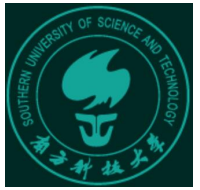
Artificial Intelligence (CS303)

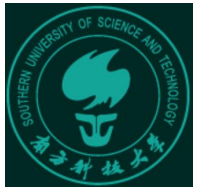
Lecture 1: Solving Problem by Searching

Outline

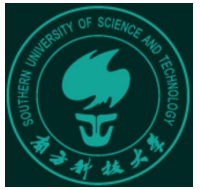


- What is search
- From searching to search tree
- Uninformed Search Methods
- Heuristic (informed) Search
- Further Studies on Heuristics

In last lecture, we discussed



- Agent
- Agent function and program
- Task Specifications (Performance, Environments, Actuator, Sensors)
- Representation



I. What is search

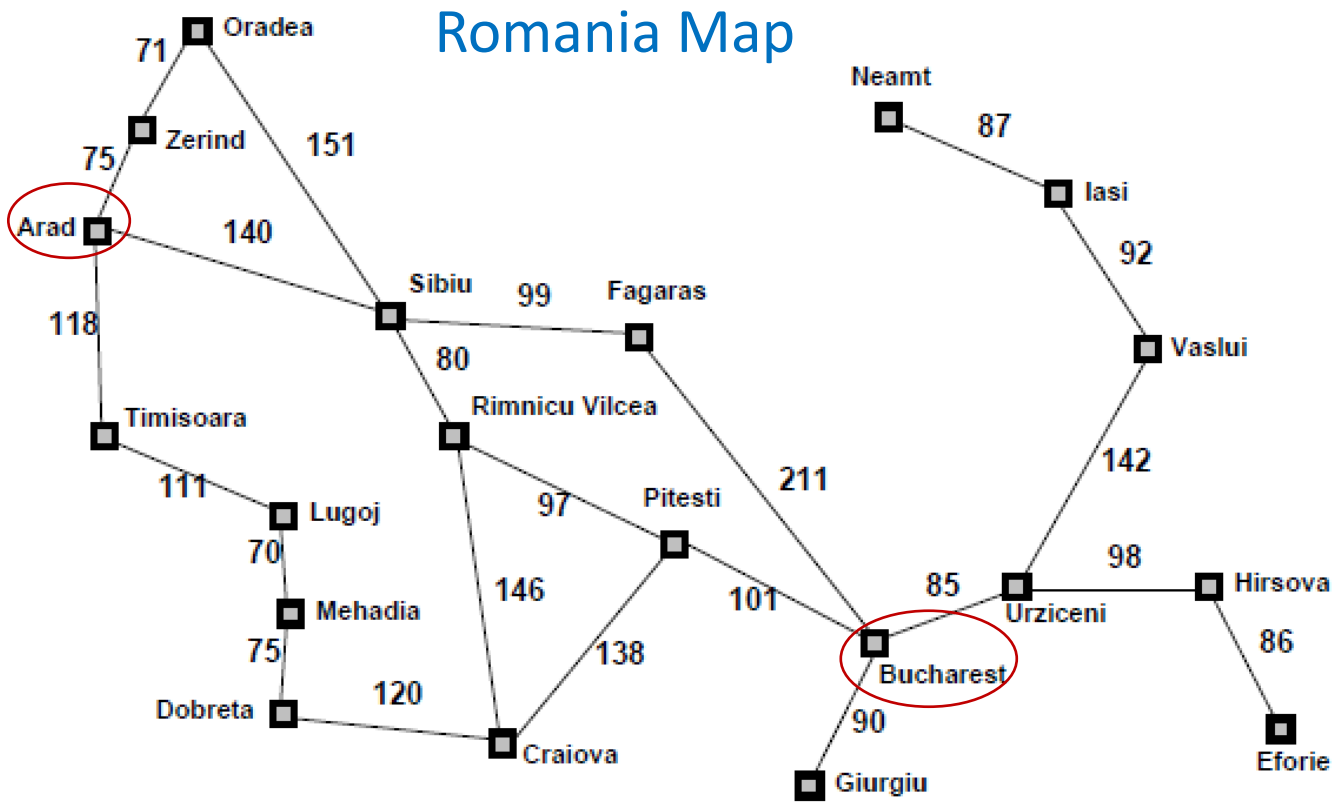
What is search?

- Given the PEAS and representation of a task, an agent needs to looking for (search) a solution that fulfils our definition of ”good performance” (e.g., successfully reach a goal, acquire sufficiently high utility).

- Search is ubiquitous (our thought process can usually be viewed as a search process).

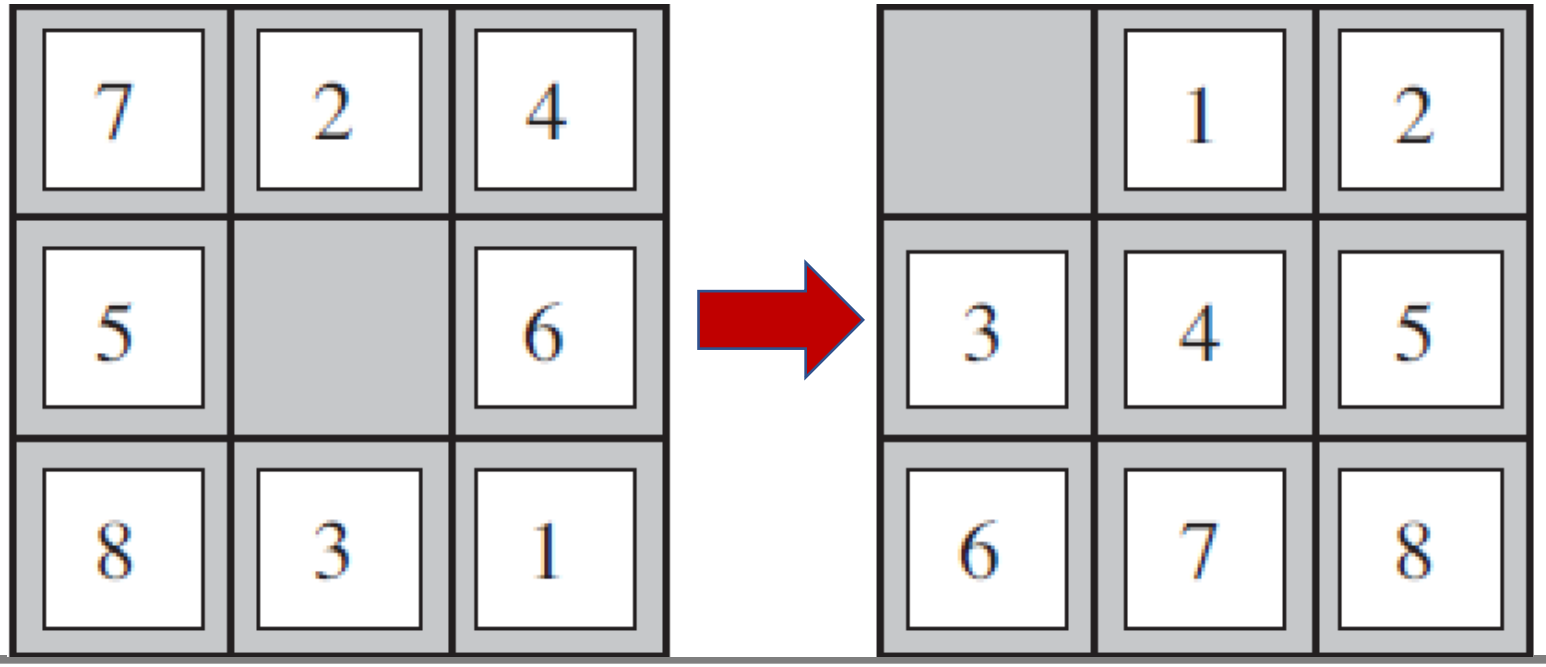
Example: Route Planning

- Find the ‘best’ route from Arad to Bucharest.
- The value on each edge is the distance.



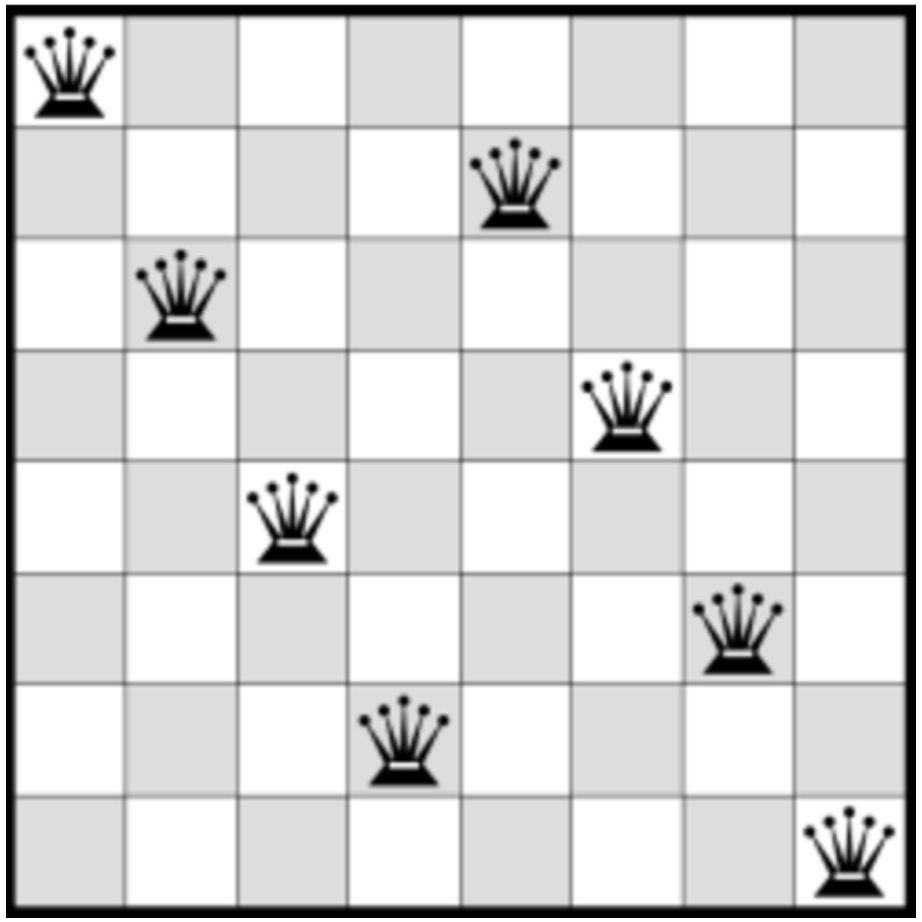
Example: 8-puzzle Problem

- Status: 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.



