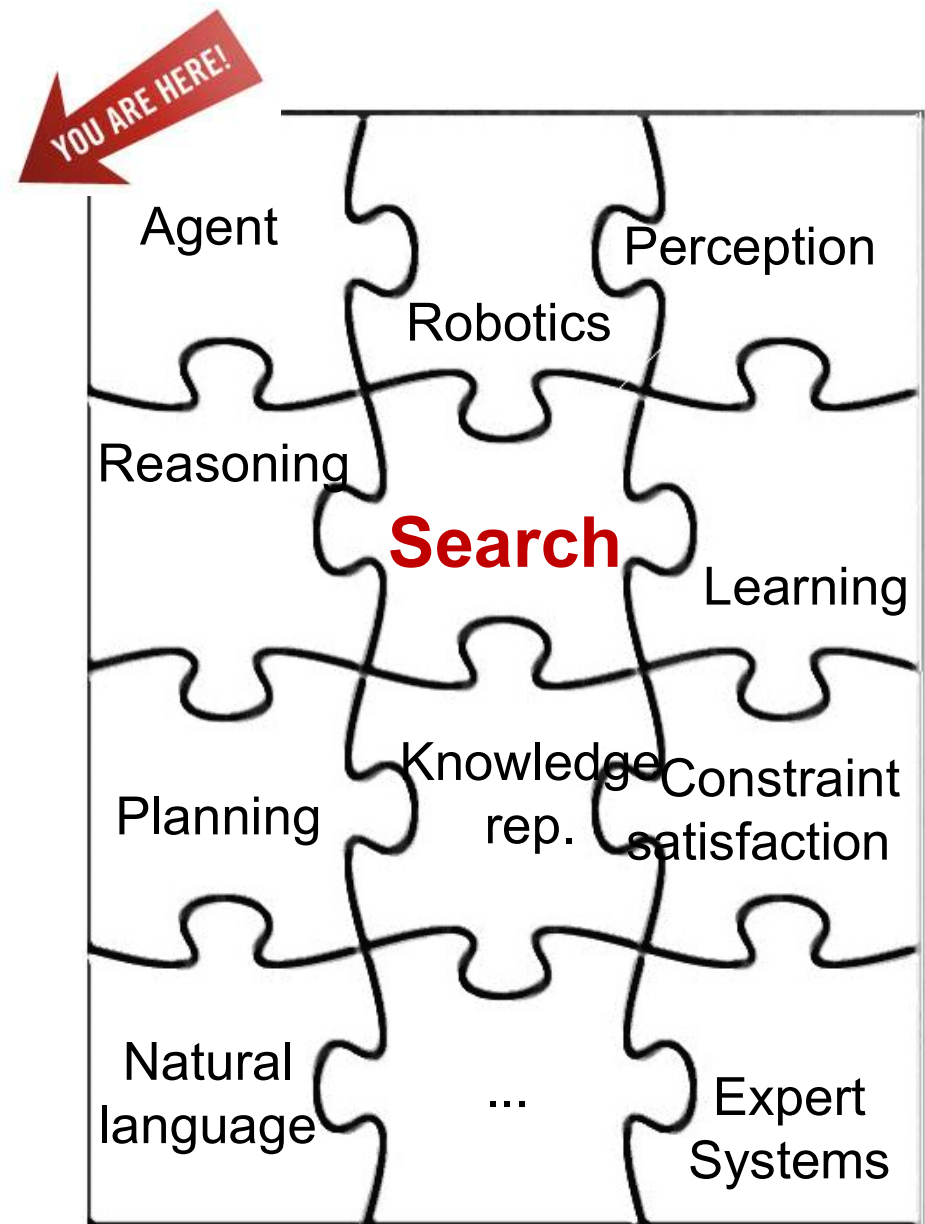
# Agent



- A rational agent acts in order to reach its predefined goal or to maximize its utility (goals ~ utility)
- The **Environment** for a chess player agent is the (virtual) chess board, and actions are moves
- Agent and Environment make the **World**

# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Uniform Cost
  - Informed search
    - Hill climbing
    - Best-First
    - (Designing Heuristics)
    - A\*
- Summary



# Example: 8-Puzzle

**State:** Any arrangement of 8 numbered tiles and an empty tile on a 3x3 board

8	2	
3	4	7
5	1	6

Initial state

1	2	3
4	5	6
7	8	

Goal state



there are several standard goals states for the 8-puzzle

1	2	3
4	5	6
7	8	

1	2	3
8		4
7	6	5

...

# $(n^2-1)$ -puzzle

8	2	
3	4	7
5	1	6

8-puzzle

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

15-puzzle

■ ■ ■ ■

# 15-Puzzle

Invented in 1874 by Noyes Palmer Chapman  
... but Sam Loyd claimed he invented it!



**SAM LOYD,**

Journalist and Advertising Expert,

ORIGINAL

Games, Novelties, Supplements, Souvenirs,  
Etc., for Newspapers.

Unique Sketches, Novelties, Puzzles, &c.,  
FOR ADVERTISING PURPOSES.

Author of the famous

"Get Off The Earth Mystery," "Trick Donkeys,"

"Big Black Puzzle," "Pigs In Clover,"

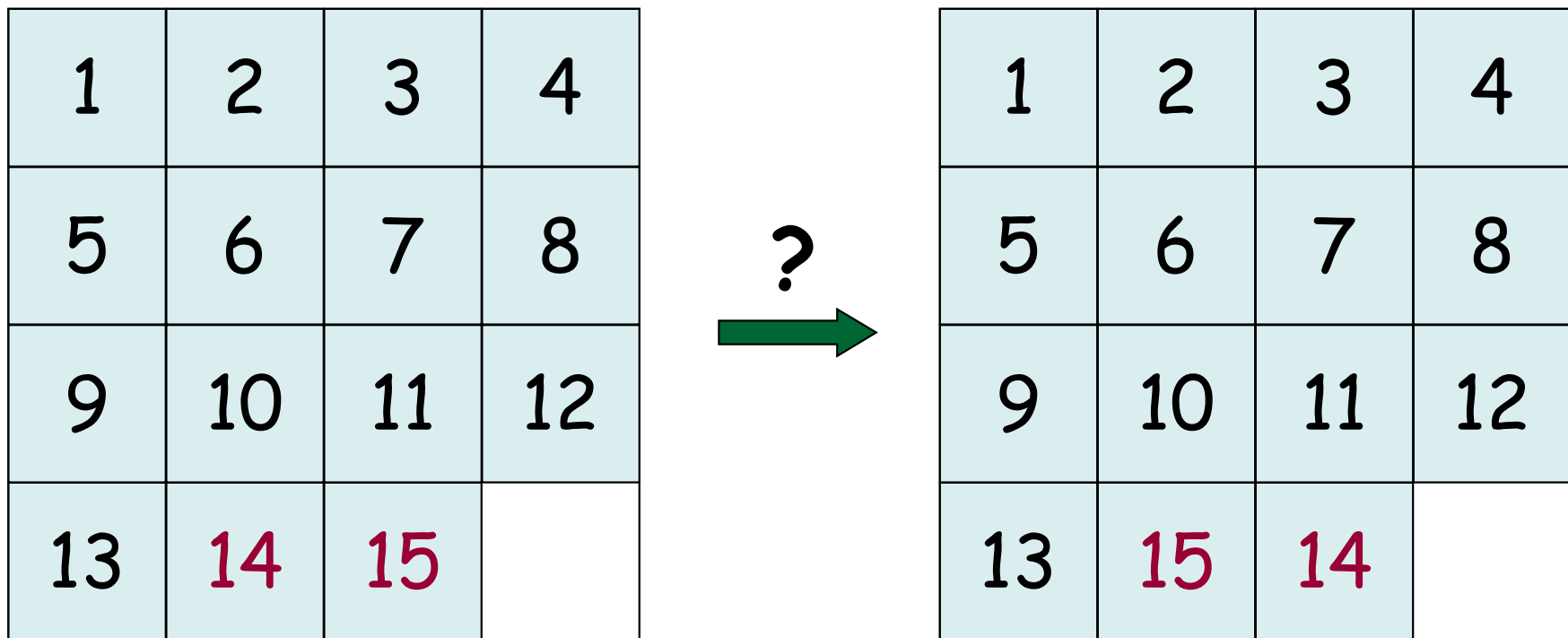
"Paradees," Etc., Etc.,

P. O. BOX 876.

New York, *April 15* 1903

# 15-Puzzle

Sam Loyd even offered \$1,000 of his own money to the first person who would solve the following problem:





But no one ever won the prize...



# State Space

- Many AI problems, can be expressed in terms of going from an **initial state** to a **goal state**
  - *Ex: to solve a puzzle, to drive from home to Concordia...*
- Often, there is no direct way to find a solution to a problem
- but we can list the possibilities and search through them
  1. Brute force search:
    - generate and search all possibilities (but inefficient)
  2. Heuristic search:
    - only try the possibilities that you *think* (based on your current best guess) are more likely to lead to good solutions

# State Space

- Problem is represented by:
  1. Initial State
    - starting state
    - ex. unsolved puzzle, being at home
  2. Set of operators
    - actions responsible for transition between states
  3. Goal test function
    - Applied to a state to determine if it is a goal state
    - ex. solved puzzle, being at Concordia
  4. Path cost function
    - Assigns a cost to a path to tell if a path is preferable to another
- Search space: the set of all states that can be reached from the initial state by any sequence of action
- Search algorithm: how the search space is visited

# Example: The 8-puzzle

8	2	
3	4	7
5	1	6

Initial state

1	2	3
4	5	6
7	8	

Goal state

Set of operators:

blank moves up, blank moves down, blank moves left, blank moves right

Goal test function:

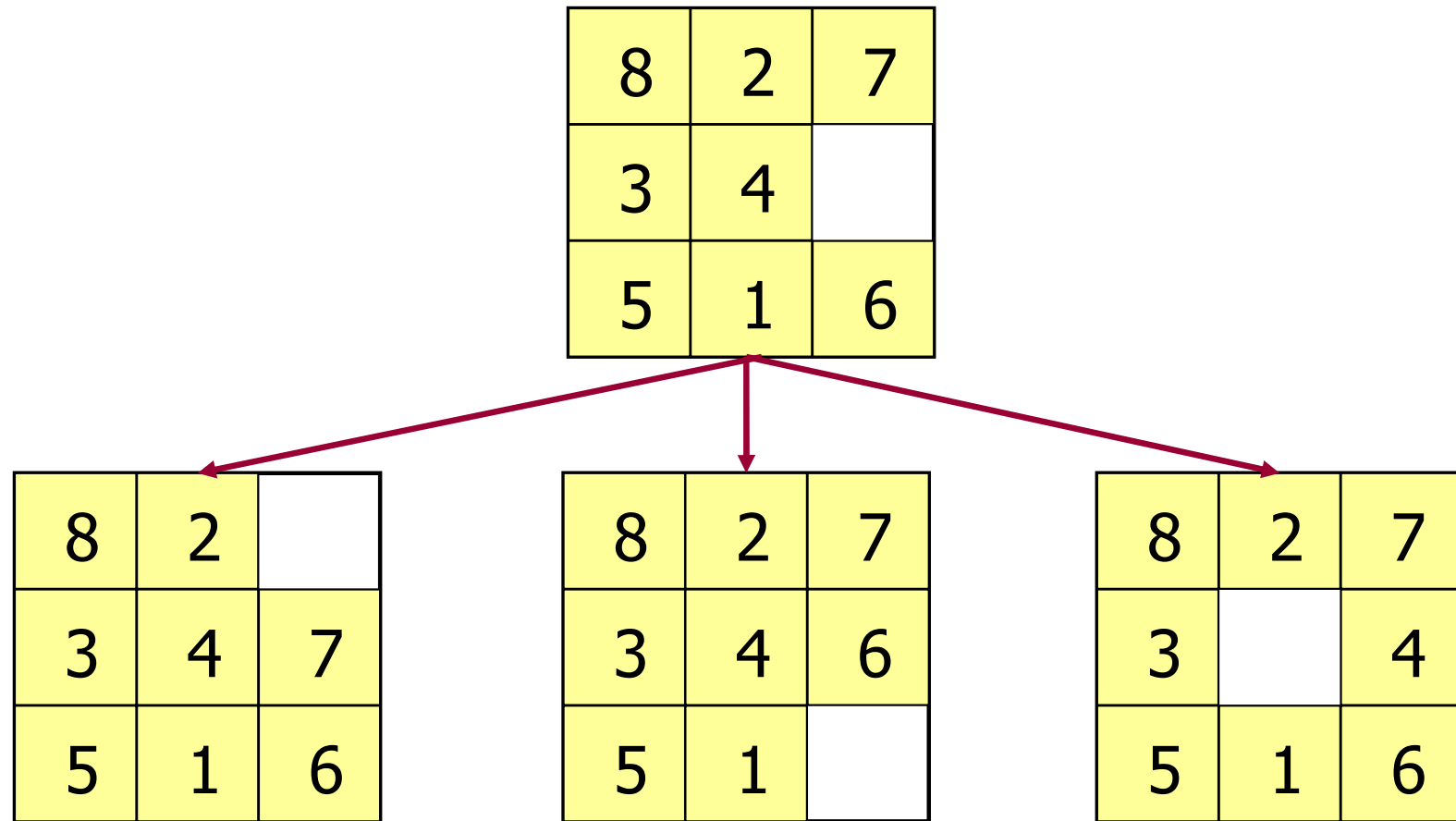
state matches the goal state

Path cost function:

each movement costs 1

so the path cost is the length of the path (the number of moves)

