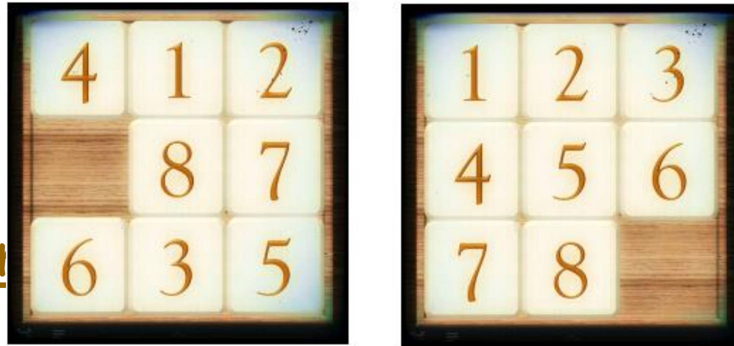
States and Searching

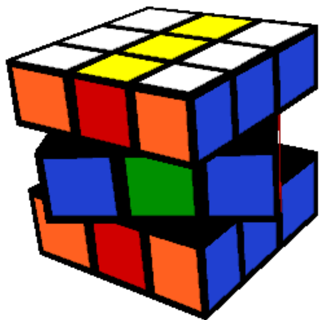
- Poole, chap. 3
- *Some slides and/or pictures in the following are adapted from slides ©2012. Prof. L. Kosseim;*

Motivation



Play the 8-puzzle online

Flow Free online



Rubik's cube

Google Self-Driving Car Route Planning

Today

- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed search
    - Best-First
    - A*
- Summary

Problem Solving

- To solve problems is an intelligent capability
- We are able to solve very different problems
  - To find a path in a labyrinth
  - To solve a crossword
  - To play a game
  - To diagnose an illness
  - To decide if invest in the stock market
  - ...
- The goal is to have programs also able to solve these problems

- We want to represent any problem so we can solve it automatically
- To do so we need:
  - A common representation for all the problems
  - Algorithms that use some strategy to solve the problems using the common representation

Example: 8-Puzzle

8	2	
3	4	7
5	1	6

Initial state

1	2	3
4	5	6
7	8	

Goal state

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

(n^2-1)-puzzle

8	2	
3	4	7
5	1	6

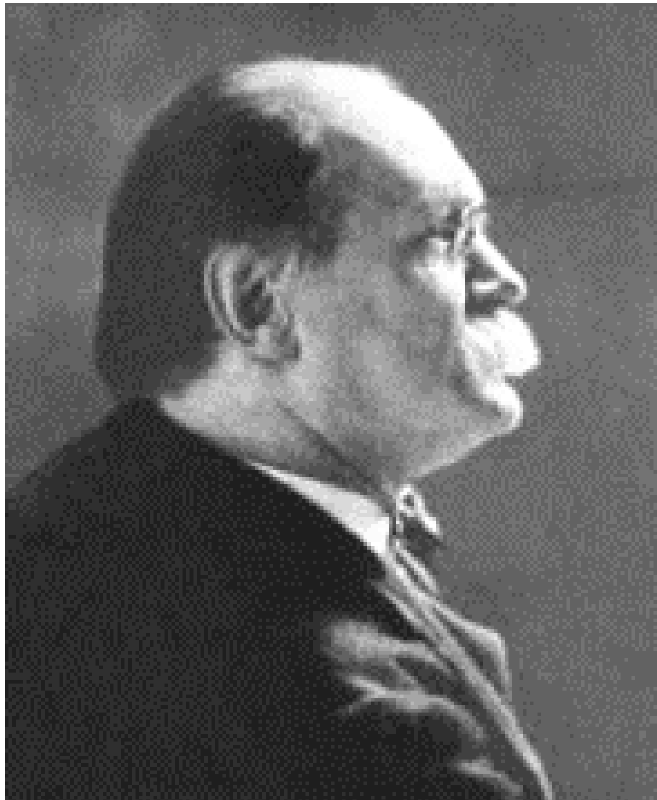
8-puzzle

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

15-puzzle

15-Puzzle

Introduced (?) in 1878 by Sam Loyd



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,"
"15 Block Puzzle," "Pigs in Clover,"
"Parcheesi," Etc., Etc.

P. O. BOX [illegible].

New York, April 15 1903

15-Puzzle

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

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

?

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

But no one ever won the prize…

Representing a problem: Searching

- If we analyze the nature of a problem we can identify:
  - A starting point
  - A goal to achieve
  - Actions that we can use to solve the problem
  - Constraints for the goal
  - Elements relevant to the problem defined by the characteristics of the domain

- Often we are not given an algorithm to solve a problem, but only a specification of what is a solution — we have to search for a solution.
- A typical problem is when the agent is in one state, it has a set of deterministic actions it can carry out, and wants to get to a goal state.
- Many AI problems can be abstracted into the problem of finding a path in a directed graph.
- Often there is more than one way to represent a problem as a graph.

Today

- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed search
    - Best-First
    - A*
- Summary

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
- Brute force search:
  - generate and search <u>all</u> possibilities (but inefficient)
- 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

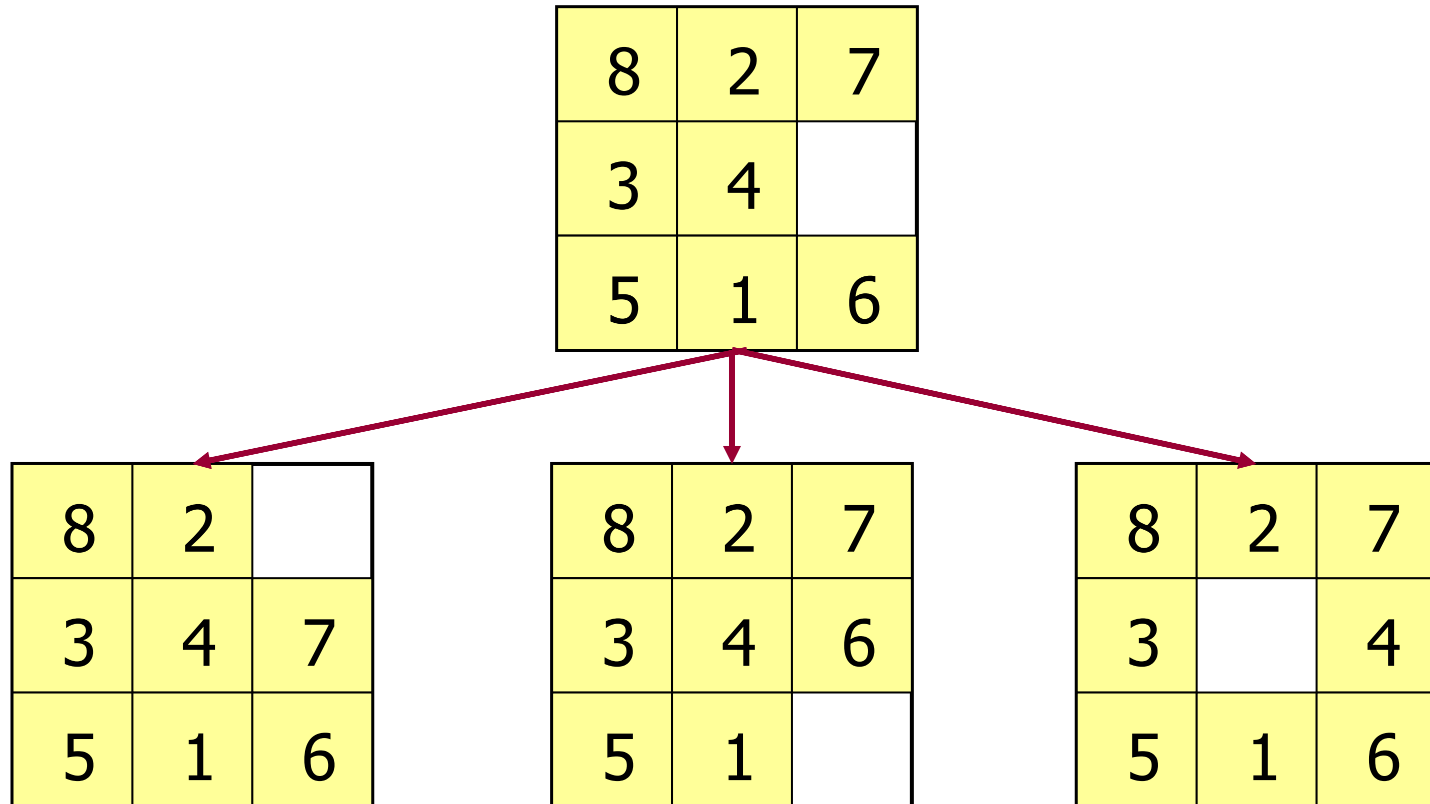
Initial state

1	2	3
4	5	6
7	8	

Goal state

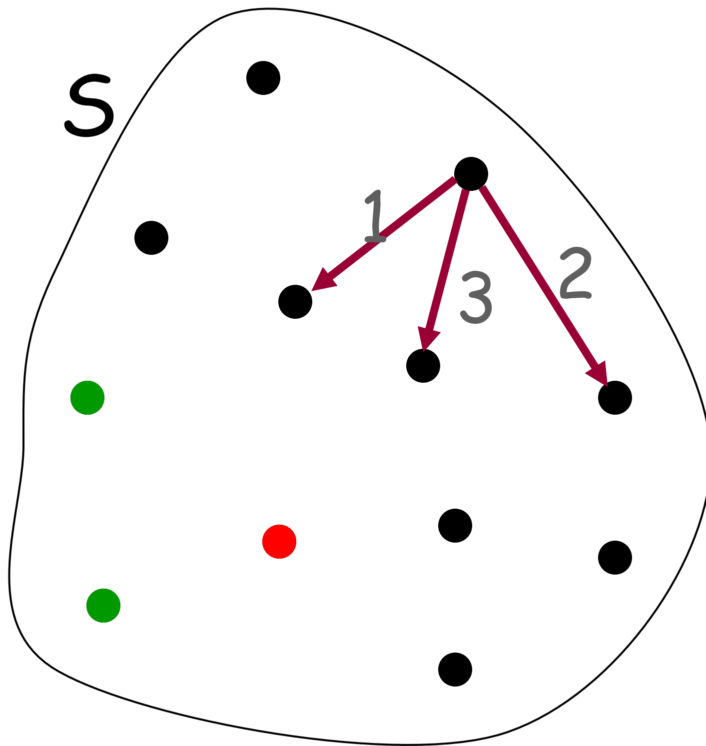
- Set of operators:
  - blank moves up, blank moves down, blank moves left, blank moves right
- Goal test function:
  - state matches the goal state
- Path cost function:
- each movement costs 1
- so the path cost is the length of the path (the number of moves)

8-Puzzle: Successor Function



- Search is about the exploration of alternatives

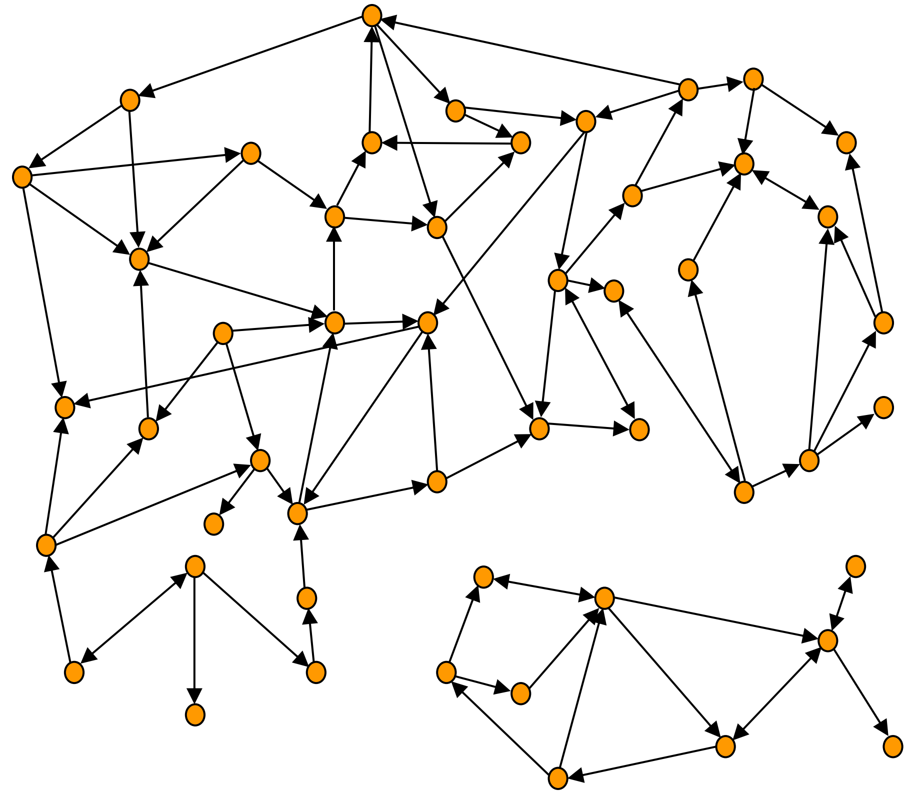
State Space Representation



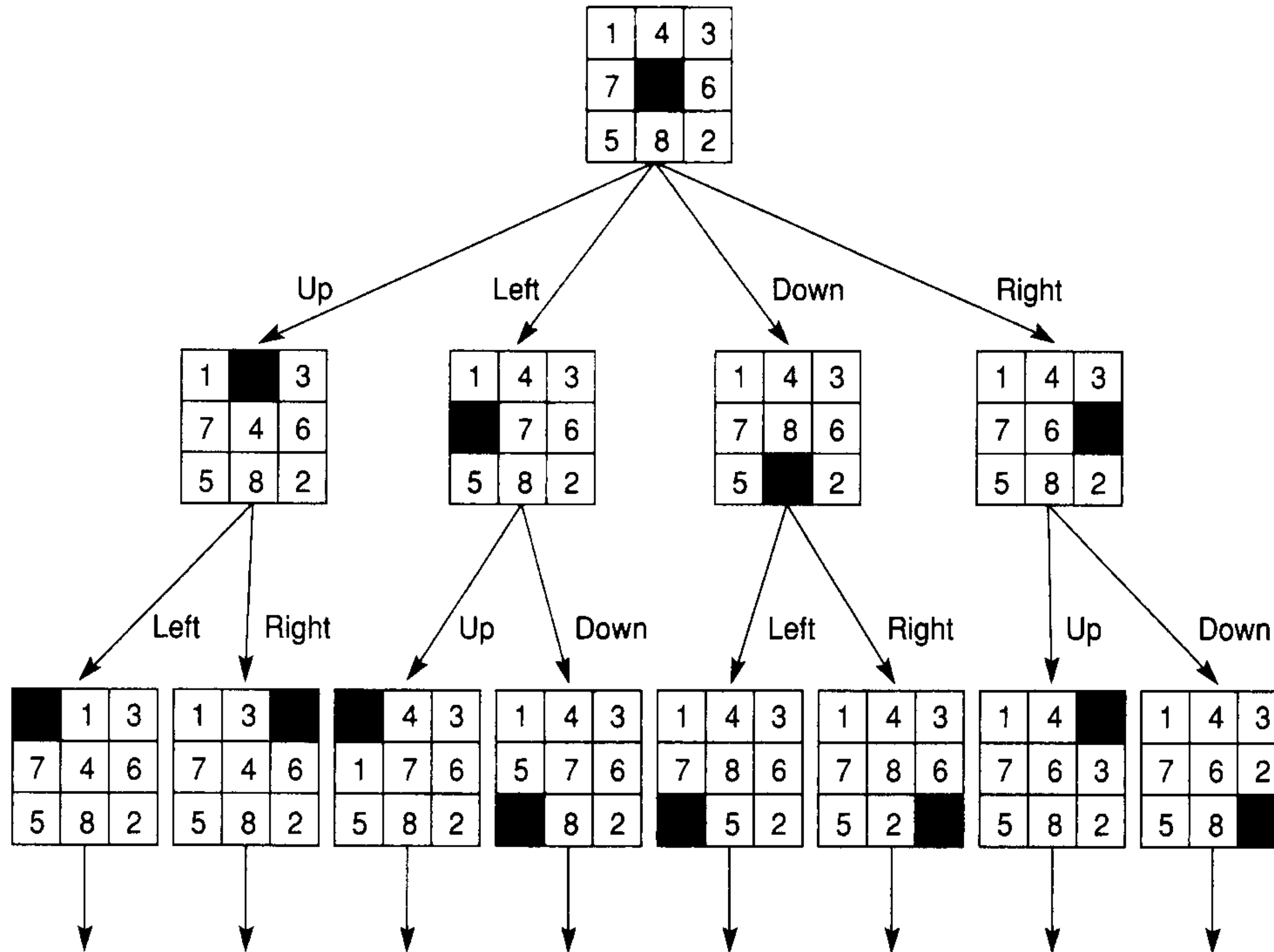
- State space S
- Successor function:
  $x \in S \rightarrow$ SUCCESSORS(x)
- Initial state s_0
- Goal states:
- $x \in S \rightarrow$ GOAL?(x) = T or F
- Arc cost