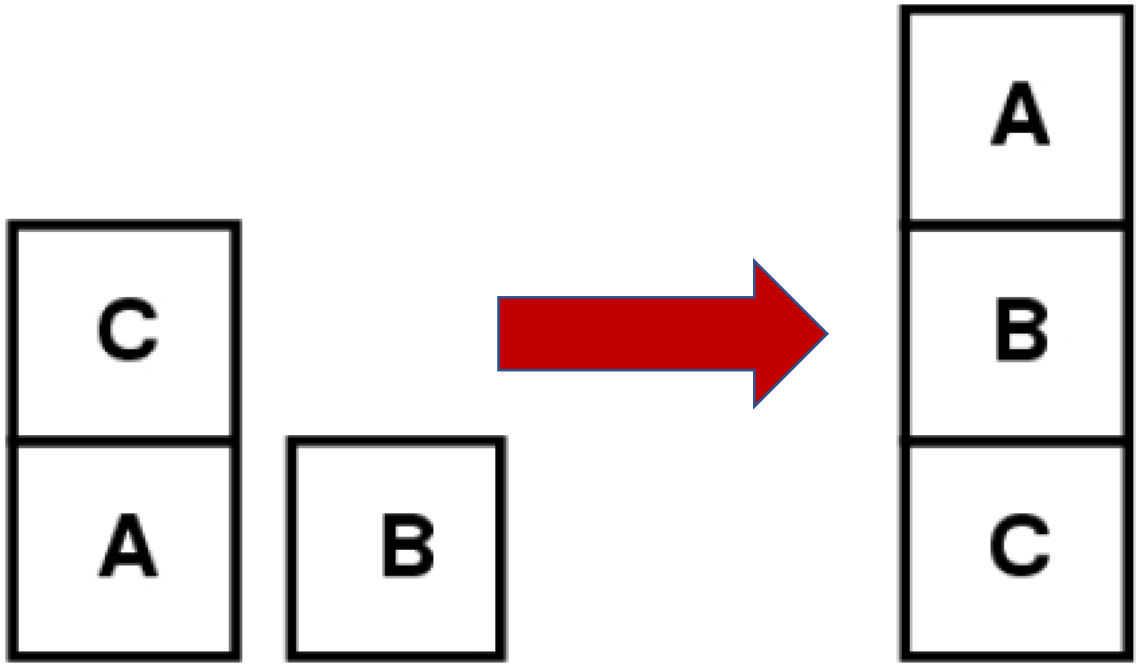
Example: 8-queen Problem

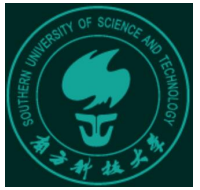
- Objective: On a chess board, place 8 queens so that no queen is attaching by any other horizontally, vertically or diagonally.



Example: Planning in the Block World

- Objective: How can we get from Left to Right?

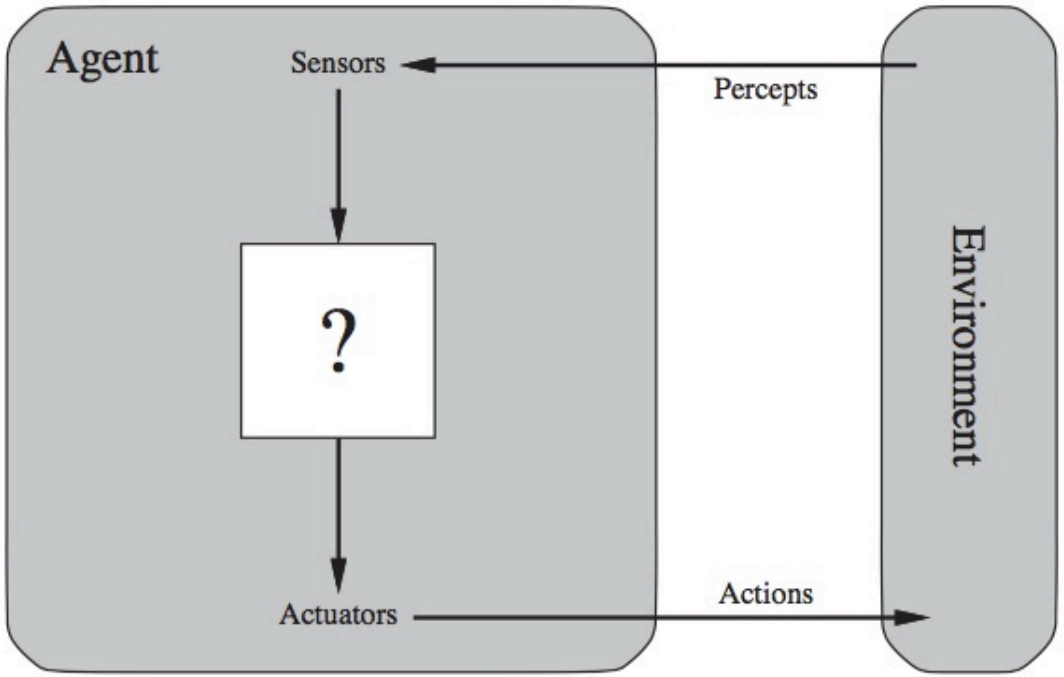




Any common features between different tasks?

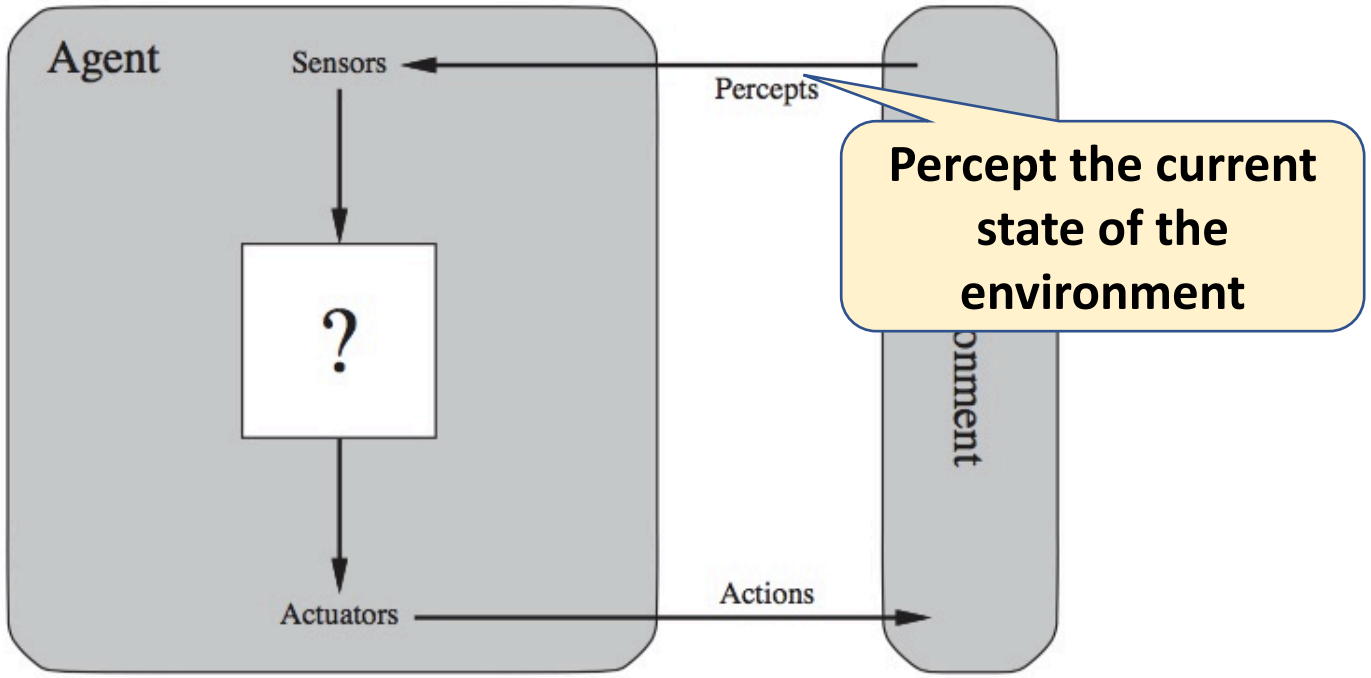
What is search?

- The state-action description of a task/problem.



What is search?

- The state-action description of a task/problem.