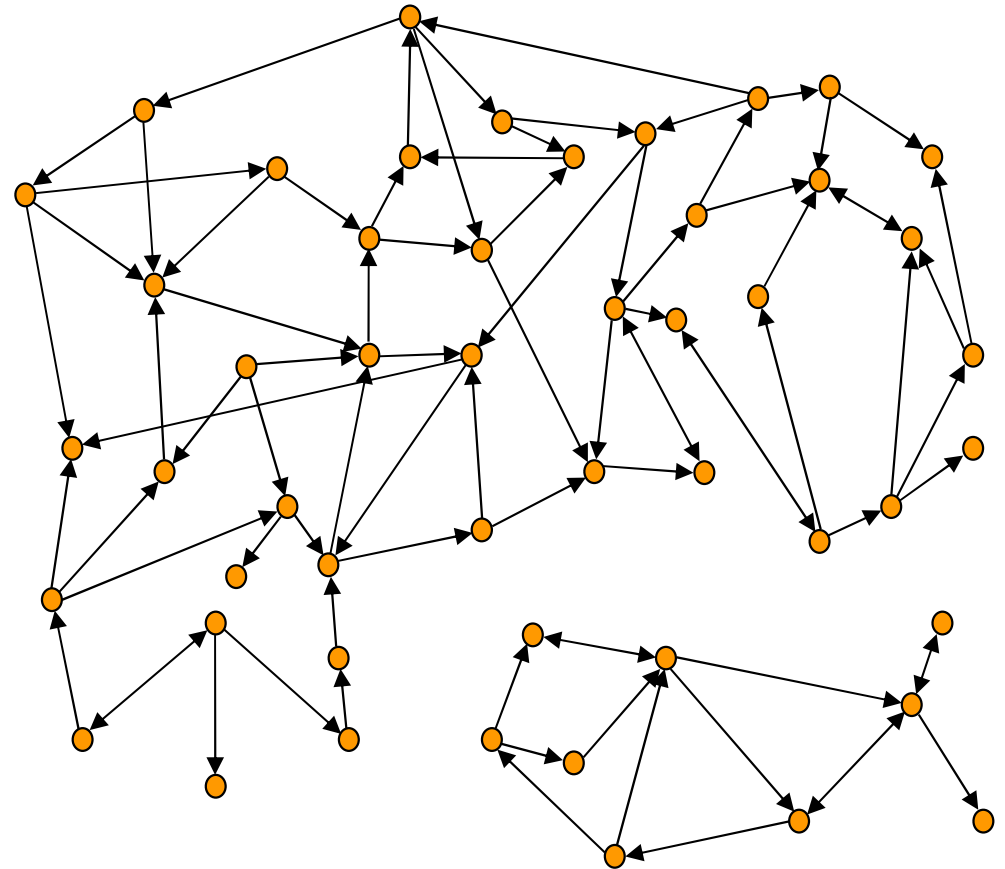
# 8-Puzzle: Successor Function



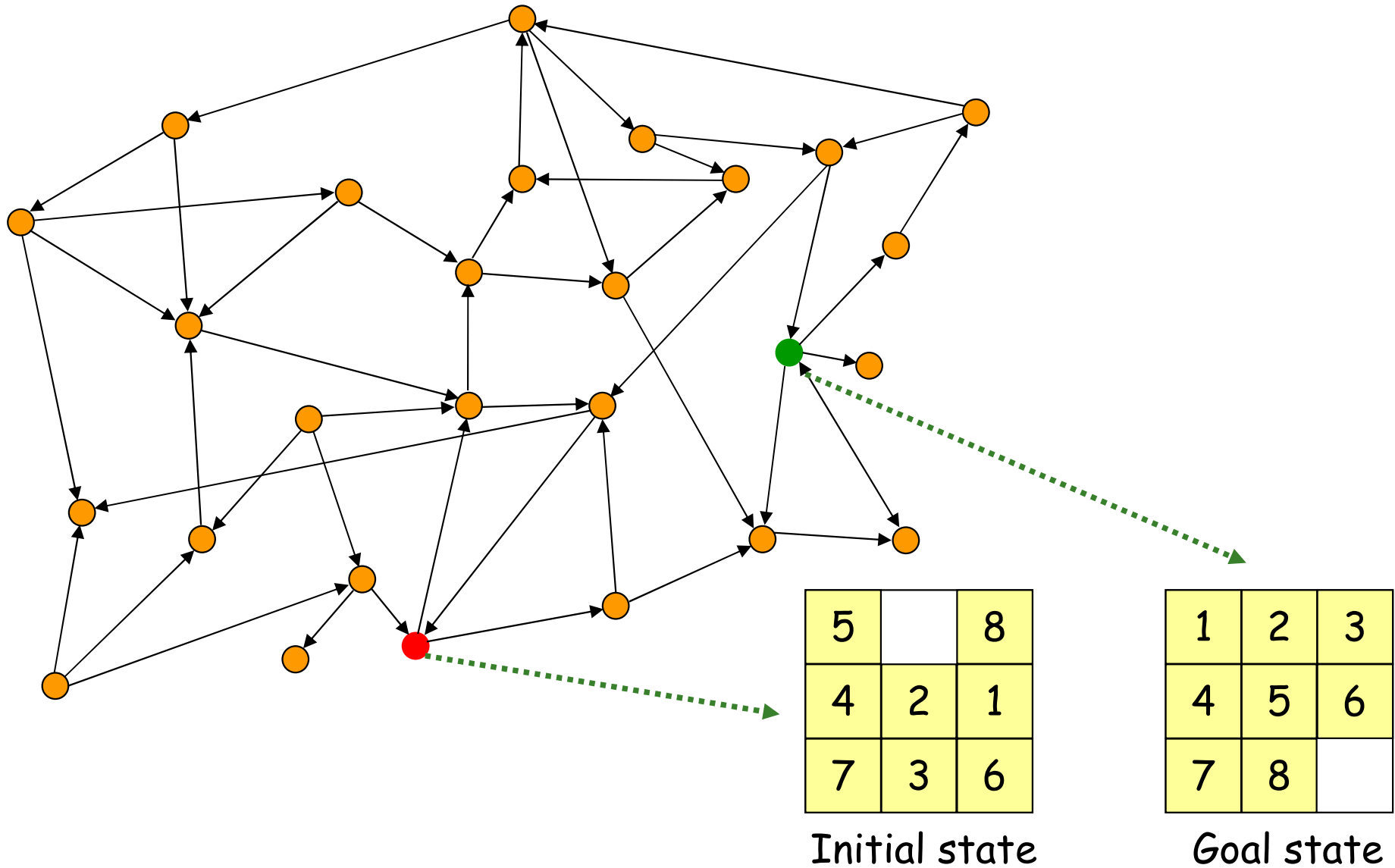
Search is about the exploration of alternatives

# State Graph

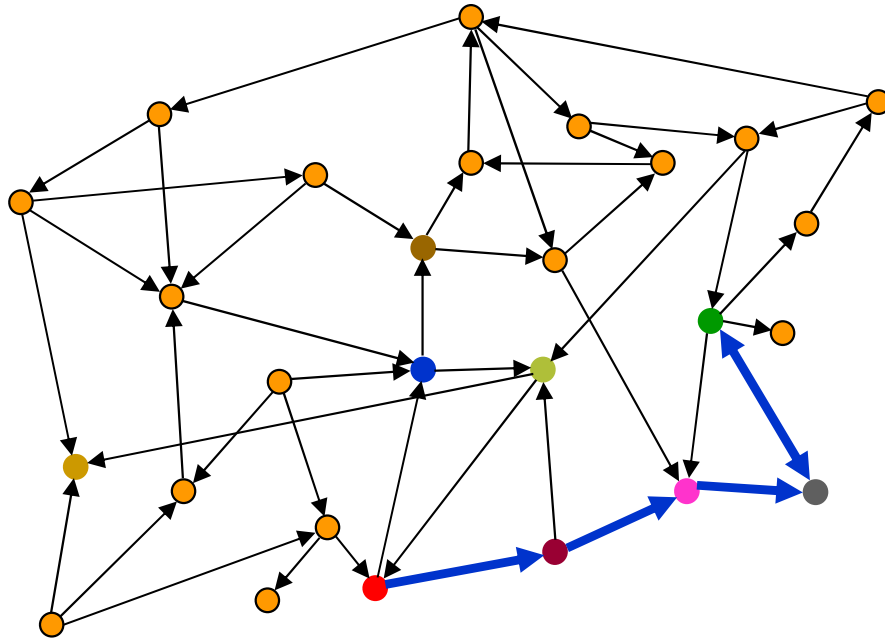
- Each state is represented by a distinct node
- An arc (or edge) connects a node  $s$  to a node  $s'$  if  $s' \in \text{SUCCESSOR}(s)$
- The state graph may contain more than one connected component



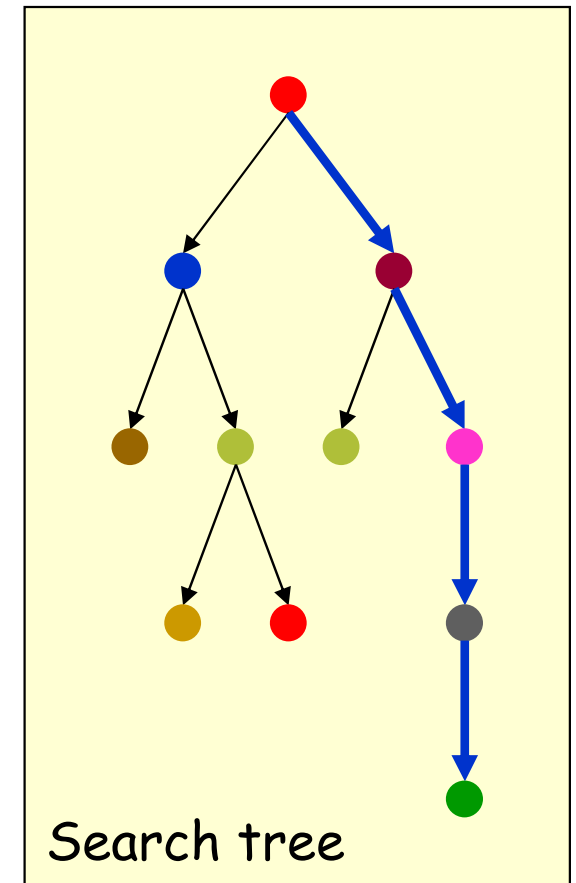
# Just to make sure we're clear...



# State Space as a Search Tree

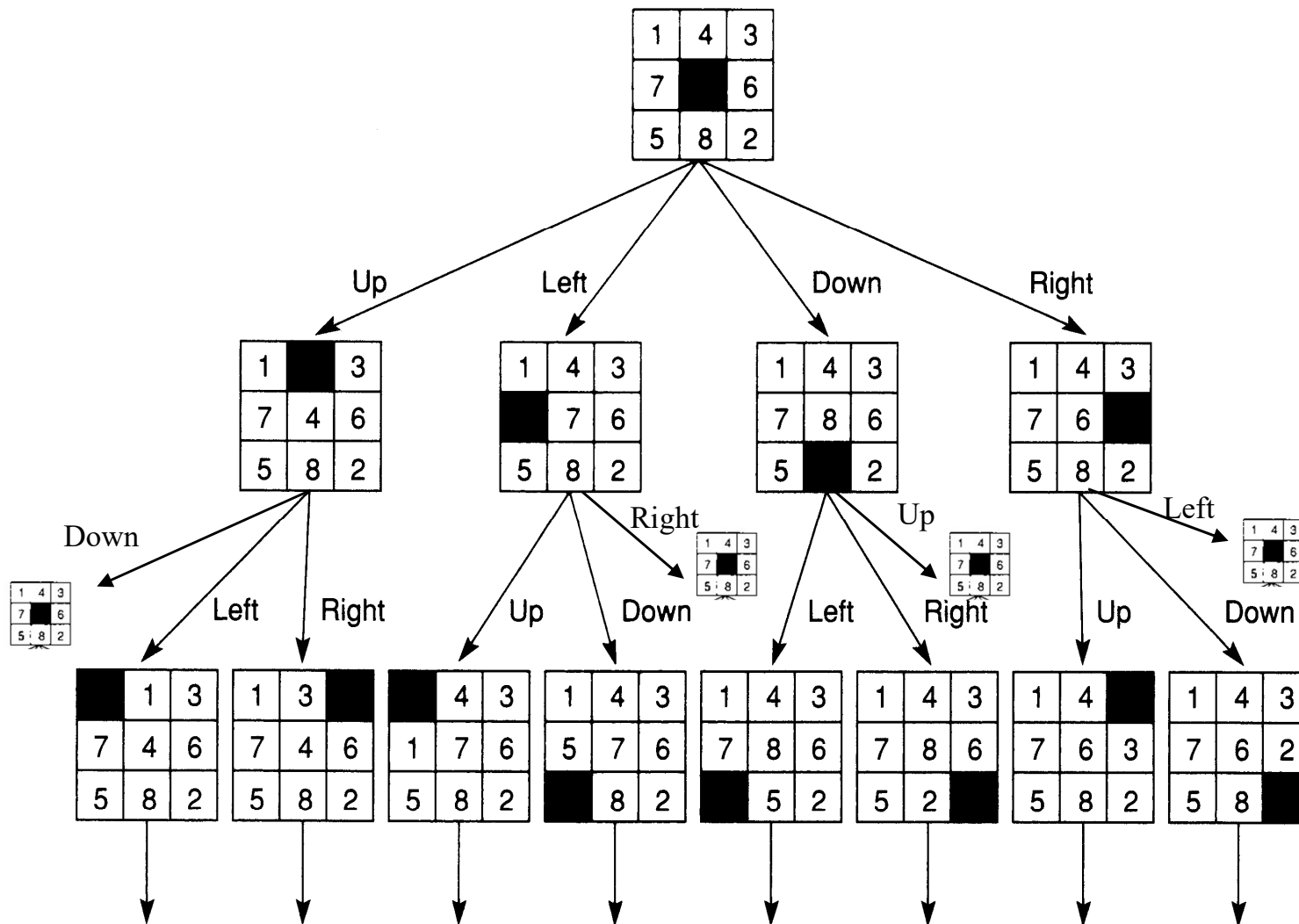


- In graph representation, cycles can prevent termination
  - Blind search without cycle check may never terminate
- Use a tree representation, and check for cycles





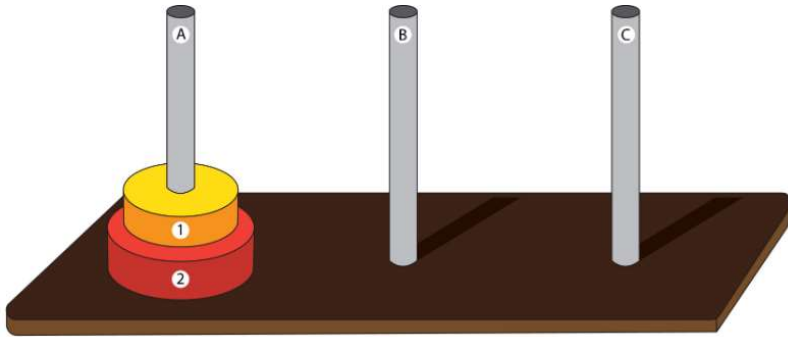
# State Space for the 8-puzzle



# How large is the state space of the $(n^2-1)$ -puzzle?

- Nb of states:
  - 8-puzzle -->  $9! = 362,880$  states
  - 15-puzzle -->  $16! \sim 2.09 \times 10^{13}$  states
  - 24-puzzle -->  $25! \sim 10^{25}$  states
- At 100 millions states/sec:
  - 8-puzzle --> 0.036 sec
  - 15-puzzle -->  $\sim 55$  hours
  - 24-puzzle -->  $> 10^9$  years

# Another example: Tower of Hanoi



- For a 2-disk Tower:
  - Nb of states?
  - Goal state?
  - Illegal states?
  - Actions?
  - State representation? (programming)

# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Uniform Cost
  - Informed search
    - Hill climbing
    - Best-First
    - (Designing Heuristics)
    - A\*
- Summary



# Uninformed VS Informed Search

## ■ Uninformed search

- We systematically explore the alternatives
- aka: systematic/exhaustive/blind/brute force search
  - Breadth-first
  - Depth-first
  - Uniform-cost
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search
  - ...

## ■ Informed search (heuristic search)

- We try to choose smartly
  - Hill climbing
  - Best-First
  - $A^*$
  - ...