State Graph

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



State Space for the 8-puzzle

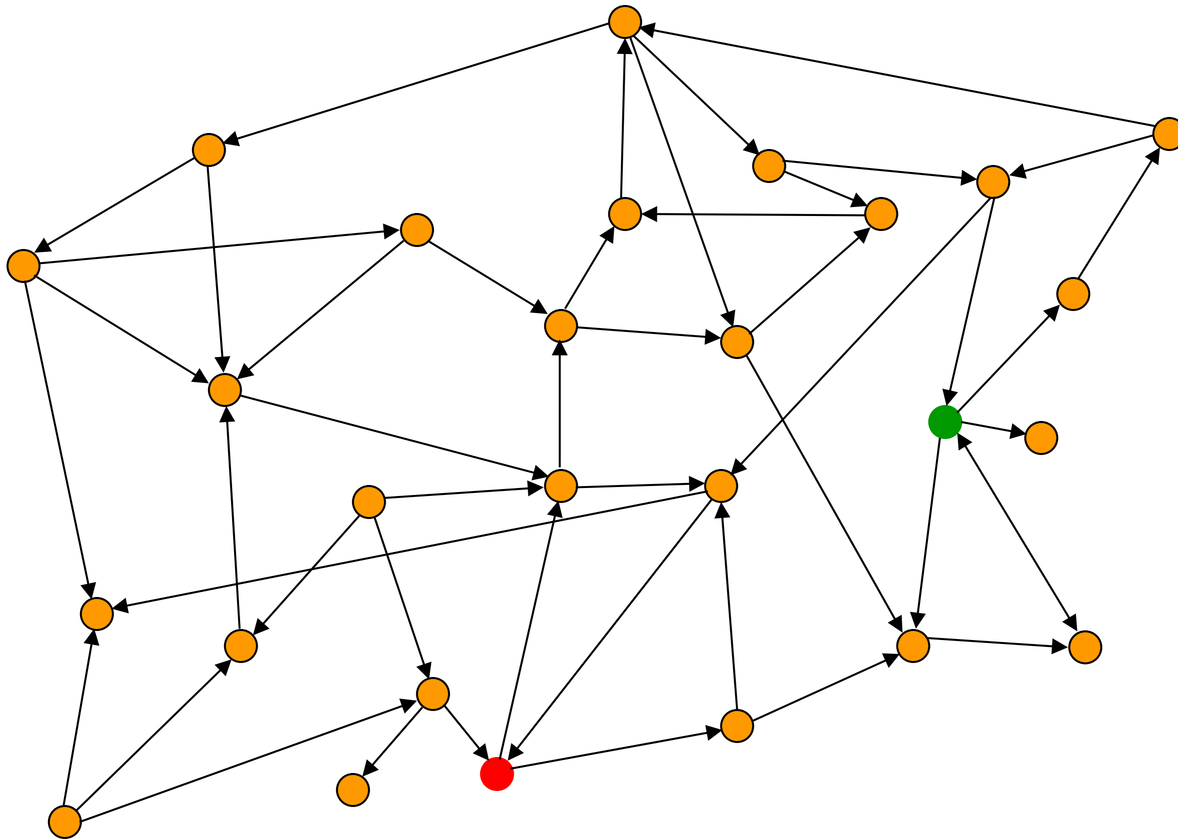


How big is the state space of the (n^2-1)-puzzle?

- Nb of states:
  - 8-puzzle --> 9! = 362,880 states
  - 15-puzzle --> 16! ~ 2.09×10^{13} states
  - 24-puzzle --> 25! ~ 10^{25} states

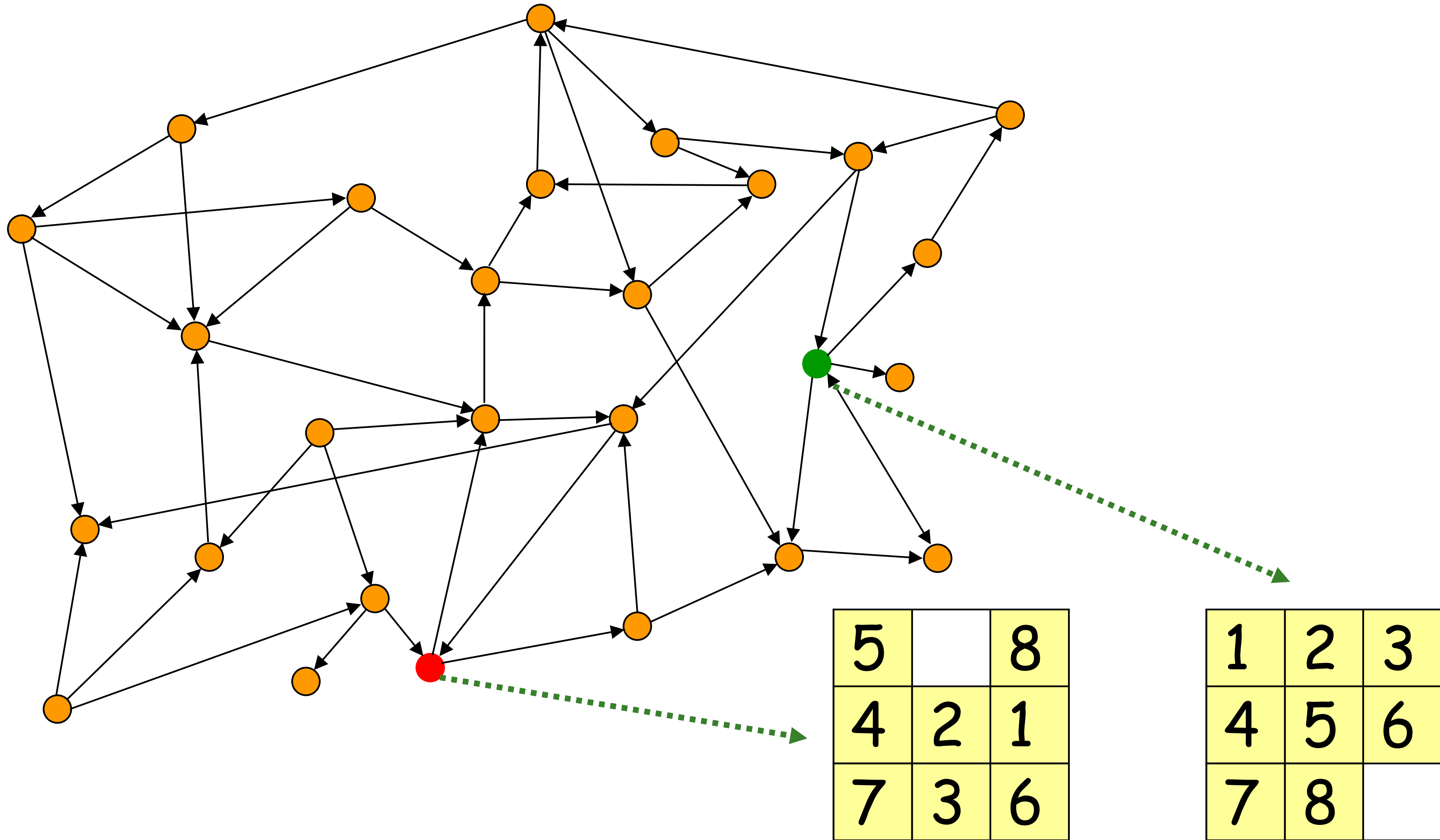
- At 100 millions states/sec:
  - 8-puzzle --> 0.036 sec
  - 15-puzzle --> ~ 55 hours
  - 24-puzzle --> > 10^9 years

Searching the State Space



- It is often not feasible (or too expensive) to build a complete representation of the state graph

Just to make sure we're clear…



Today

- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed search
    - Best-First
    - A*
- Summary

State-Space Search

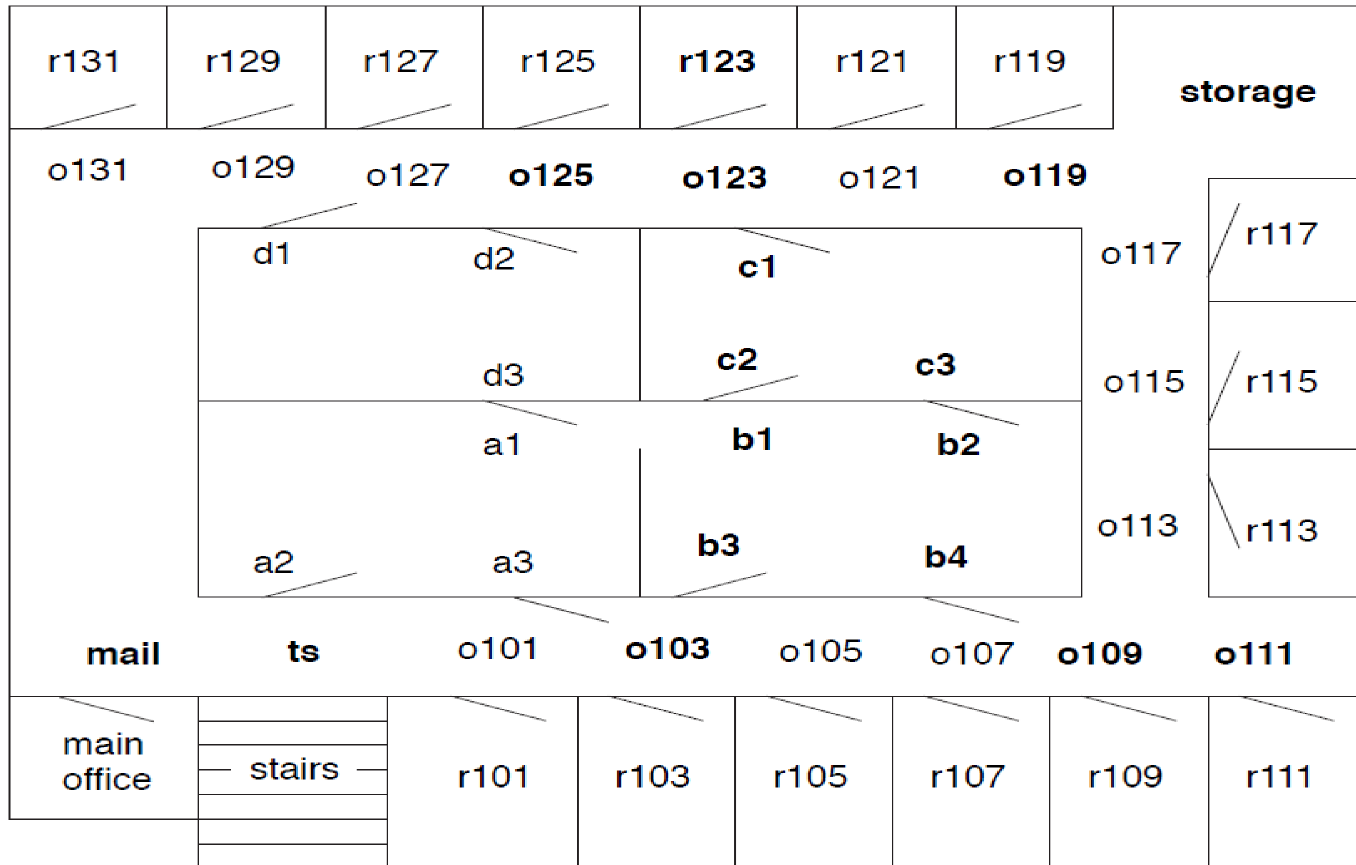
- flat or modular or hierarchical
- explicit states or features or individuals and relations
- static or finite stage or indefinite stage or infinite stage
- fully observable or partially observable
- deterministic or stochastic dynamics
- goals or complex preferences
- single agent or multiple agents
- knowledge is given or knowledge is learned
- perfect rationality or bounded rationality

Directed Graphs

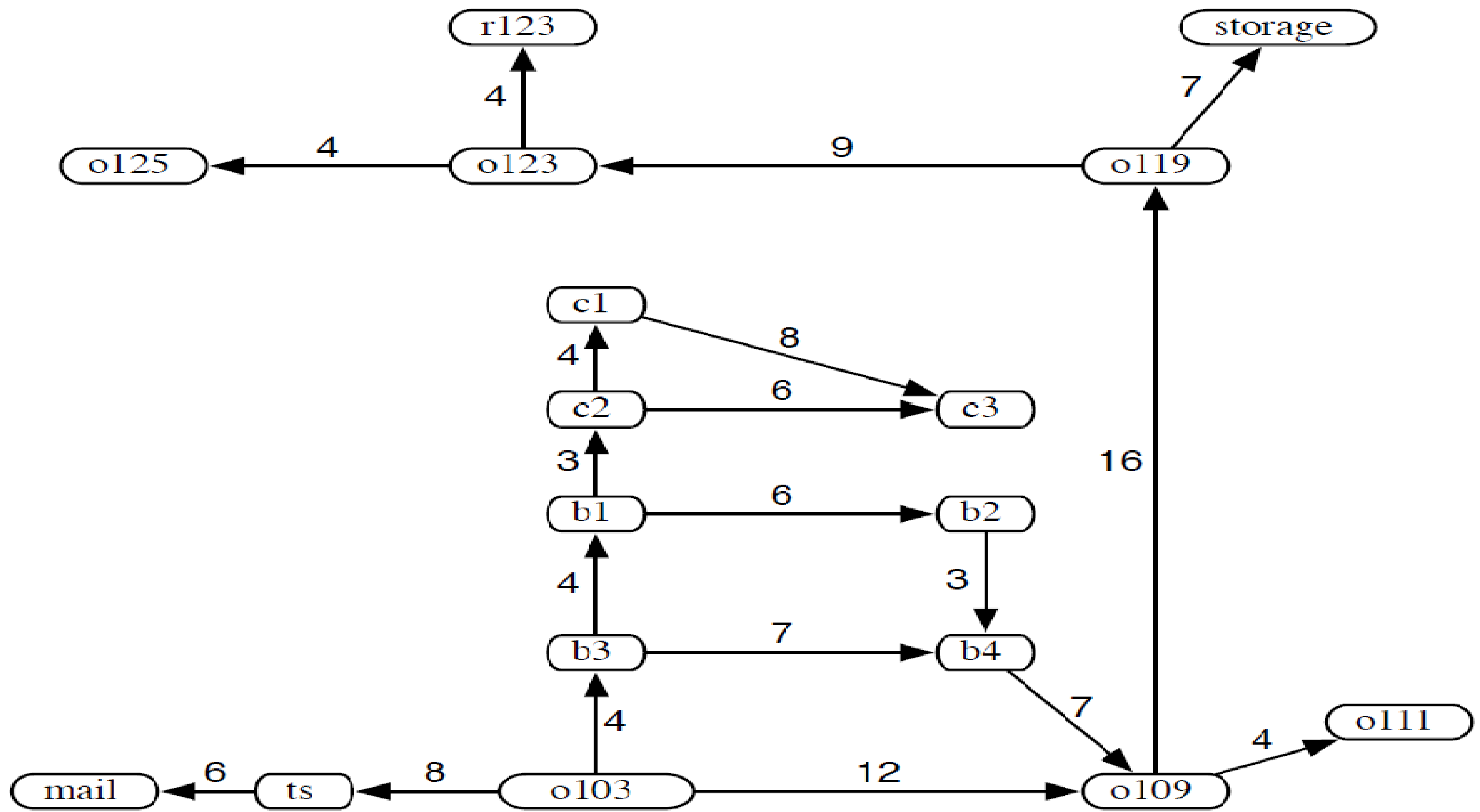
- A graph consists of a set N of nodes and a set A of ordered pairs of nodes, called arcs.
- Node n_2 is a neighbor of n_1 if there is an arc from n_1 to n_2. That is, if $\langle n_1, n_2 \rangle \in A$.
- A path is a sequence of nodes $\langle n_0, n_1, \ldots, n_k \rangle$ such that $\langle n_{i-1}, n_i \rangle \in A$.
- The length of path $\langle n_0, n_1, \ldots, n_k \rangle$ is k.
- Given a set of start nodes and goal nodes, a solution is a path from a start node to a goal node.