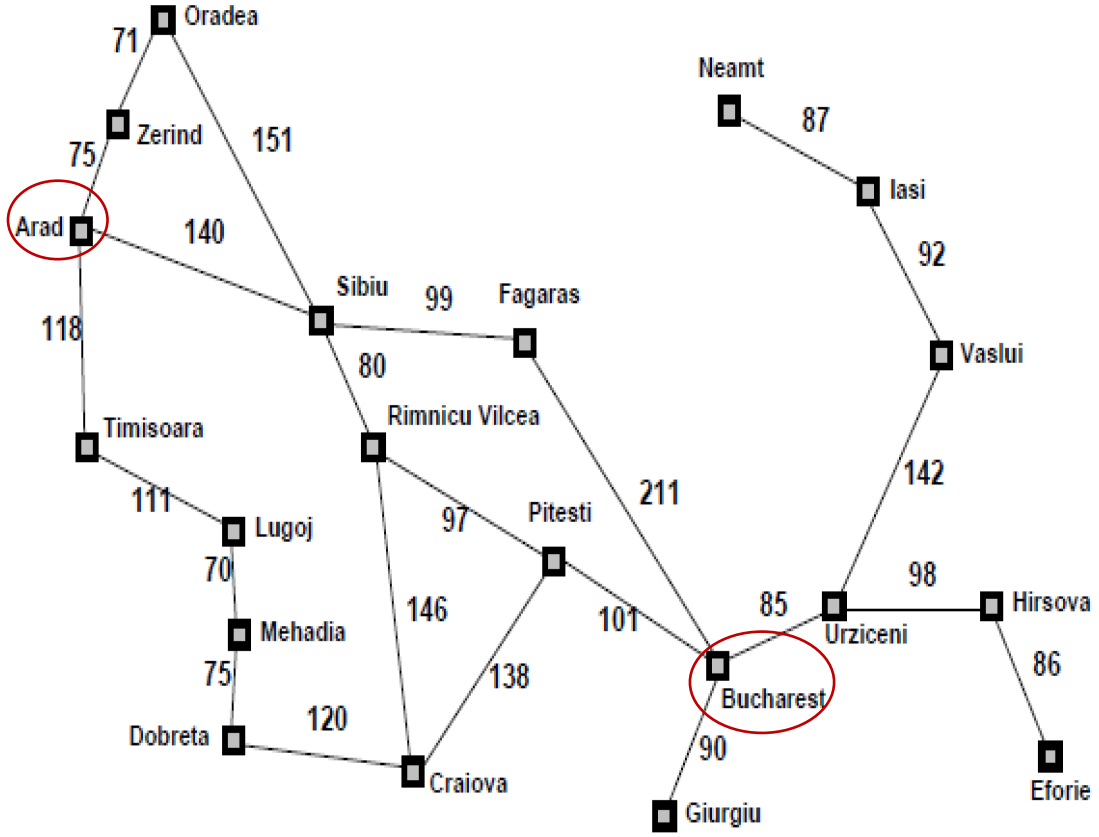


The State-Space Formulation

- States: all reachable states from the initial by any sequence of actions.
- Initial state: the state where the problem starts.
- Actions: the legal actions between states.
- Transition model: a description of what each action does $RESULT(s, a)$.
- Goal test: determine whether a given state is a goal state.
- Path cost: function that assigns a numeric cost to each path regarding PF.

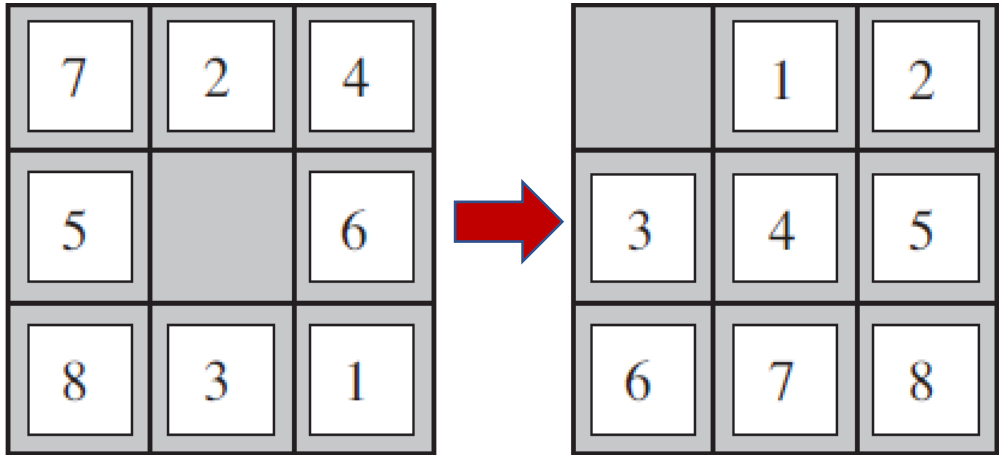
Formulation: Route Planning

- States: {(cur_city, walk_dist)}.
- Initial state: (‘Arad’, 0).
- Actions: walk to the adjacent city.
- Transition: new city & walk distance.
- Goal test: reach ‘Bucharest’?
- Path cost: accumulate walk distance.



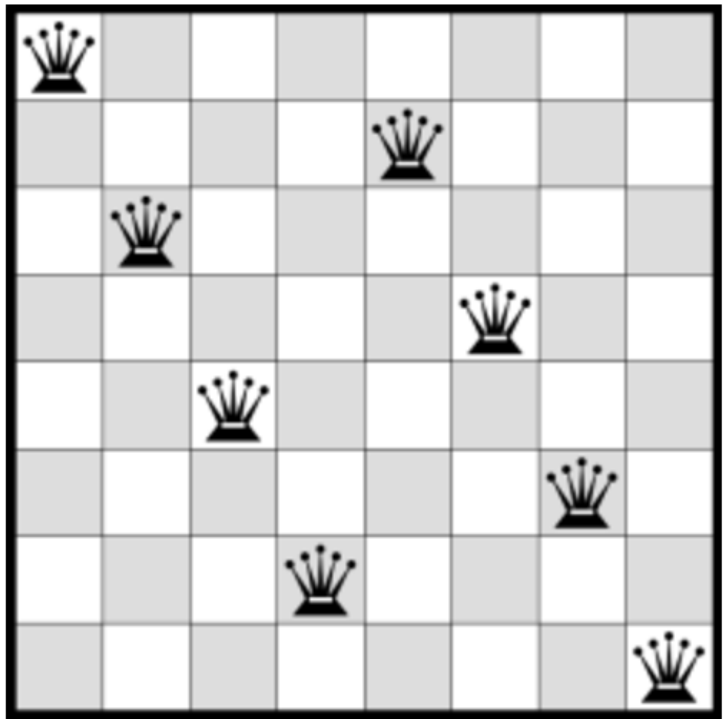
Formulation: 8-puzzle Problem

- States: all allocations of the 9 tiles.
- Initial state: the allocation of the Left.
- Actions: the movement of the blank space $\{left, right, up, down\}$.
- Transition: updated allocations of the tiles.
- Goal test: the allocation of the Right?
- Path cost: #moves.



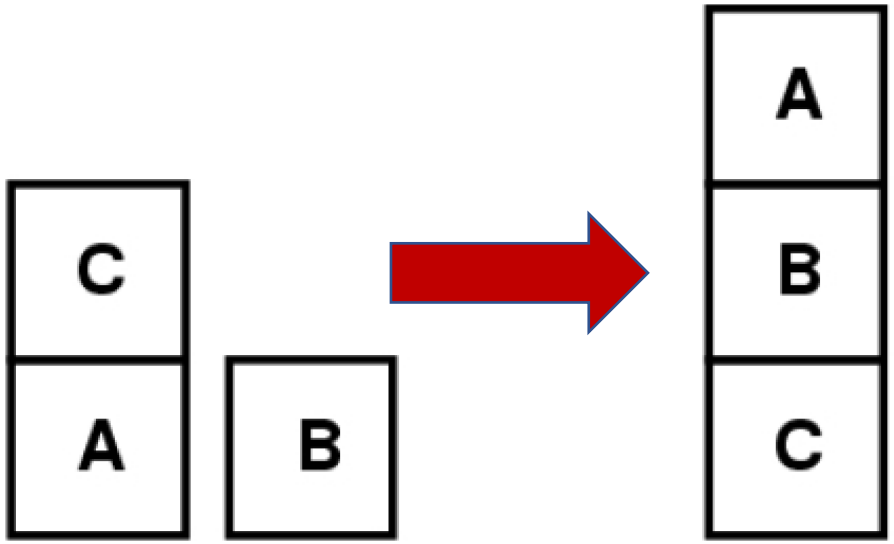
Formulation: 8-queen Problem

- States: All arrangements of 8 queens on the board, $\# = C_{64}^{8}$.
- Initial state: No queen on the board.
- Actions: Add a queen to any empty square.
- Transition: updated board.
- Goal test: 8 queens on board without attacked?
- Path cost: #search.



Formulation: Planning in the Blocks

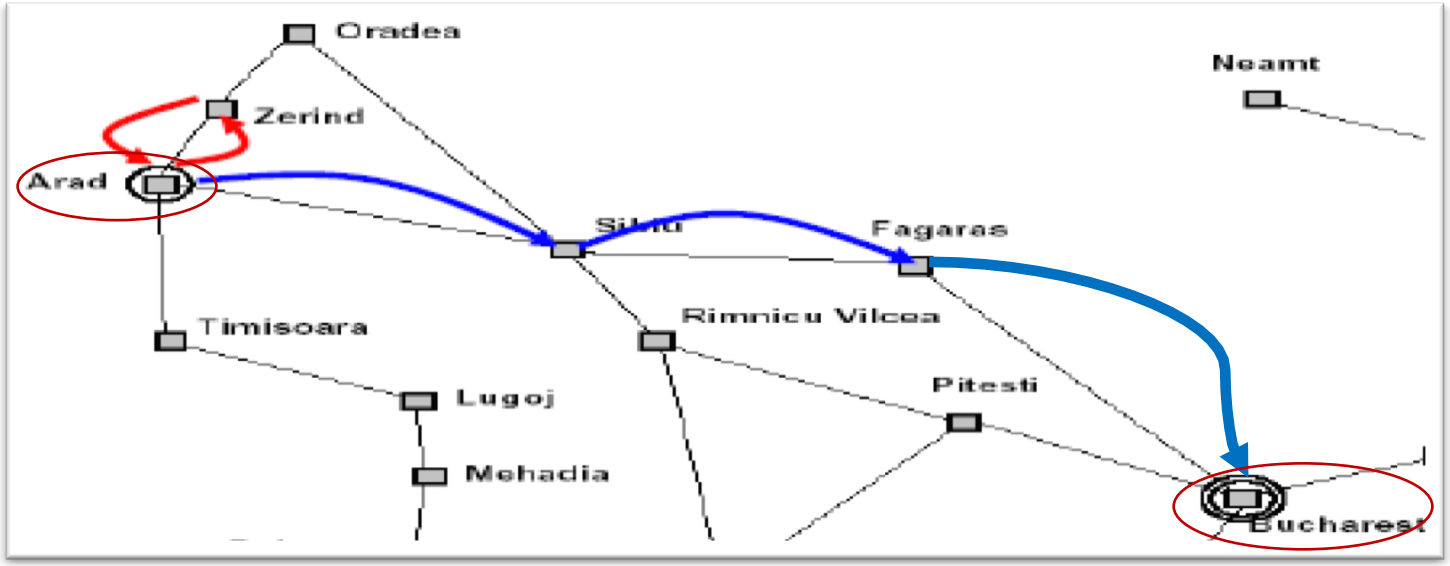
- States: ?
- Initial state: ?
- Actions: ?
- Transition: ?
- Goal test: ?
- Path cost: ?