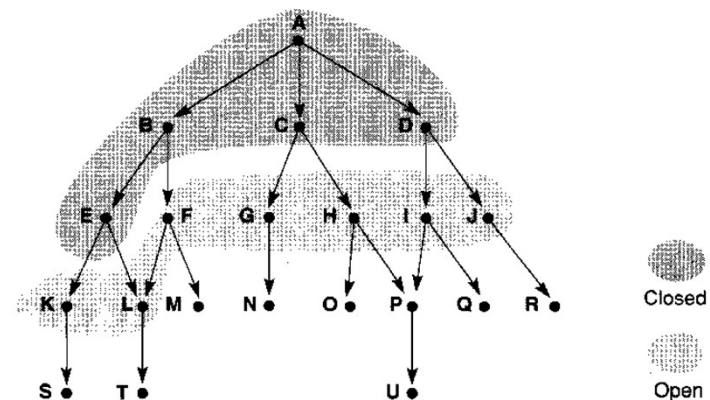
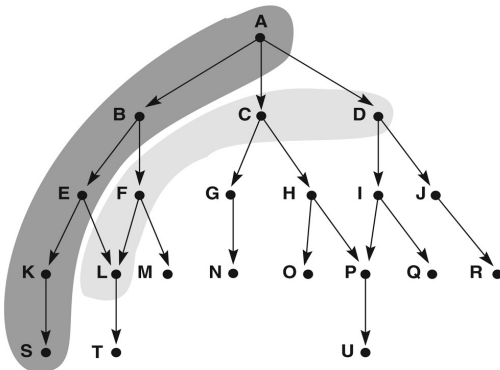
# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Uniform Cost
  - Informed search
    - Hill climbing
    - Best-First
    - (Designing Heuristics)
    - A\*
- Summary



# Breadth-first vs Depth-first Search

- Determine order for examining states
  - Depth-first:
    - visit successors before siblings
  - Breadth-first:
    - visit siblings before successors
    - ie. visit level-by-level

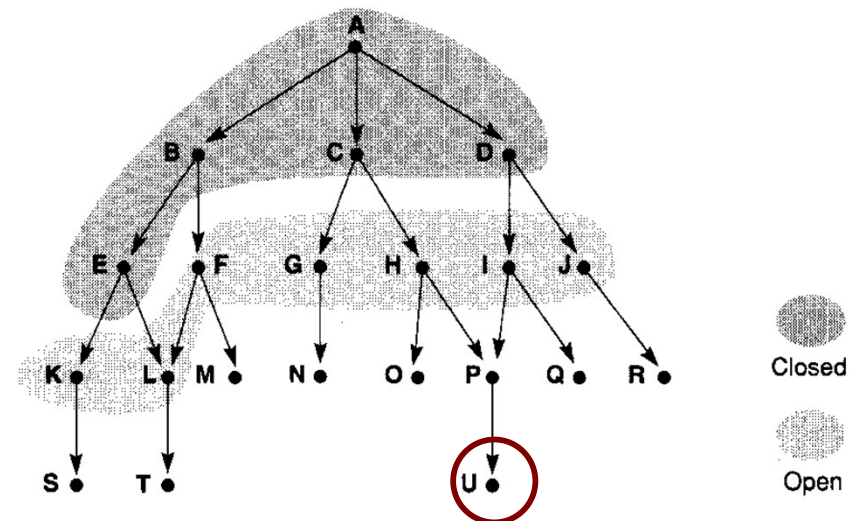


# Data Structures

- In all search strategies, you need:
  - **open list** (aka the **frontier**)
    - lists generated nodes not yet expanded
    - order of nodes controls order of search
  - **closed list** (aka the **explored set**)
    - stores all the nodes that have already been visited (to avoid cycles).
- ex:

Closed = [A, B, C, D, E]

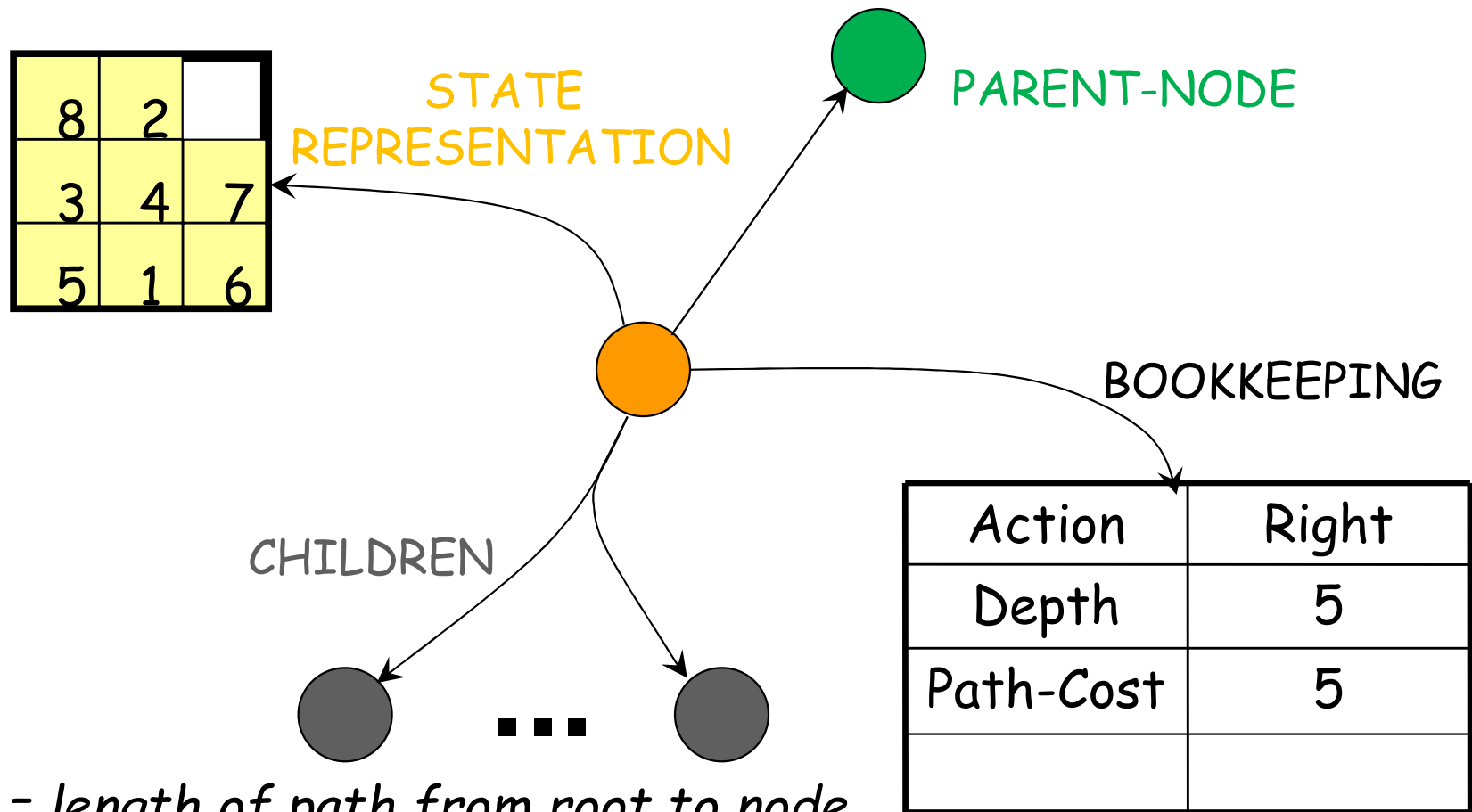
Open = [F, G, H, I, J, K, L]





# Data Structures

- To trace back the entire path of the solution after the search, each node in the lists contain:



*Depth = length of path from root to node*

# Generic Search Algorithm

1. Initialize the **open list** with the initial node  $s_0$  (top node)
2. Initialize the **closed list** to **empty**
3. Repeat
  - a) If the **open list** is **empty**, then **exit** with failure.
  - b) Else, take the first node  $s$  from the **open list**.
  - c) If  $s$  is a **goal state**, **exit** with success. Extract the solution path from  $s$  to  $s_0$
  - d) Else, insert  $s$  in the **closed list** ( $s$  has been visited /expanded)
  - e) Insert the **successors of  $s$**  in the **open list** in a **certain order** if **they are not** already in the closed and/or open lists (to avoid cycles)

## Notes:

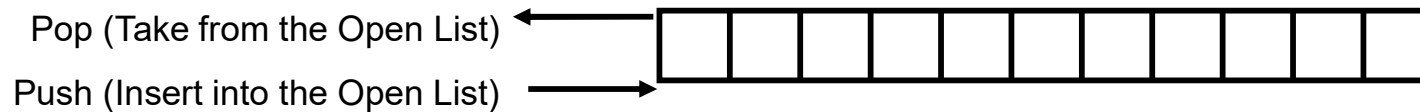
- The **order** of the nodes in the open list depends on the search strategy

# DFS and BFS

- DFS and BFS differ only in the way they order nodes in the open list:

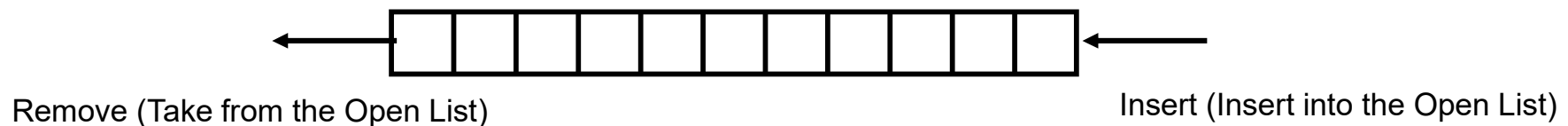
- DFS uses a **stack**:

- nodes are added on the top of the list.



- BFS uses a **queue**:

- nodes are added at the end of the list.



# Breadth-First Search

```
begin
  open := [Start];                                % initialize
  closed := [];
  while open ≠ [] do                               % states remain
    begin
      remove leftmost state from open, call it X;
      if X is a goal then return SUCCESS           % goal found
      else begin
        generate children of X;
        put X on closed;
        discard children of X if already on open or closed;
        put remaining children on right end of open
      end
    end
  end
  return FAIL
end.
```

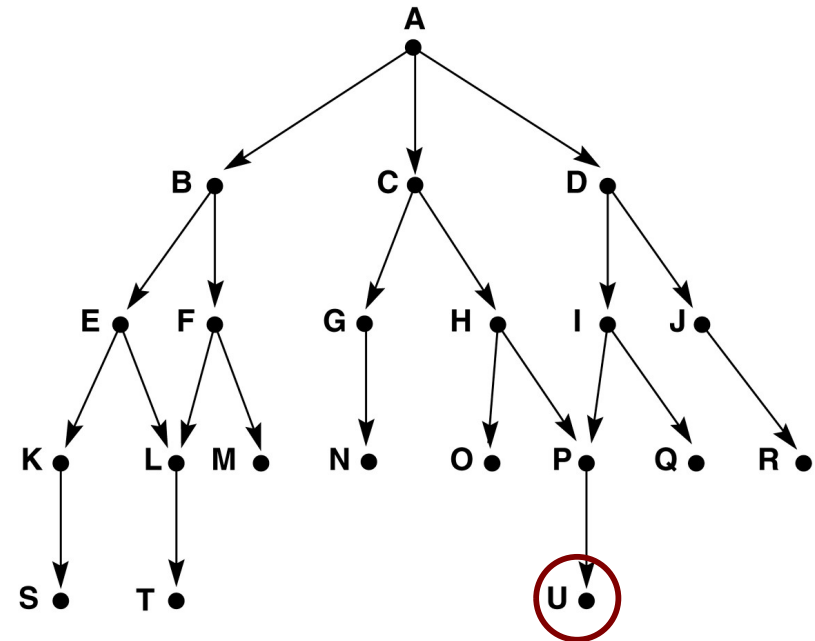
% loop check  
% queue  
% no states left

# Breadth-First Search Example

## ■ BFS: (open is a queue)

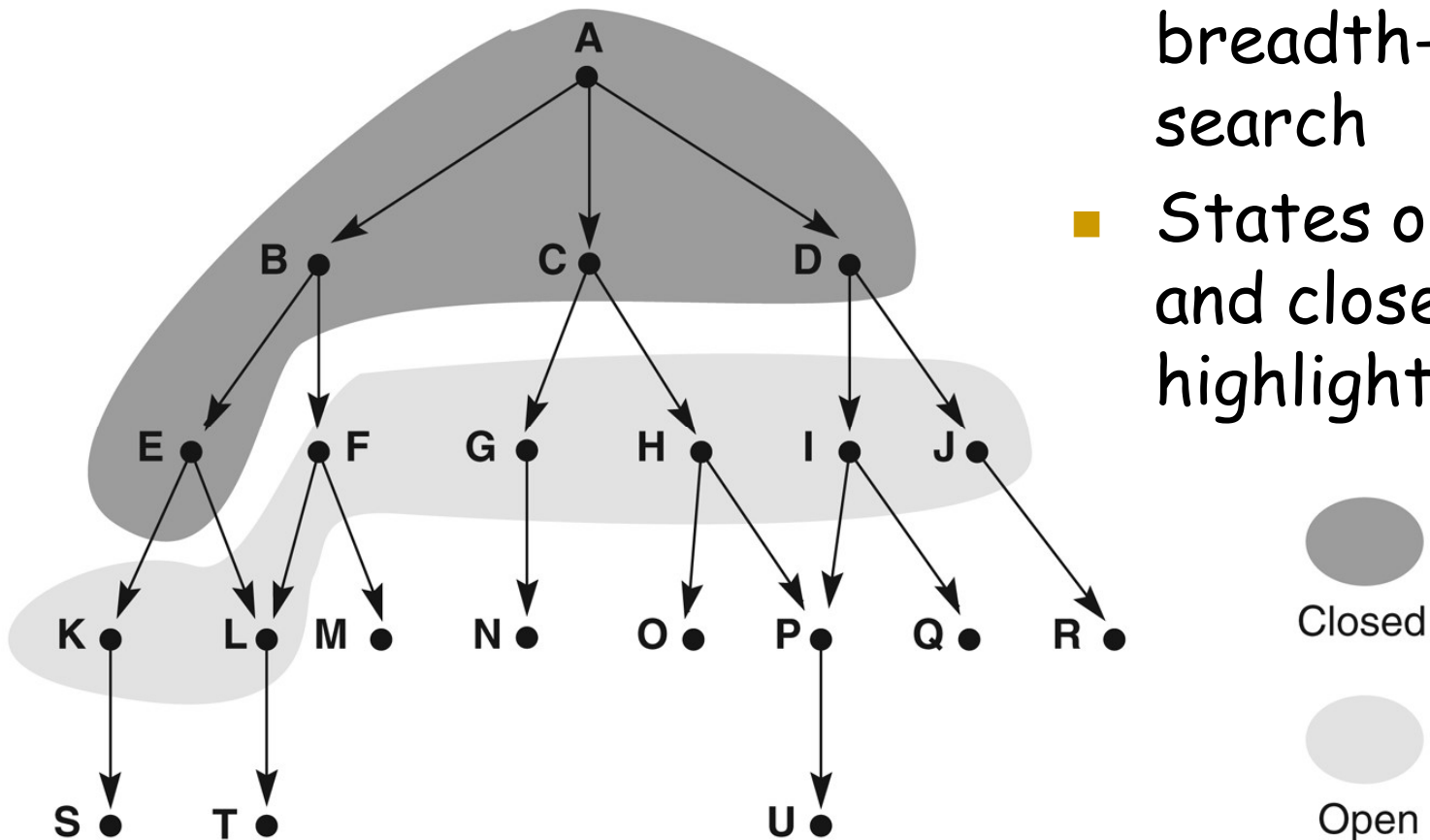
Assume U is goal state

1. open = [A-null] closed = []
2. open = [B-A C-A D-A] closed [A]
3. open = [C-A D-A E-B F-B] closed = [B A]
4. open = [D-A E-B F-B G-C H-C] closed = [C B A]
5. open = [E-B F-B G-C H-C I-D J-D] closed = [D C B A]
6. open = [F-B G-C H-C I-D J-D K-E L-E] closed = [E D C B A]
7. open = [G-C H-C I-D J-D K-E L-E M-F] as L is already in open closed = [F E D C B A]
8. and so on until either U is found or open = []



# Snapshot of BFS

- Search graph at iteration 6 of breadth-first search
- States on open and closed are highlighted