Example Problem for Delivery Robot

The robot wants to get from outside room 103 to the inside of room 123.



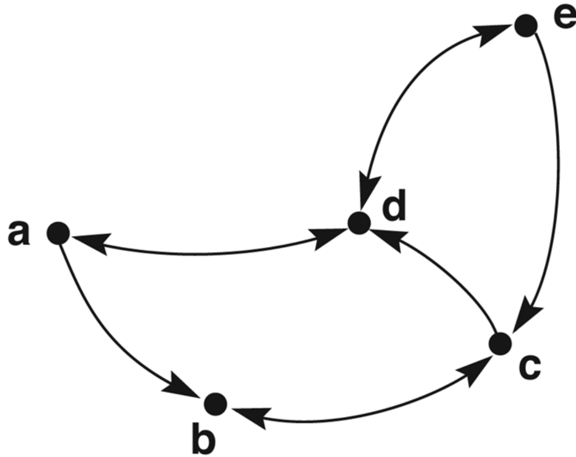
State-Space Graph for the Delivery Robot

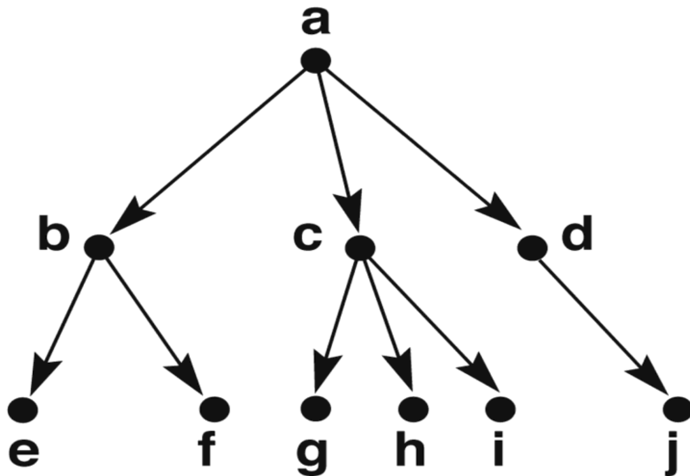


Graph Searching

- Generic search algorithm: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.
- Maintain a frontier of paths from the start node that have been explored.
- As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.
- The way in which the frontier is expanded defines the search strategy.

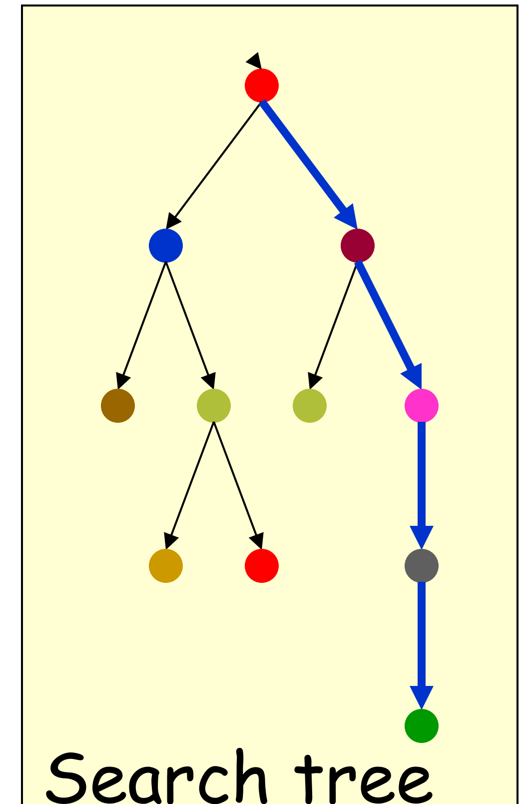
Problems with Graph Search



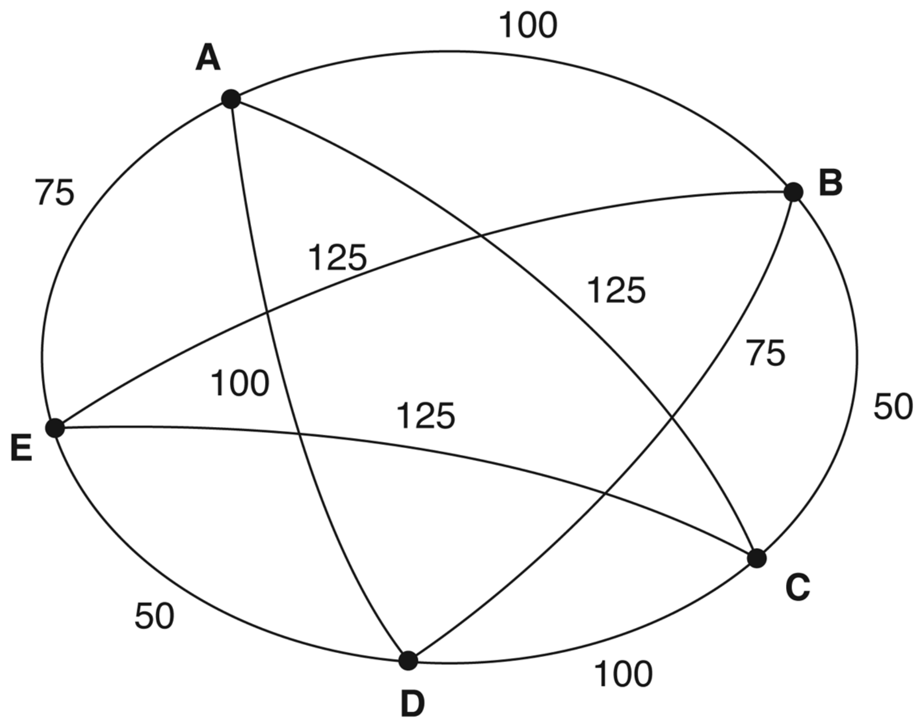


- Several paths to same nodes
  - Need to select best path according to needs of problem
- Cycles in path can prevent termination
  - Blind search without cycle check will never terminate (ex. see graph on the top)
  - There is no cycle problems with trees

State Space as a Search Tree

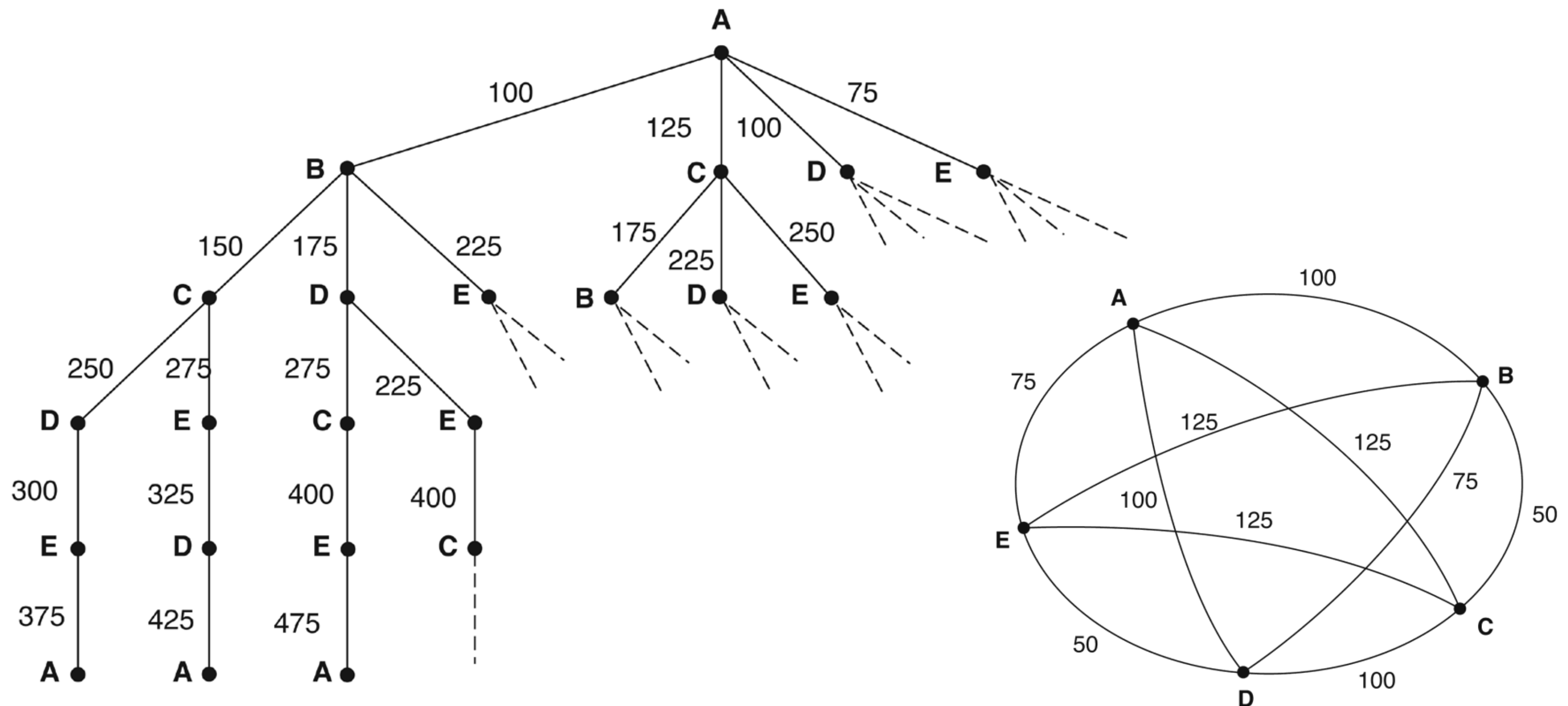


State Search of Travelling Salesperson



- Salesperson has
  - to visit 5 cities
  - must return home afterwards
- Goal: find shortest path for travel
  - minimize cost and/or time of travel
- Nodes represent cities
- Weighted arcs represent cost of travel
- Simplification: salesperson lives in city **A** and will return there
- Goal is to go from **A** to **A** with minimal cost

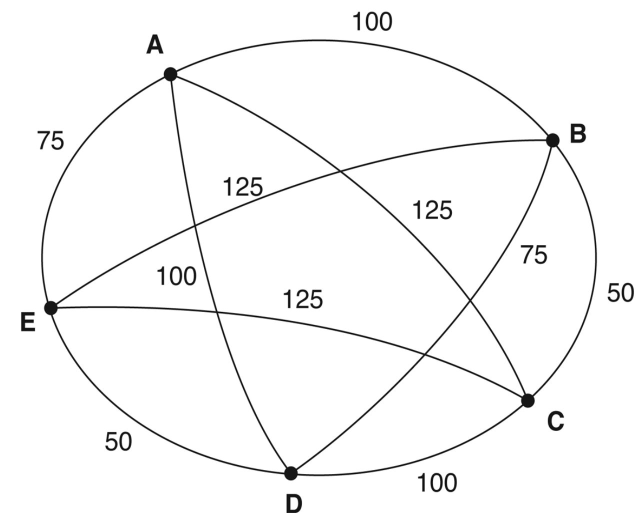
Search of TSP problem



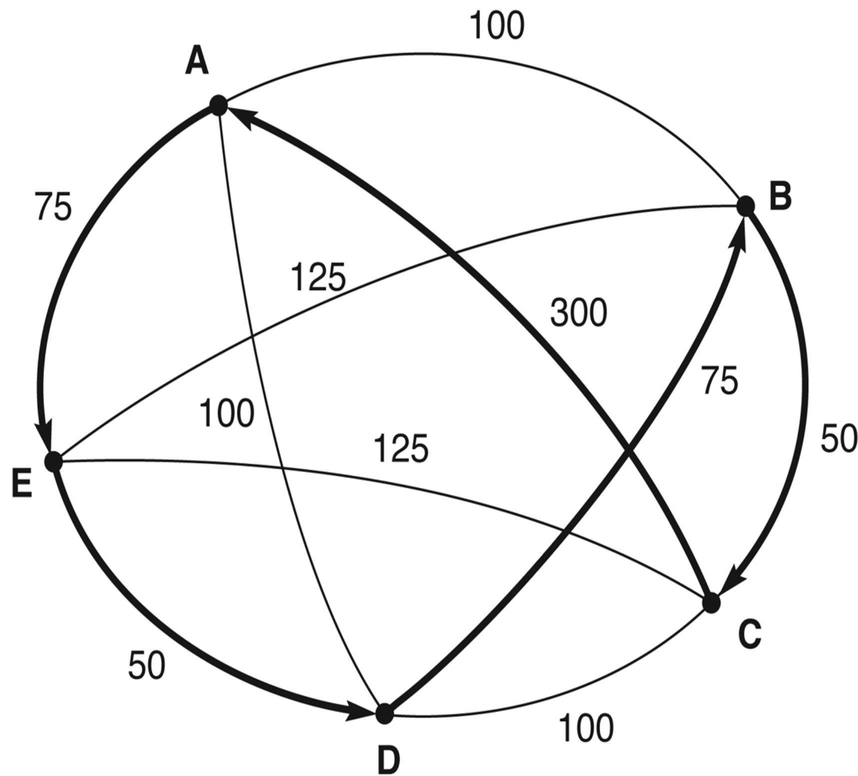
Each arc is marked with the total weight of all paths from the start node (**A**) to its endpoint (**A**)

Size of Search Space

- Complexity: (**N** - 1)! with **N** the number of cities
  - interesting problem size: 50 cities
  - simple exhaustive search becomes intractable
- Several techniques can reduce complexity
  - --> heuristic search (see later)
  - ex: Nearest neighbour heuristic
    - go to closest unvisited city
    - highly efficient since only one path is tried
    - might fail (see next example)

