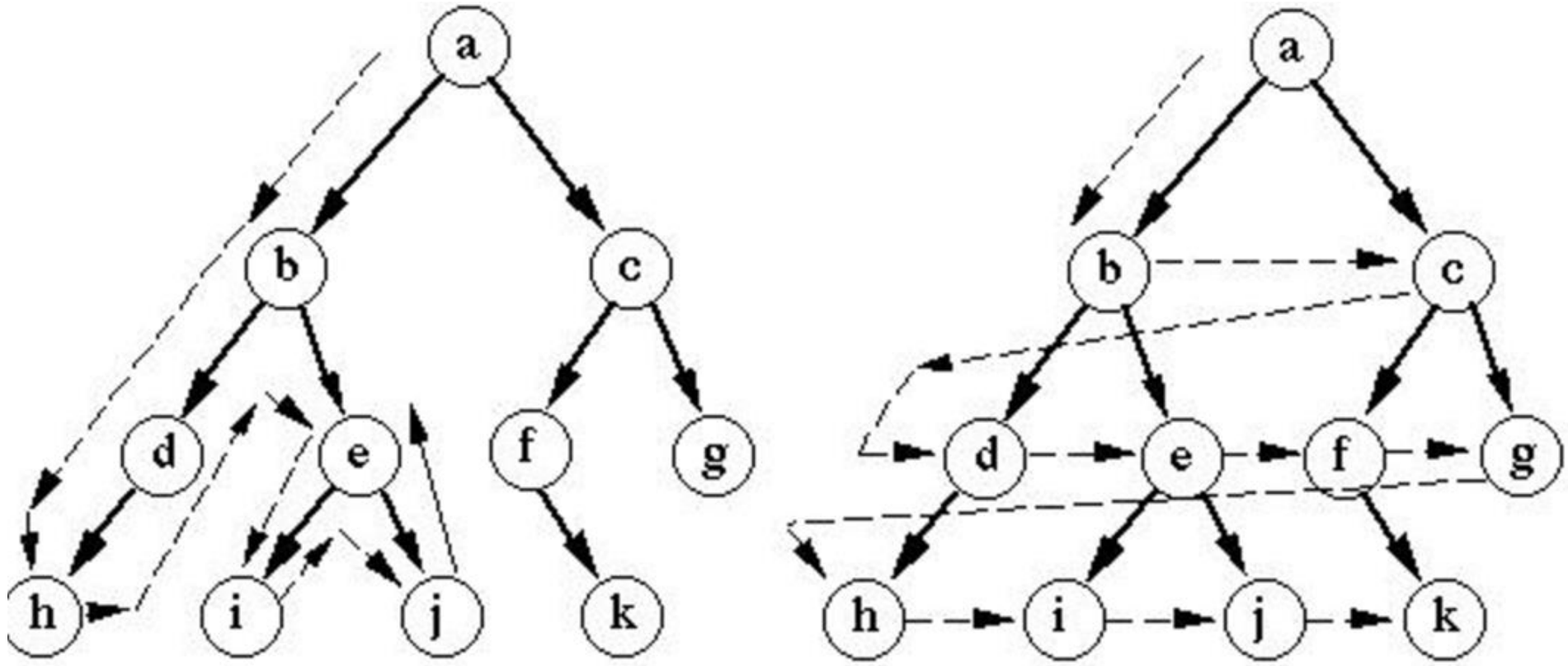
Basic Search



Outline

- Motivating Examples
- Problem Formulation
- From Searching to Search Tree
- Uninformed Search Methods
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search

I. Motivating Examples

-- search is everywhere

Example:
Route Planning

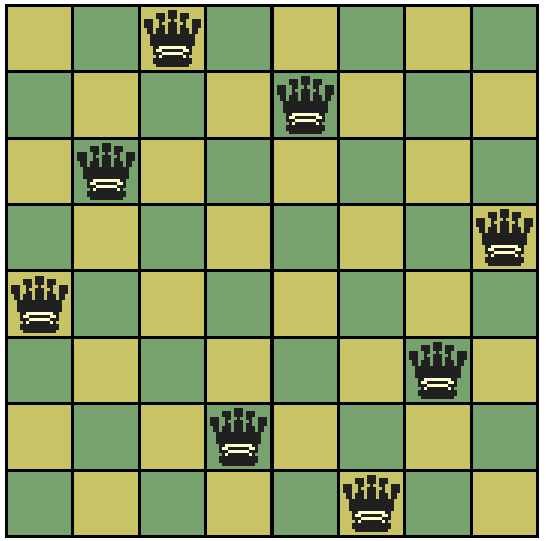
- How to find the ‘best’ route from the *swimming pool* to the *teaching building 3*?
- What’s the definition of a ‘good’ route?
- What if there is some temporal construction work?

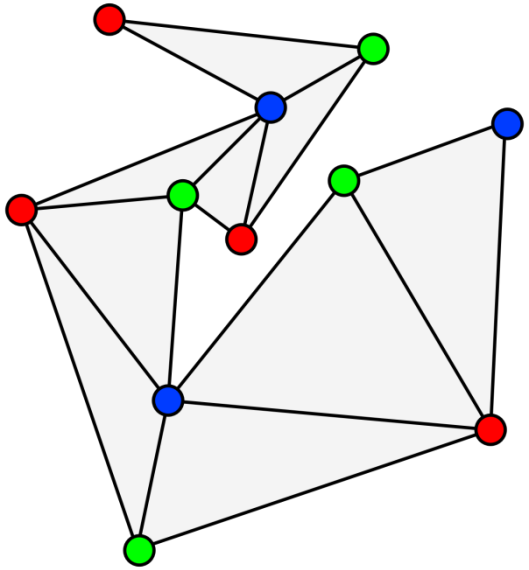
Example: Robot Navigation

- Automated guided vehicle
- Legged robot
- Robotic vacuum cleaner
- …

节\星期	星期一	星期二	星期三	星期四	星期五
1					
2					
3		【智能数据分析智能数据分析　全周(1-16周) 3-4(茗园6栋403)】		【研究方法研究方法　全周(1-16周) 3-4(茗园1栋202)】	
4		【智能数据分析智能数据分析　全周(1-16周) 3-4(茗园6栋403)】		【研究方法研究方法　全周(1-16周) 3-4(茗园1栋202)】	
5		【智能数据分析智能数据分析　双周(1-16周) 5-6(茗园6栋406机房)】		【高级优化算法高级优化算法　全周(1-16周) 5-6(一教301)】	【高级优化算法高级优化算法　全周(1-16周) 5-6(二教201)】
6		【智能数据分析智能数据分析　双周(1-16周) 5-6(茗园6栋406机房)】		【高级优化算法高级优化算法　全周(1-16周) 5-6(一教301)】	【高级优化算法高级优化算法　全周(1-16周) 5-6(二教201)】
7	【密码学与网络安全密码学与网络安全　全周(1-16周) 7-8(一教303)】				
8	【密码学与网络安全密码学与网络安全　全周(1-16周) 7-8(一教303)】				
9					
10					

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9





Example: Constraint Satisfaction Problems

II. Problem Formulation

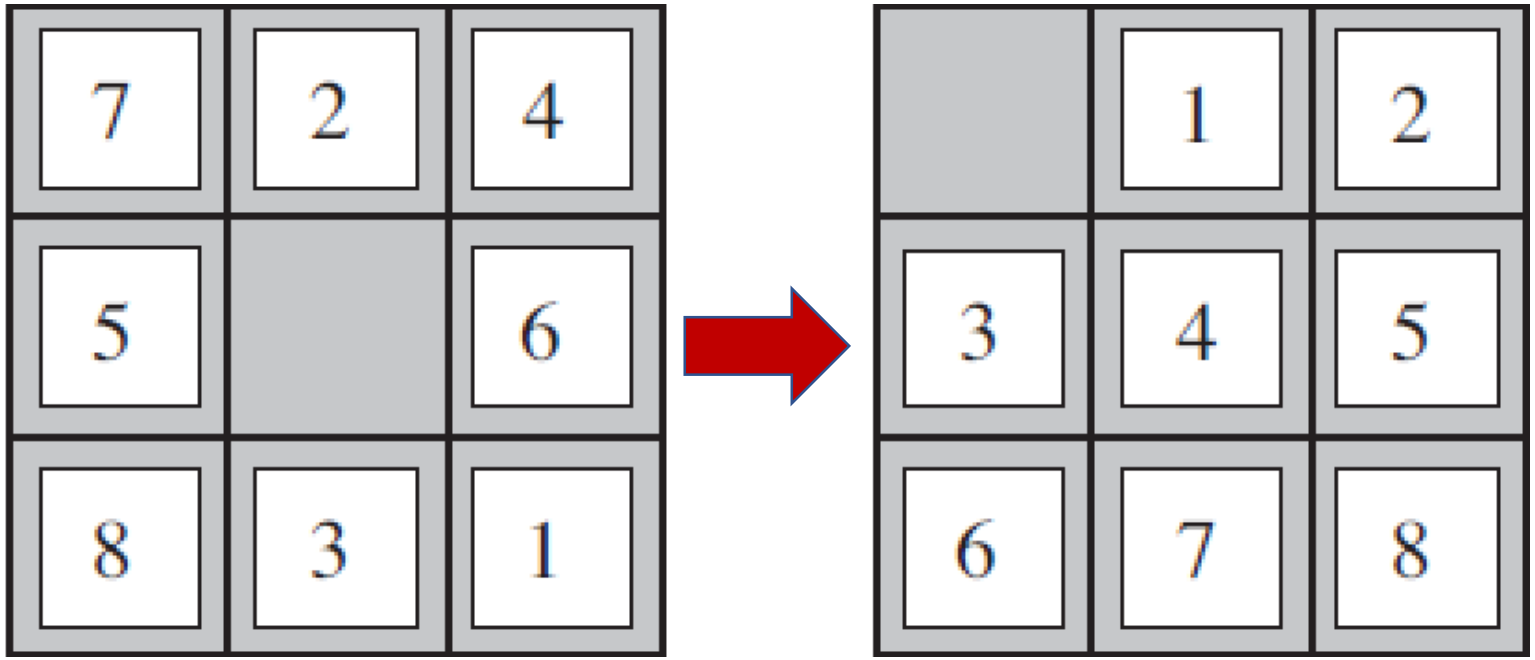
The State-Space Formulation

- States: all reachable states from the initial state by any sequence of actions.
  - $RESULT(s, a)$ – the state that results from doing an action a in state s.
  - $c(s, a, s')$: the step cost of taking action a in state s to reach state s'.
- Initial state: the state where the problem starts.
- Actions: legal/available actions on states.
- Transition model: next state when giving the current state and action
- Goal test: determine whether a given state is a goal state.
- Path cost: function that assigns a numeric cost to each path.

A solution is a sequence of actions leading from the initial state to a goal state.

[Example] 8-puzzle Problem

- States: 3×3 board with 8 numbered & one blank tiles.
- Rule: A tile adjacent to the blank space can slide into the space.
- Objective: Reach a specified goal state.