# Function Depth-First Search

Function depth\_first\_search algorithm

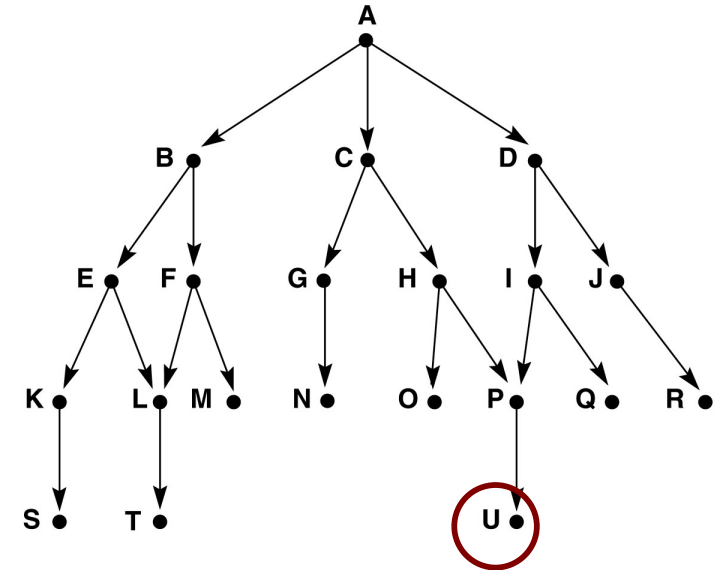
```
begin
  open := [Start];                                % initialize
  closed := [ ];
  while open ≠ [ ] do                             % states remain
    begin
      remove leftmost state from open, call it X;
      if X is a goal then return SUCCESS           % goal found
      else begin
        generate children of X;
        put X on closed;
        discard children of X if already on open or closed;
        put remaining children on left end of open % loop check
                                                    % stack
      end
    end
  end;
  return FAIL                                     % no states left
end.
```

# Depth-First Search Example

## ■ DFS: (open is a stack)

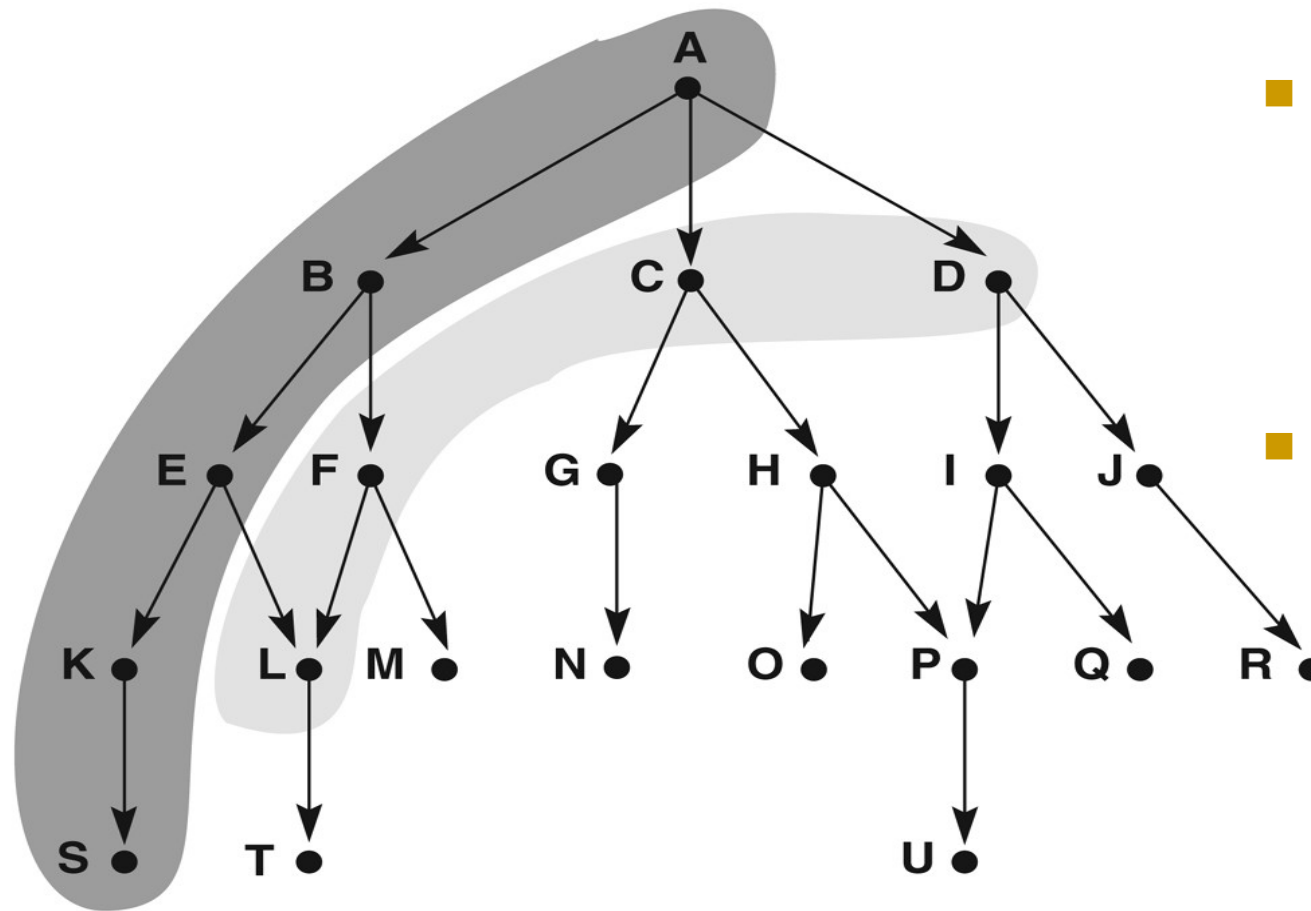
Assume U is goal state

1. open = [A-null] closed = []
2. open = [B-A C-A D-A] closed = [A]
3. open = [E-B F-B C-A D-A] closed = [B A]
4. open = [K-E L-E F-B C-A D-A] closed = [E B A]
5. open = [S-K L-E F-B C-A D-A] closed = [K E B A]
6. open = [L-E F-B C-A D-A] closed = [S K E B A]
7. open = [T-L F-B C-A D-A] closed = [L S K E B A]
8. open = [F-B C-A D-A] closed = [T L S K E B A]
9. open = [M-F C-A D-A] *as L is already on closed* closed = [F T L S K E B A]
10. open = [C-A D-A] closed = [M F T L S K E B A]
11. open = [G-C H-C D-A] closed = [C M F T L S K E B A]





# Snapshot of DFS



- Search graph at iteration 6 of depth-first search
- States on open and closed are highlighted

Closed

Open

# Depth-first vs. Breadth-first

- Breadth-first:
  - Optimal: will always find shortest pathBut:
  - inefficient if *branching factor* **B** is very high
  - memory requirements high -- exponential space for states required:  $B^n$
- Depth-first:
  - Not optimal (no guarantee to find the shortest path)But:
  - Requires less memory
- But both search are impractical in real applications because search space is too large!

# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breadth-first and Dep
    - Depth-limited Search
    - Iterative Deepening
    - Uniform Cost
  - Informed search
    - Hill climbing
    - Best-First
    - (Designing Heuristics)
    - A\*
- Summary



# Depth-Limited Search

## Compromise for DFS :

- Do depth-first but with **depth cutoff**  $k$  (depth at which nodes are not expanded)
- Three possible outcomes:
  - Solution
  - Failure (no solution)
  - Cutoff (no solution within cutoff)

# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breath-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Uniform Cost
  - Informed search
    - Hill climbing
    - Best-First
    - (Designing Heuristics)
    - A\*
- Summary



# Iterative Deepening

Compromise between BFS and DFS:

- use depth-first search, but
- with a maximum depth before going to next level
- Repeats depth first search with gradually increasing depth limits
  - Requires little memory (fundamentally, it's a depth first)
  - Finds the shortest path (limited depth)
- Preferred search method when there is a large search space and the depth of the solution is unknown

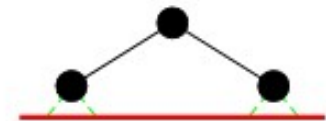
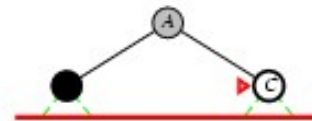
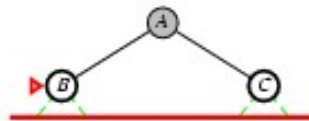
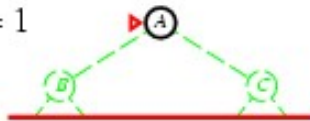
# Iterative Deepening: Example

Limit = 0



# Iterative Deepening: Example

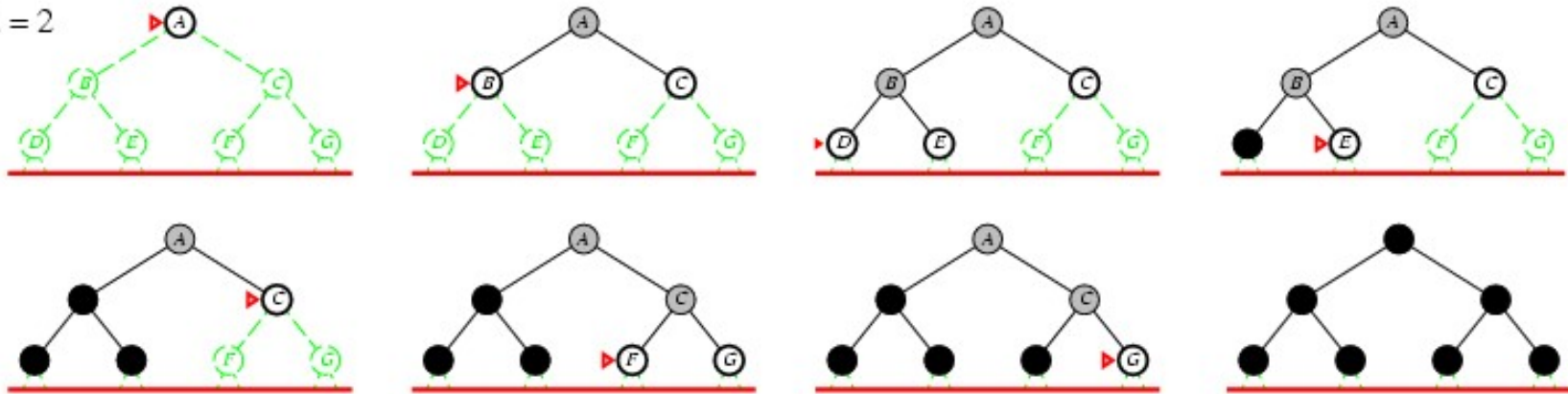
Limit = 1





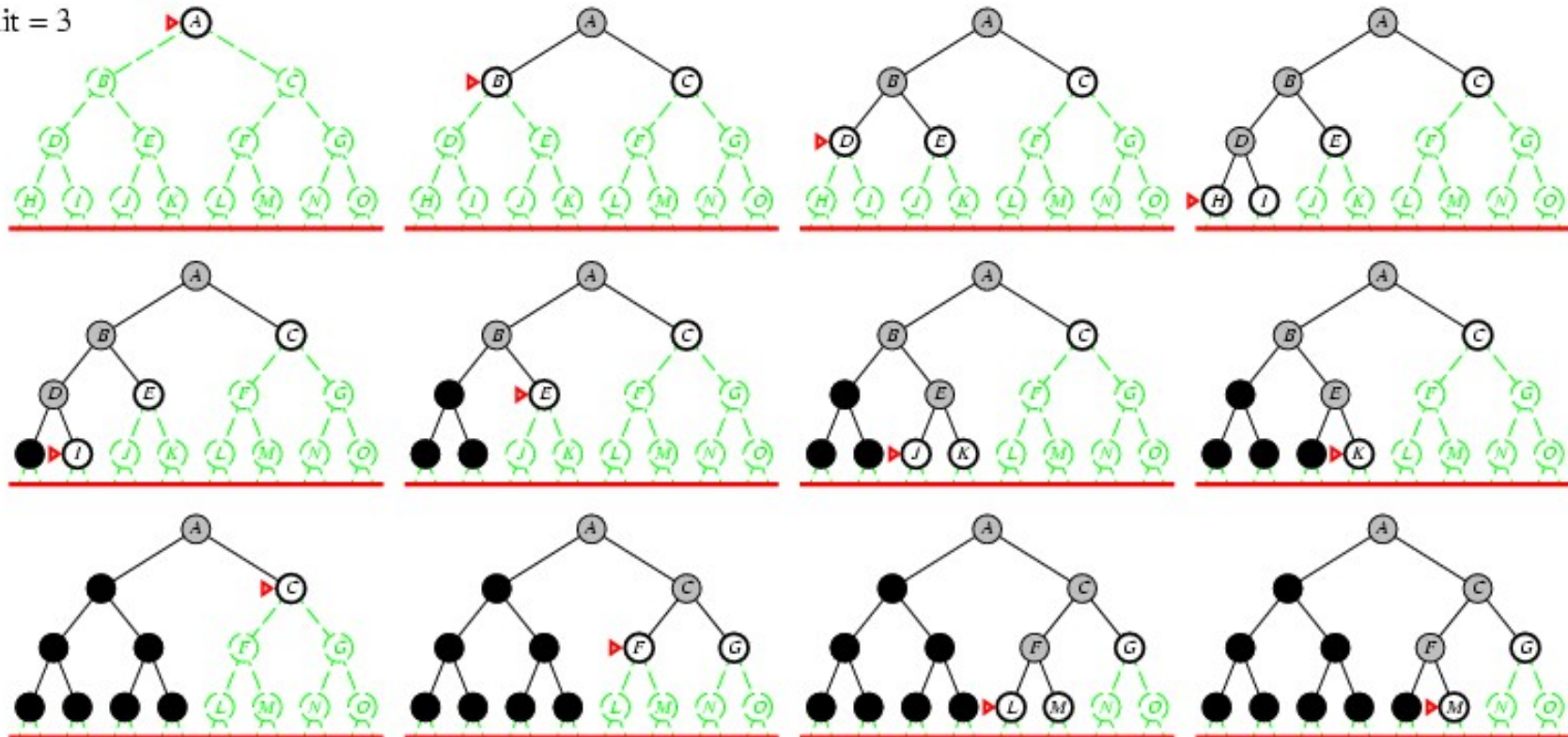
# Iterative Deepening: Example

Limit = 2



# Iterative Deepening: Example

Limit = 3



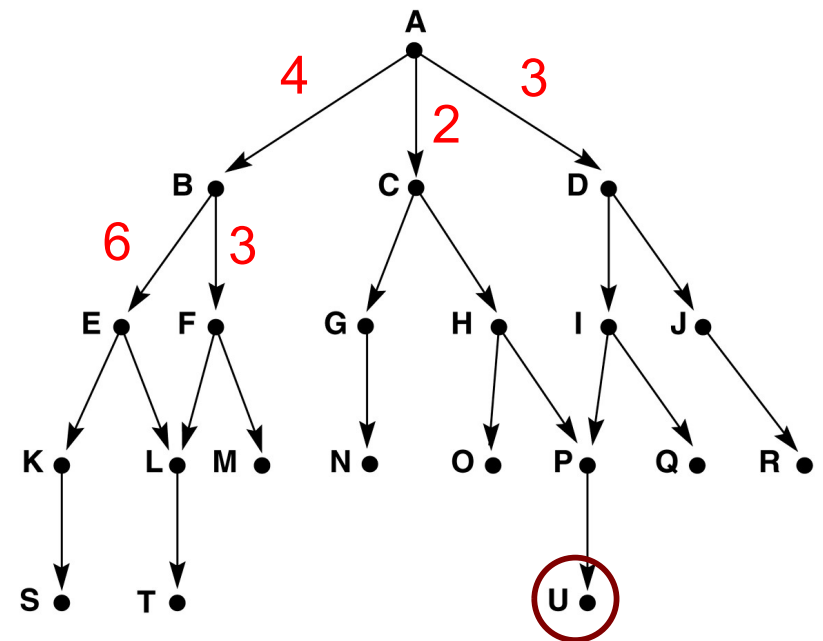
# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breath-first and Depth-first
    - Depth-limited
    - Iterative Deepening
    - Uniform Cost
  - Informed search
    - Hill climbing
    - Best-First
    - (Designing Heuristics)
    - A\*
- Summary