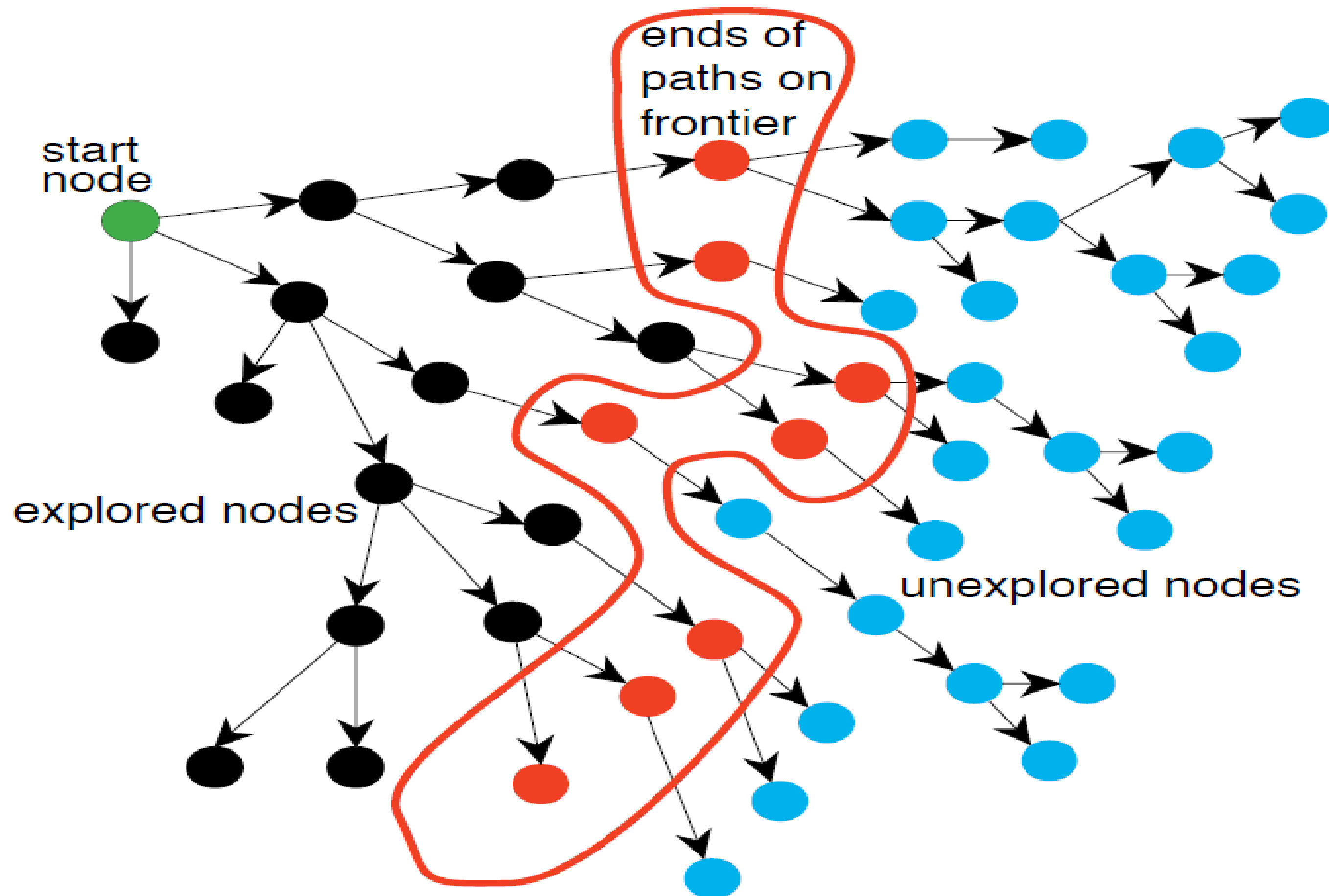


Nearest Neighbour on TSP



- nearest neighbour path in bold
- this path (A, E, D, B, C, A) has a cost of 550
- but it is <u>not</u> the shortest path
- the high cost of (C, A) defeated the heuristic "nearest neighbour"

Problem Solving by Graph Searching



Basic Graph Search Algorithm

Input: a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node.
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$;
while *frontier* is not empty:
    select and **remove** path $\langle n_0, \ldots, n_k \rangle$ from *frontier*;
    if $goal(n_k)$
        return $\langle n_0, \ldots, n_k \rangle$;
    for every neighbor n of n_k
        add $\langle n_0, \ldots, n_k, n \rangle$ to *frontier*;
end while

- Which value is selected from the frontier at each stage defines the search strategy.
- The neighbors define the graph.
- *goal* defines what is a solution.
- If more than one answer is required, the search can continue from the return.

Basic algorithm

- **Open nodes:** States generated but not yet expanded
- **Closed nodes:** States already expanded
- We will need a data structure to store the open nodes
- The policy of insertion of the data structure will determine the search strategy
- If we are searching in a graph it could be necessary to detect repeated states (this means we also need a data structure to store the closed nodes). It is worth to have it if the number of different states is small compared to the number of paths

Characteristics of the algorithms

- Characteristics:
    - **Completeness:** Is it guaranteed to find a solution if there is one?
    - **Temporal Complexity:** How long does it take to find a solution?
    - **Spatial Complexity:** How much memory is needed?
    - **Optimality:** does it find the optimal solution?

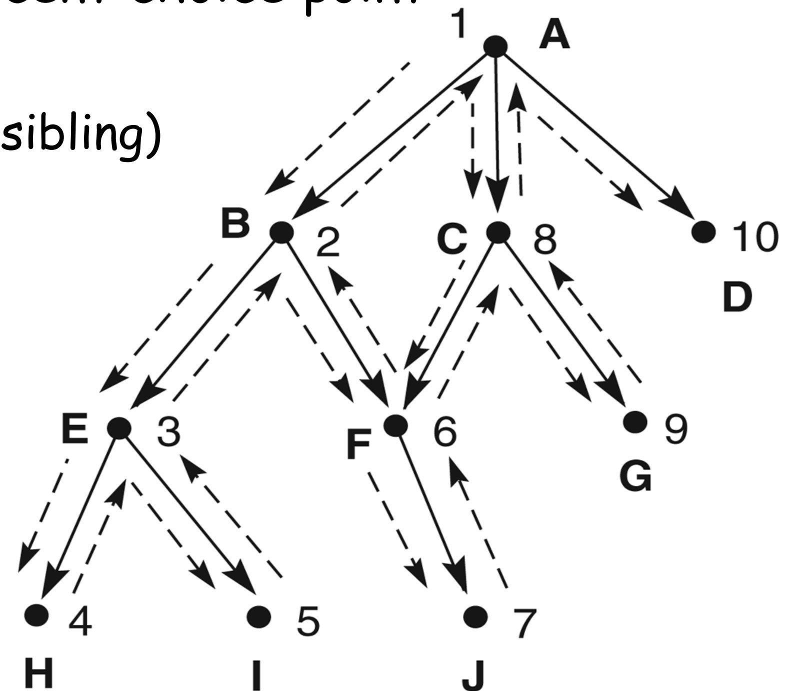
Implementing Graph Search

- Problem: Search a given graph and construct path to goal
  - must try all possible paths
  - must not get stuck in cycles

- Solution: Backtrack Algorithm
- Idea
  - keep track of visited nodes
  - apply recursion to get out of dead ends

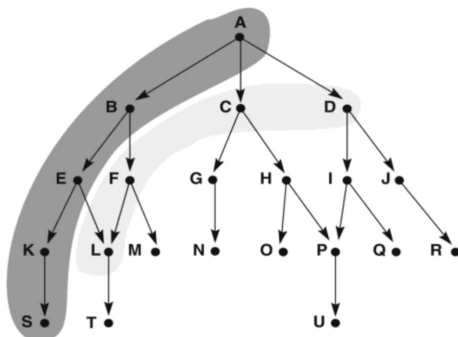
Idea of Backtracking

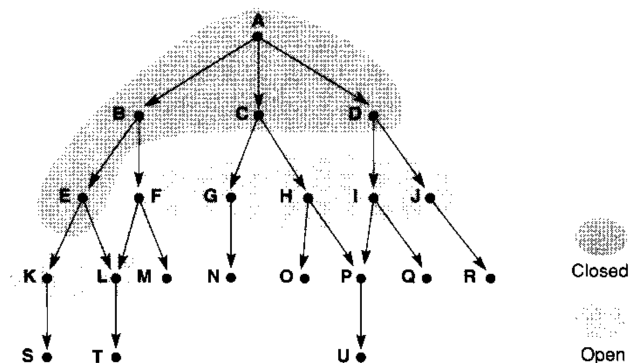
- Exhaustive search is required
- If a dead end is found:
  - dead end: state or path does not match solution criteria
  - algorithm backtracks to most recent choice point
  - alternative path possible?
    - yes: explore this path (another sibling)
    - no: continue backtracking
- Termination
  - algorithm continues until either
  - a solution/goal is found **or**
  - state space is exhausted



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





Generic Search Algorithm

1. Initialize the open list with the initial node s_o (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 path from s to s_o
   d) 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/open lists (to avoid cycles)

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

Today

- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
  - Informed search
    - Best-First
    - A*

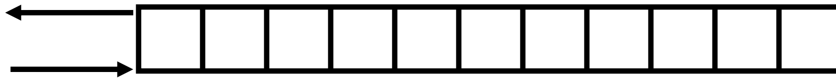
- Summary

Search strategies

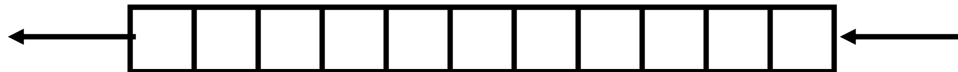
- Uninformed search
  - We systematically explore the alternatives
  - aka: systematic/exhaustive/blind/brute force search
    - Breath-first
    - Depth-first
    - Uniform-cost
    - Depth-limited search
    - Iterative deepening search
    - Bidirectional search
    - …
- Informed search (heuristic search)
  - We try to choose smartly
    - Best-First
    - A*
    - …

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.



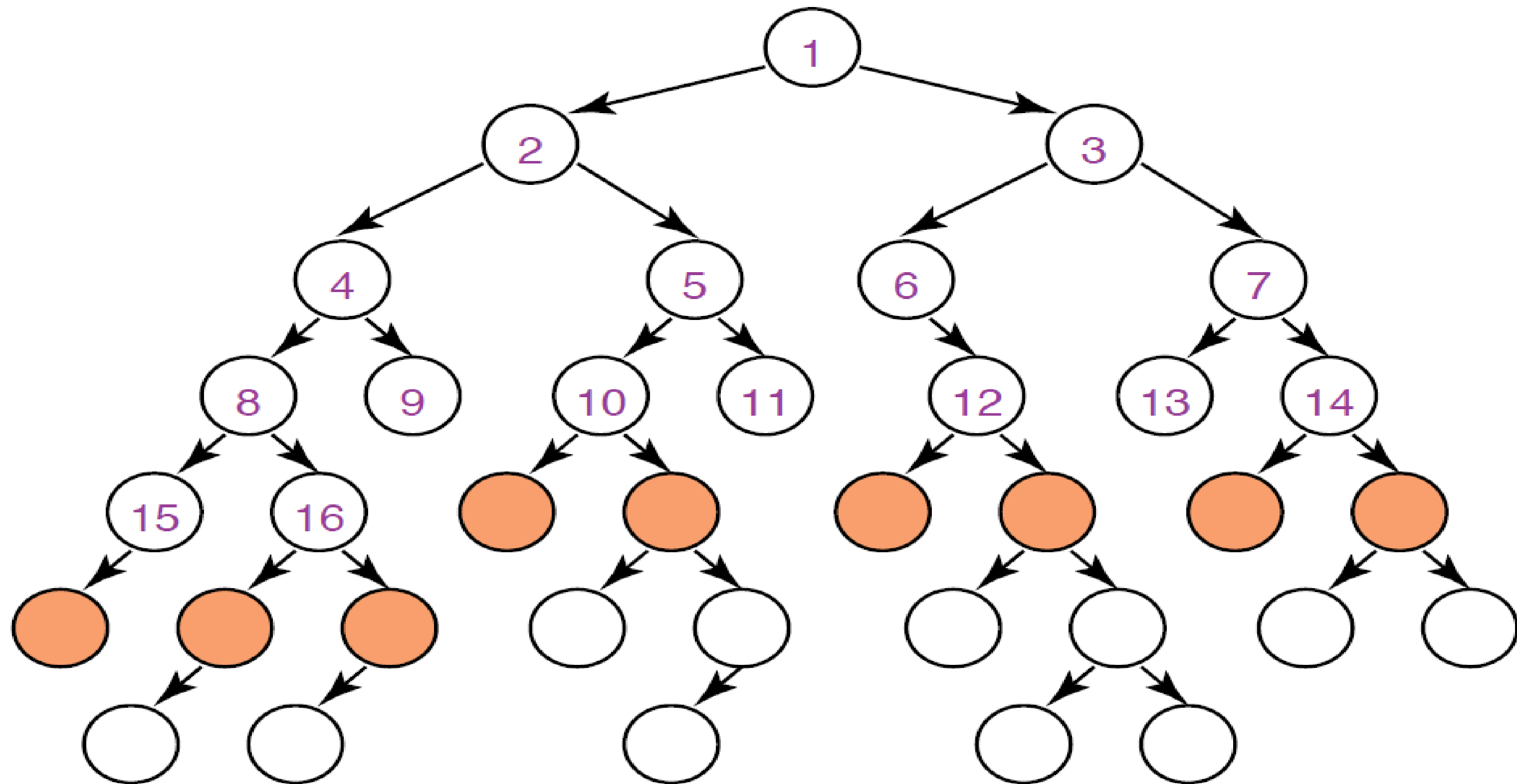
Today

- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed search
    - Best-First
    - A*
- Summary

Breadth-first Search

- Breadth-first search treats the frontier as a queue.
- It always selects one of the earliest elements added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots, p_r]$:
  - p_1 is selected. Its neighbors are added to the end of the queue, after p_r.
  - p_2 is selected next.