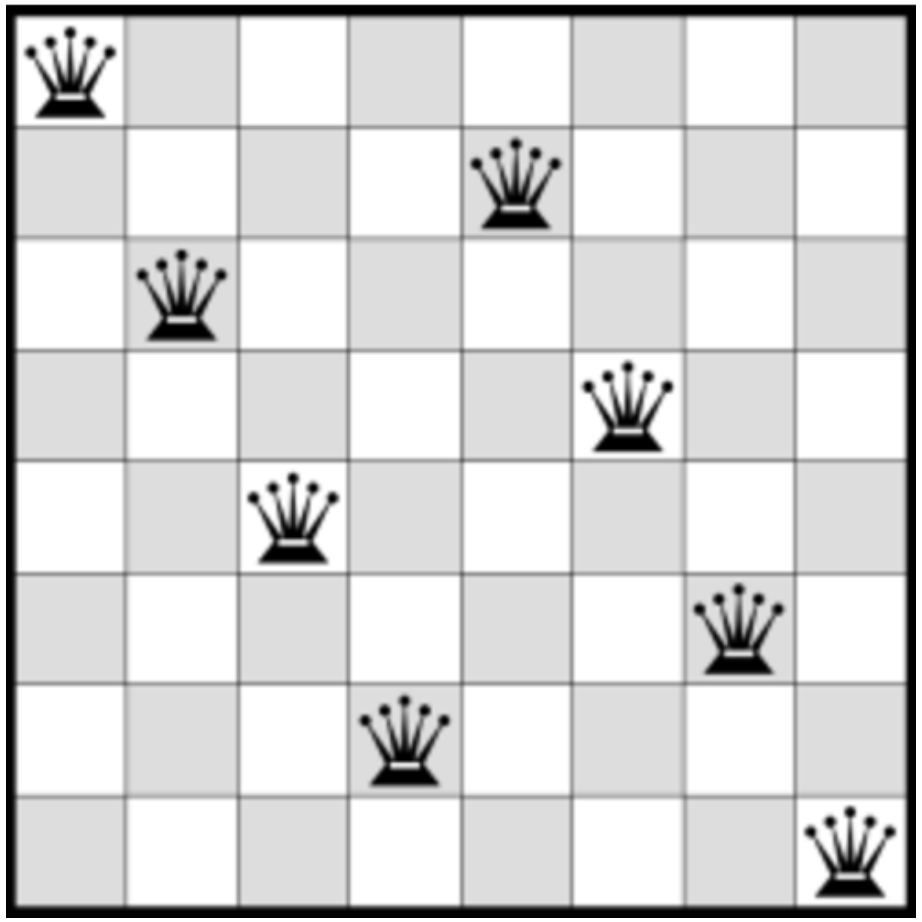


[Example] Formulation of 8-puzzle Problem

- States: all allocations of the 9 tiles.
- Initial state: left allocation.
- Actions: move blank tile $\{left, right, up, down\}$.
- Transition model: return the next state, thus the updated allocations.
- Goal test: right allocation.
- Path cost: #moves.

[Example] 8-queen Problem

- Objective: On a chess board, place 8 queens so that no queen is attacked by any others horizontally, vertically or diagonally.



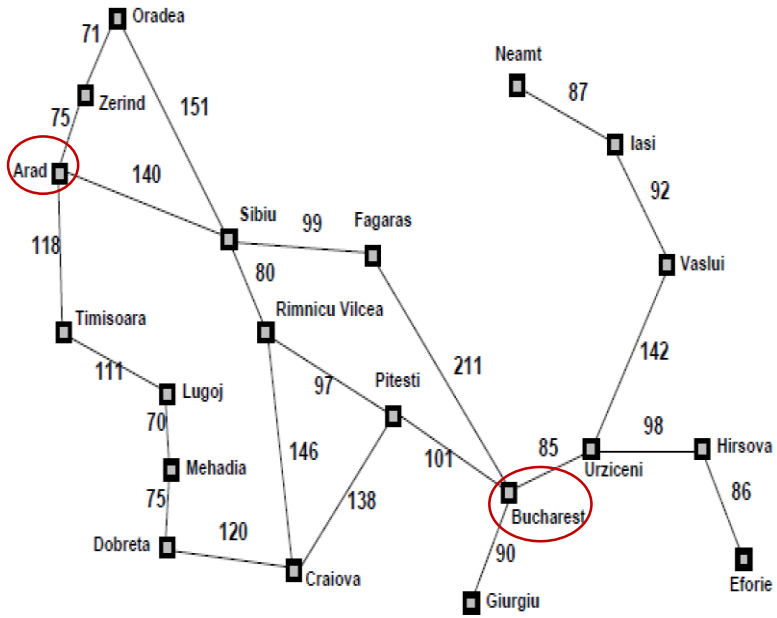
[Example] Formulation of 8-queen Problem

- States: all allocations of 8 queens.
- Initial state: no queen on the board.
- Actions: add a queen to any empty square.
- Transition model: return the next state, thus the updated allocations.
- Goal test: 8 queens on board without attacked?

[Example] Formulation of Route Planning

Objective: Find the shortest path from *Arad* to *Bucharest*.

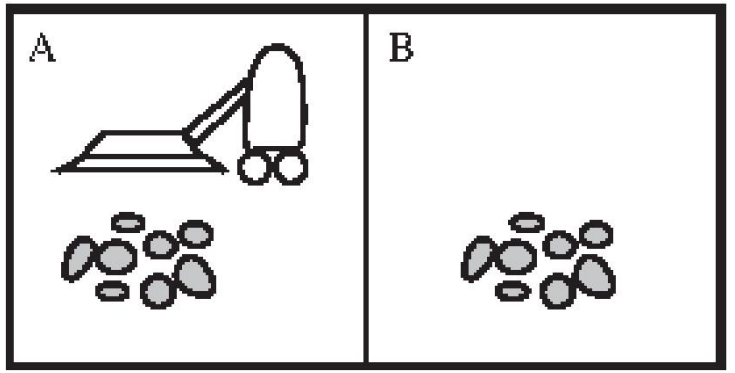
- States: $\{(cur\_city, walk\_dist)\}$
- Initial state: $(Arad, 0)$
- Actions: walk to an adjacent city.
- Next state: e.g. $RESULT((Arad, 0), go\_to\_Sibiu)$
- Goal test: reach '*Bucharest*'?
- Path cost: accumulated walking distance.



[Example] Formulation of Robot Navigation

Simplified version: A vacuum-cleaner world with just 2 locations (A & B).

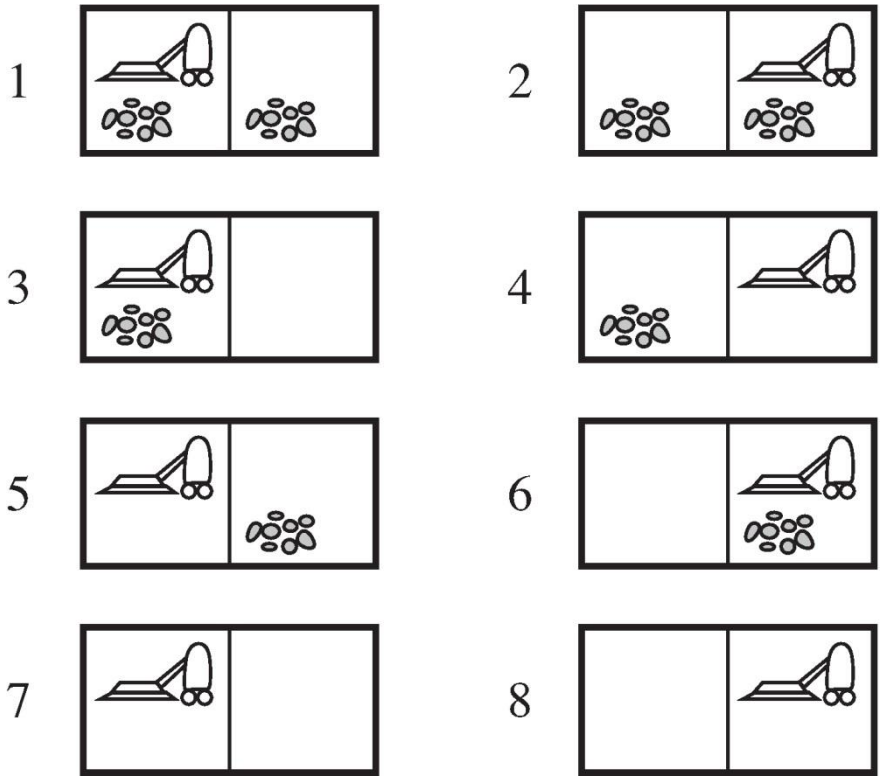
- States: $\{(locaion, state\_of\_floor)\}$
- Initial state: $(A, dirty)$
- Actions:
  - move to an adjacent location;
  - suck.
- Next state:
  - e.g. $RESULT((A, dirty), suck)$
  - or $RESULT((A, dirty), move\_to\_B)$
- Goal test: All parts of floor cleaned? $(A, clean)$ and $(B, clean)$
- Path cost:
  - accumulated move distance;
  - energy cost;
  - time spent to clean the floor.



Multi-objective optimization!!!
We will learn more in future lectures.