Searching for Solution

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

➢ We can find the problem solution by searching.

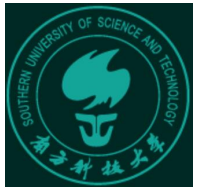


Searching for Solution

- The space in which the agent search for a solution is defined by solution representation.

- Is there a **universal representation** for all possible (intelligent) tasks?

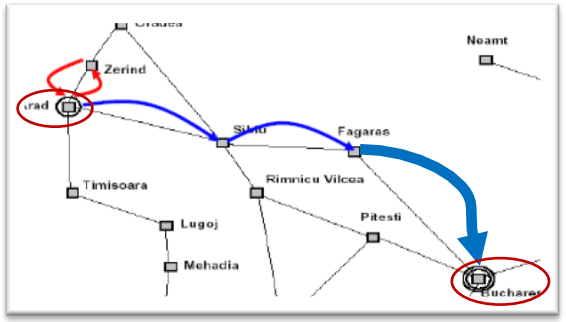
- The solution/search space could be overwhelmingly huge for some tasks (regardless of the representation used), that's why we need more than naïve search methods.

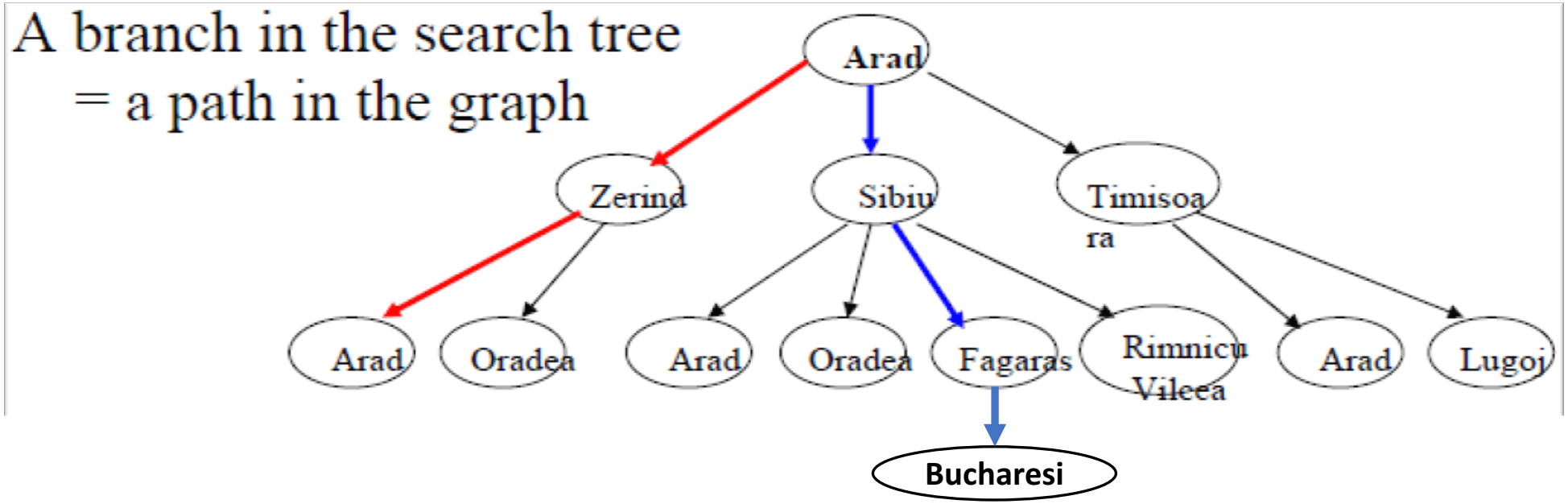


II. From Searching to Search Tree

Searching for Solution by Tree

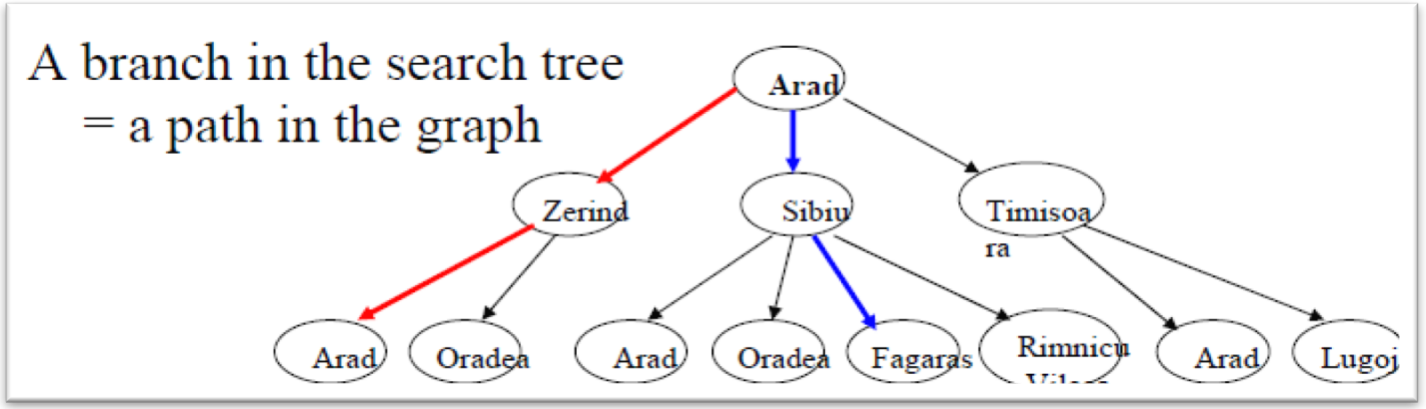
- Question: How can we formalize a searching process?
- Answer: Search in a tree.





Search Tree

- Search tree models the sequence of legal actions.
  - Root: initial state.
  - Nodes: the states resulting from actions.
  - For a given state, its node has a child node with a follow-up state.
  - Branch: a sequence of state (and thereby sequence of actions).
- Expand: create all children nodes for a given a node.



Structure of Tree Node

- Tree node: a data structure to keep track of the search tree that is being constructed.
- 4 components of node n:
  - n.STATE: node n's state.
  - 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.

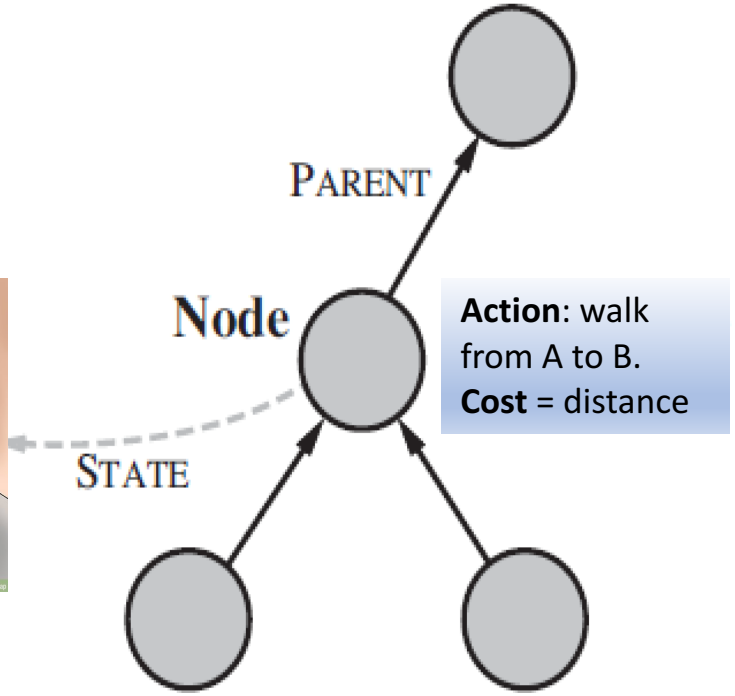
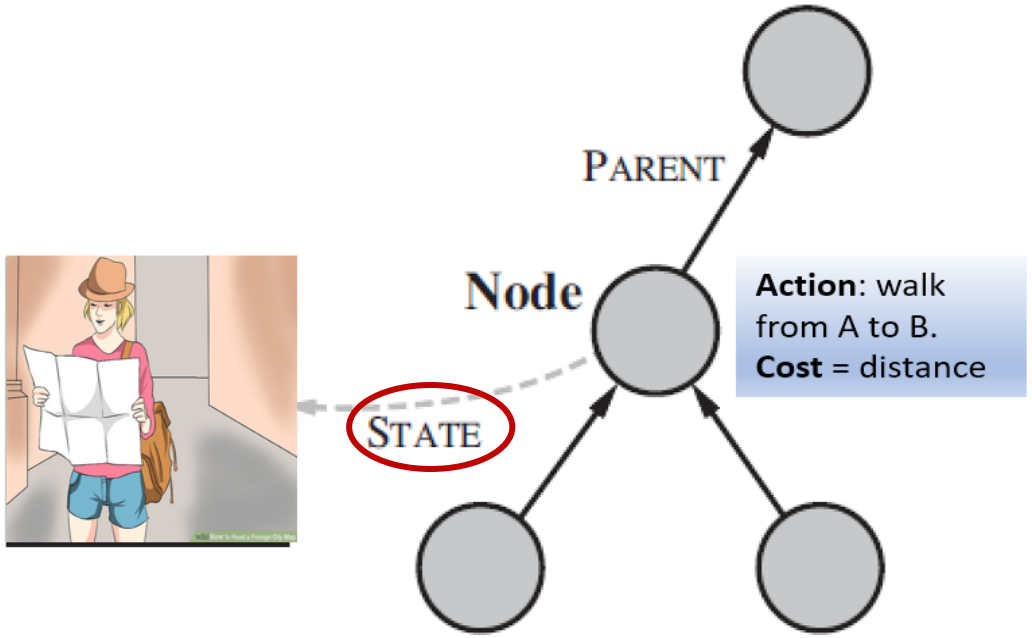


Fig. Structure of tree node with which search tree is constructed. Arrows point from child to parent.