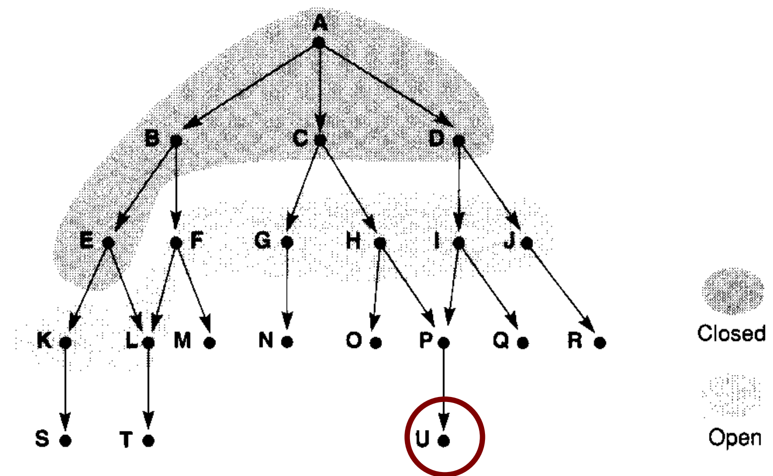
Illustrative Graph - Breadth-first Search



Data Structures

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

Closed = [A, B, C, D, E]
Open = [K, L, F, G, H, I, J]

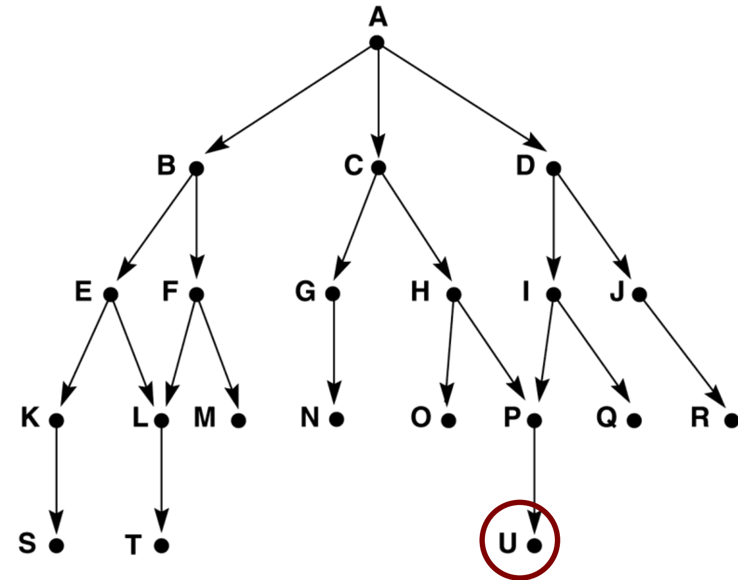


Breadth-First Search

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

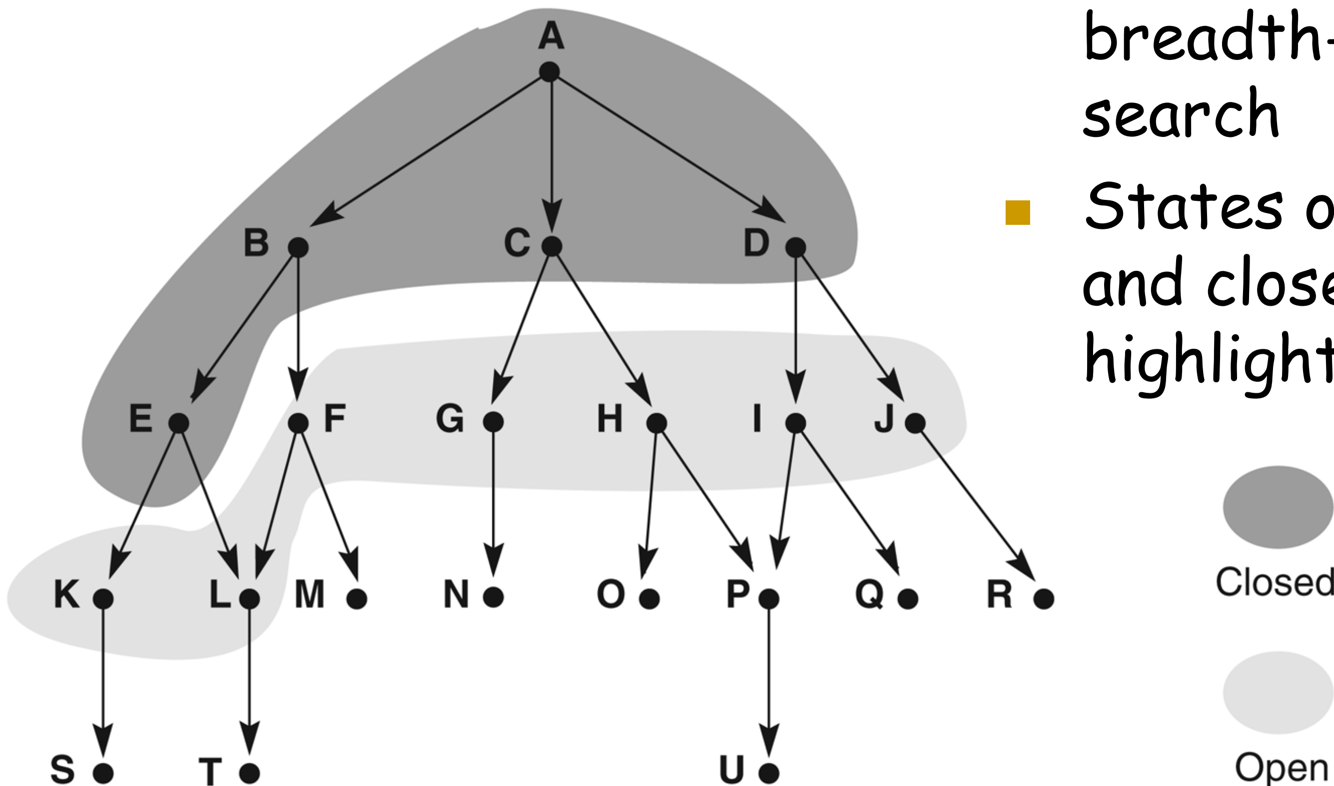
Breadth-First Search Example

- BFS: (open is a queue)



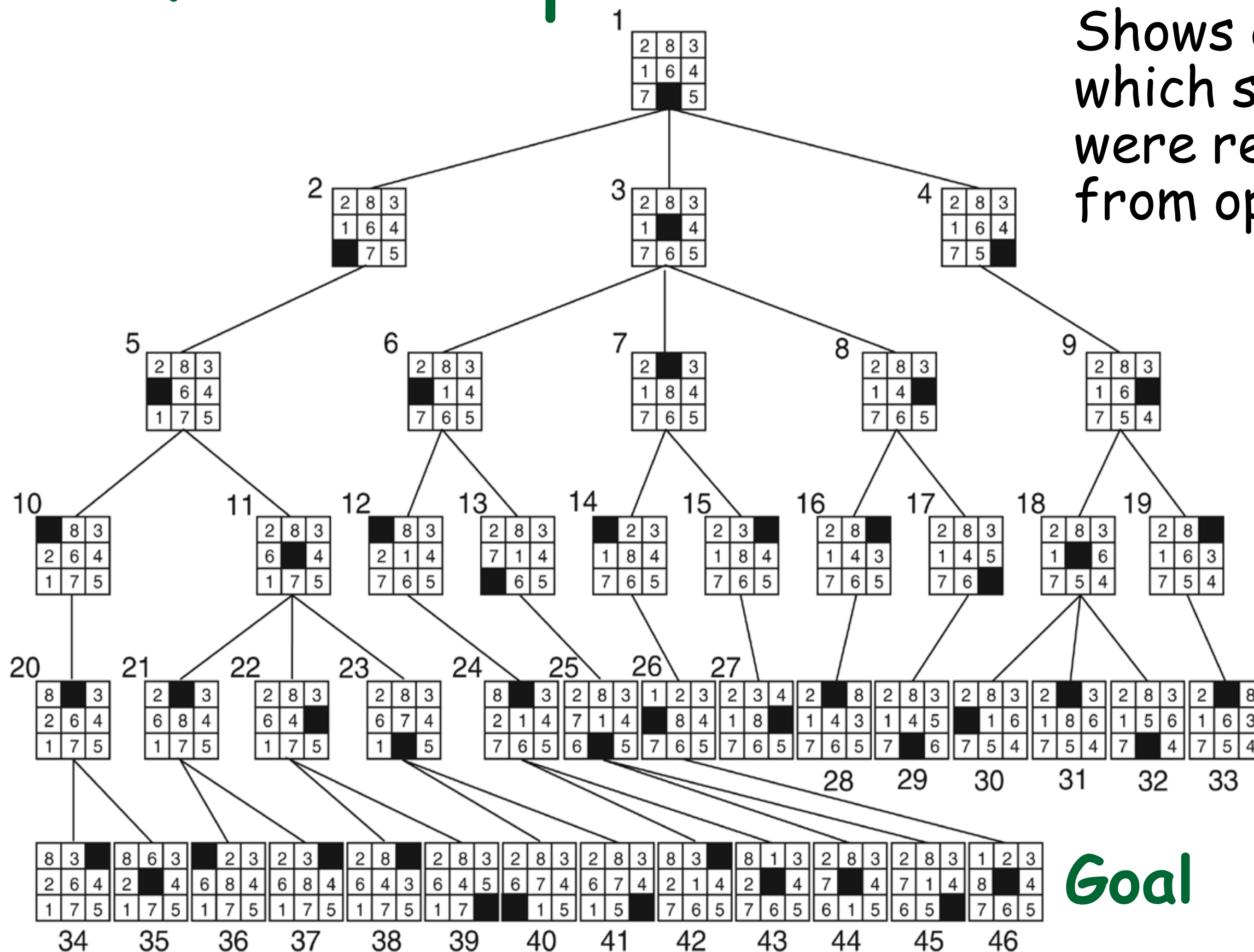
1. **open = [A]; closed = [ ]**
2. **open = [B,C,D]; closed = [A]**
3. **open = [C,D,E,F]; closed = [B,A]**
4. **open = [D,E,F,G,H]; closed = [C,B,A]**
5. **open = [E,F,G,H,I,J]; closed = [D,C,B,A]**
6. **open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]**
7. **open = [G,H,I,J,K,L,M]** (as L is already on open); **closed = [F,E,D,C,B,A]**
8. **open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]**
9. 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

BFS of the 8-puzzle

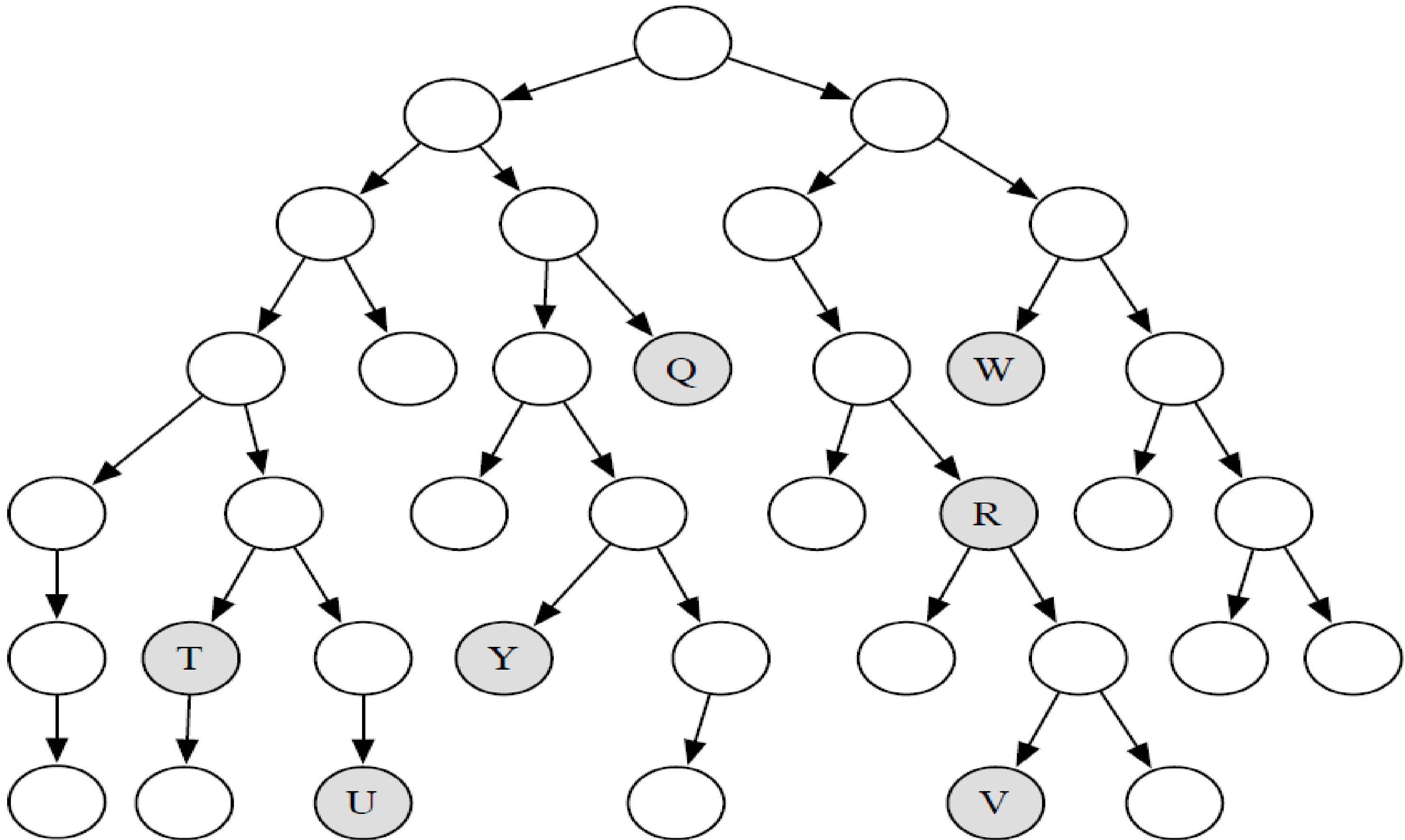


Shows order in which states were removed from open

Breadth-first search

- Nodes are generated and visited level wise
- The structure to store the open nodes is a queue (FIFO)
- A node is visited when all the nodes from the previous level and its previous siblings in generation order have been visited
- Characteristics:
  - Completeness: The algorithm always find a solution
  - Temporal Complexity: Bounded by an exponential function of the branching factor over the depth of the solution $O(b^d)$
  - Spatial Complexity: Bounded by an exponential function of the branching factor over the depth of the solution $O(b^d)$
  - Optimality: The solution is optimal on the number of levels from the root of the search tree

Which shaded goal will a breadth-first search find first?



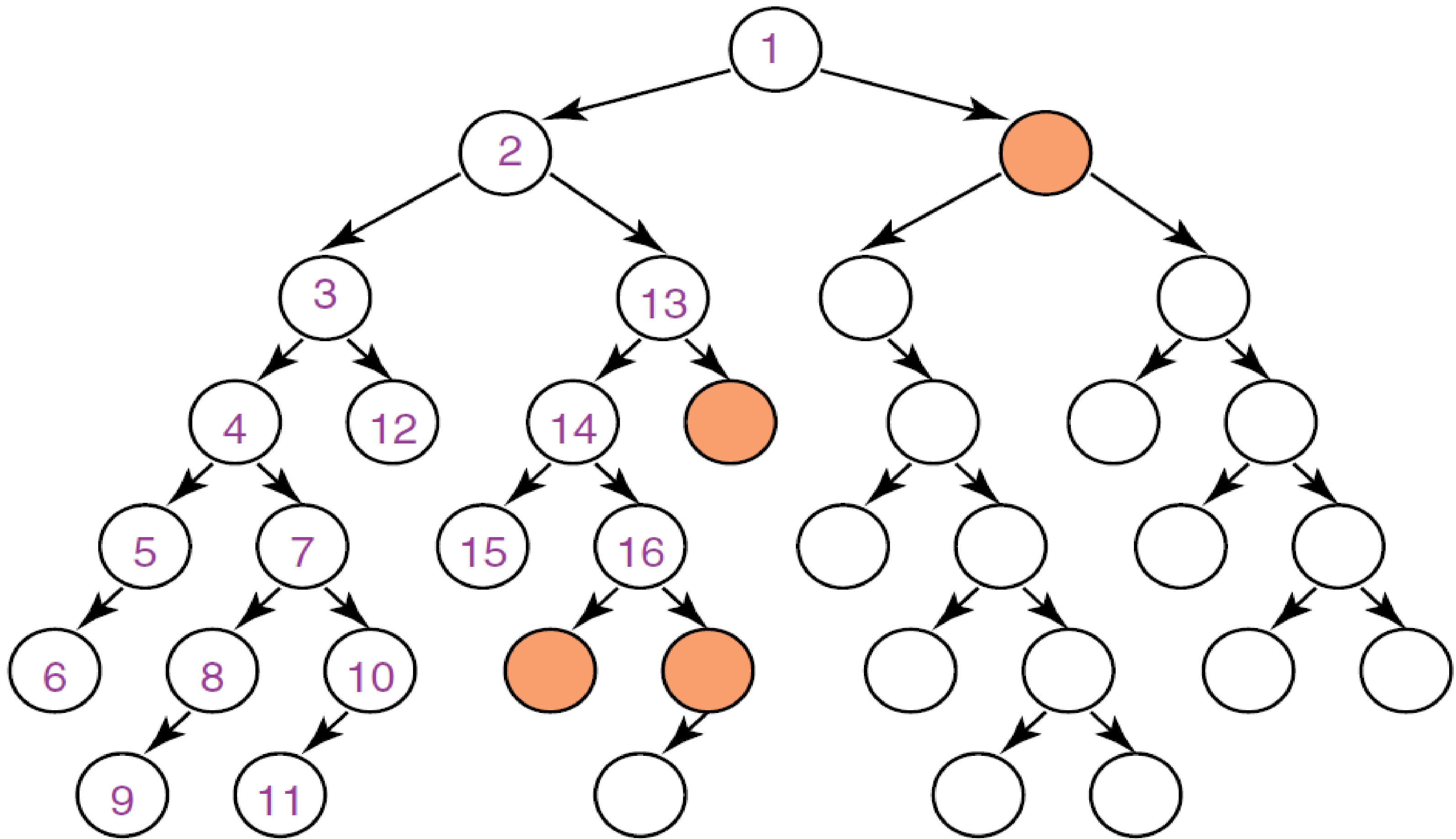
Today

- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-f
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed search
    - Best-First
    - A*
- Summary

Depth-first Search

- Depth-first search treats the frontier as a stack
- It always selects one of the last elements added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots]$
  - p_1 is selected. Paths that extend p_1 are added to the front of the stack (in front of p_2).
  - p_2 is only selected when all paths from p_1 have been explored.