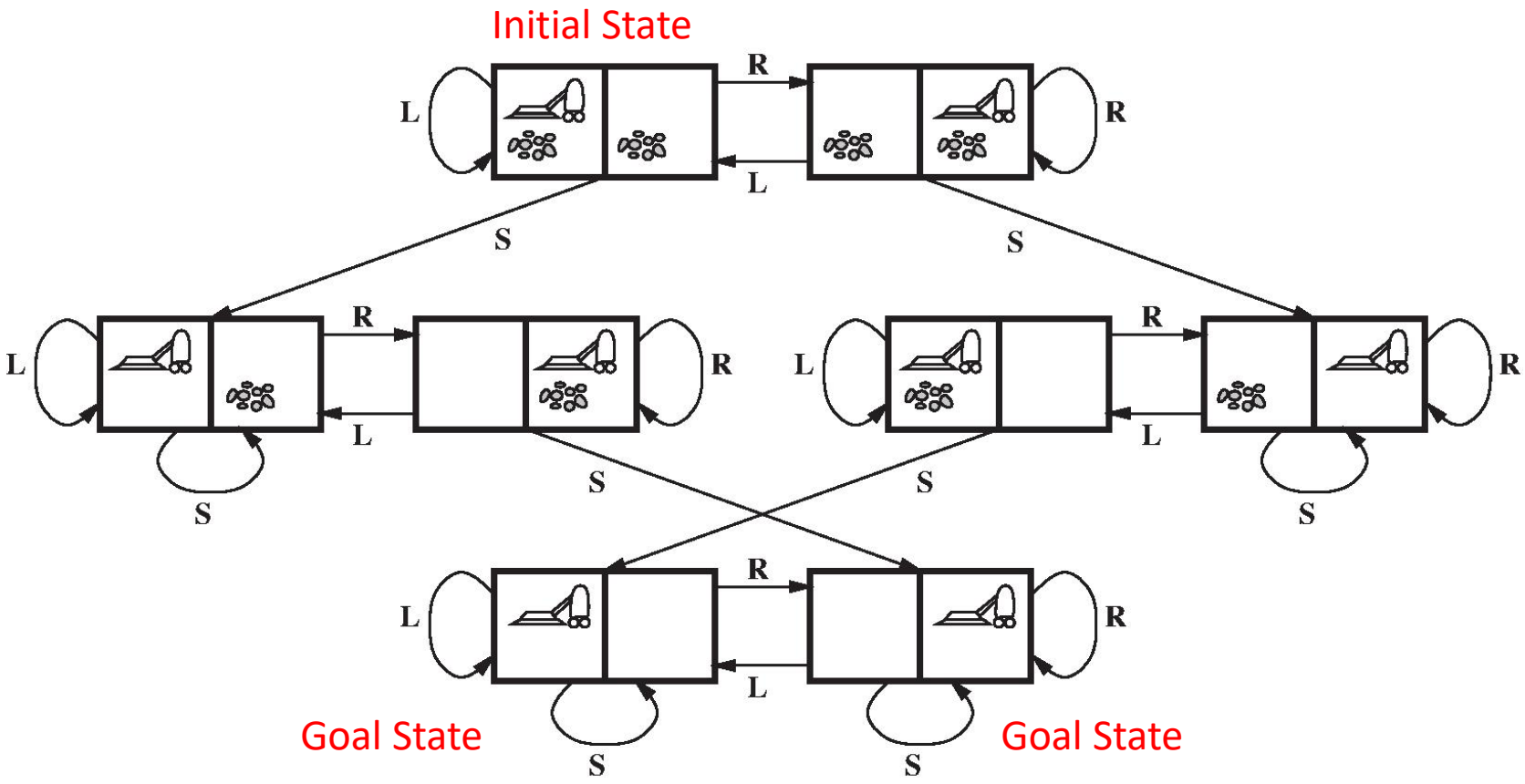
[Example] Formulation of Robot Navigation

- Question: What are the possible states for this problem?

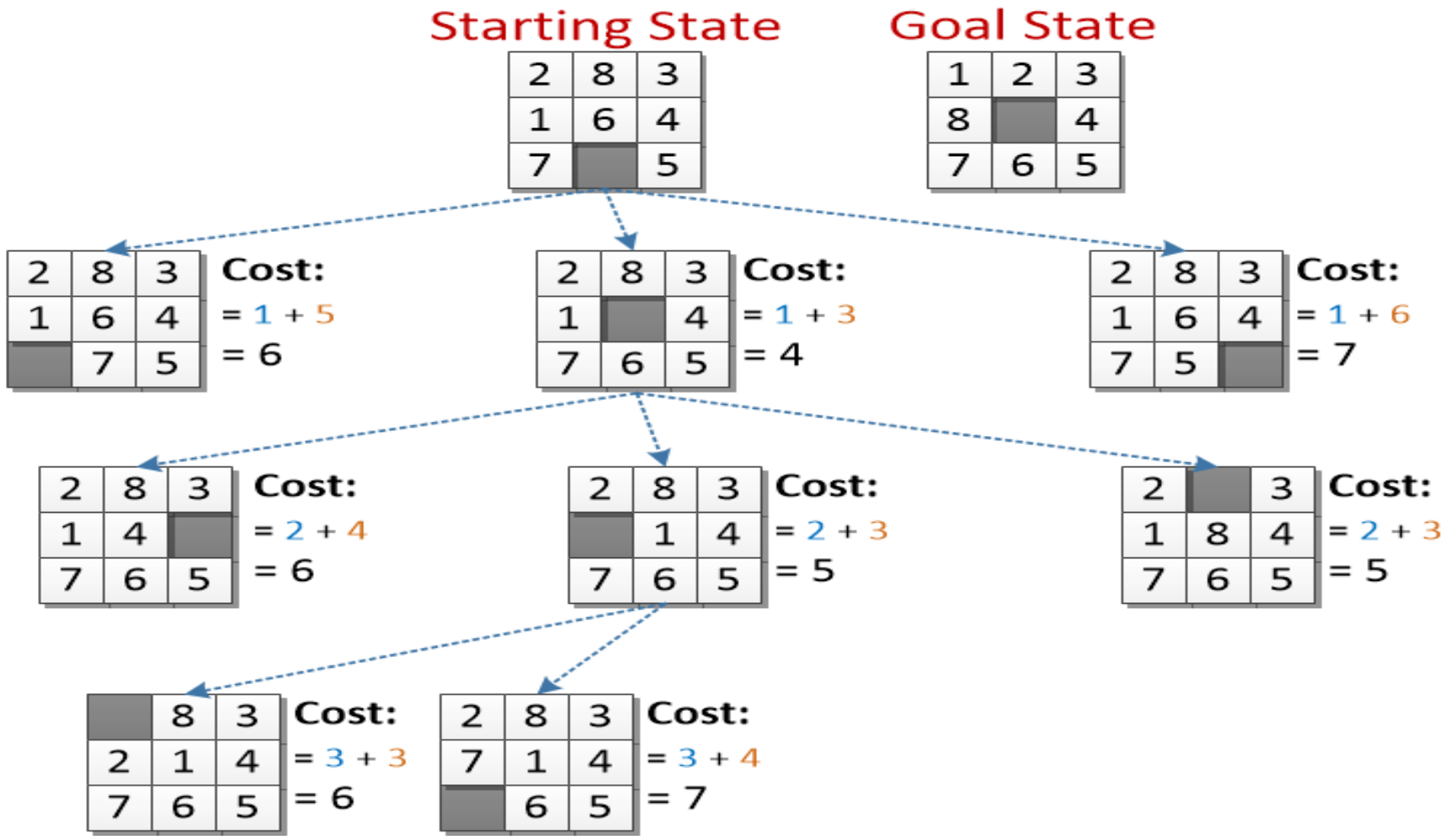


III. From Searching to Search Trees

Searching for a Solution: Vacuum-cleaner



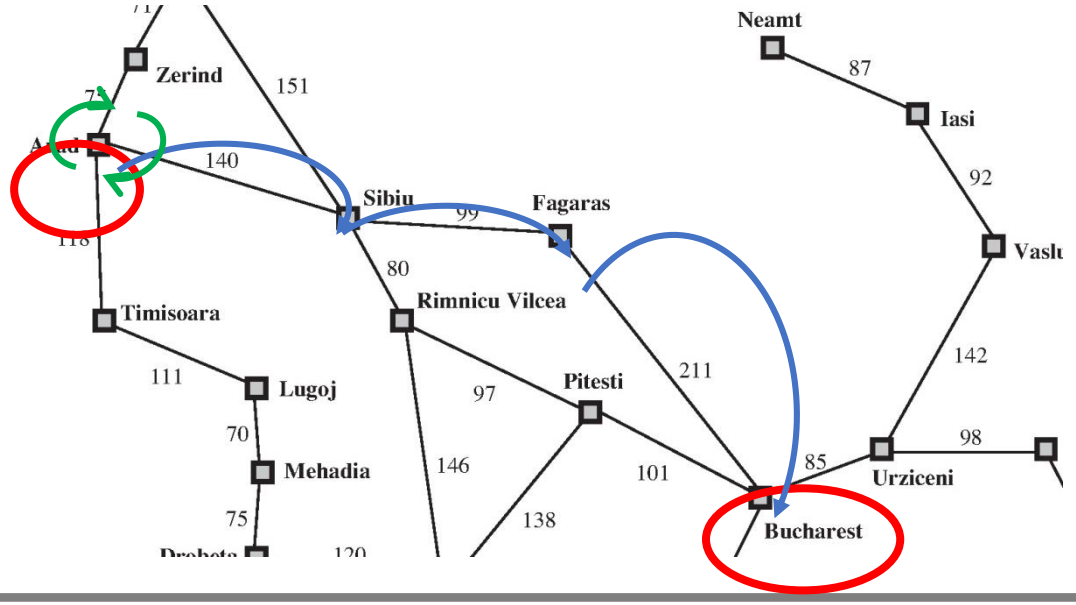
Searching for a Solution: 8-puzzle Problem



Searching for a Solution

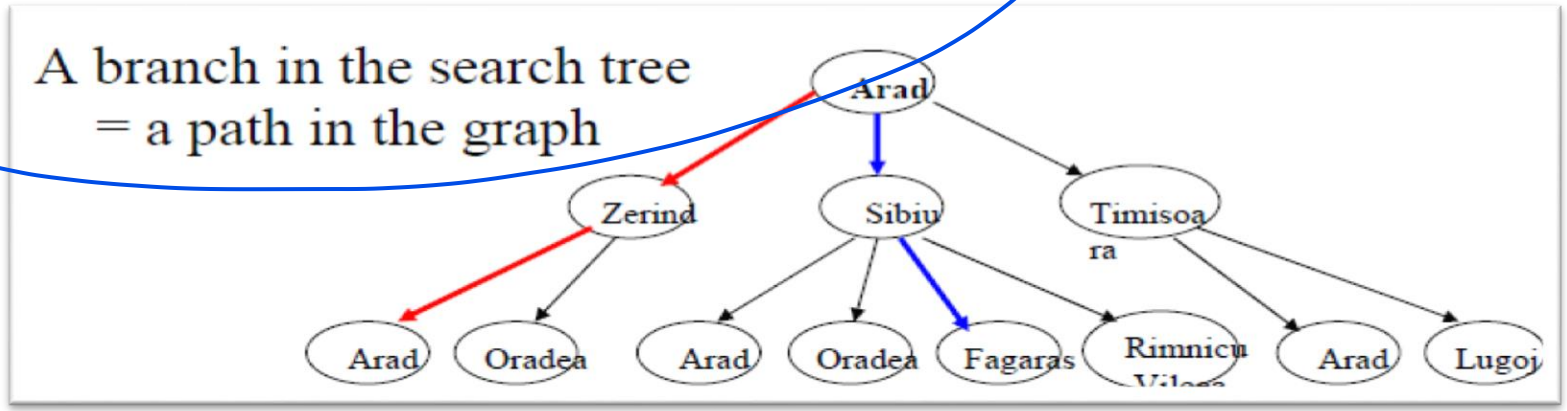
- Problem solution: a sequence of actions that lead to the goal.
- Searching: explore the state space by a sequence of actions to reach the goal from the initial state.

➢ We can find the solution by searching.



Search Trees

- A search tree models the sequence of legal actions.
  - Root: initial state.
  - Nodes: the states resulting from actions.
  - Child nodes: the follow-up states of a previous node.
  - Branch: a sequence of states (and thereby a sequence of actions).
- Expand: create all child nodes for a given node.