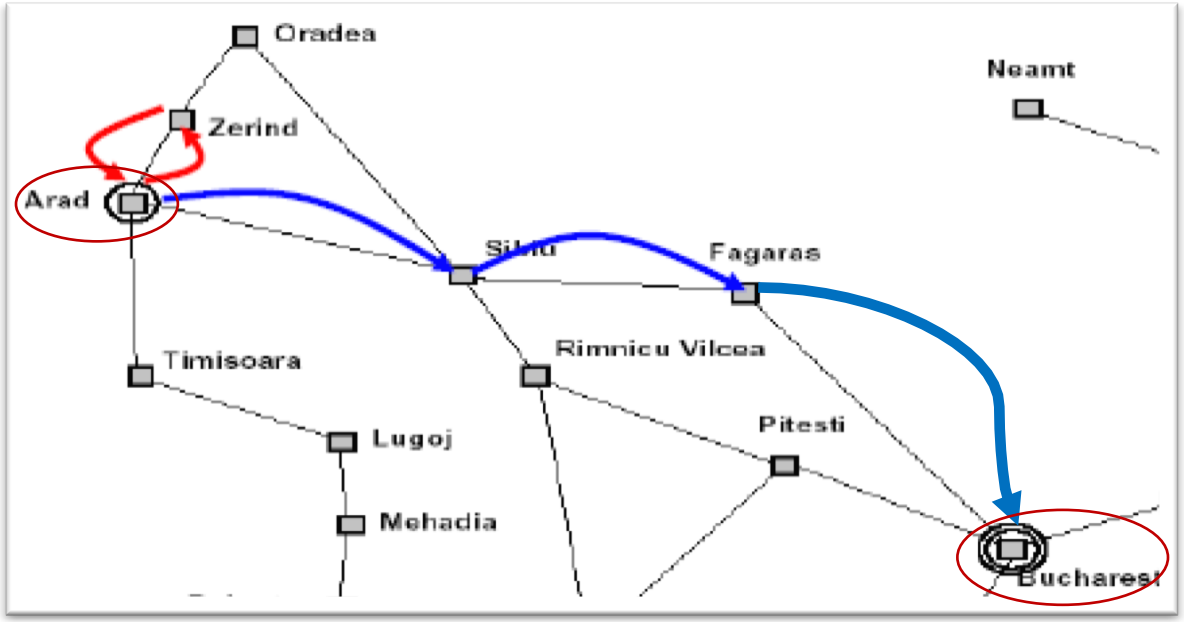
Nodes vs States

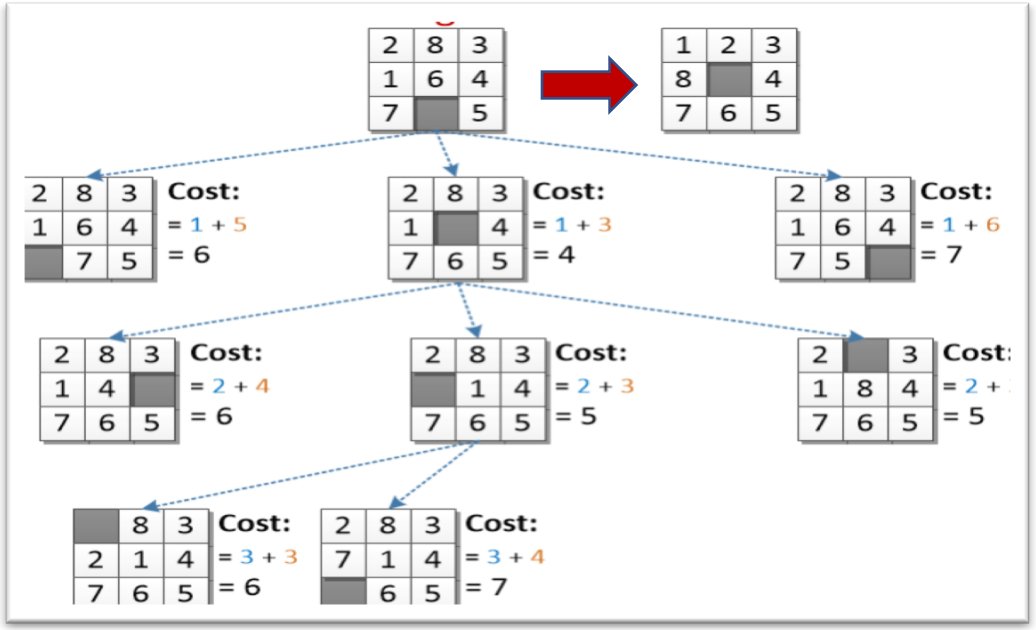
- State is a component of node.
  - A node is a data structure constituting part of a search tree.
  - A state is the current status of a node.



Search Method

- Search methods:
  - defined by picking the order of node expansion.
  - aim to identify the 'best' solution.





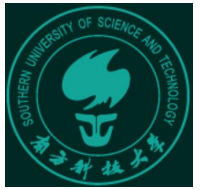
Search Methods: Performance Metrics

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

Search Methods: Performance Metrics

In general, time and space complexity depend on:

- 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.

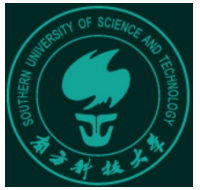


III. Uninformed Search Methods

Search Methods

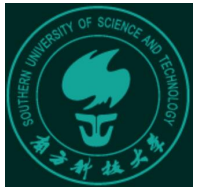
- Search methods differ in how they explore the space, i.e., how they choose the node to expand NEXT.

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 shallowest unexpected node.
- Implementation: a FIFO queue.

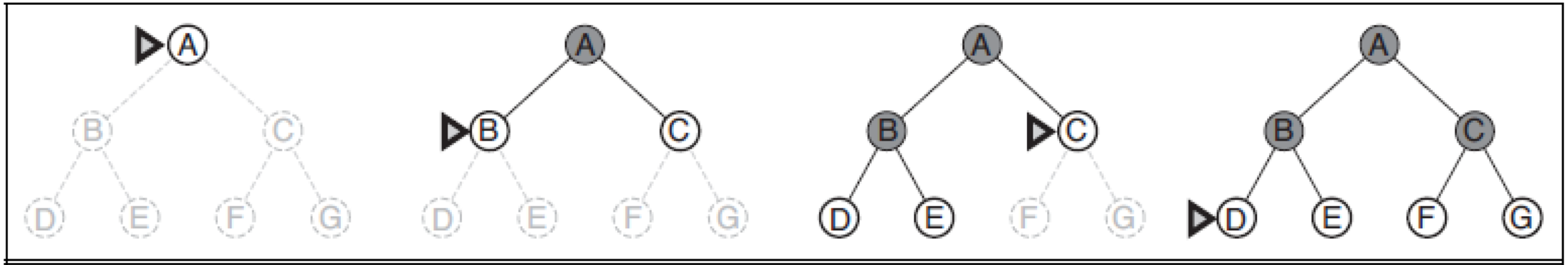
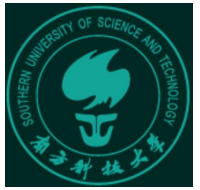


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

BFS: PF Metrics

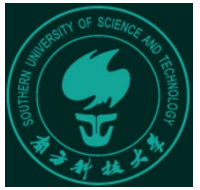


b – maximum # successors of any node in search tree.
d – depth of the least-cost solution.
m – maximum length of any path in the state space.

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

➢ Memory + exponential time complexities are the biggest handicaps of BFS.

BFS: Pseudo-code

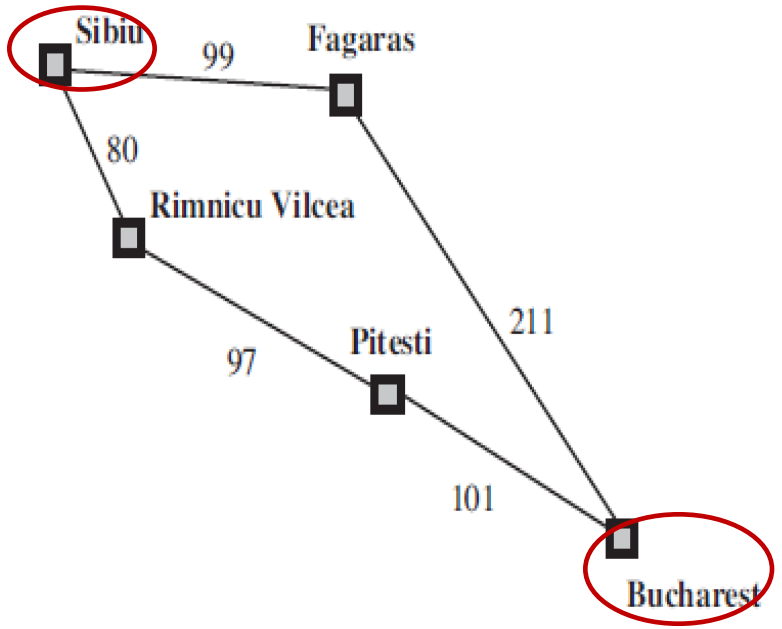



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop 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)
```

Uniform-cost Search (UCS)

- The costs in the search tree may be different.
- Expand cheapest unexpanded node.
- Implementation: a queue ordered by path cost, 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]}



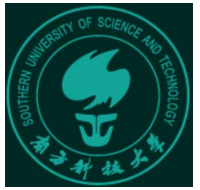
UCS: PF Metrics

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 $O(b^{d+1})$.
  - When all step costs are equal, $O(b^{1+\lfloor C^*/\epsilon \rfloor}) = O(b^{d+1})$.

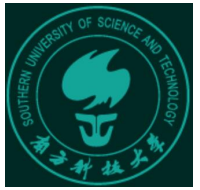
➢ When all step costs are equal, UCS is similar to BFS.

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
```