# Uniform Cost Search

- Breadth First Search
  - Open is a priority queue sorted using the depth of the nodes from the root
  - guarantees to find the shortest solution path
- But what if all edges/moves do not have the same cost?
- Uniform Cost Search
  - uses a priority queue sorted using the cost from the root to node  $n$  - later called  $g(n)$
  - guarantees to find the lowest cost solution path



# Uniform Cost Search

```
begin
  open := [Start];                                % initialize
  closed := [];
  while open ≠ [] do                               % states remain
    begin
      remove leftmost state from open, call it X;
      if X is a goal then return SUCCESS           % goal found
      else begin
        generate children of X;
        put X on closed;
        discard children of X if already on open or closed;
        put remaining children on right end of open
        end      Put the remaining children and reorder
                  (sort) the open list wrt g(n)
      end
    end
  return FAIL
end.
```

% loop check  
Priority queue  
(based on g(n))

% no states left

# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breath-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Uniform Cost
  - Informed search
    - Hill climbing
    - Best-First
    - (Designing Heuristics)
    - A\*
- Summary

# Informed Search (aka heuristic search)

- Most of the time, it is not feasible to do an exhaustive search, search space is too large
  - e.g. state space of all possible moves in chess =  $10^{120}$ 
    - $10^{75}$  = nb of molecules in the universe
    - $10^{26}$  = nb of nanoseconds since the "big bang"
- so far, all search algorithms have been uninformed (general search)
- so need an **informed/heuristic search**
- Idea:
  - choose "best" next node to expand
  - according to a selection function (i.e. a heuristic function  $h(n)$ )
- But: heuristic might fail

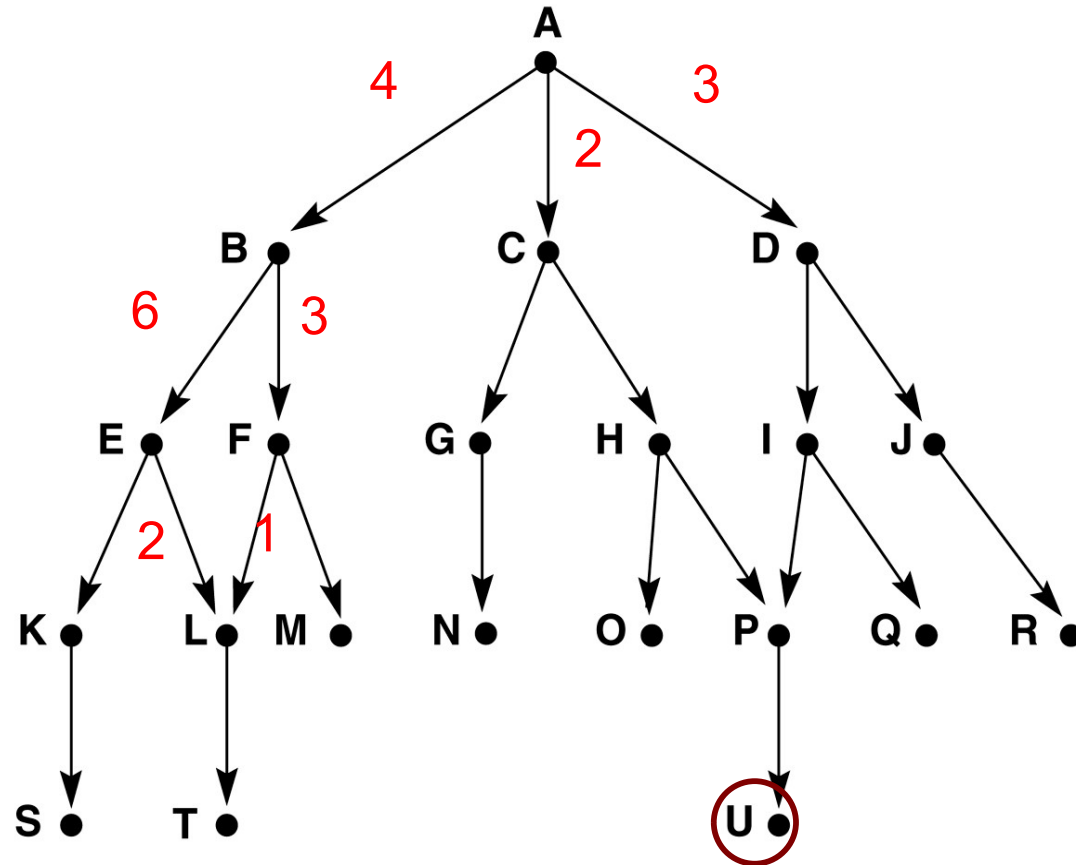
# Heuristic - Heureka!



- Heuristic:
  - a rule of thumb, a good bet
  - but has no guarantee to be correct whatsoever!
- Heuristic search:
  - A technique that improves the efficiency of search, possibly sacrificing on completeness
  - Focus on paths that seem most promising according to some function
  - Need an evaluation function (heuristic function) to score a node in the search tree
- Heuristic function  $h(n)$  = an **approximation** of the lowest cost from node  $n$  to the goal

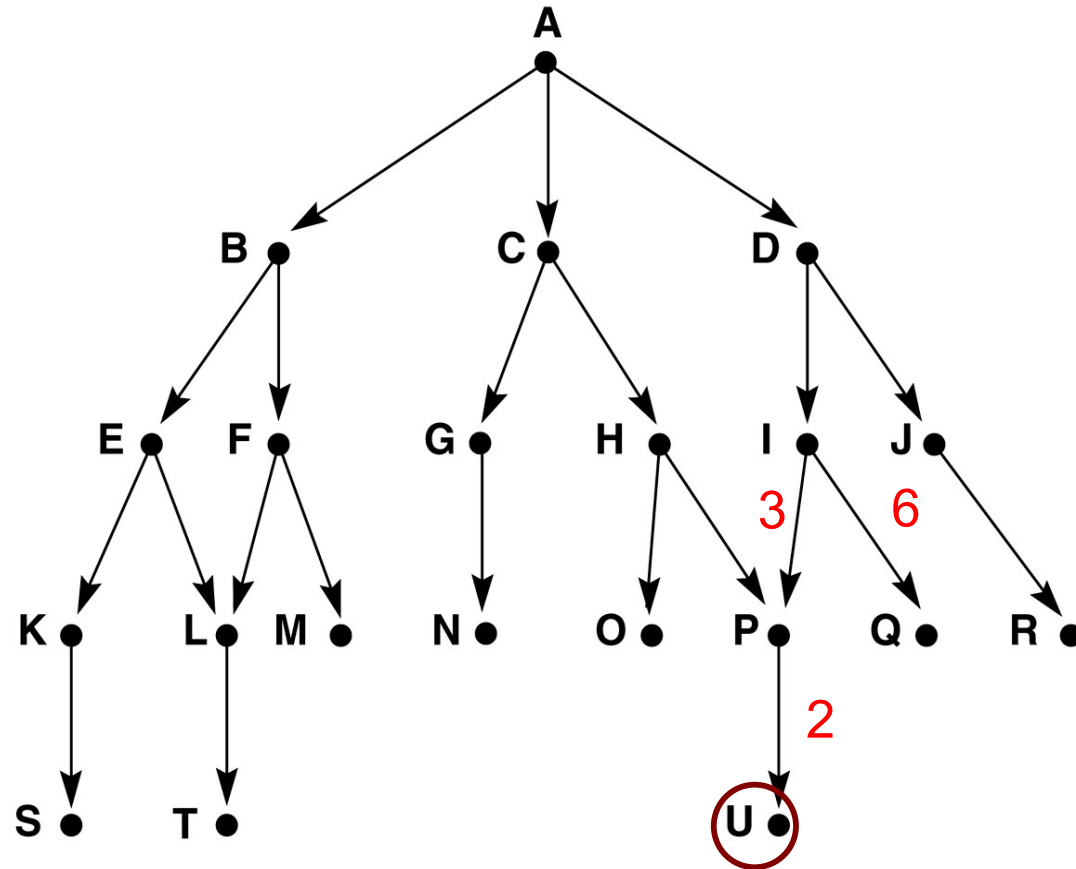


$g(n)$



- $g(n)$  = cost of current path from start to node  $n$

# $h(n)$



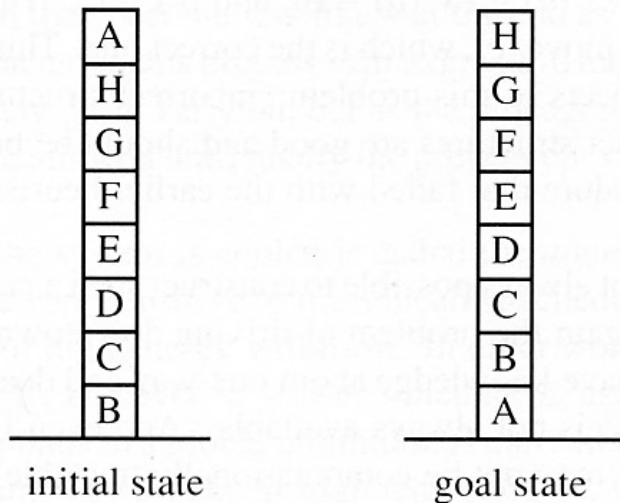
□  $h(n)$  = *estimate* of the lowest cost from  $n$  to *goal*

# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breath-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
  - Informed search
    - Hill Climbing
    - Best-First
    - (Designing Heuristics)
    - A\*
- Summary



# Example: Hill Climbing with Blocks World



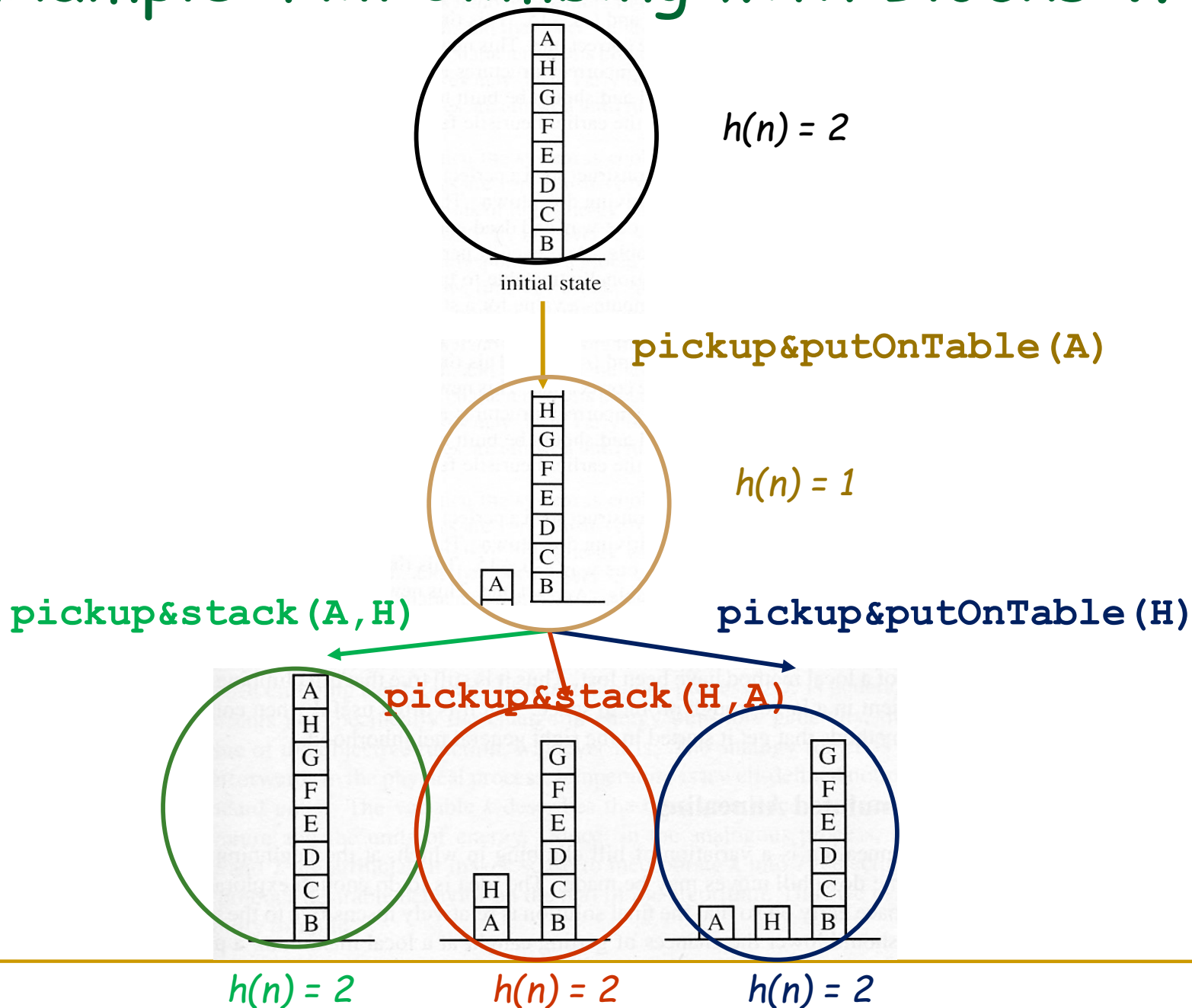
## ■ Operators:

- ❑ `pickup&putOnTable(Block)`
- ❑ `pickup&stack(Block1,Block2)`

## ■ Heuristic:

- ❑ Opt if a block is sitting (on another block) where it is supposed to sit
- ❑ +1pt if a block is NOT sitting where it is supposed to sit
- ❑ so lower  $h(n)$  is better
  - $h(\text{initial}) = 2$
  - $h(\text{goal}) = 0$

# Example: Hill Climbing with Blocks World



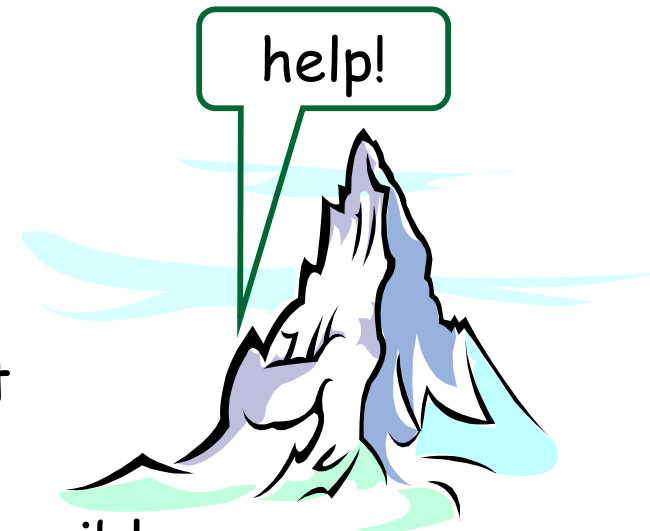
# Hill Climbing

- General hill climbing strategy:

- as soon as you find a position that is better than the current one, select it.
- Does not maintain a list of next nodes to visit (an open list)
- Similar to climbing a mountain in the fog with amnesia ... always go higher than where you are now, but never go back...