Structure of Tree Node

- Tree node: a data structure to keep track of constructing search tree.
- 4 components of node n:
  - n.STATE: node n's state(s).
  - n.PARENT: node that generated n.
  - n.ACTION: the action applied to the *parent* to generate node n.
  - n.PATHCOST: the cost of the entire path from the initial state.

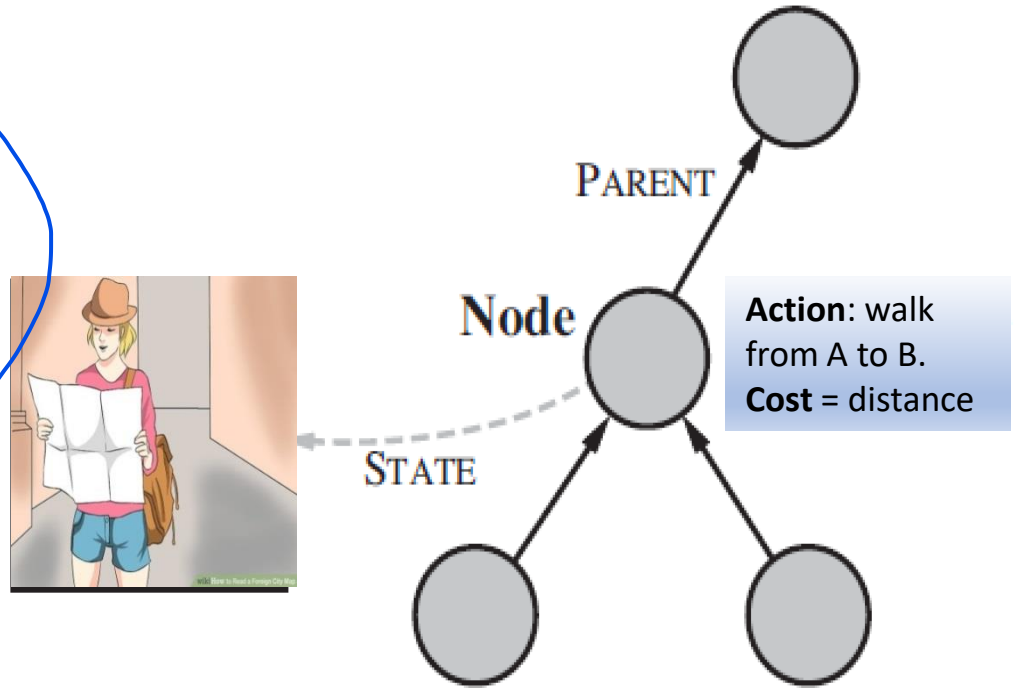


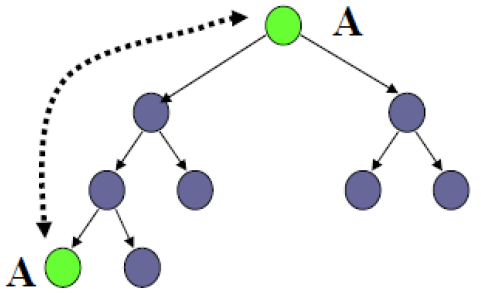
Fig. Structure of tree node with which search tree is constructed. Arrows point from child to parent.

Pseudocode for Tree Search

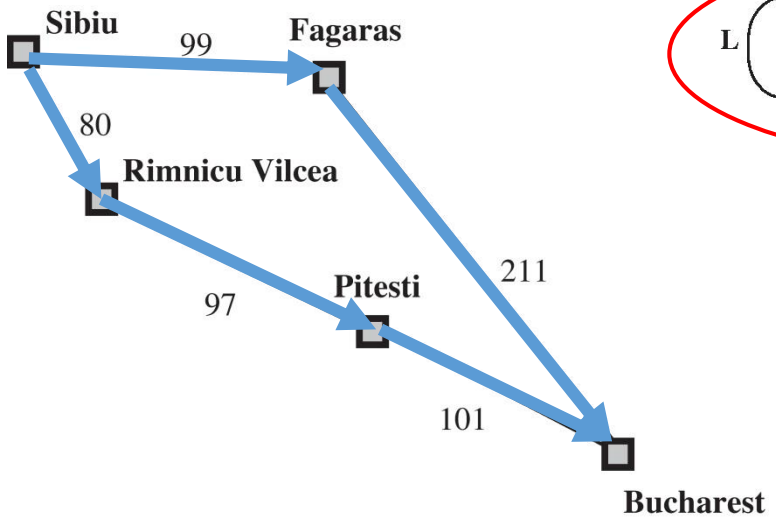
```
1: function TreeSearch(problem, strategy) returns a solution or failure
2:     frontier → initial state of the problem
3:     loop
4:         if frontier is empty then
5:             return failure
6:         Choose a leaf node according to strategy and remove it from the frontier
7:         if the node contains a goal state then
8:             return the corresponding solution
9:         Expand the chosen node and add the resulting nodes to the frontier
```

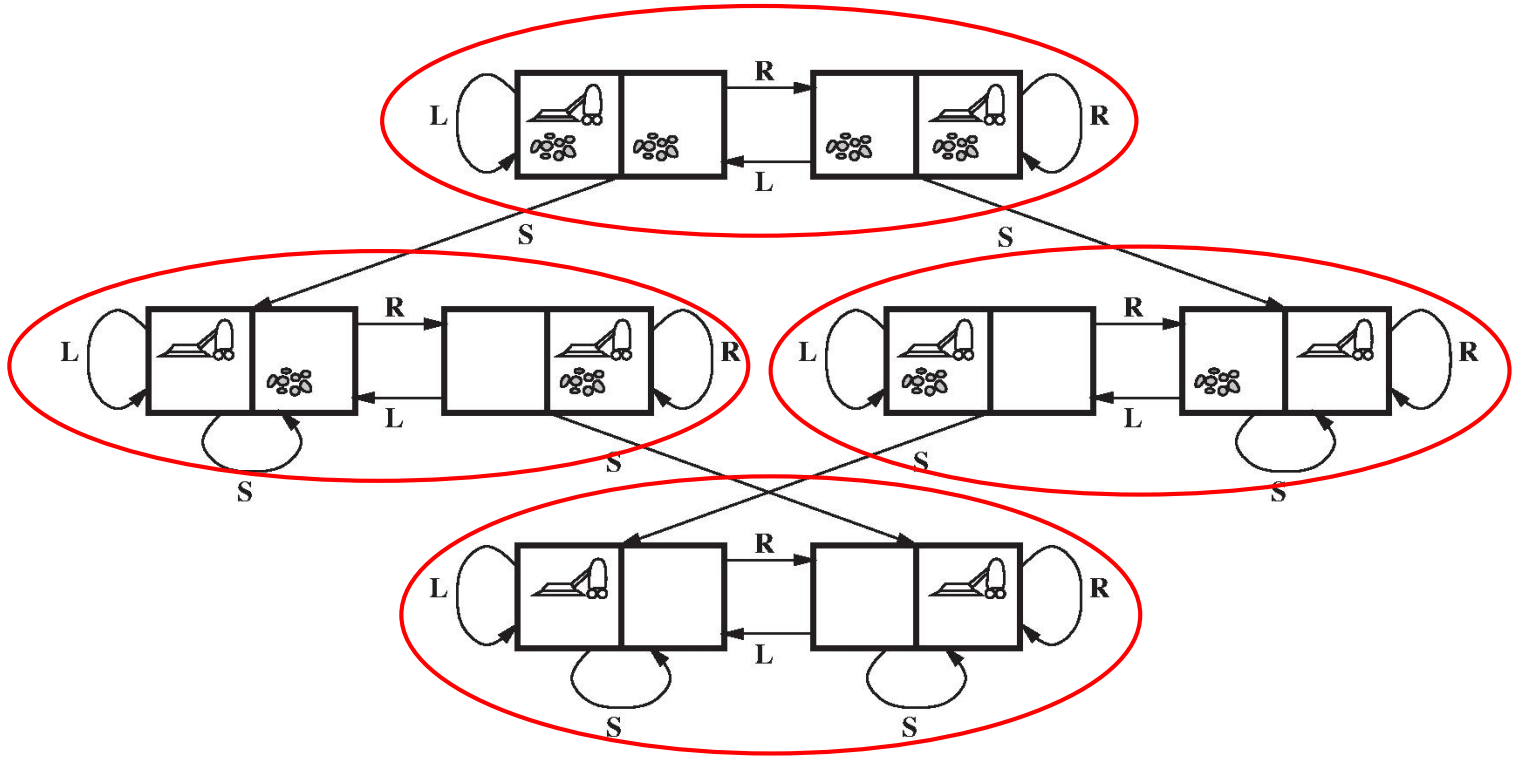
- Key points:
  - Frontier: the set of candidate tree nodes for expansion.
  - Expansion: searching all children of a given node
- Main question: which leaf node to expand? => Search strategy

Potential Problems



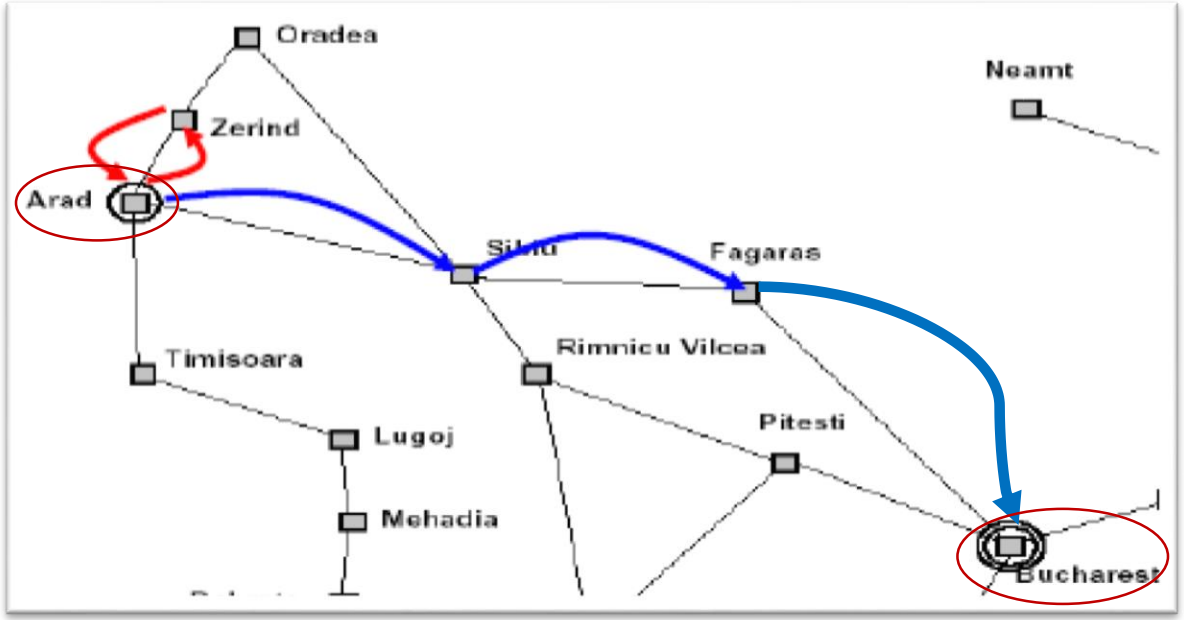
- Redundant data due to the repeated states
- Infinite loops
- Redundant paths