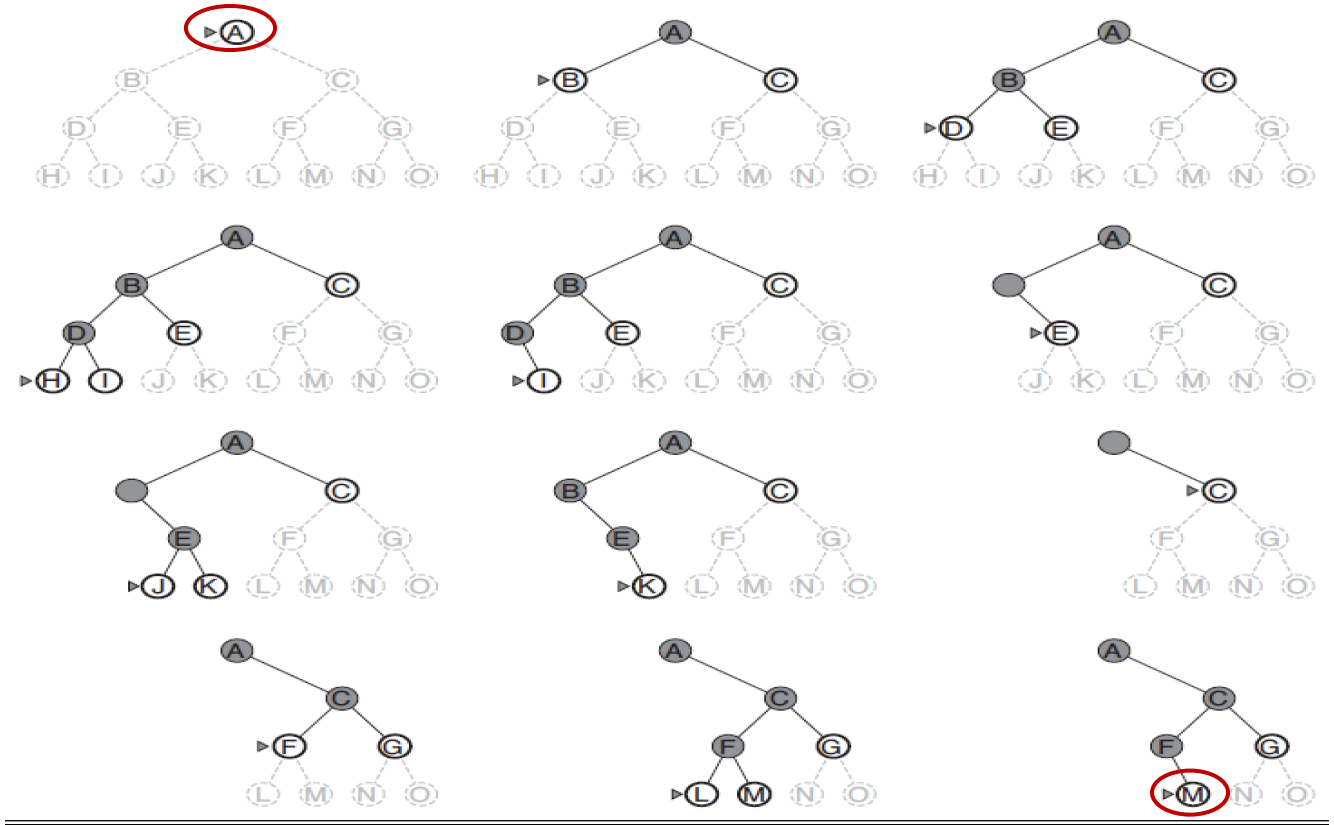


Depth-related Search Methods

Depth-first Search (DFS)

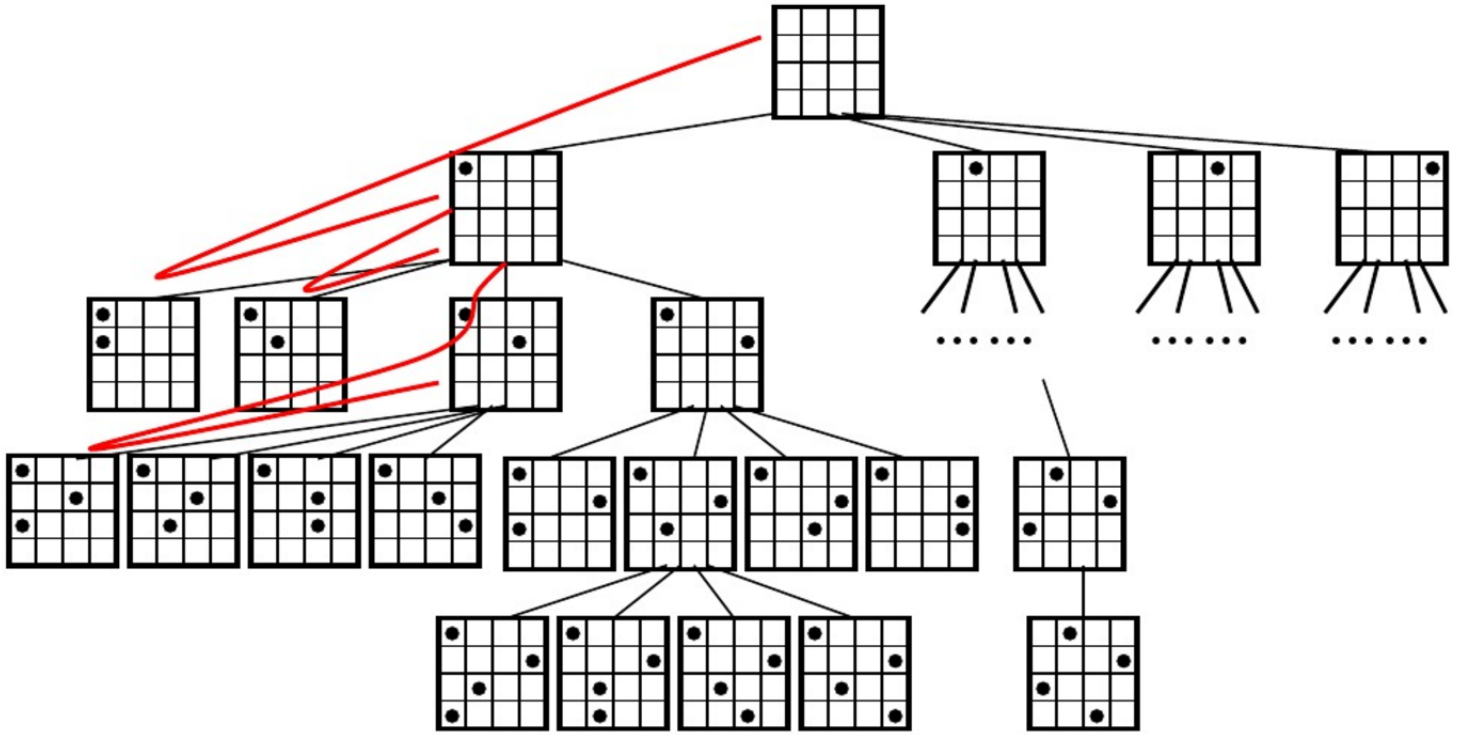
- Expand deepest unexpanded node.
- Implementation: LIFO stack.

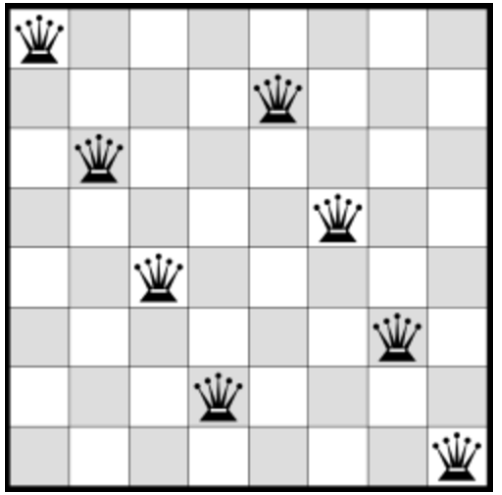
- Task: Search from A to M.
- Note: Once a node is expanded, it is removed from memory asap all its children are explored.



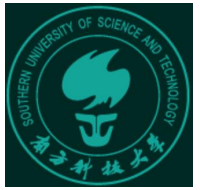
Problem Solving as a Tree Search

- Example: 4-queen solving by DFS





DFS: PF Metrics



- Complete? No, fail in infinite-depth space and space with loops.
- Optimal? No.
- 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.

DFS: Pseudo-code

```
DFS(G,v)   ( v is the vertex where the search starts )
   Stack S := {};   ( start with an empty stack )
   for each vertex u, set visited[u] := false;
   push S, v;
   while (S is not empty) do
      u := pop S;
      if (not visited[u]) then
         visited[u] := true;
         for each unvisited neighbour w of u
            push S, w;
      end if
   end while
END DFS()
```

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 traveller problem with 20 cities → $l < 20$.
  - DLS is the variant of DFS.