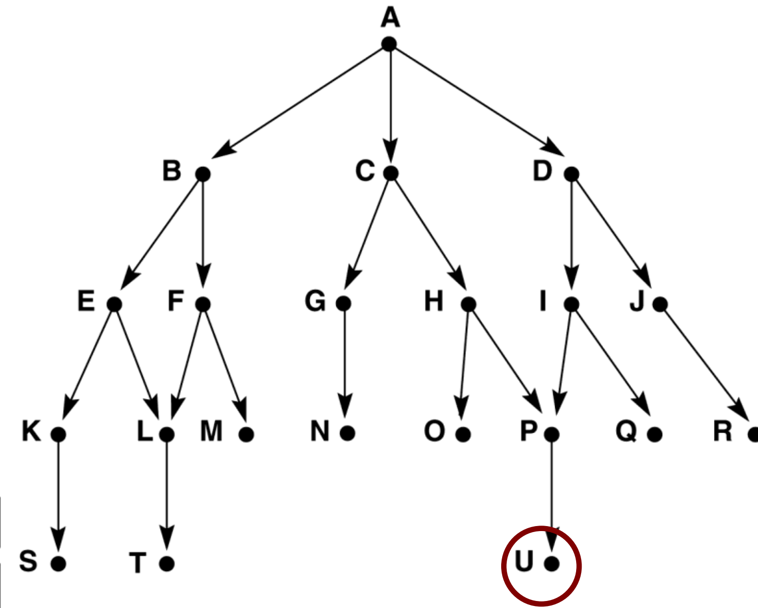
Illustrative Graph --Depth-first Search



Function Depth-First Search

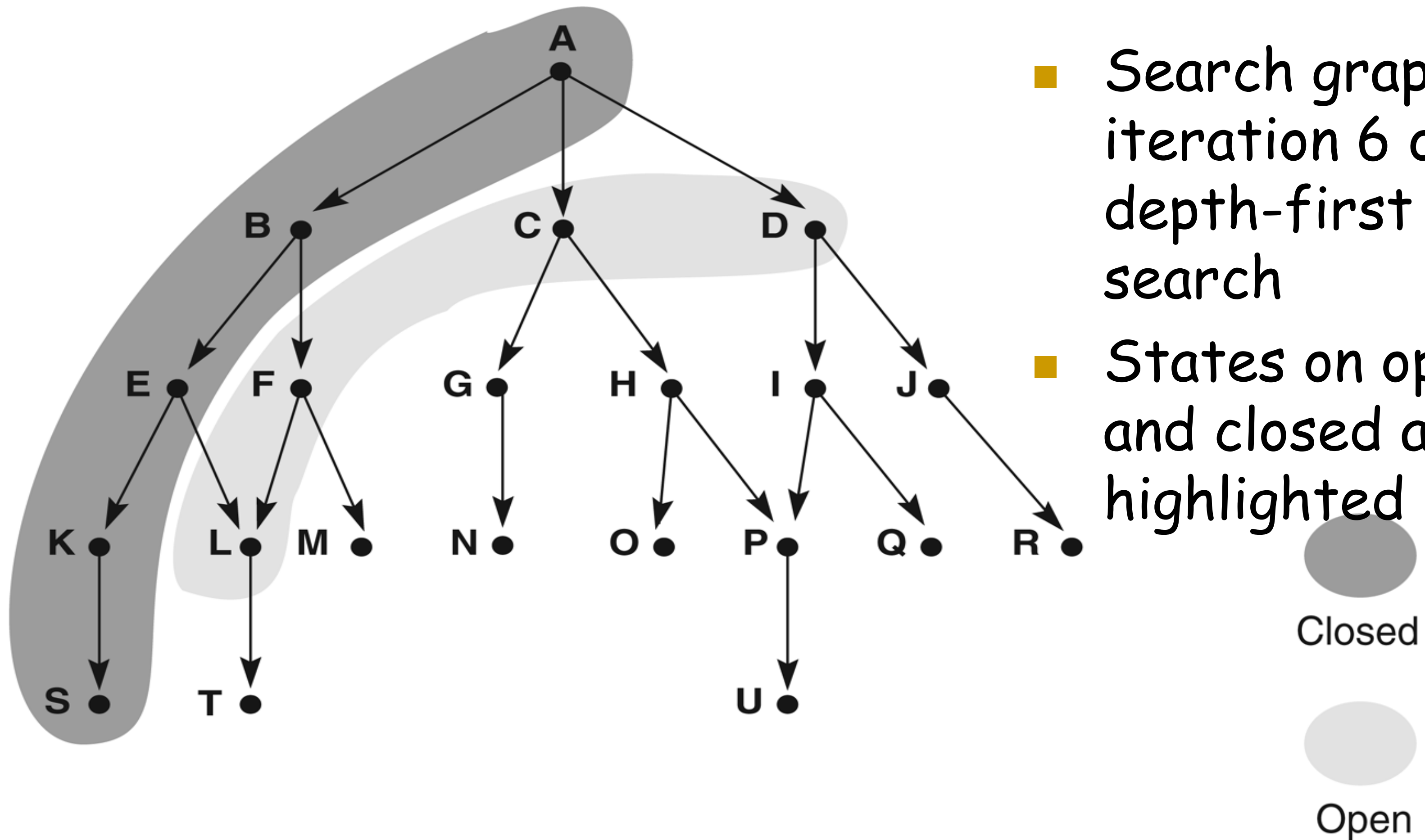
```
begin
   open := [Start];                                             % initialize
   closed := [ ];
a) while open ≠ [ ] do                                         % states remain
      begin
   b) remove leftmost state from open, call it X;
   c) if X is a goal then return SUCCESS              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

Trace of Depth-First Search



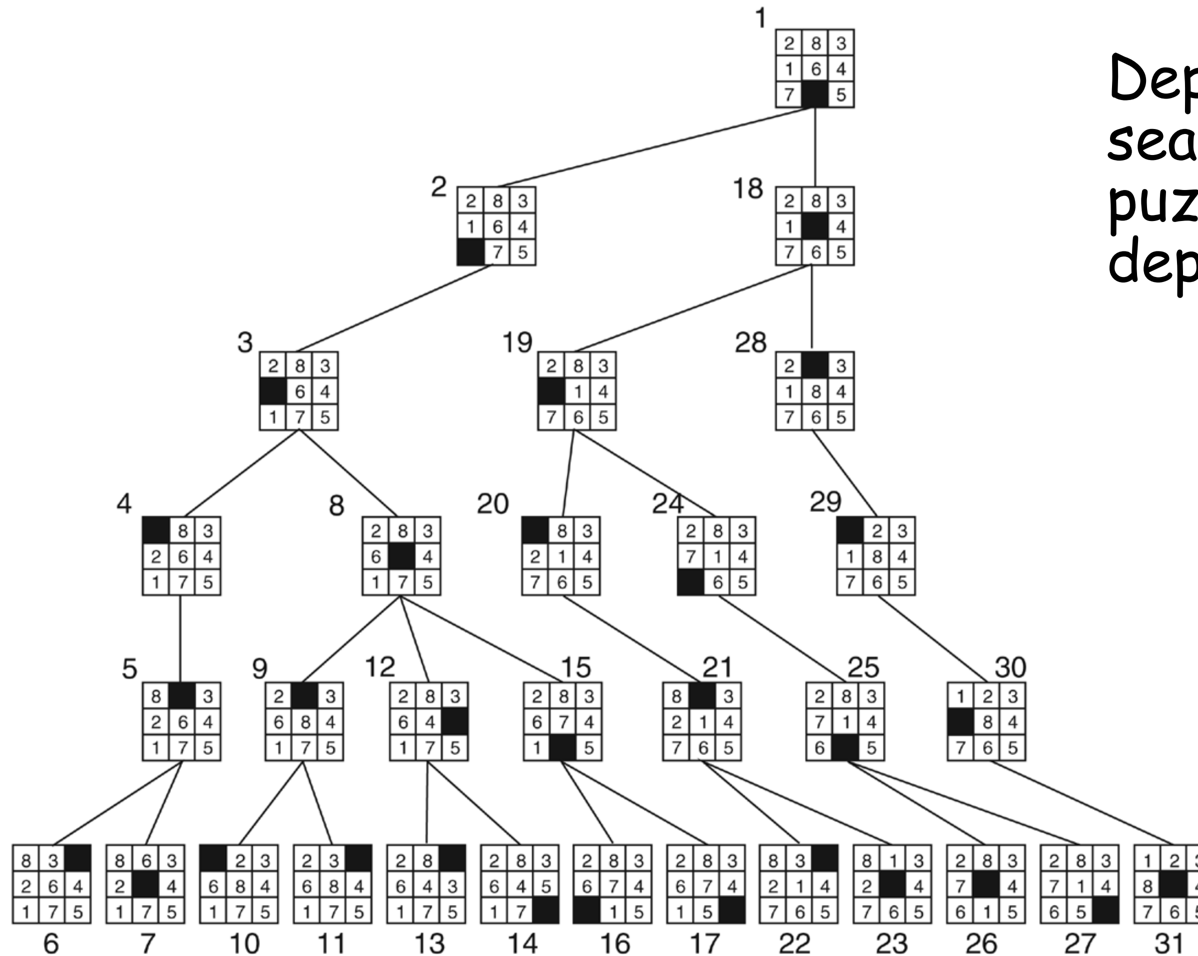
1. **open = [A]; closed = [ ]**
2. **open = [B,C,D]; closed = [A]**
3. **open = [E,F,C,D]; closed = [B,A]**
4. **open = [K,L,F,C,D]; closed = [E,B,A]**
5. **open = [S,L,F,C,D]; closed = [K,E,B,A]**
6. **open = [L,F,C,D]; closed = [S,K,E,B,A]**
7. **open = [T,F,C,D]; closed = [L,S,K,E,B,A]**
8. **open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

DFS on the 8-puzzle



Goal

Depth-first search of the 8-puzzle with a depth bound of 5

Depth-first vs. Breadth-first

- Breadth-first:
  - Complete: always finds a solution if it exists
  - Optimal: always finds shortest path

  But:
  - inefficient if branching factor **B** is very high
  - memory requirements high -- exponential space for states required: $\mathbf{B^n}$
- Depth-first:
  - Not complete (ex. may get stuck in an infinite branch)
  - Not optimal (will not find the shortest path)

  But:
  - Requires less memory -- only memory for states of one path needed: $\mathbf{B \times n}$
  - May find the solution without examining much of the search space
  - Efficient if solution path is known to be long

Depth-first vs. Breadth-first

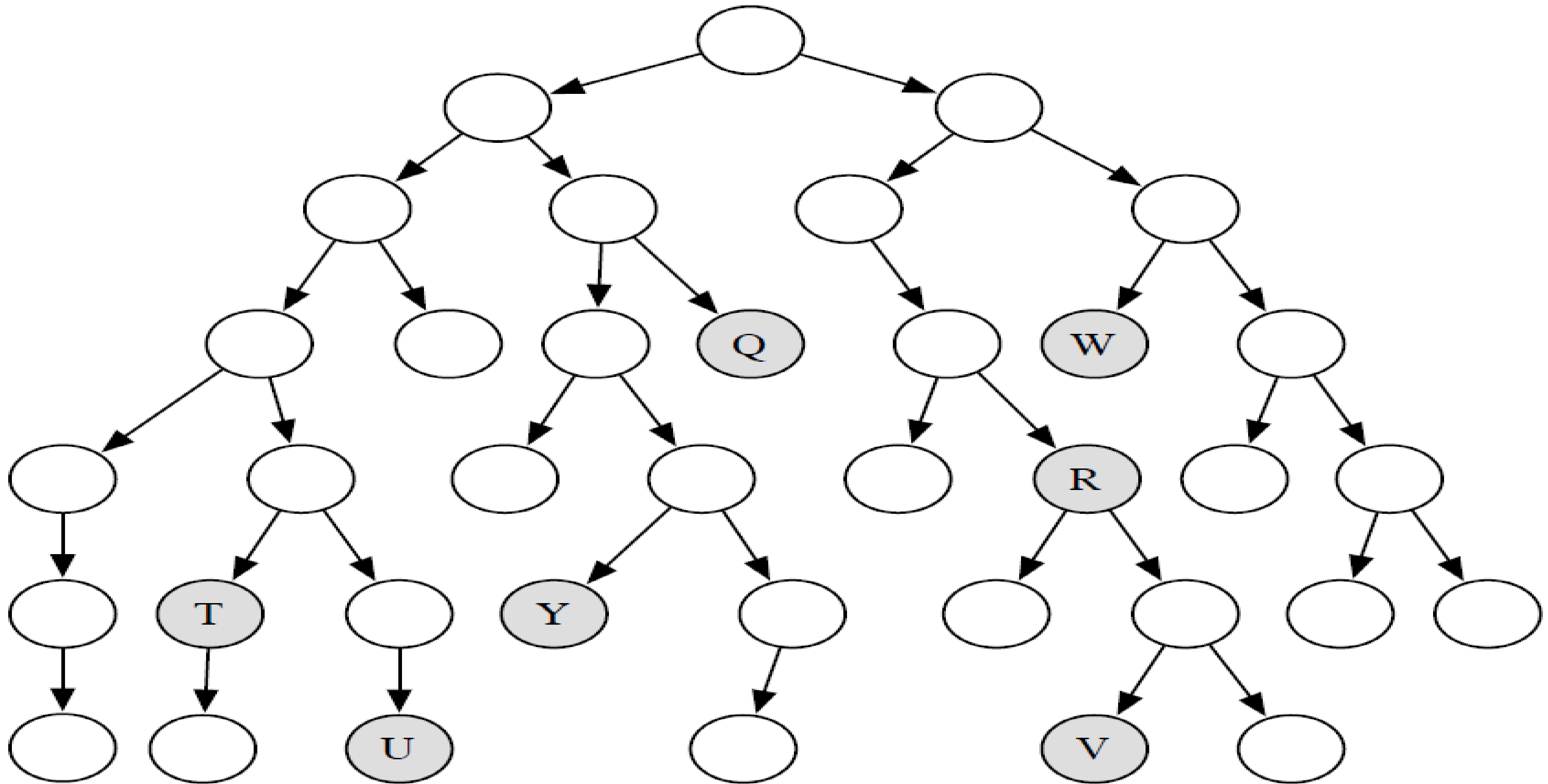
- If you want the shortest path --> breadth first
- If memory is a problem --> depth first

- Many variants of breadth & depth first:
  - Depth-limited Search
  - Iterative deepening search
  - Uniform-cost search
  - Bidirectional search

Depth-first search

- The nodes are visited and generated always looking for the deepest node
- The structure to store the open nodes is a stack (LIFO)
- In order to assure that the algorithm stops, a maximum depth of exploration is used (to avoid infinite paths)
- Characteristics
  - Completeness: The algorithm finds a solution if a maximum depth of exploration is used and there is a solution above this limit
  - Temporal Complexity: Bounded by an exponential function of the branching factor over the maximum depth $O(b^m)$
  - Spatial Complexity: If duplicated nodes are not controlled, it is bounded by an linear function of the branching factor times the maximum depth $O(b \times m)$. If duplicated nodes are treated the space is the same than breadth-first search. If we use a recursive implementation of the algorithm (no stack needed), it is bounded by an linear function of the maximum depth $O(m)$
  - Optimality: There is no guarantee that the solution found is optimal

Which shaded goal will a depth-first search find first?



Today

- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Dep
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed search
    - Best-First
    - 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)

Depth-first limited search