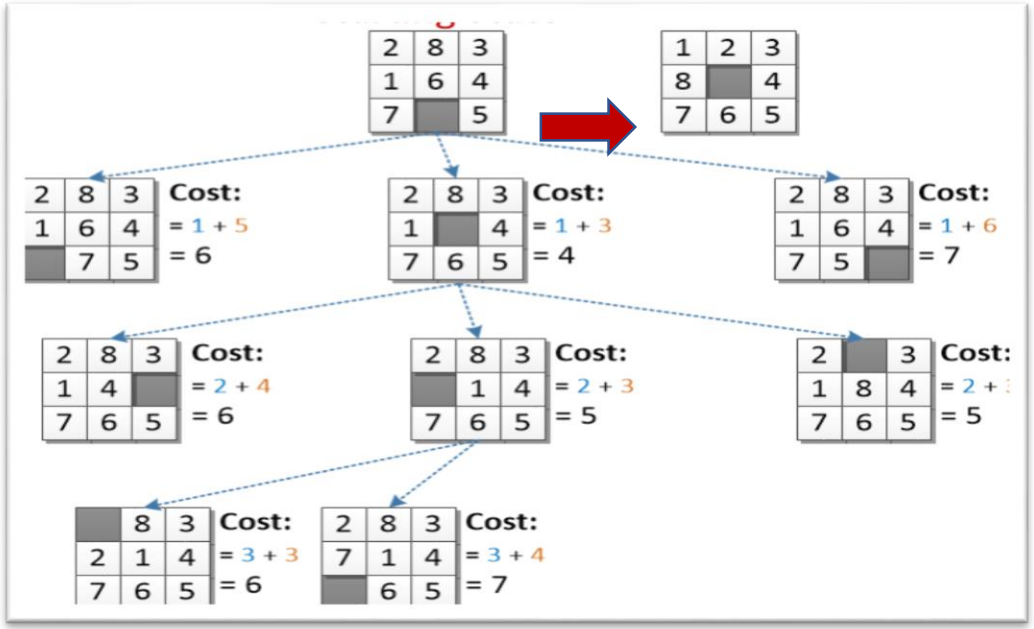




Search Strategies/Methods

- Defined by the order of node expansion.
- Aim to identify the 'best' solution.





Search Methods: Performance Metrics

How to evaluate a search method?

- Completeness: Does it always find a solution if it exists?
- Optimality: Does it always find the least-cost (optimal) solution?
- Time complexity: # nodes generated/expanded.
- Space complexity: maximum #nodes in the memory.

Balance between time/space complexity and solution quality: sometimes you need to give up some solution quality to achieve a fast computation.

Search Methods: Performance Metrics

In particular, time and space complexities are measured in terms of:

- b – maximum # of successors of any node in a search tree.
- d – depth of the least-cost solution.
- m – maximum length of any path in the tree.

Big-O nation in measuring complexity:

- Non-technical: http://bigocheatsheet.com/; https://www.interviewcake.com/article/java/big-o-notation-time-and-space-complexity
- Formally, f(x) = O(g(x)) iff ∃ N and C such that |f(x)| <= C |g(x)| for all x>N.

  http://web.mit.edu/16.070/www/lecture/big_o.pdf

IV. Uninformed Search Methods

Search Methods

- [Reminder] Search methods differ in how they explore the space, i.e., how they choose the node to expand NEXT.

Uninformed Search Methods

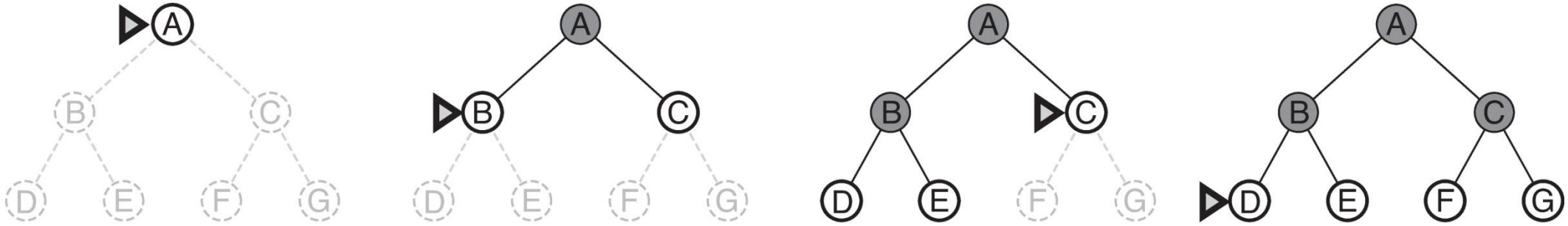
- Use only the information available in the problem definition.
- Use NO domain knowledge.

- Breadth-First Search (BFS)
- Uniform-Cost Search (UCS)
- Depth-First Search (DFS)
- Depth-Limited Search (DLS)
- Iterative Deepening Search (IDS)
- Bidirectional Search

Breadth-related Search Methods

Breadth-First Search (BFS)

- Expand the shallowest unexpanded node.
- Data structure: a FIFO queue.



[Illustrated example] Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

BFS: Pseudo-code

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

BFS: Performance Metrics

- Complete? Yes, if b is finite.
- Optimal? Yes, if costs on the edge are non-negative.
- Time? $O(b^d)$
  - $1 + b + b^2 + \cdots + b^d = \frac{b^{d+1}}{b-1} = O(b^d)$
- Space? $O(b^d)$
  - $1 + b + b^2 + \cdots + b^d = \frac{b^{d+1}}{b-1} = O(b^d)$
  - keep every node in the memory.

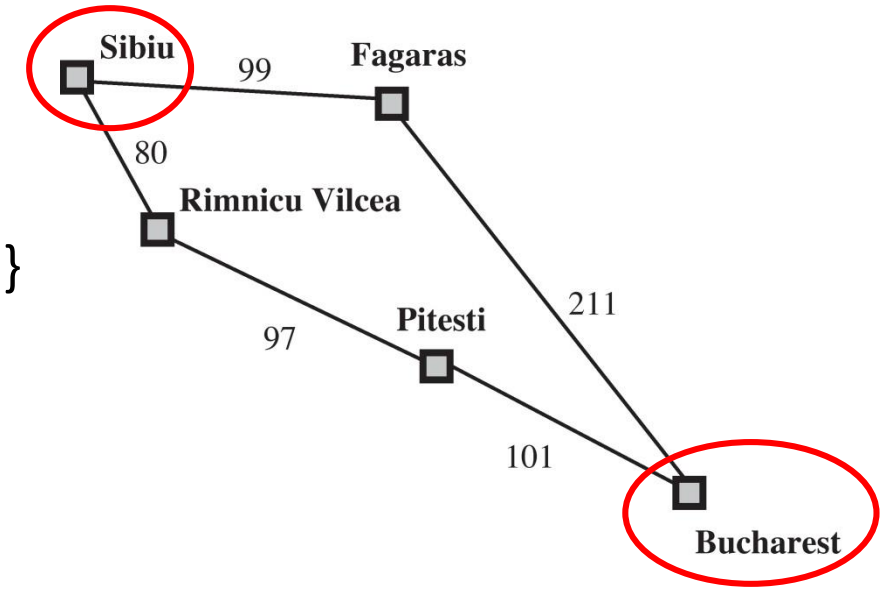
b – maximum # successors of any node in search tree.
d – depth of the least-cost solution.
m – maximum length of any path in the state space.

➢ Space and time complexities are the biggest handicaps of BFS.

Uniform-cost Search (UCS)

- The path costs in the search tree may be different.
- Expand the cheapest unexpanded node.
- Data structure: a queue ordered by the path cost, the lowest first.

- **Task**: from Sibiu to Bucharest.
- [0] {[Sibiu, 0]}
- [1] {[Sibiu→Rimnicu, 80]; [Sibiu→Fagaras, 99]}
- [2] {[Sibiu→Rimnicu→Pitesti, 177]; [Sibiu→Fagaras, 99]}
- [3] {[Sibiu→Rimnicu→Pitesti, 177];
  [Sibiu→Fagaras→Bucharest, 310]}
- [4] {[Sibiu→Rimnicu→Pitesti→Bucharest, 278];
  [Sibiu→Fagaras→Bucharest, 310]}



UCS: Pseudo-code

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

UCS: Performance Metrics

b – maximum # successors of any node in search tree.
d – depth of the least-cost solution.
m – maximum length of any path in the state space.

- Complete? Yes, if every step cost $\geq \epsilon$.
- Optimal? Yes, if costs on the edge are non-negative.
- Time? Space? $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
  - C^*: the cost of the optimal solution.
  - every action costs at least ϵ.
  - $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ can be much greater than $O(b^d)$.
  - When all step costs are equal, $O(b^{1+\lfloor C^*/\epsilon \rfloor}) = O(b^{d+1})$.

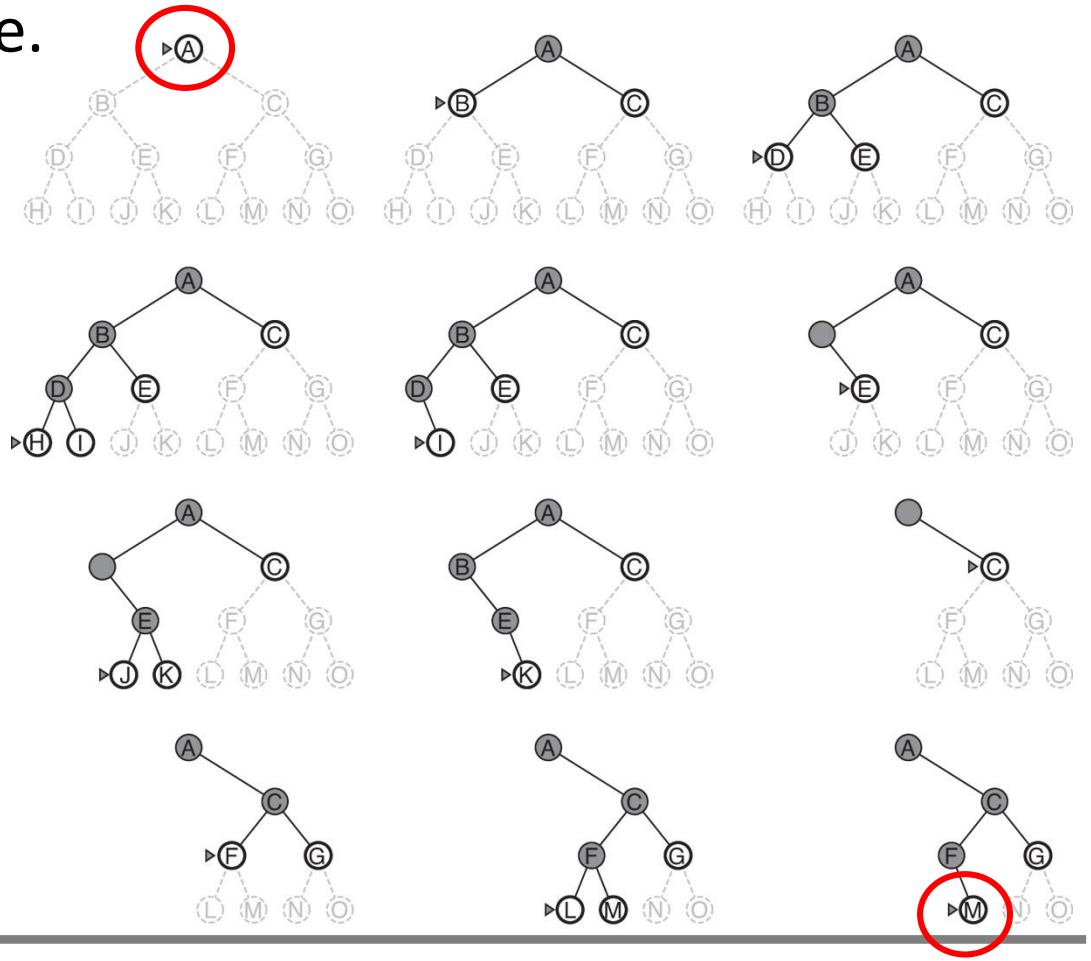
➢ When all step costs are equal, UCS is similar to BFS.