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

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

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
```

Iterative Deepening Search (IDS)

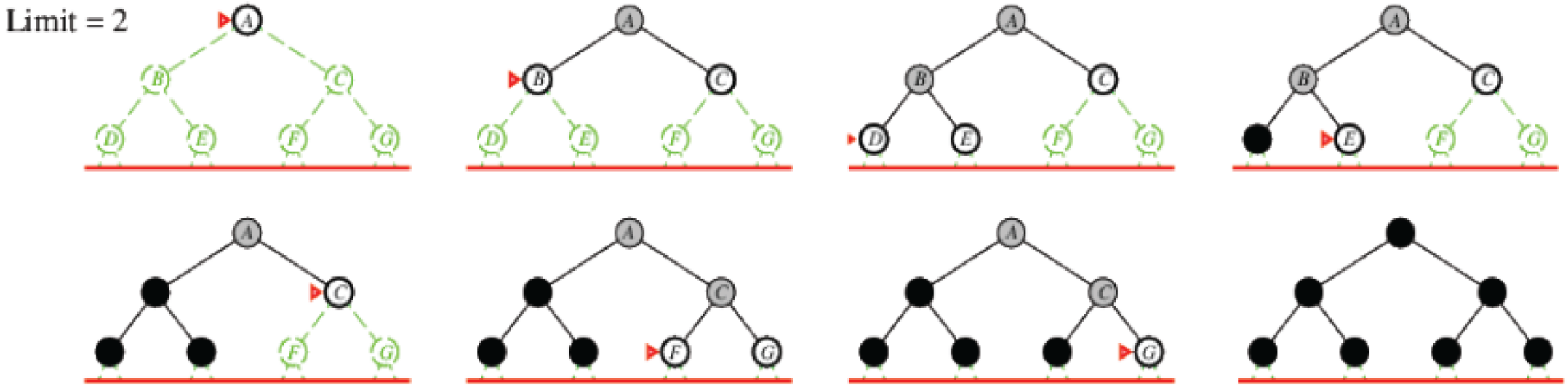
- Apply DLS with increasing limits.
- Combine the benefits of BFS and DFS.
  - Like BFS, complete when b is finite & optimal when the path cost is non-decreasing regarding depth of the nodes.
  - Like DFS, space complexity is $O(bd)$.

Iterative Deepening Search (IDS)

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

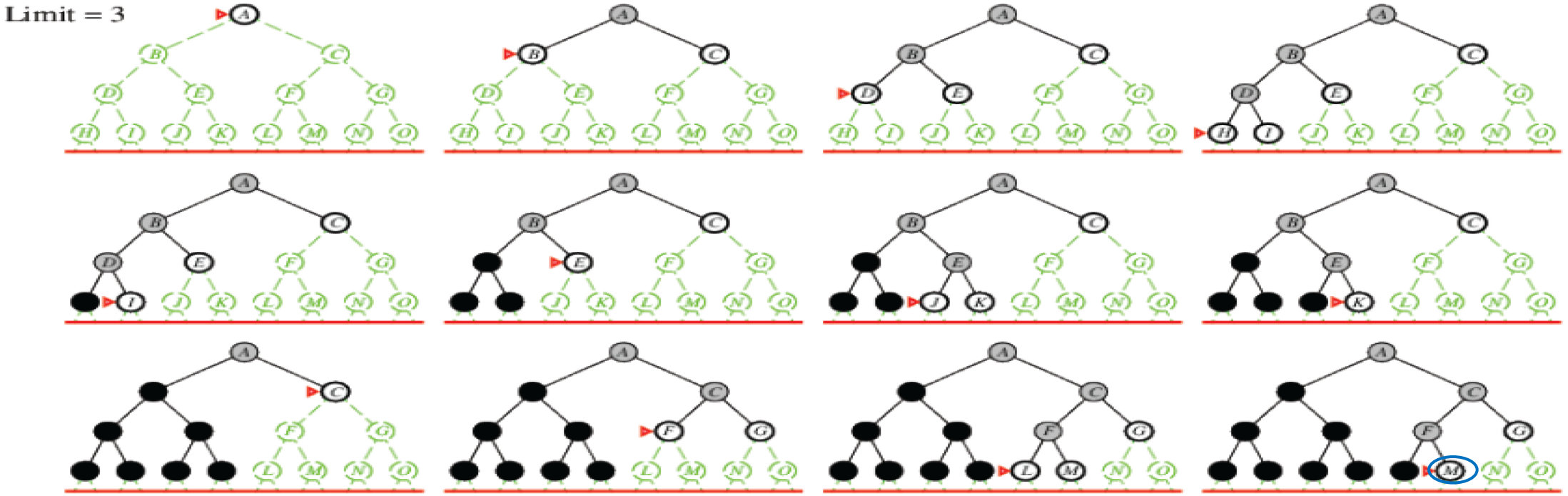
Iterative Deepening Search (IDS)

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

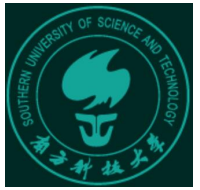


Iterative Deepening Search (IDS)

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



IDS: PF 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.

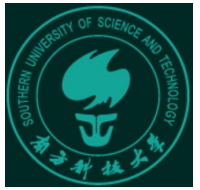
IDS: Pseudo-code

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

  for $depth = 0$ **to** ∞ **do**

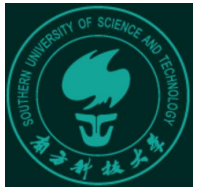
    $result \leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)

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



Bidirectional Search Method

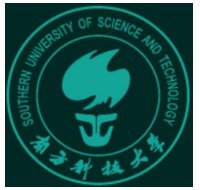
Bidirectional Search



- Search from both directions simultaneously.
- Replace 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-graphs intersect.

Bidirectional Search: PF Metrics

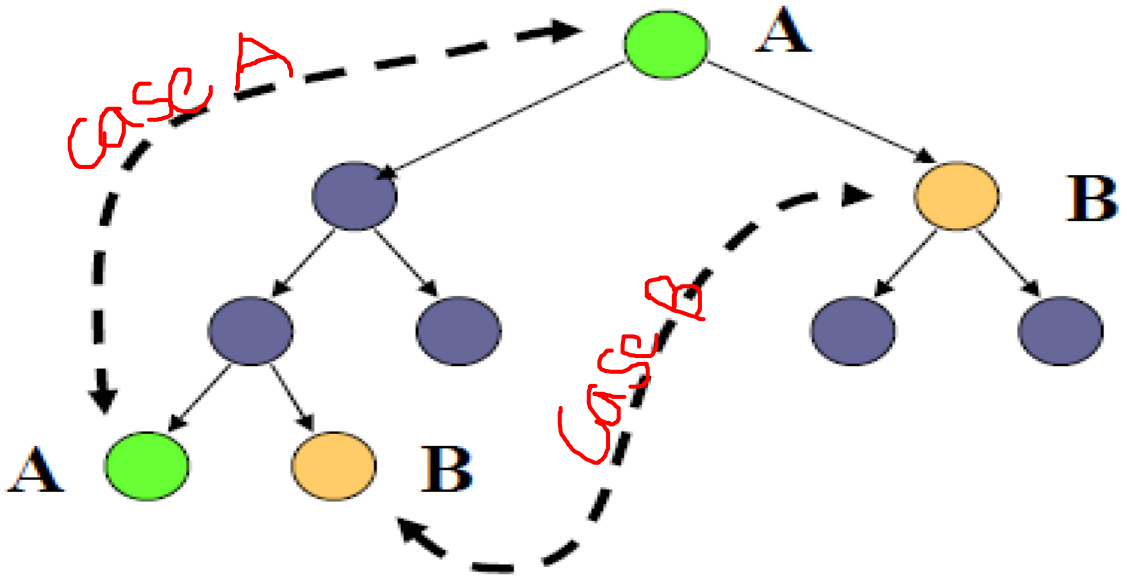
- Complete? Yes, if BFS is used in both search.
- Optimal? Yes, if BFS is used & paths have uniform cost.
- Time? Space? $O(b^{d/2})$
- Disadvantage? Not always applicable (Reversible actions? Explicitly stated goal state).



Elimination of State Repeats

Two Cases of State Repeats

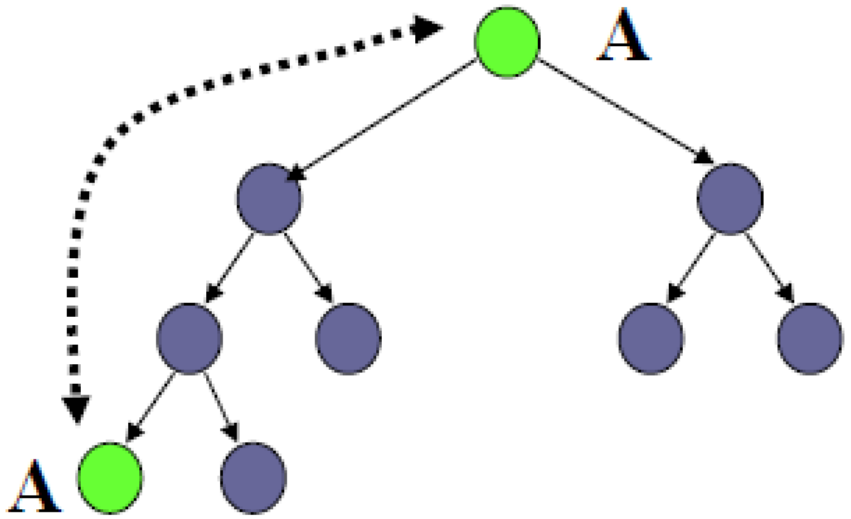
- [Case A] Cyclic state repeats. [Case B] Non-cyclic state repeats.
- Question: Is it necessary to keep & expand all copies of the repeated states in the search tree?



Case A: Cyclic State Repeats

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

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



Case A: Elimination

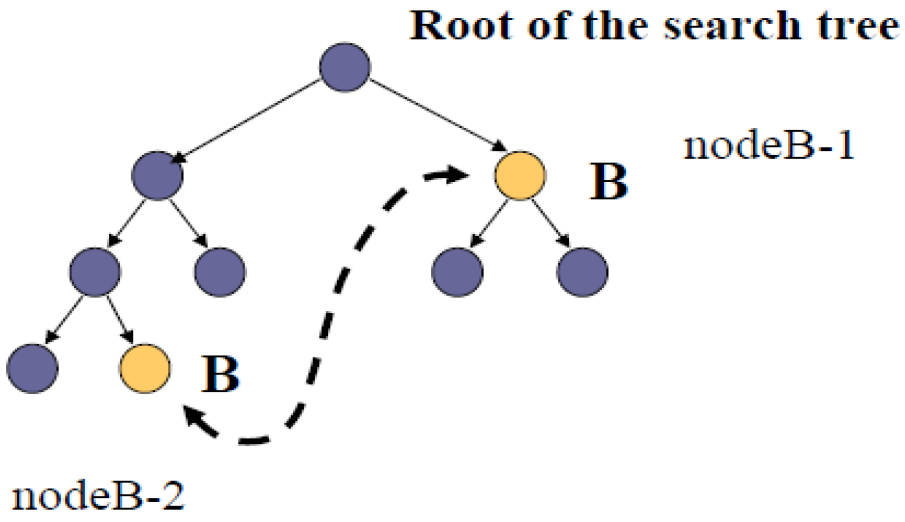
- Question: How to check for the cyclic states?
- Strategy: Check ancestors in the tree structure.
- Con: Need to keep the tree.