```
Procedure: Depth-first limited search (limit: integer)
St_open.add(initial state)
Current ← St_open.first()
while not is_goal?(Current) and not St_open.empty?() do
    St_open.delete_first()
    St_closed.add(Current)
    if depth(current) ≤ limit then
        Successors ← generate_successors (Current)
        Successors ← treat_duplicated (Successors, St_closed, St_open)
        St_open.add(Successors)
    end
    Current ← St_open.first()
end
```

- The structure for the open nodes is a stack
- A node is not expanded when its depth is greater than the depth limit
- This guaranties that the algorithm stops
- If duplicated nodes are treated we lose the space complexity advantage

Today

- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed search
    - Best-First
    - A*
- Summary

Iterative Deepening search (ID)

- Combines the space complexity of Depth-first search and the optimality of solutions of Breadth-first search
- The algorithm performs successive depth-first searches with limited depth that is increased each iteration
- This strategy gives a behaviour similar to breadth-first search but without its spatial complexity because each exploration is depth-first, although all the nodes are generated each iteration
- This strategy allows to avoid the cases when depth-first algorithm does not end (infinite paths)
- The first iteration the maximum depth is 1 and this value will be incremented until a solution is found
- In order to guarantee that the algorithm ends if there is no solution a depth limit can be imposed

Iterative Deepening

- Completeness: The algorithm always finds a solution (if it exists)
- Temporal Complexity: Same as Breadth-first search. To regenerate the nodes each iteration only adds a constant factor to the cost $O(b^d)$
- Spatial Complexity: Same as Depth-first search
- Optimality: Optimal as Breadth-first search

Iterative Deepening

Compromise between BFS and DFS:

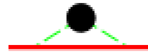
- use depth-first search, but
- with a maximum depth before going to next level

- Repeats depth first search with gradually increasing depth limits
  - Requires little memory (fundamentally, it's a depth first)
  - Finds the shortest path (limited depth)

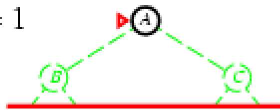
- Preferred search method when there is a large search space and the depth of the solution is unknown

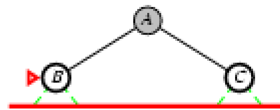
Iterative Deepening: Example

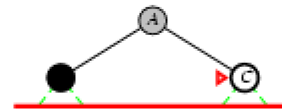
Limit = 0

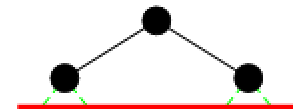


Iterative Deepening: Example

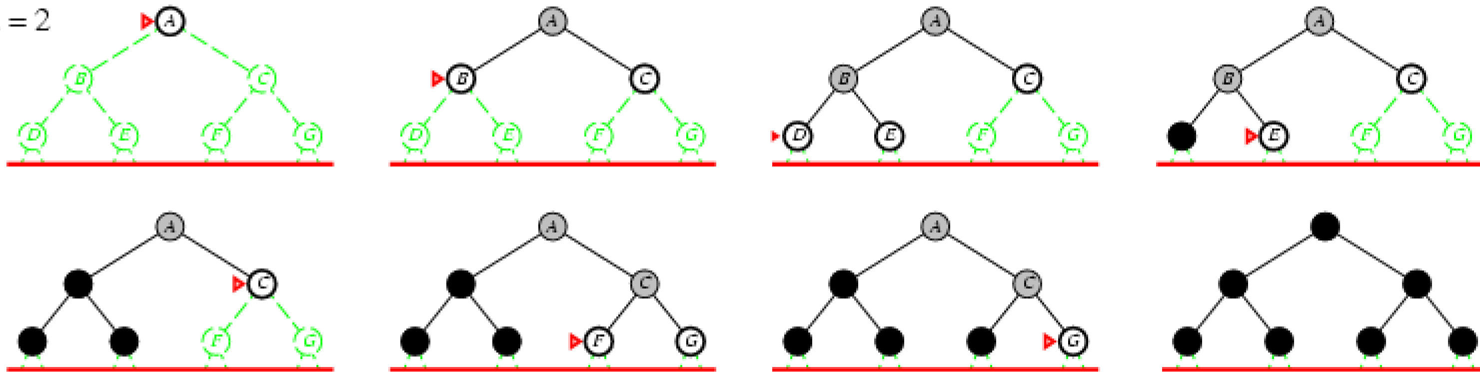




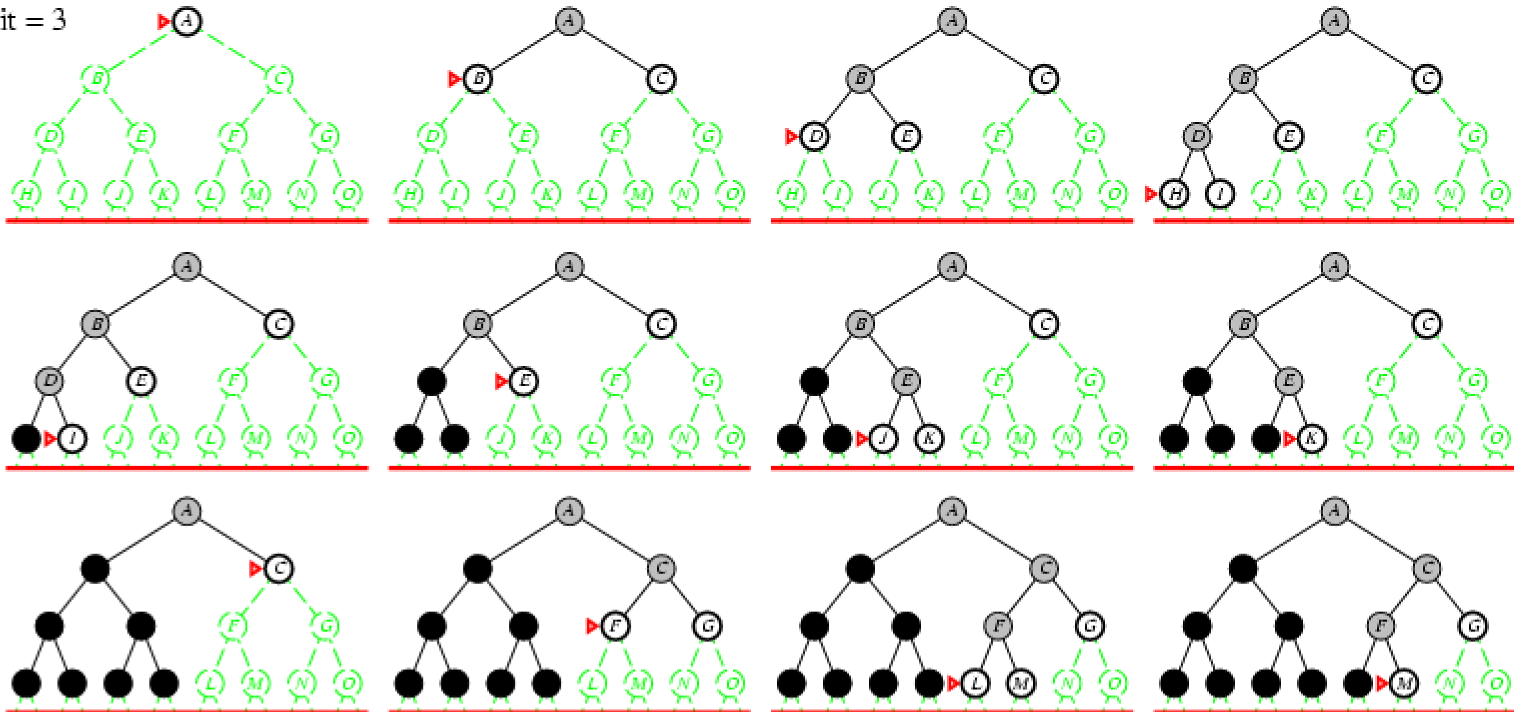




Iterative Deepening: Example



Iterative Deepening: Example



Today

- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed search
    - Best-First
    - A*
- Summary

Lowest-cost-first Search

- Sometimes there are costs associated with arcs. The cost of a path is the sum of the costs of its arcs.

$$cost(\langle n_0, \ldots, n_k \rangle) = \sum_{i=1}^{k} |\langle n_{i-1}, n_i \rangle|$$

  An optimal solution is one with minimum cost.
- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost.
- The frontier is a priority queue ordered by path cost.
- It finds a least-cost path to a goal node.
- When arc costs are equal $\Longrightarrow$ breadth-first search.

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added	No	No	Linear
Breadth-first	First node added	Yes	No	Exp
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp

Complete — if there a path to a goal, it can find one, even on infinite graphs.

Halts — on finite graph (perhaps with cycles).

Space — as a function of the length of current path

Today

- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed search
    - Best-First
    - 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
- 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 (heuristic)

- 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 to a function that gives the true evaluation of the node n

Why do we need heuristics?

1. A problem may not have an exact solution
   - ex: ambiguity in the problem statement or the data
2. The computational cost of finding the exact solution is prohibitive
   - ex: search space is too large
   - ex: state space representation 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"

Heuristic Search

- Idea: don't ignore the goal when selecting paths.
- Often there is extra knowledge that can be used to guide the search: heuristics.
- $h(n)$ is an estimate of the cost of the shortest path from node n to a goal node.
- $h(n)$ needs to be efficient to compute.
- h can be extended to paths: $h(\langle n_0, \ldots, n_k \rangle) = h(n_k)$.
- $h(n)$ is an underestimate if there is no path from n to a goal with cost less than $h(n)$.
- An admissible heuristic is a nonnegative heuristic function that is an underestimate of the actual cost of a path to a goal.

Example Heuristic Functions

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance from n to the closest goal.
- If the nodes are locations and cost is time, we can use the distance to a goal divided by the maximum speed.