Depth-related Search Methods

Depth-First Search (DFS)

- Expand the deepest unexpanded node.
- Data structure: LIFO stack.

- Task: Search from A to M.
- Note: Once a node is expanded, it is removed from the memory asap, and all its children are added.
- Note: DFS needs to save all children of a node.



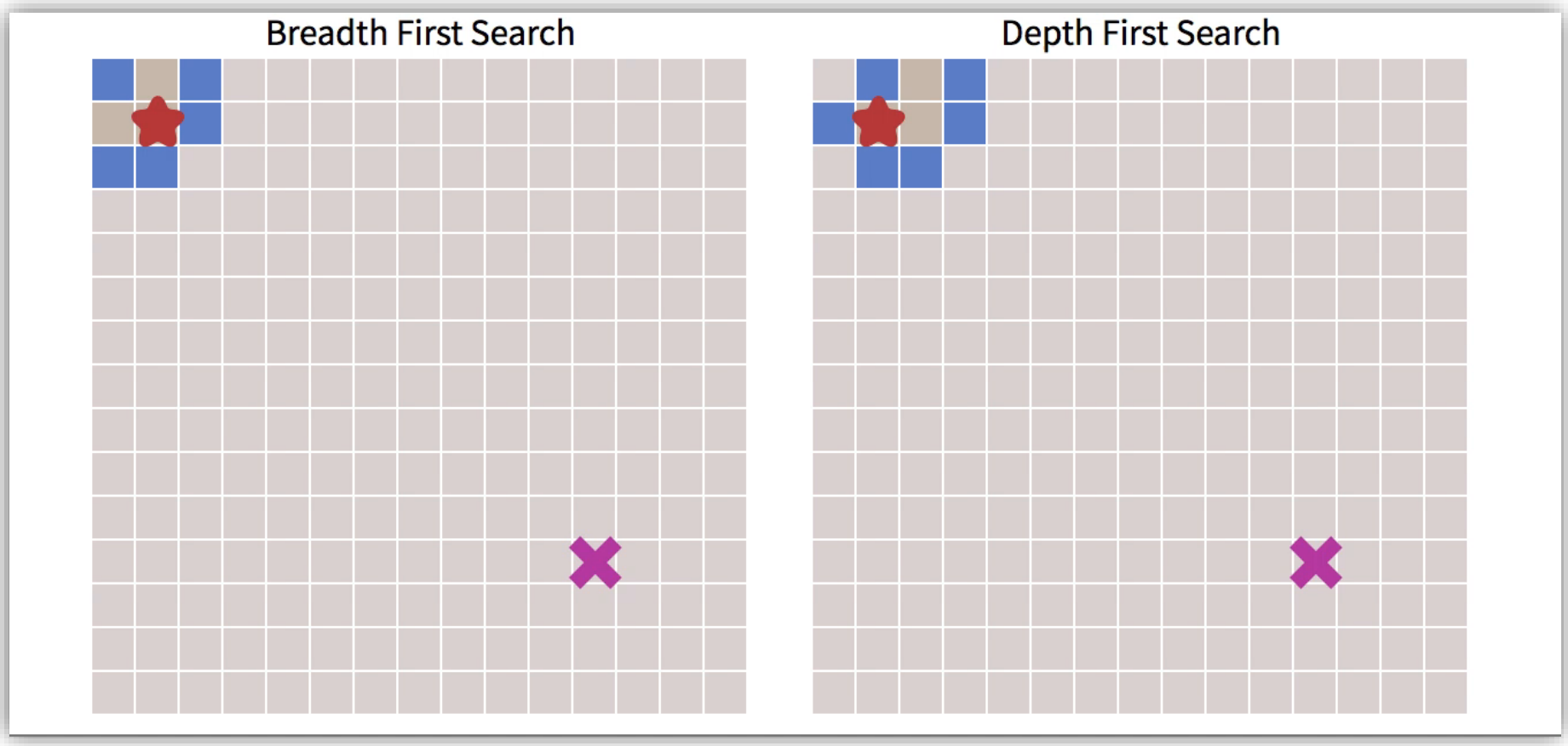
DFS: Performance Metrics

- Complete? No, fail in an infinite-depth space and the space with loops.
- Optimal? No. (Why? Think of a case.)
- Time? $O(b^m)$
  - Terrible if m is much larger than d.
  - If solutions are dense, may be much faster than BFS.
- Space? $O(bm)$ – linear!

➢ Space complexity is much lower than BFS.

b – maximum # successors of any node in search tree.
d – depth of the least-cost solution.
m – maximum length of any path in the state space.

BFS v.s. DFS



*Video recorded from https://cs.stanford.edu/people/abisee/tutorial/bfsdfs.html

Depth-Limited Search (DLS)

- DFS with depth limit l: nodes at depth l have no successors.
  - Limit l is defined based on domain knowledge.
    - e.g. a traveling salesman problem with 20 cities → $l < 20$.
  - DLS is a variant of DFS.

- DLS overcomes the failure of DFS in an infinite-depth space.