- Steepest ascent hill climbing:

- instead of moving to the first position that is better than the current one
- pick the best position out of all the next possible moves



# Steepest Ascent Hill Climbing

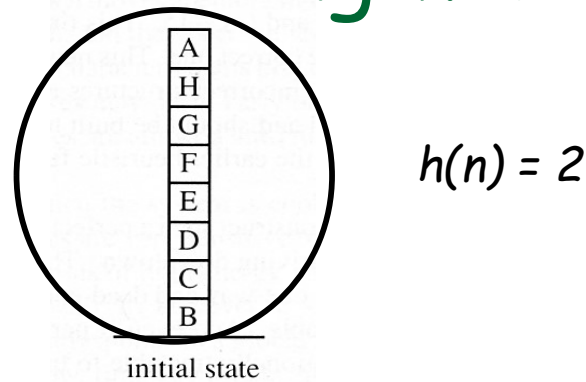
```
currentNode = startNode;
loop do
  L = CHILDREN(currentNode);
  nextEval = INFINITY;
  nextNode = NULL;

  for all c in L
    if (HEURISTIC-VALUE(c) < nextEval) // lower h is better
      nextNode = c;
      nextEval = HEURISTIC-VALUE(c);

  if nextEval >= HEURISTIC-VALUE(currentNode)
    // Return current node since no better child state exists
    return currentNode;

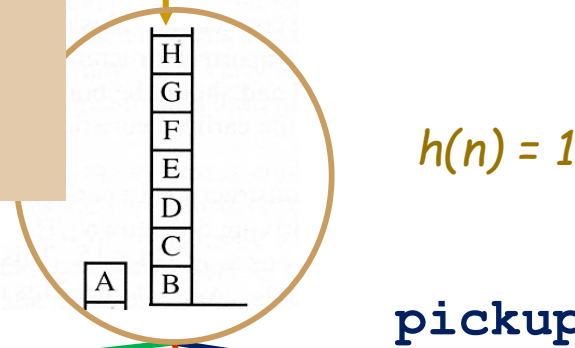
currentNode = nextNode;
```

# Example: Hill Climbing with Blocks World



hill-climbing will stop,  
because all children have  
higher  $h(n)$  than the  
parent... --> local minimum

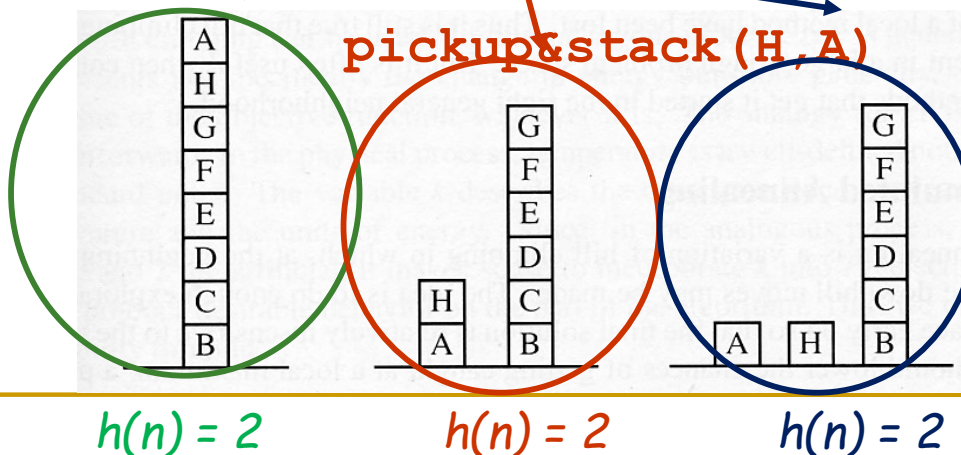
**pickup&putOnTable (A)**



**pickup&stack (A, H)**

**pickup&putOnTable (H)**

**pickup&stack (H, A)**



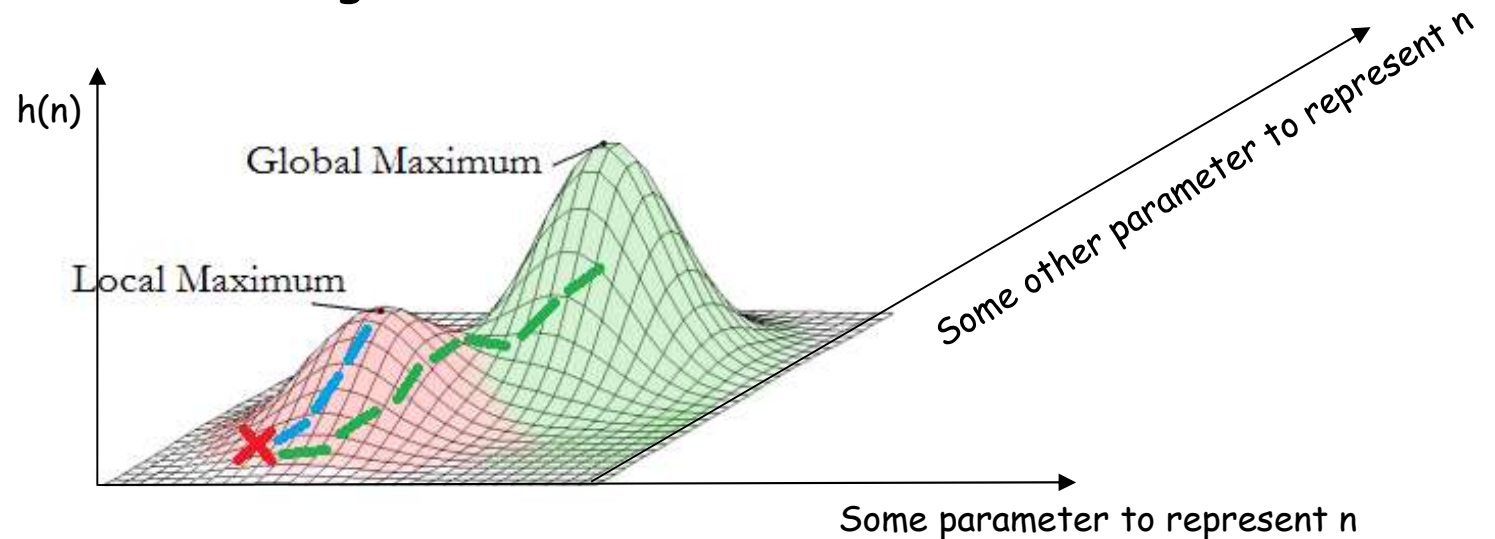
Don't be confused...  
a lower  $h(n)$  is better...



# Problems with Hill Climbing

## ■ Foothills (or local maxima)

- ❑ reached a local maximum, not the global maximum
- ❑ a state that is better than all its neighbors but is not better than some other states farther away.
- ❑ at a local maximum, all moves appear to make things worse.
- ❑ ex: 8-puzzle: we may need to move tiles temporarily out of goal position in order to place another tile in goal position
- ❑ ex: TSP: "nearest neighbour" heuristic



# Problems with Hill Climbing

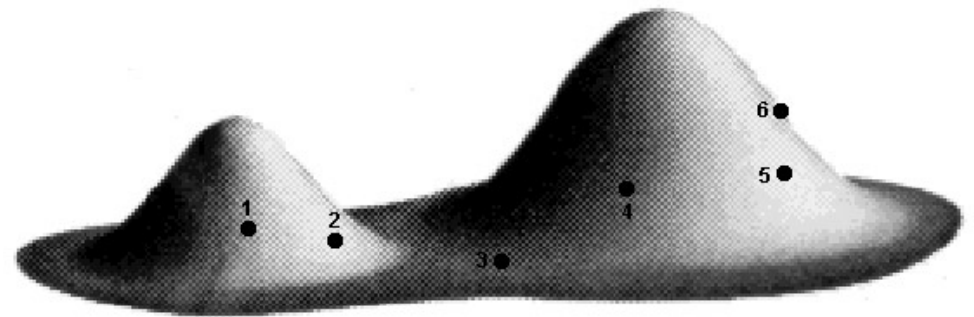
## ■ Plateau

- a flat area of the search space in which the next states have the same value.
- it is not possible to determine the best direction in which to move by making local comparisons.

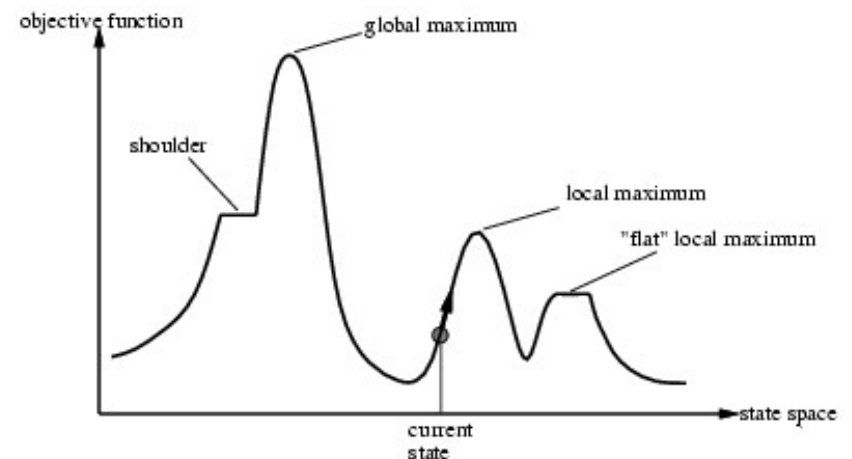


# Some Solutions to Hill-Climbing

- Random-restart hill-climbing
  - random initial states are generated
  - run each until it halts or makes no significant progress.
  - the best result is then chosen.



- keep going even if the best successor has the same value as current node
  - works well on a "shoulder"
  - but could lead to infinite loop on a plateau



# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
  - Informed
    - Hill Climbing
    - Best-First
    - (Designing Heuristics)
    - $A^*$
- Summary



# Best-First Search

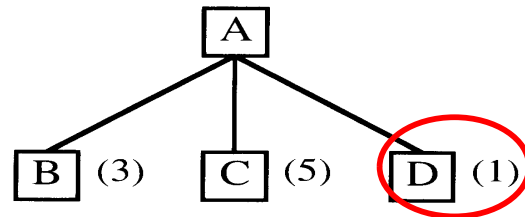
- problem with hill-climbing:
  - one move is selected and all others are forgotten.
- solution to hill-climbing:
  - use "open" as a priority queue
  - this is called best-first search
- Best-first search:
  - Insert nodes in *open* list so that the nodes are sorted in ascending  $h(n)$
  - Always choose the next node to visit to be the one with the best  $h(n)$  -- regardless of where it is in the search space

# Best-First: Example

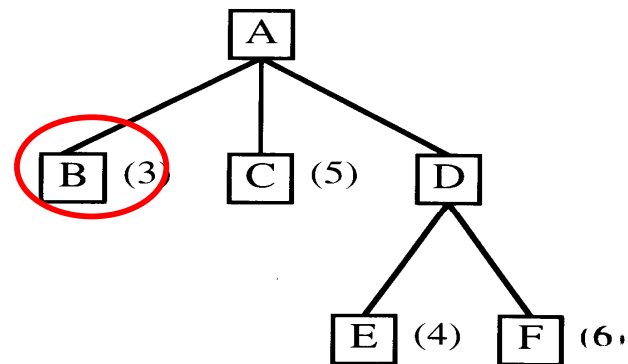
Step 1



Step 2

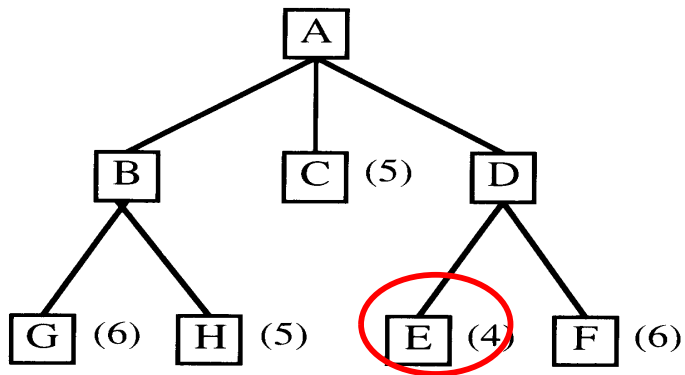


Step 3

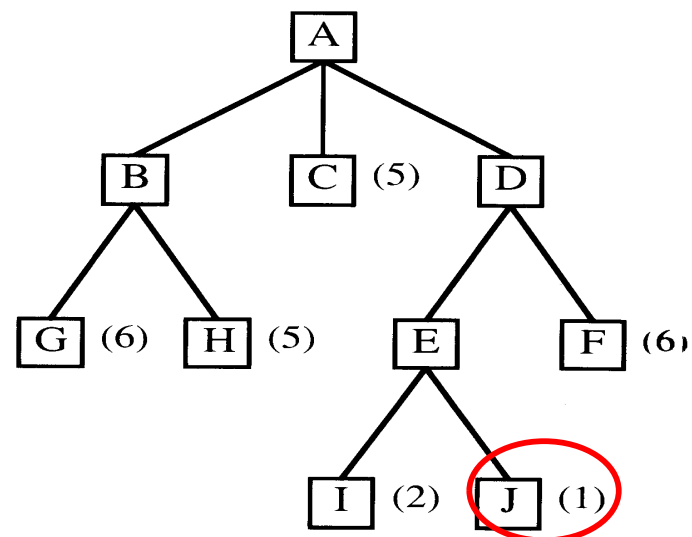


Lower  $h(n)$  is better

Step 4



Step 5



# Notes on Best-first

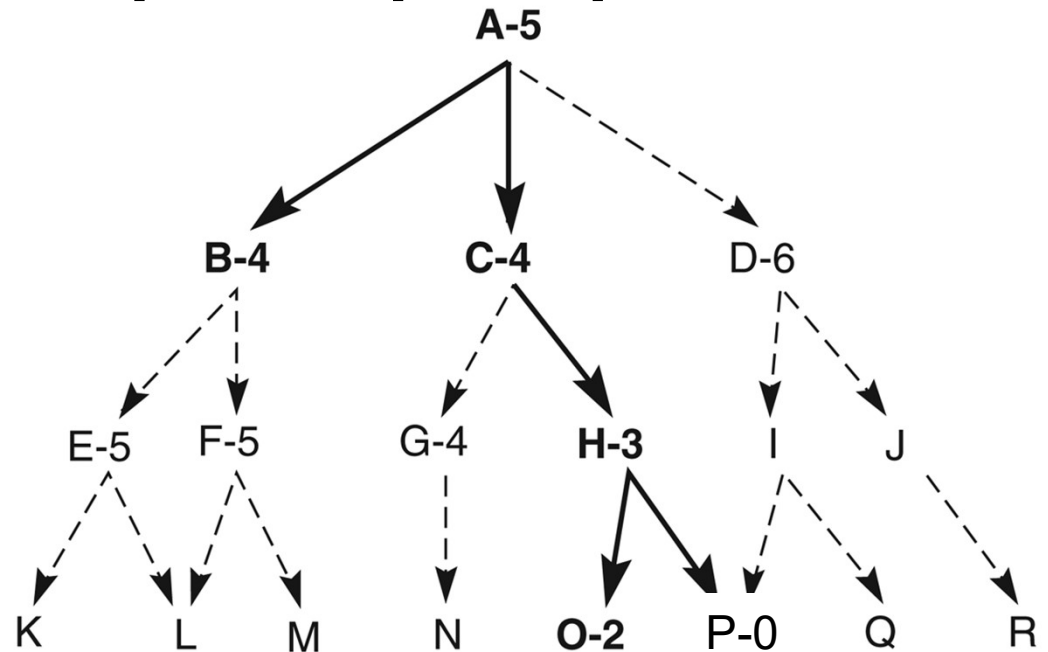
- If you have a good  $h(n)$ , best-first can find the solution very quickly
- The first solution may not be the best, but there is a good chance of finding it quickly
- It is an exhaustive search ...
  - will eventually try all possible paths