- If the goal is to collect all of the coins and not run out of fuel, the cost is an estimate of how many steps it will take to collect the rest of the coins, refuel when necessary, and return to goal position.
- A heuristic function can be found by solving a simpler (less constrained) version of the problem.

Today

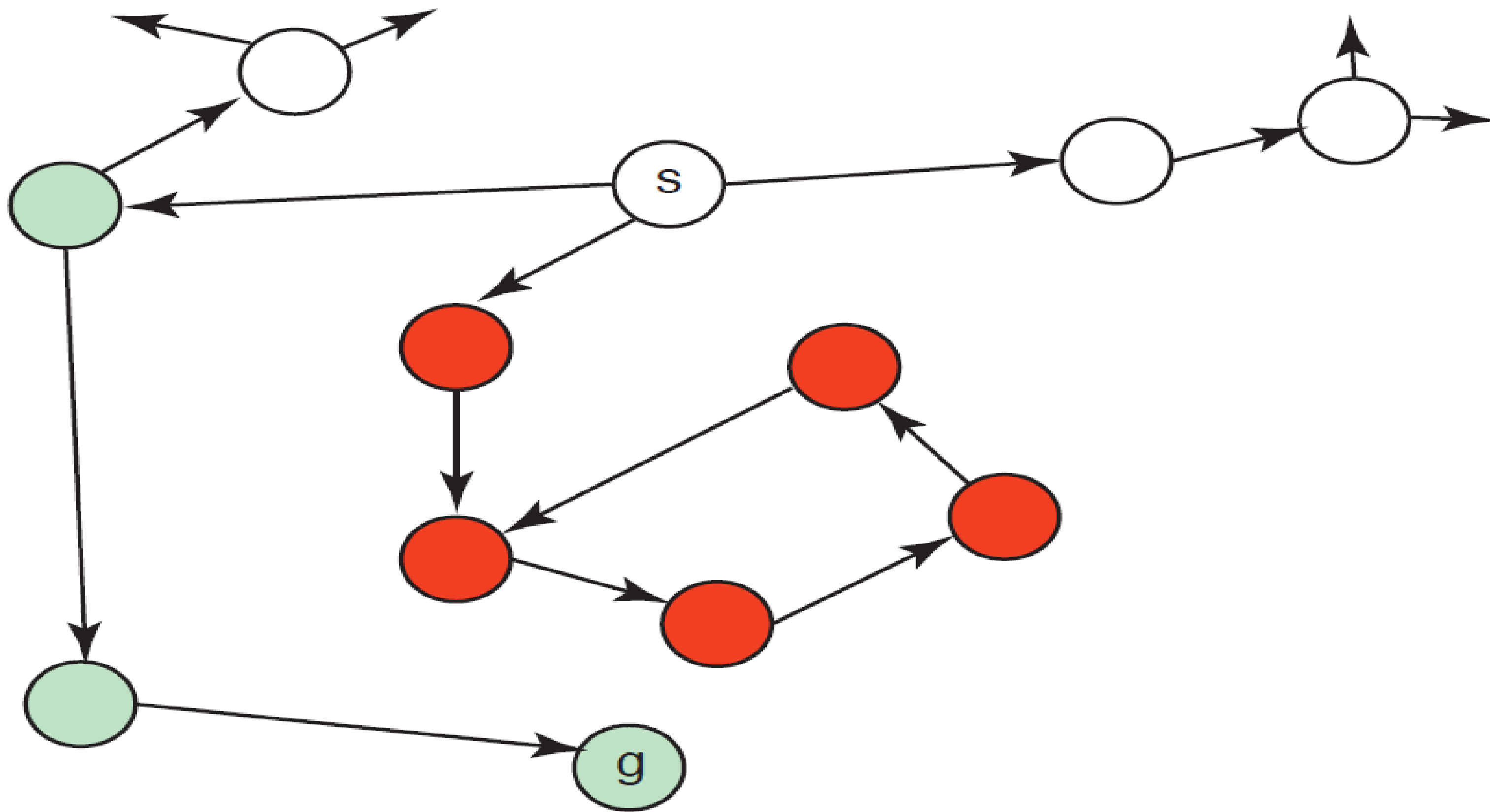
- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed sea
    - Best-First
    - A*

- Summary

Best-first Search

- Idea: select the path whose end is closest to a goal according to the heuristic function.
- Best-first search selects a path on the frontier with minimal h-value.
- It treats the frontier as a priority queue ordered by h.

Illustrative Graph ---Best-first Search



Complexity of Best-first Search

- Does best-first search guarantee to find the shortest path or the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

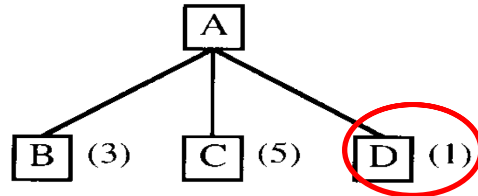
Best-First Search

- Best-first search:
  - Insert nodes in *open* list so that the nodes are sorted in best (ascending/descending) **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

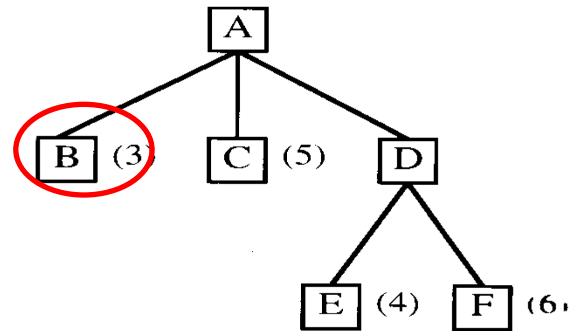
Best-First: Example

Step 1

Step 2

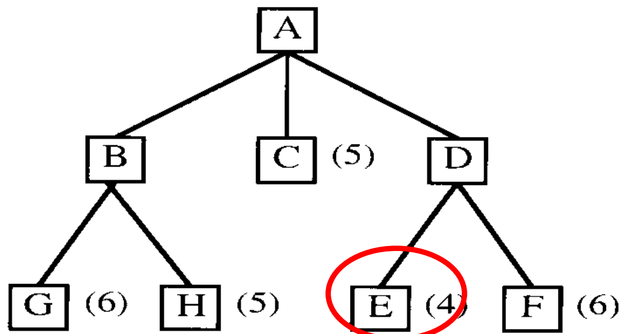


Step 3

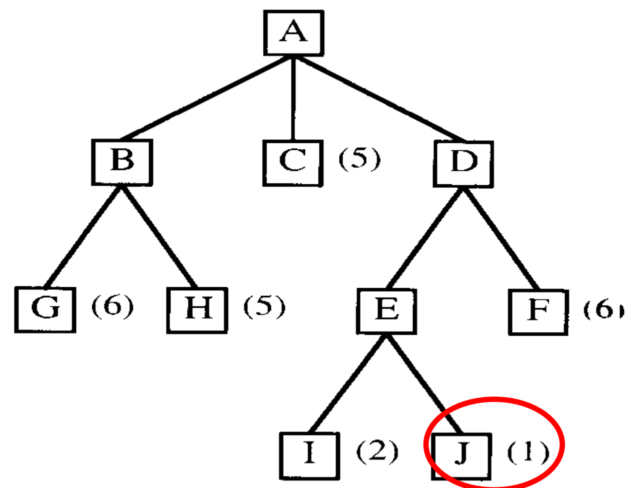


- Lower h(n) is better

Step 4



Step 5

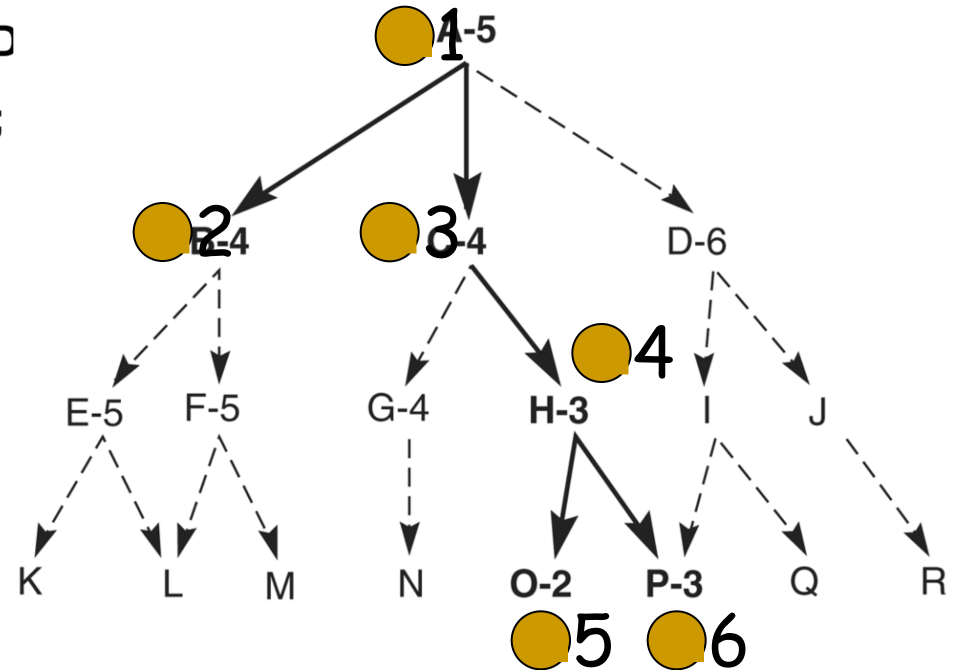


Notes on Best-first

- If you have a good evaluation function, 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)

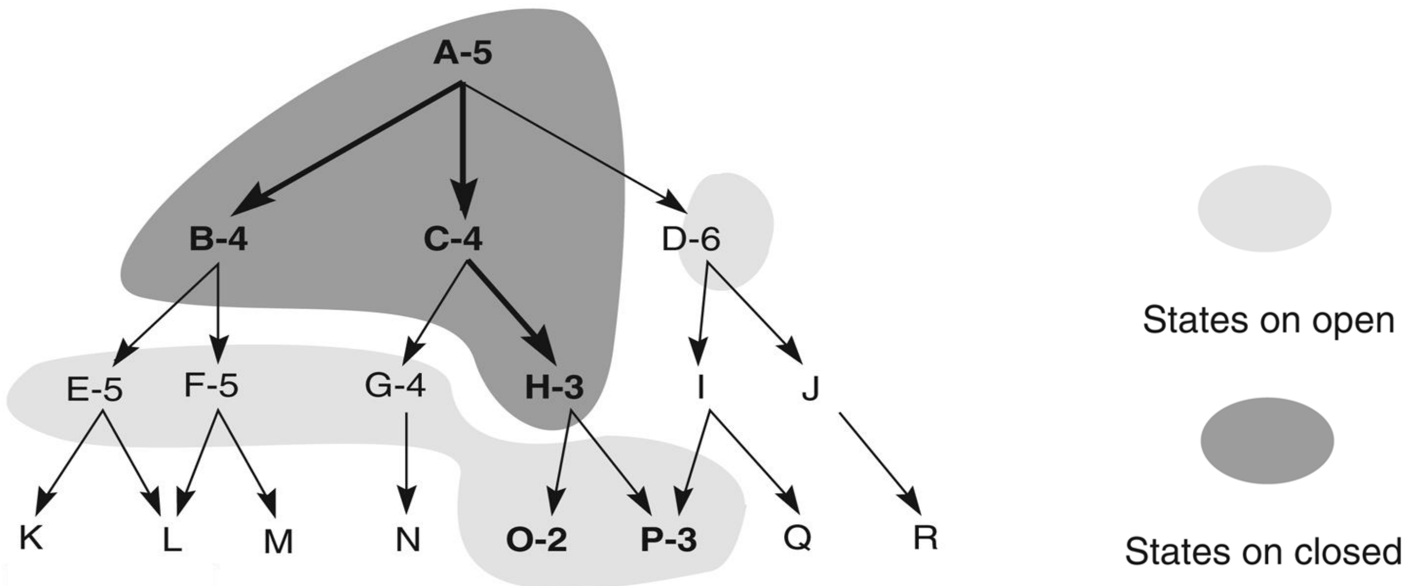
1. **open = [A5]; closed = [ ]**
2. **evaluate A5; open = [B4,C4,D6]; closed = [A5]**
3. **evaluate B4; open = [C4,E5,F5,D6]; closed = [B4,A5]**
4. **evaluate C4; open = [H3,G4,E5,F5,D6]; closed = [C4,B4,A5]**
5. **evaluate H3; open = [O2,P3,G4,E5,F5,D**
6. **evaluate O2; open = [P3,G4,E5,F5,D6];**
7. **evaluate P3; the solution is found!**



- Lower h(n) is better

Best-First Search: Example (2)

1. **open = [A5]; closed = [ ]**
2. **evaluate A5; open = [B4,C4,D6]; closed = [A5]**
3. **evaluate B4; open = [C4,E5,F5,D6]; closed = [B4,A5]**
4. **evaluate C4; open = [H3,G4,E5,F5,D6]; closed = [C4,B4,A5]**
5. **evaluate H3; open = [O2,P3,G4,E5,F5,D6]; closed = [H3,C4,B4,A5]**

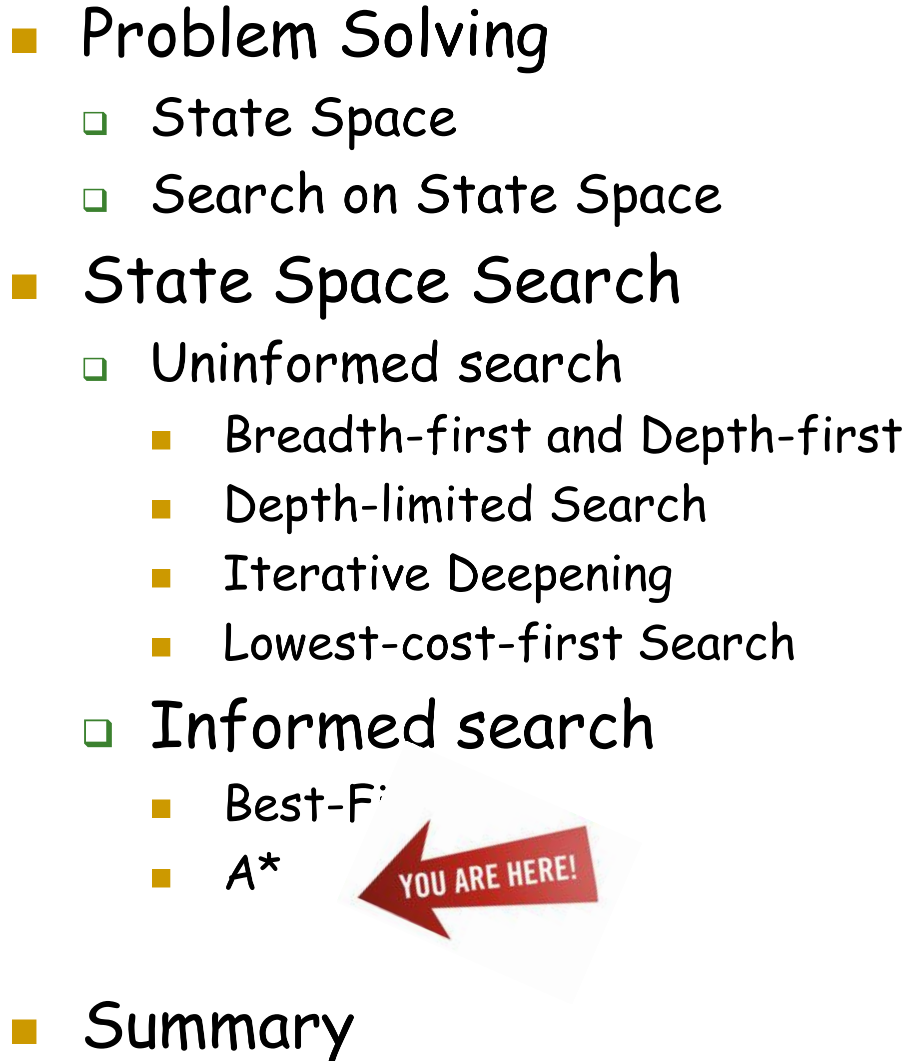


Designing Heuristics

- 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

Today

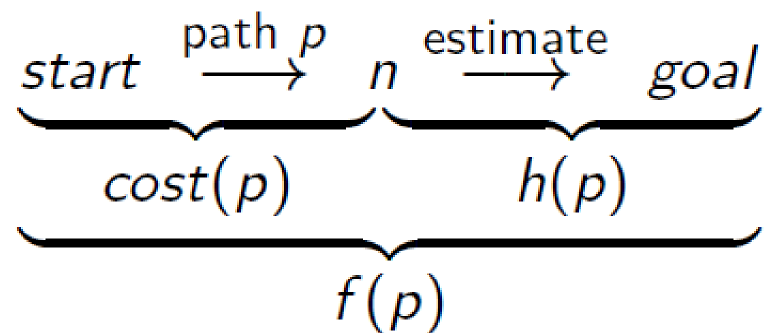
- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed search
    - Best-F
    - A*



- Summary

A* Search

- A^* search uses both path cost and heuristic values
- $cost(p)$ is the cost of path p.
- $h(p)$ estimates the cost from the end of p to a goal.
- Let $f(p) = cost(p) + h(p)$.
  $f(p)$ estimates the total path cost of going from a start node to a goal via p.



A* Search Algorithm

- A^* is a mix of lowest-cost-first and best-first search.
- It treats the frontier as a priority queue ordered by $f(p)$.
- It always selects the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node.

Complexity of A^* Search

- Does A^* search guarantee to find the shortest path or the path with fewest arcs?
- What happens on infinite graphs or on graphs with cycles if there is a solution?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

Admissibility of A*

If there is a solution, A^* always finds an optimal solution —the first path to a goal selected— if

- the branching factor is finite
- arc costs are bounded above zero (there is some $\epsilon > 0$ such that all of the arc costs are greater than ϵ), and
- $h(n)$ is nonnegative and an underestimate of the cost of the shortest path from n to a goal node.

Why is A* admissible?

- If a path p to a goal is selected from a frontier, can there be a shorter path to a goal?
- Suppose path p' is on the frontier. Because p was chosen before p', and $h(p) = 0$:

$$cost(p) \leq cost(p') + h(p').$$

- Because h is an underestimate:

$$cost(p') + h(p') \leq cost(p'')$$

  for any path p'' to a goal that extends p'.
- So $cost(p) \leq cost(p'')$ for any other path p'' to a goal.

Why is A* admissible?

A^* can always find a solution if there is one:

- The frontier always contains the initial part of a path to a goal, before that goal is selected.
- A^* halts, as the costs of the paths on the frontier keeps increasing, and will eventually exceed any finite number.

How do good heuristics help?

Suppose c is the cost of an optimal solution. What happen to a path p where

- $cost(p) + h(p) < c$
- $cost(p) + h(p) = c$
- $cost(p) + h(p) > c$

How can a better heuristic function help?

Summary of Search Strategies

Strategy	Frontier Selection	Complete	Halts	Space
Depth-first	Last node added	No	No	Linear
Breadth-first	First node added	Yes	No	Exp
Heuristic depth-first	Local min $h(p)$	No	No	Linear
Best-first	Global min $h(p)$	No	No	Exp
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp
A^*	Minimal $f(p)$	Yes	No	Exp

Complete — if there a path to a goal, it can find one, even on infinite graphs.
Halts — on finite graph (perhaps with cycles).
Space — as a function of the length of current path

Today

- Problem Solving
  - State Space
  - Search on State Space
- State Space Search
  - Uninformed search
    - Breadth-first and Depth-first
    - Depth-limited Search
    - Iterative Deepening
    - Lowest-cost-first Search
  - Informed search
    - Best-First
    - A*
- Summary