DLS: Pseudo-code

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else if limit = 0 then return cutoff
    else
        cutoff_occurred? ← false
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            result ← RECURSIVE-DLS(child, problem, limit − 1)
            if result = cutoff then cutoff_occurred? ← true
            else if result ≠ failure then return result
        if cutoff_occurred? then return cutoff else return failure
```

DLS: Performance Metrics

- Complete? No, (Eps. $l < d$).
- Optimal? No.
- Time? $O(b^l)$
- Space? $O(bl)$

b – maximum # successors of any node in search tree.
d – depth of the least-cost solution.
m – maximum length of any path in the state space.

Iterative Deepening Search (IDS)

- Apply DLS with increasing limits.
- Combine the benefits of BFS and DFS.
  - Like BFS, complete when b is finite & optimal when the path cost is non-decreasing regarding the depth of the nodes.
  - Like DFS, space complexity is $O(bd)$.

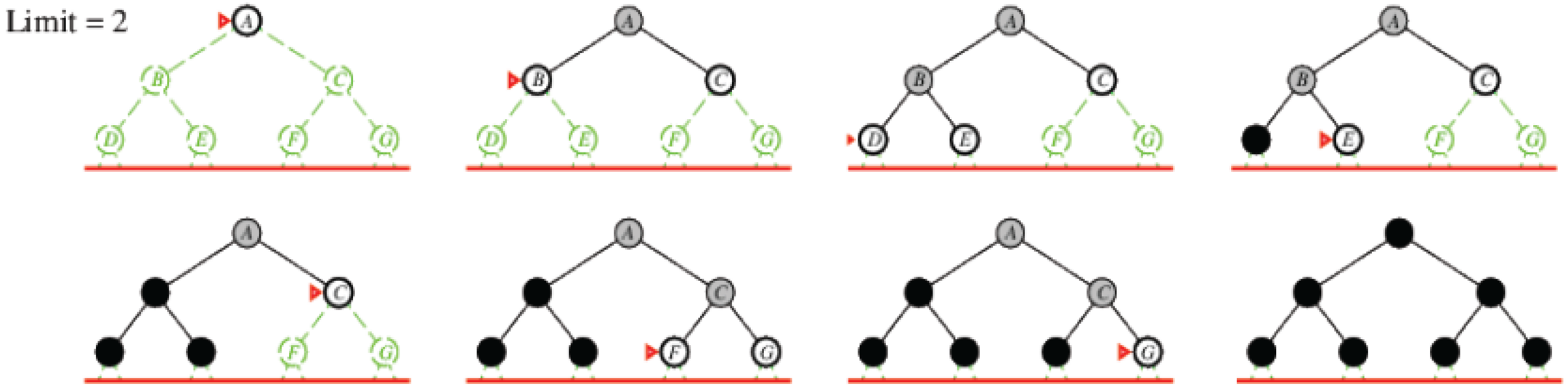
Limit = 0

Iterative Deepening Search (IDS)

- Apply DLS with increasing limits.
- Combine the benefits of BFS and DFS.
  - Like BFS, complete when b is finite & optimal when the path cost is non-decreasing regarding the depth of the nodes.
  - Like DFS, space complexity is $O(bd)$.

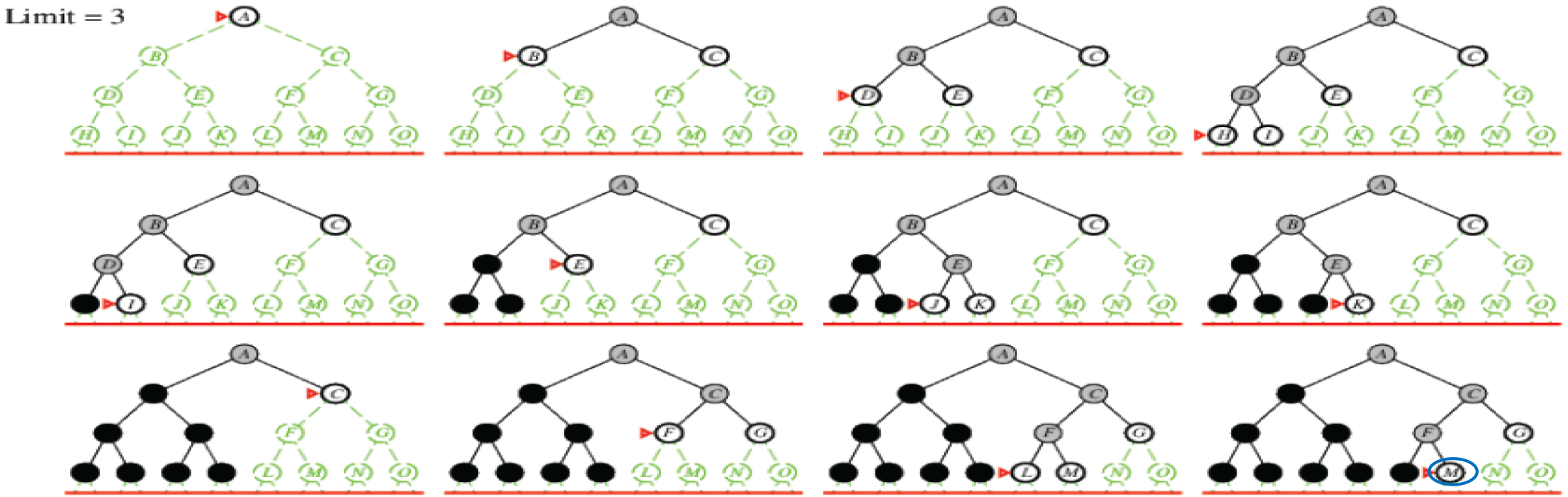
Iterative Deepening Search (IDS)

- Apply DLS with increasing limits.
- Combine the benefits of BFS and DFS.



Iterative Deepening Search (IDS)

- Apply DLS with increasing limits.
- Combine the benefits of BFS and DFS.



IDS: Pseudo-code

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure

    for *depth* = 0 **to** ∞ **do**

        result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)

        if *result* $\neq$ cutoff **then return** *result*

IDS: Performance Metrics

b – maximum # successors of any node in search tree.
d – depth of the least-cost solution.
m – maximum length of any path in the state space.

- Complete? Yes.
- Optimal? Yes, if costs on the edge are non-negative.
- Time? $O(b^d)$
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \cdots + b^d = O(b^d)$.
- Space? $O(bd)$

➢ IDS is the preferred uninformed search method when the search space is large and the depth of the solution is unknown.

Bidirectional Search Method

Bidirectional Search

- Search from forward & backward directions simultaneously.
- Replace a single search tree with two smaller sub-trees.
  - Forward tree: forward search from source to goal.
  - Backward tree: backward search from goal to source.
- Goal test: two sub-trees intersect.

Bidirectional Search: Performance Metrics

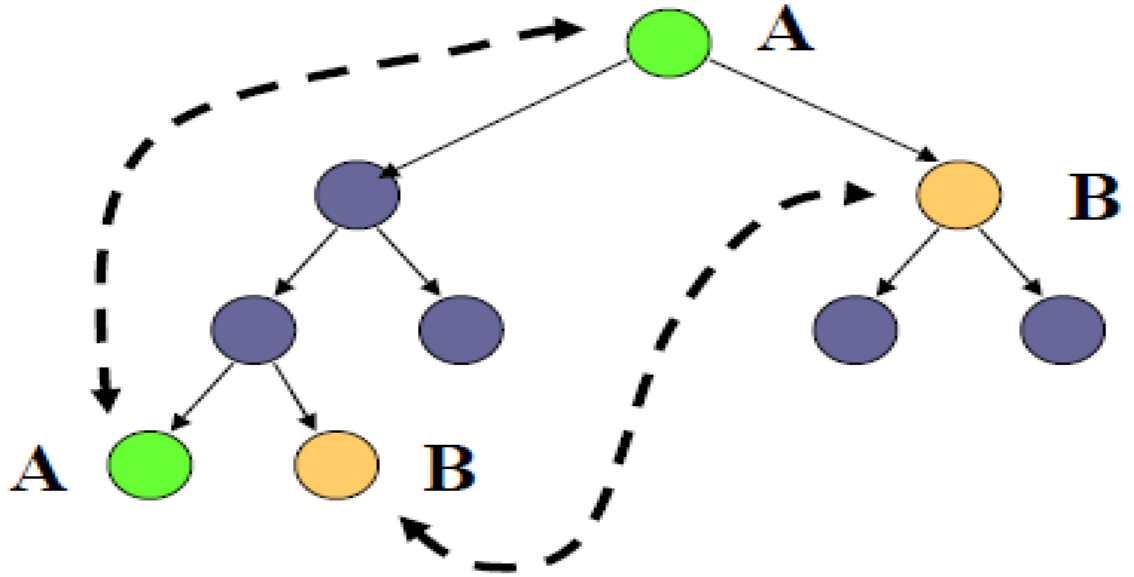
- Complete? Yes, if BFS is used in both search.
- Optimal? Yes, if BFS is used & paths have a uniform cost.
- Time? Space? $O(b^{d/2})$

b – maximum # successors of any node in search tree.
d – depth of the least-cost solution.
m – maximum length of any path in the state space.

Elimination of State Repetition

Two Cases of State Repetition

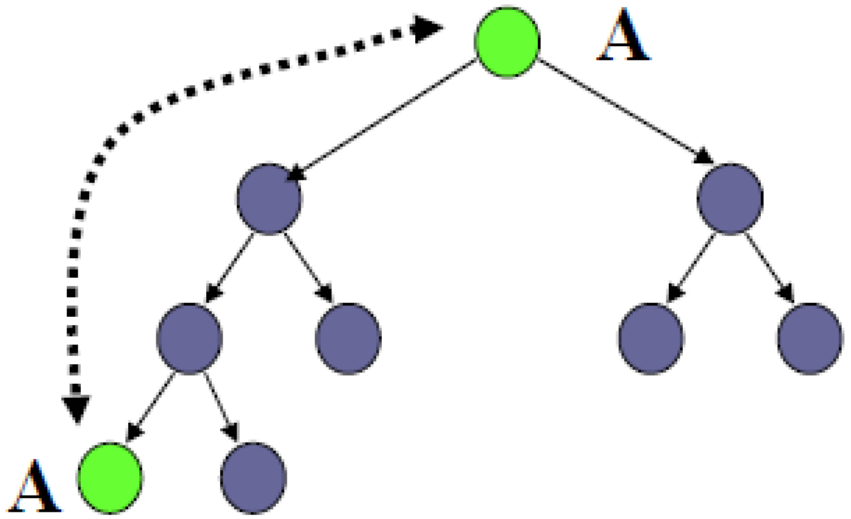
- [Case A] Cyclic state repetition.
- [Case B] Non-cyclic state repetition.
- Question: Is it necessary to keep & expand all copies of the repeated states in the search tree?



Case A: Cyclic State Repetition

- Question: Can the branch with the cyclic state be a part of optimal path?
- Answer: NO!

➢Branches with cycles cannot be part of the optimal solution & can be deleted.



Case A: Elimination

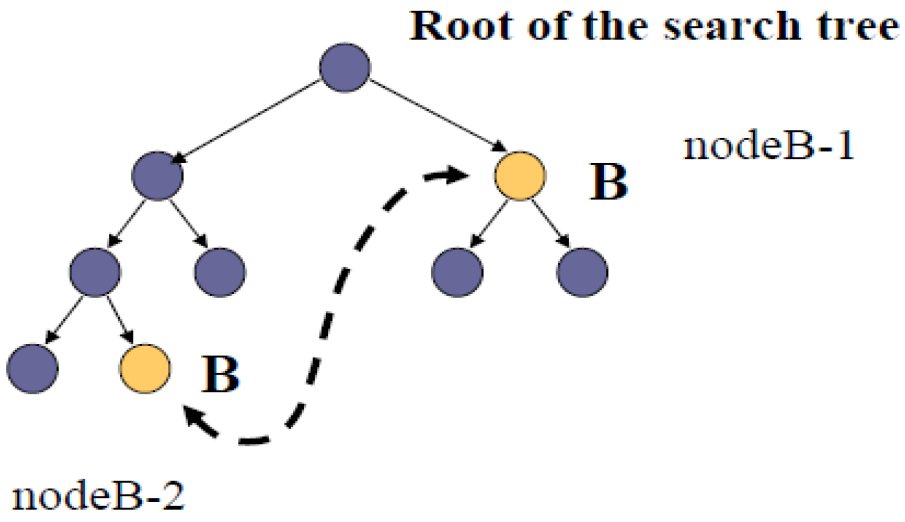
- Question: How to check for the cyclic states?
- Strategy: Check ancestors in the tree structure.
- Con: Need to keep the tree.

➢ Implementation: Do not expand the node that has the same state as one of its ancestors.

Case B: Non-cyclic State Repetition