# Best-First Search: Example

1. open = [A-null-5] closed = []
2. open = [B-A-4 C-A-4 D-A-6] (*arbitrary choice*) closed [A]
3. open = [C-A-4 E-B-5 F-B-5 D-A-6] closed = [B A]
4. open = [H-C-3 G-C-4 E-B-5 F-B-5 D-A-6] closed = [C B A]
5. open = [P-H-0 O-H-2 G-C-4 E-B-5 F-B-5 D-A-6] closed = [H C B A]
6. goal P found

solution path: A C H P

Lower  $h(n)$  is better





# Today

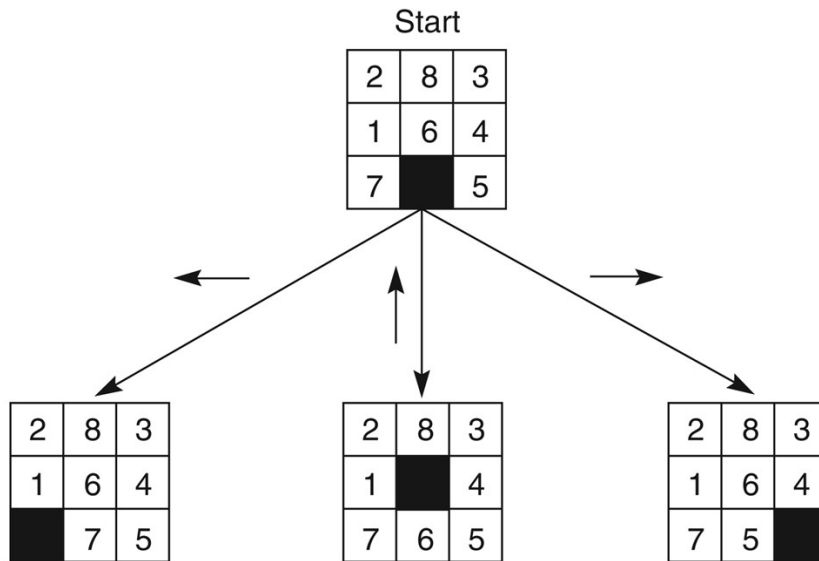
- State Space Representation
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
  - Informed search
    - Hill Climbing
    - Best-First
    - (Designing Heuristics)
    - $A^*$
- Summary



# Designing Heuristics

- Heuristic evaluation functions are highly dependent on the search domain
- In general: the more informed a heuristic is, the better the search performance
- Bad heuristics lead to frequent backtracking
- So how do we design a “good” heuristic?

# Example: 8-Puzzle - Heuristic 1



- $h_1$ : Simplest heuristic
  - Hamming distance : count number of tiles out of place when compared with goal

5		8
4	2	1
7	3	6

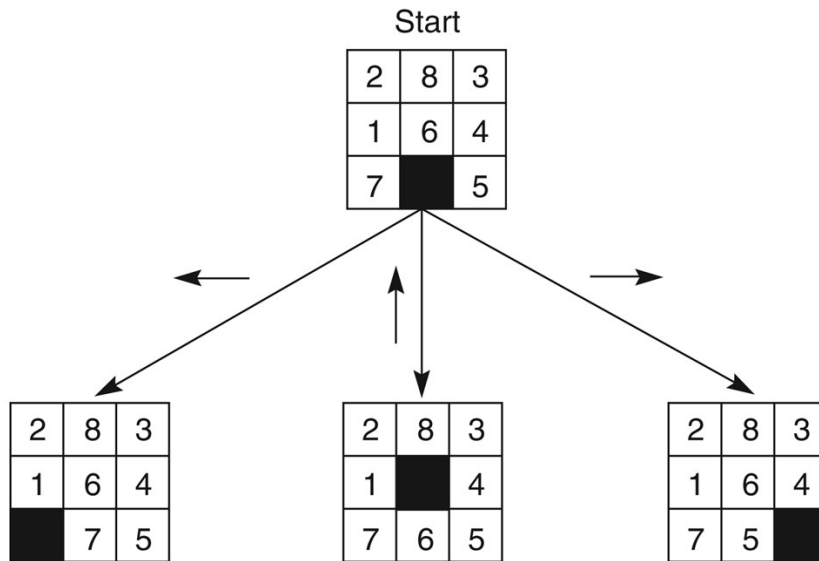
STATE n

1	2	3
4	5	6
7	8	

Goal state

- $h_1(n) = 6$ 
  - does not consider the distance tiles have to be moved

# Example: 8-Puzzle - Heuristic 2



- $h_2$ : Better heuristic
  - Manhattan distance: sum up all the **distances** by which tiles are **out of place**

5		8
4	2	1
7	3	6

STATE n

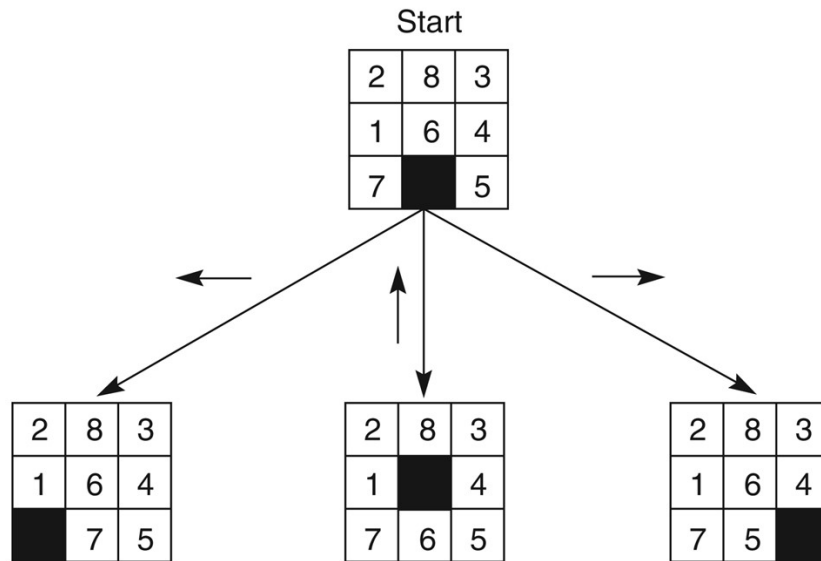
1	2	3
4	5	6
7	8	

Goal state

- $$h_2(n) = 2+3+0+1+3+0+3+1$$
$$= 13$$

(starting from 5, row-wise)

# Example: 8-Puzzle - Heuristic 3

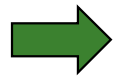


- $h_3$ : Even Better
  - sum of permutation inversions
  - See next slide...

# $h_3(N)$ = sum of permutation inversions

5		8
4	2	1
7	3	6

STATE n



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

- For each numbered tile, count how many tiles on its right should be on its left in the goal state.

- $$\begin{aligned} h_3(n) &= n_5 + n_8 + n_4 + n_2 + n_1 + n_7 + n_3 + n_6 \\ &= 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0 \\ &= 16 \end{aligned}$$

1	2	3
4	5	6
7	8	

Goal state

# Heuristics for the 8-Puzzle

5		8
4	2	1
7	3	6

STATE n

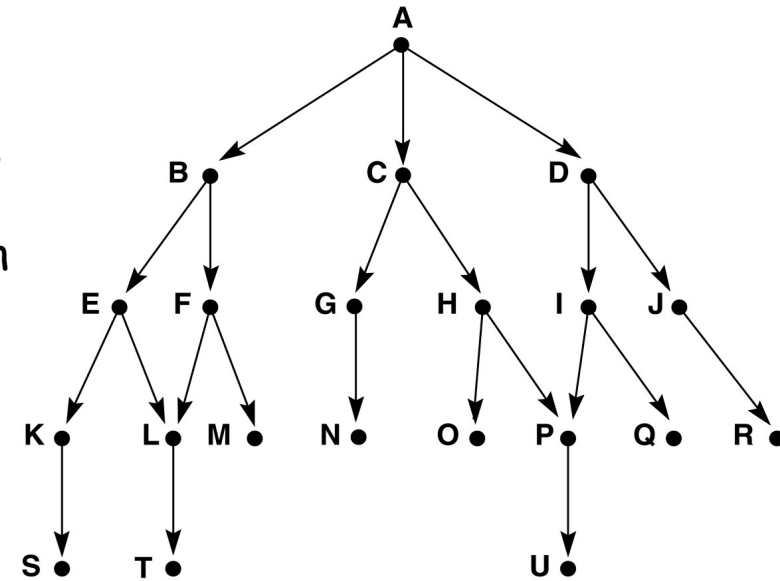
1	2	3
4	5	6
7	8	

Goal state

- $h_1(n)$  = misplaced numbered tiles  
= 6
- $h_2(n)$  = Manhattan distance  
=  $2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 = 13$
- $h_3(n)$  = sum of permutation inversions  
=  $n_5 + n_8 + n_4 + n_2 + n_1 + n_7 + n_3 + n_6$   
=  $4 + 6 + 3 + 1 + 0 + 2 + 0 + 0 = 16$

# $g(n)$ , $h(n)$ and $f(n)$

- Evaluation function  $f(n) = g(n) + h(n)$  for node  $n$ :
  - $g(n)$  current cost from *start* to node  $n$
  - $h(n)$  **estimate** of the lowest cost from  $n$  to *goal*
  - $f(n)$  **estimate** of the lowest cost of the solution path (from *start* to *goal* passing through  $n$ )
- Now consider  $f^*(n) = g^*(n) + h^*(n)$ :
  - $g^*(n)$  cost of **lowest cost** path from *start* to node  $n$
  - $h^*(n)$  **actual** lowest cost from  $n$  to *goal*
  - $f^*(n)$  **actual** cost of lowest cost of the solution path (from *start* to *goal* passing through  $n$ )





# Evaluating Heuristics

## 1. Admissibility:

- ❑ “optimistic”
- ❑ never overestimates the actual cost of reaching the goal
- ❑ guarantees to find the lowest cost solution path to the goal (if it exists)

## 1. Monotonicity:

- ❑ “local admissibility”
- ❑ guarantees to find the lowest cost path to each state  $n$  encountered in the search

## 1. Informedness:

- ❑ measure for the “quality” of a heuristic
- ❑ the more informed, the better

# Admissibility

- A heuristic is **admissible** if it never overestimates the cost of reaching the goal
- i.e.:
  - $h(n) \leq h^*(n)$  for all  $n$
- guarantees to find the lowest cost solution path to the goal (if it exists)
- Note: does not guarantee to find the lowest cost search path.
- e.g.: breadth-first is admissible -- it uses  $f(n) = g(n) + 0$

# Example: 8-Puzzle

5		8
4	2	1
7	3	6

n

1	2	3
4	5	6
7	8	

goal

- $h1(n)$  = Hamming distance = number of misplaced tiles = 6  
--> admissible
- $h2(n)$  = Manhattan distance = 13  
--> admissible

# Monotonicity (aka consistent)

- An admissible heuristics may temporarily reach non-goal states along a suboptimal path
- A heuristic is **monotonic** if it always finds the optimal path to each state the 1<sup>st</sup> time it is encountered !
- guarantees to find the lowest cost path to each state  $n$  encountered in the search
- $h$  is monotonic if for every node  $n$  and every successor  $n'$  of  $n$ :
  - $h(n) \leq c(n, n') + h(n')$
- i.e.  $f(n)$  is non-decreasing along any path

# Informedness

- Intuition: number of misplaced tiles is less informed than Manhattan distance
- For two admissible heuristics  $h_1$  and  $h_2$ 
  - if  $h_1(n) \leq h_2(n)$ , for all states  $n$
  - then  $h_2$  is **more informed** than  $h_1$
  - $h_1(n) \leq h_2(n) \leq h^*(n)$

More informed heuristics search smaller space to find the solution path

- However, you need to consider the computational cost of evaluating the heuristic...
- The time spent computing heuristics must be recovered by a better search

# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
  - Informed search
    - Hill Climbing
    - Beam Search
    - (Dijkstra's) Heuristics
    - A\*
- Summary



YOU ARE HERE!

# Algorithm A

- Heuristics might be wrong:
  - so search could continue down a wrong path
- Solution:
  - Maintain depth/cost count, i.e., give preference to shorter/least expensive paths
- Modified evaluation function  $f$ :
$$f(n) = g(n) + h(n)$$
  - $f(n)$  estimate of total cost along path through  $n$
  - $g(n)$  actual cost of path from start to node  $n$
  - $h(n)$  estimate of cost to reach goal from node  $n$

# Algorithm A on the 8-puzzle

$g(n) = 0$

Start

2	8	3
1	6	4
7		5

$g(n) = 1$

2	8	3
1	6	4
	7	5

2	8	3
1		4
7	6	5

2	8	3
1	6	4
7	5	

Values of  $f(n)$  for each state,

**6**

**4**

**6**

where:

$f(n) = g(n) + h(n)$ ,

$g(n)$  = actual distance from  $n$   
to the start state, and

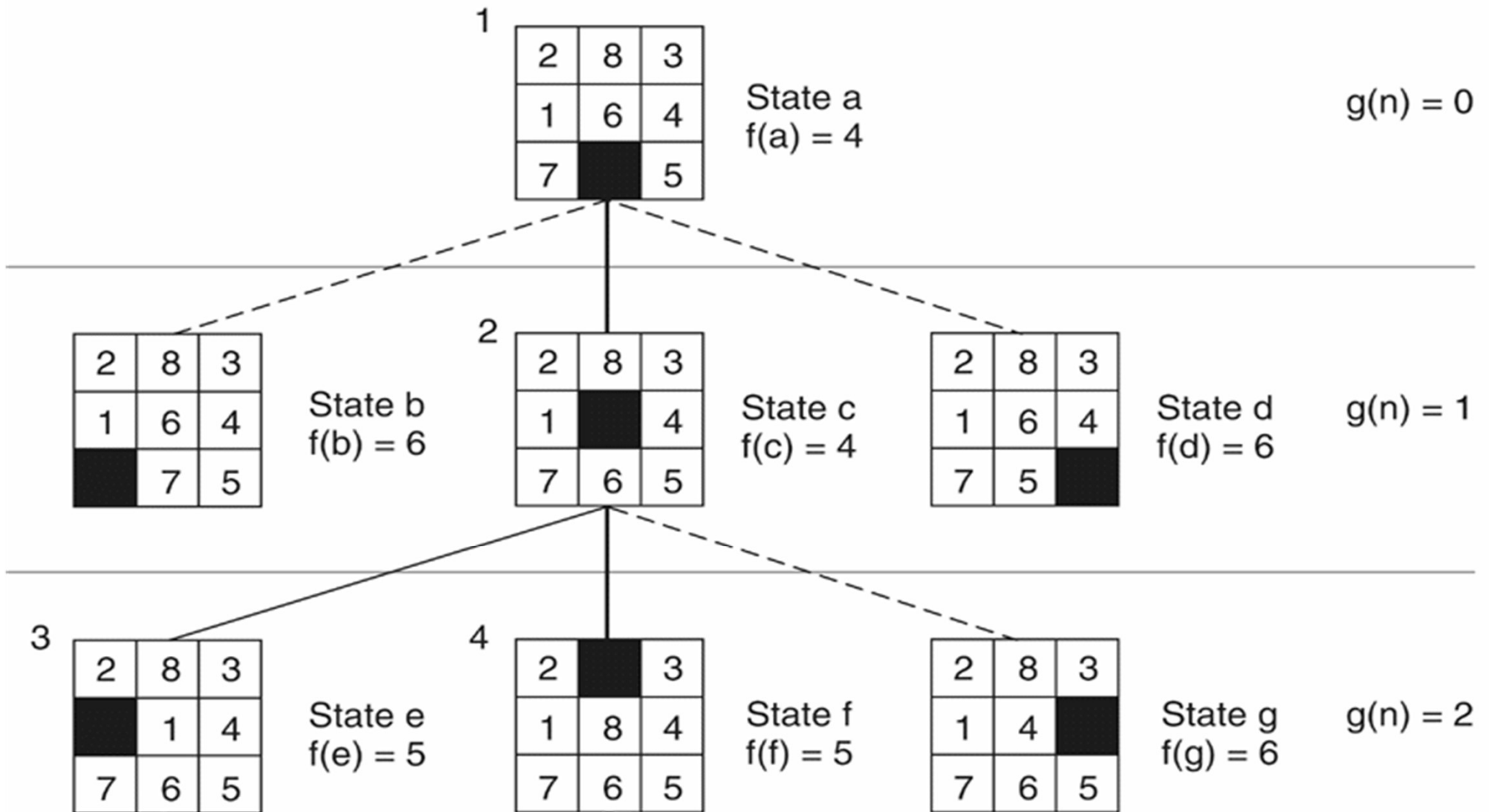
$h(n)$  = number of tiles out of place.

1	2	3
8		4
7	6	5

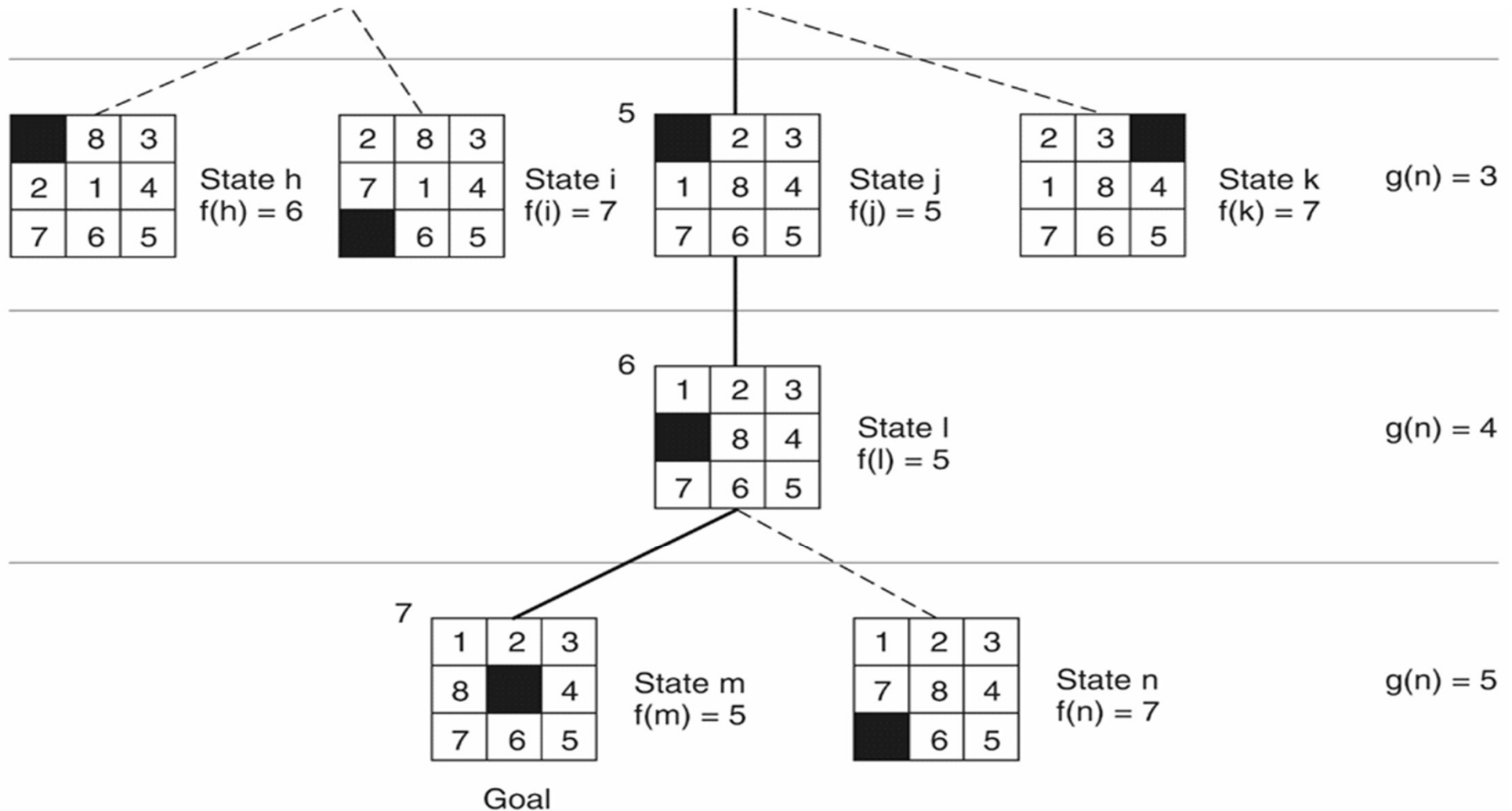
Goal



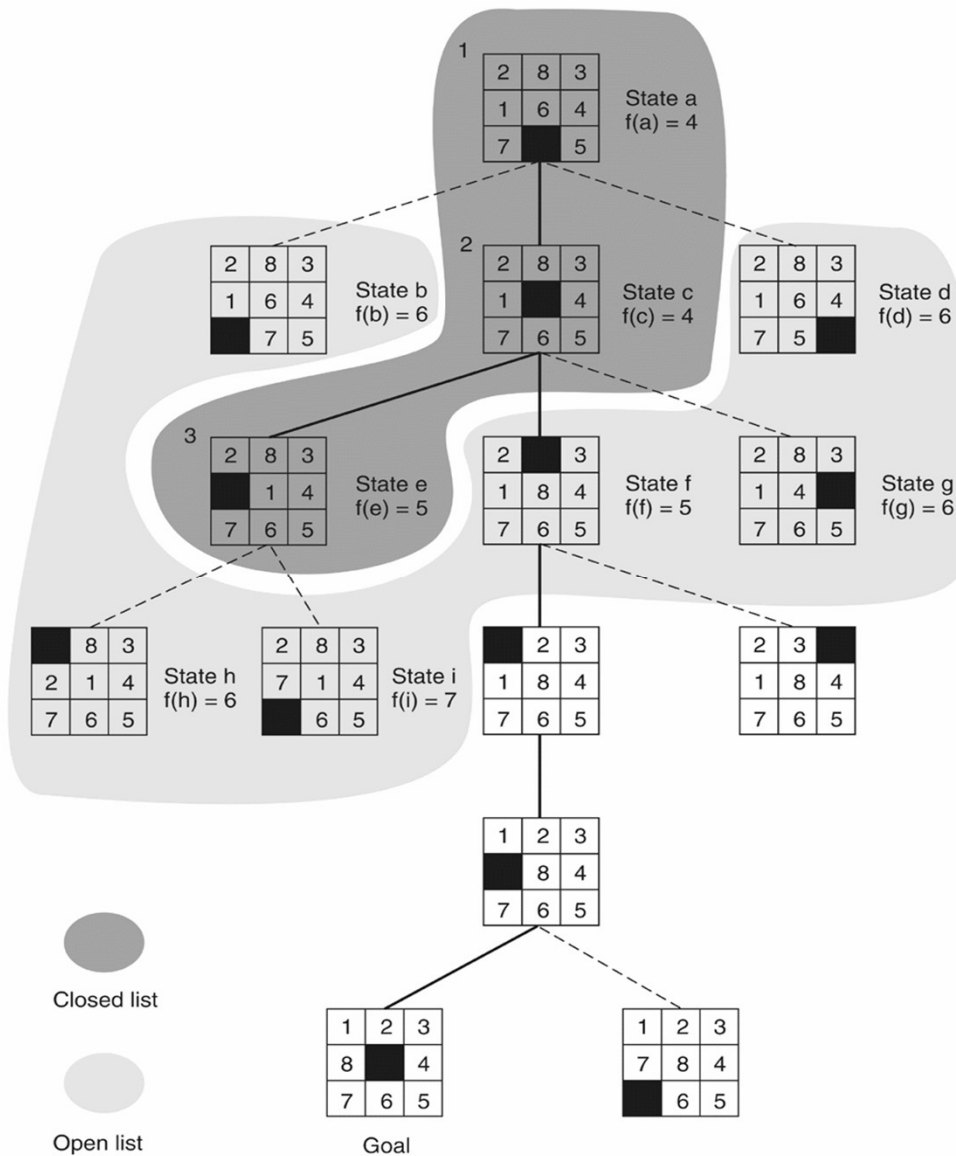
# Algorithm A on the 8-puzzle

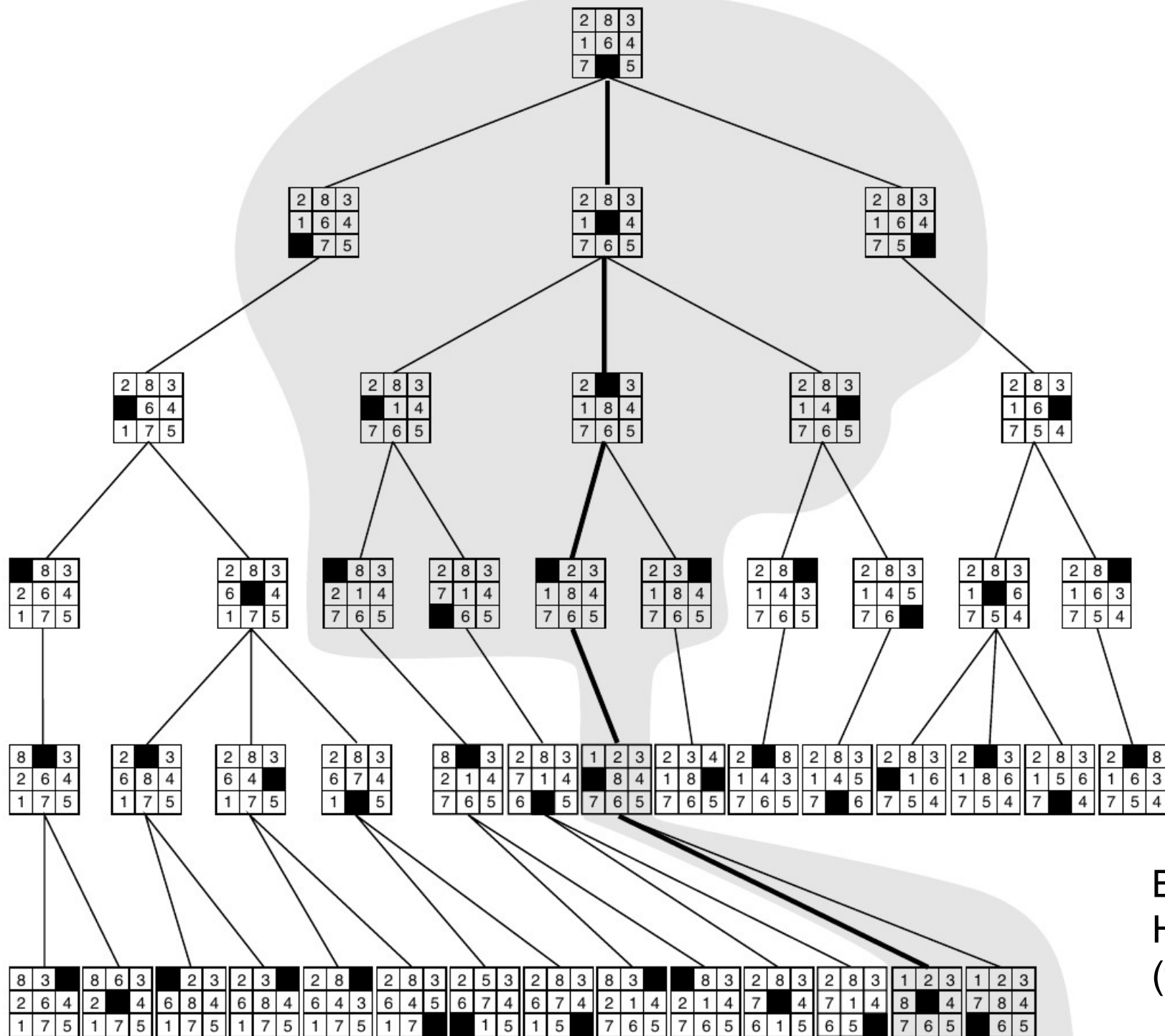


# Algorithm A on the 8-puzzle



# Algorithm A on the 8-puzzle





Goal

BFS vs.  
Heuristic search  
(tiles out of place)

source: G. Luger (2005)

# Algorithm A vs Algorithm A\*

- if  $g(n) \geq g^*(n)$  for all  $n$ 
  - best-first search with  $f(n) = g(n) + h(n)$  is called "algorithm A"
- if  $h(n) \leq h^*(n)$  for all  $n$ 
  - i.e.  $h(n)$  never overestimates the true cost from  $n$  to a goal
  - algorithm A used with such an  $h(n)$  is called "algorithm A\*"
  - an A\* algorithm is admissible
  - i.e. it guarantees to find the lowest cost solution path from the initial state to the goal

# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breath-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
  - Informed search
    - Hill climbing
    - Best-First
    - (Designi cs)
    - A\*
- Summary



# Summary

Search	Uses $h(n)$ ?	Open is a...
Breadth-first	No	Queue
Depth-first	No	Stack
Depth-limited	No	Stack
Iterative Deepening	No	Stack
Uniform Cost	No	Priority queue sorted by $g(n)$
Hill Climbing	Yes	none
Best-First	Yes	Priority queue sorted by $h(n)$
Algorithm A - no constraints on $h(n)$	Yes	Priority queue sorted by $f(n)$ $f(n) = g(n) + h(n)$
Algorithm A* - same as A, but $h(n)$ must be admissible - guarantees to find the lowest cost solution path	Yes	Priority queue sorted by $f(n)$ $f(n) = g(n) + h(n)$

# Today

- State Space Representation
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Uniform Cost
  - Informed search
    - Hill climbing
    - Best-First
    - A\*
- Summary