- Question: Is one of nodeB-1 and nodeB-2 preferable?
- Answer: Yes, nodeB-1 has a shorter path between root and B.

➢ Since we are happy with the optimal solution, nodeB-2 can be eliminated.



Case B: Elimination for BFS

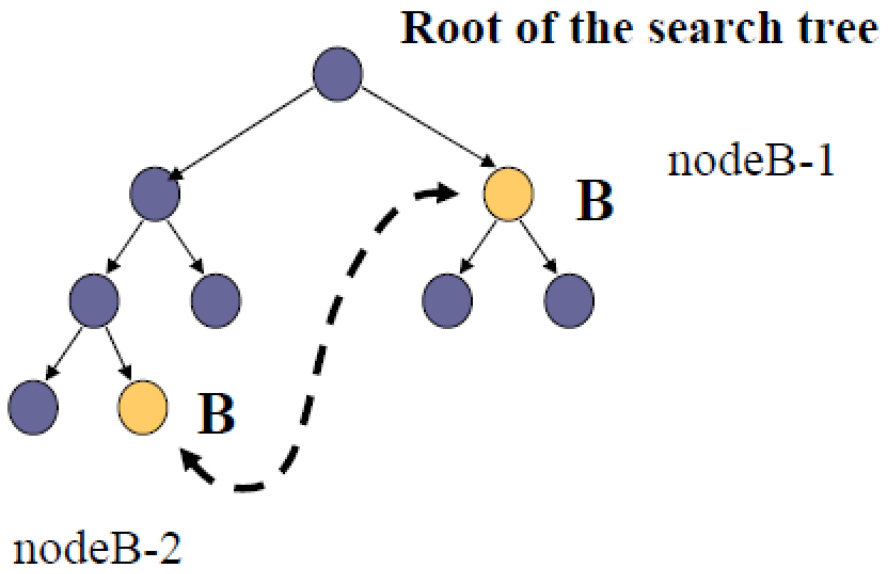
- Idea: BFS is easy as nodeB-1 is always before nodeB-2

  → Order of expansion determines the elimination strategy.

- Strategy: Eliminate all subsequent occurrences of the same state.

➢ Implementation via *marking*:

- ➢ All expanded states are marked.
- ➢ All marked states are stored in a hash table
- ➢ Check if the node has ever been expanded



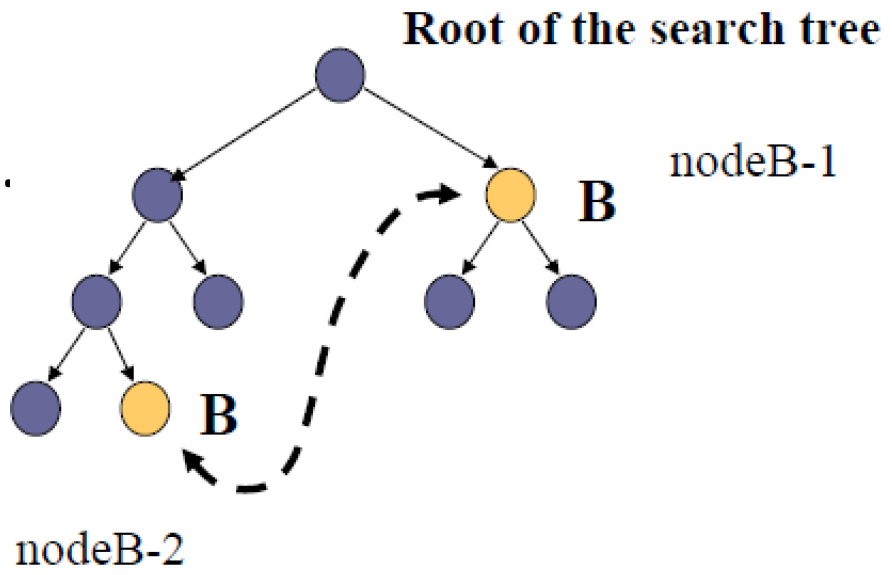
Case B: Elimination in General

- Problem: In general, nodeB-2 may be expanded before nodeB-1.
- Strategy: Eliminate a new node if ∃node with the same state & a shorter path.
- Pro: Work for any search method.
- Con: Use additional path length information.

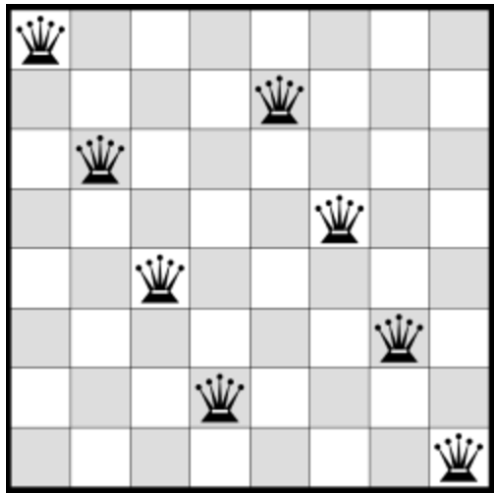
➢ Implementation: Eliminate a new node if:

  ➢ It is in the hash table, and

  ➢ Its path cost ≥ the value stored.



Case Study: n-Queens



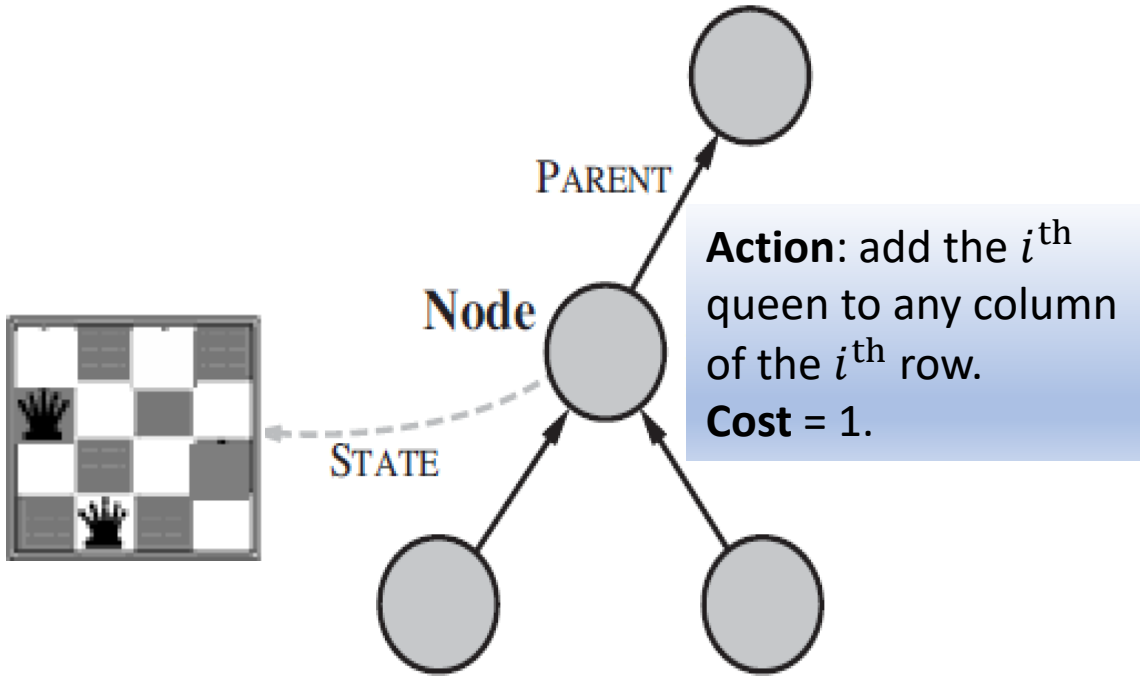
Problem Statement

- Goal: n queens placed in non-attacking positions on the board.
- Notation: x_i - a queen placed on (i, x_i).
- Constraints: for x_i, x_j $\forall i, j = 1, \cdots, n$
  - Not on the same row: one per row.
  - Not on the same column: $x_i \neq x_j$ for $i \neq j$.
  - Not diagonal: $|x_i - x_j| \neq |i - j|$.

➢CSP: Constraint satisfaction problem.

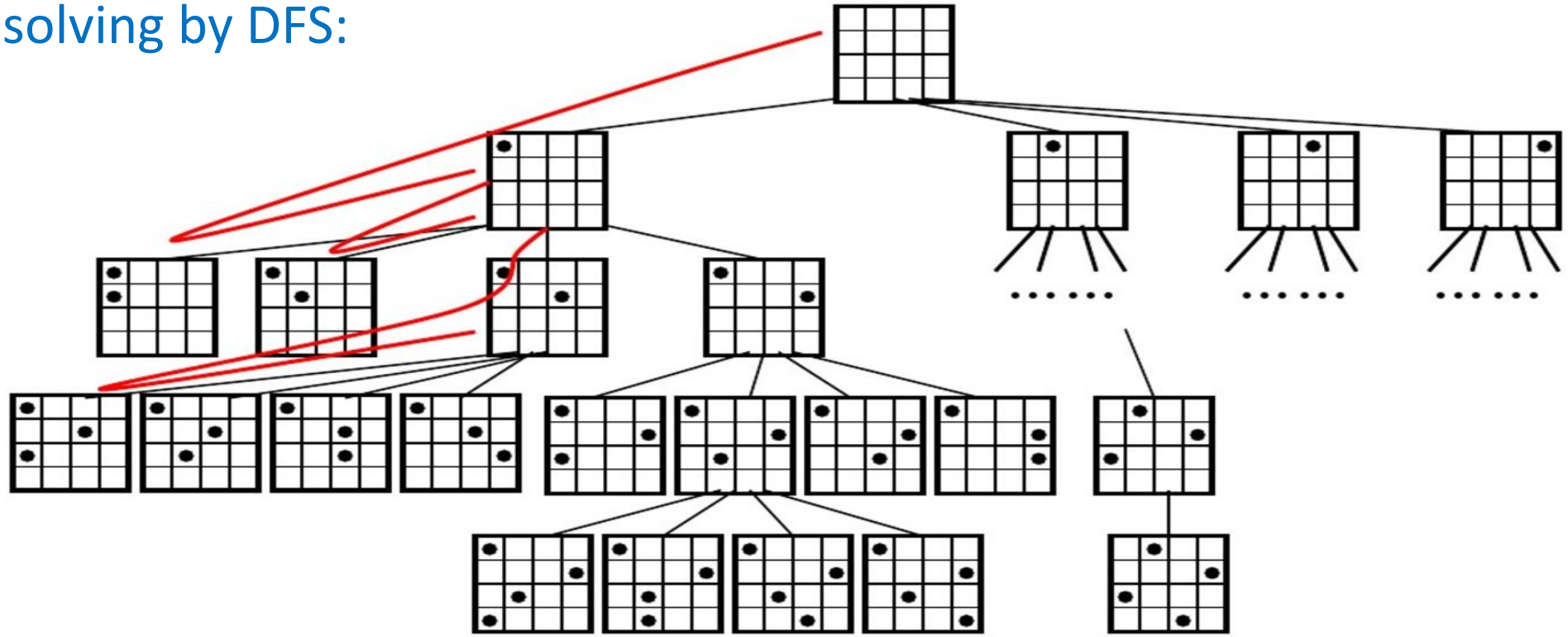
Problem Formulation

- Node: state, parent, action, path-cost



Problem Solving through Tree Search

- 4-queen solving by DFS:



- DFS in the context of CSP is also referred to as backtracking.

Summary

Basic Search Methods

- Breadth-first search (BFS): expand the shallowest node
- Uniform-cost search (UCS): expand the cheapest node
- Depth-first search (DFS): expand the deepest node
- Depth-limited search (DLS): depth first with a depth limit
- Iterative deepening search (IDS): DLS with an increasing limit
- Bidirectional search: search from both directions

Basic Search Methods: Performance

b – maximum # successors of any node in search tree.
d – depth of the least-cost solution.
m – maximum length of any path in the state space.

PF Metric	Breadth-first Search	Uniform-cost Search	Depth-first Search	Depth-limited Search	Iterative Deepening	Di-directional Search
Complete?	Yes*, if b is finite.	Yes*, if step costs$\geq \epsilon$.	No, infinite loops can occur.	No. (Eps. $l < d$)	Yes	Yes* if BFS is used for both search.
Optimal?	Yes*, if costs on the edge are non-negative.	Yes	No,	No	Yes*, if costs on the edge are non-negative.	Yes* if BFS is used & paths have a uniform cost.
Time?	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space?	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$

Basic Search Methods: Remarks

- Space and time complexities are the biggest handicaps of BFS.
- When all step costs are the same, UCS is similar to BFS.
- Space complexity of DFS is much lower than BFS.
- IDS uses only linear space and NOT much more time than other uninformed methods → preferred.

Reading materials for this lecture

- [1] AI textbook (3rd edition)
  - Chapter I.2: Intelligent Agents (pages 34-59)
  - Chapter II.3: Solving Problems by Searching (pages 64-108)
  - AI-book codes: https://github.com/aimacode
- [2] Algorithms textbook
  - Chapter 22.2: Breadth-first search (pages 595-602)
  - Chapter 22.3 Depth-first search (pages 603-612)
- [3] Basic search algorithms: http://artint.info/html/ArtInt_52.html
- [4] Chen, M. S., & Shin, K. G. (1990). Depth-first search approach for fault-tolerant routing in hypercube multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, *1*(2), 152-159.