➢ Implementation: Do no expand the node that has the same state as one of its ancestors.

Case B: Non-cyclic State Repeats

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

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

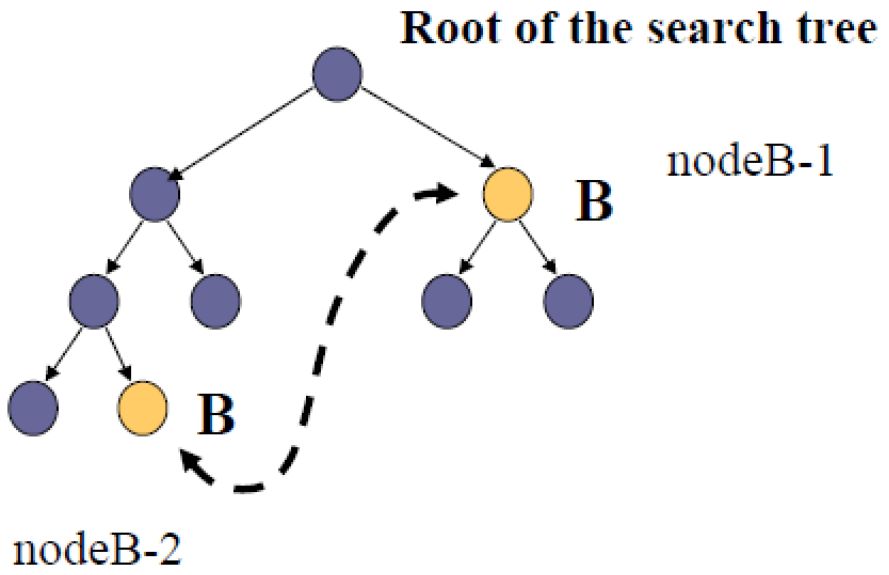


Case B: Elimination for BFS

- 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



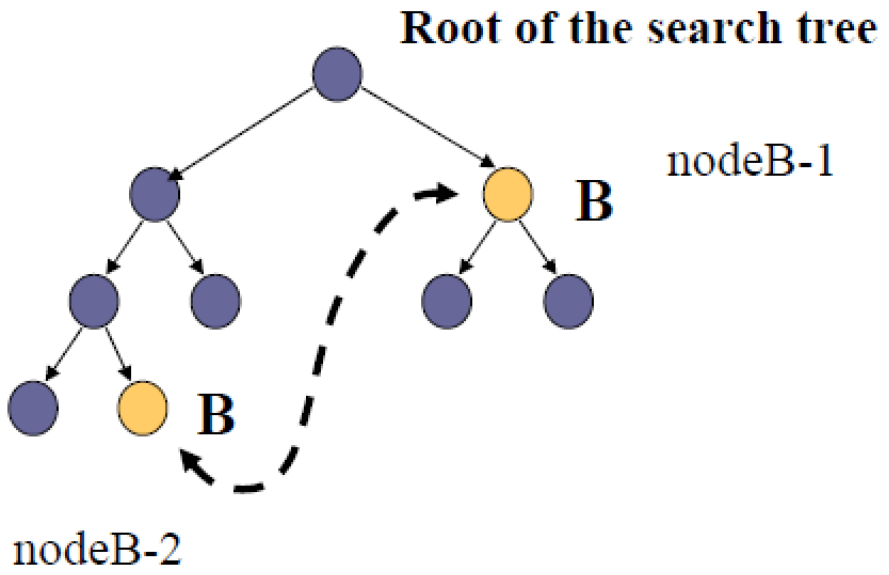
Case B: Elimination for General

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

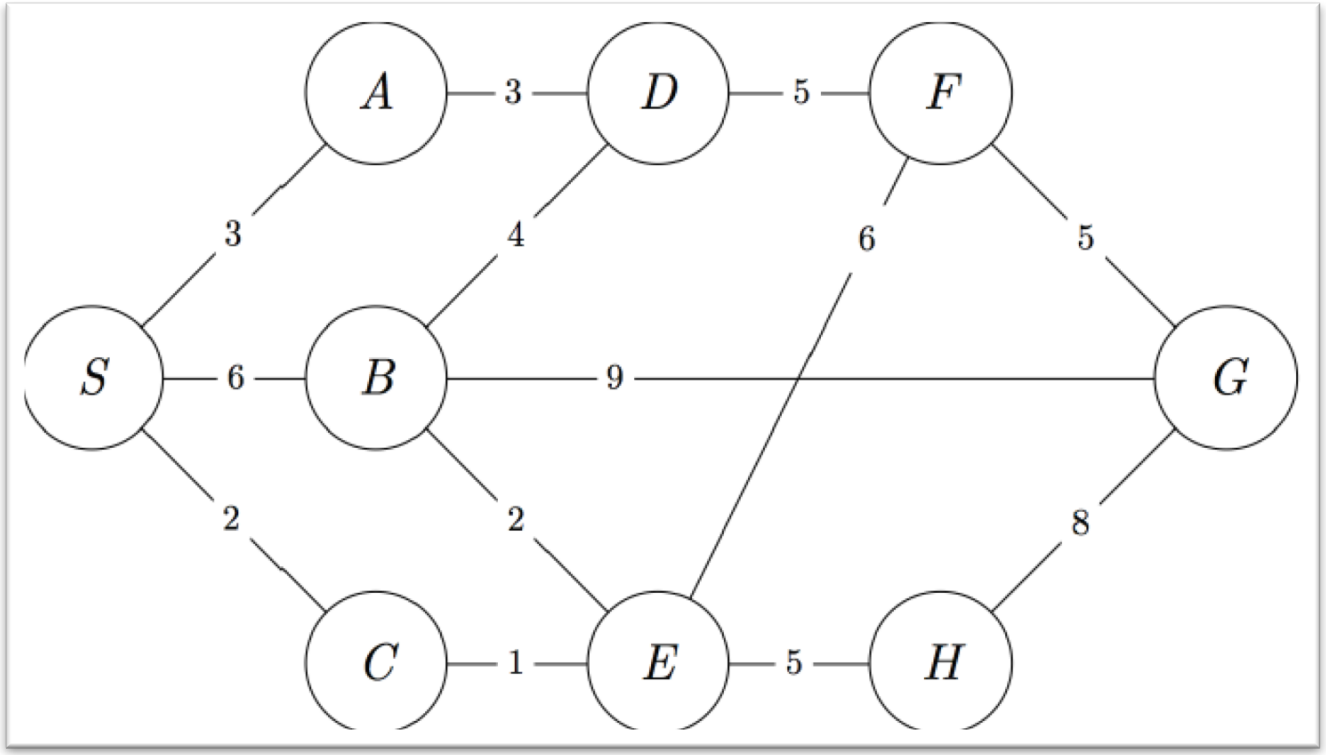
➢ Implementation: Eliminate new node if:

  ➢ It is in the hash table, and

  ➢ Its path cost ≥ the value stored.



Tree-Search Methods



Question: find search trees & paths of S → A by BFS, DFS and UCS?

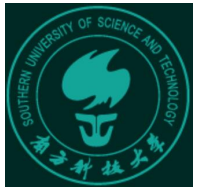
Basic Search Methods: PF

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 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 uniform cost.
Time?	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space?	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$

Basic Search Methods: Briefs

- Memory + exponential 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.



IV. Heuristic (informed) Search

From Basic Search to Heuristic Search

- Basic (uninformed) search uses NO domain knowledge.
- Which node to expand is (in part) decided arbitrarily (sometimes randomly).
- Heuristics: which part of the search space to explore? ⇒ help direct the search.
- Heuristic search uses domain knowledge.

Heuristic Function

- Question: Where to impart domain knowledge?
- Answer: Heuristic function $h(n)$ at node n.

➢ Heuristic function $h(n)$ estimates the cheapest cost from n to goal.

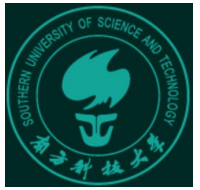
➢(1) $h(n) = 0$ if n the goal node.

➢(2) nonnegative.

➢(3) problem-specific.

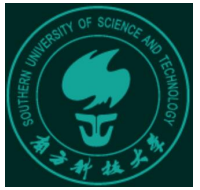
- Define a good h endows with some intelligence ➔ A little AI now.

Heuristic Function: Examples



- $h_0(n) = 0$.
- $h^*(n) =$ the true cost from n to the goal.
- $h_{SLD}(n) =$ straight-line distance from node n to goal.

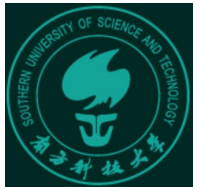
Evaluation Function



- Evaluation function $f(n)$ is constructed as a cost estimate, and the node n with the lowest $f(n)$ is expanded first.

- Most best-first search methods include a heuristic function $h(n)$ as a component of $f(n)$.

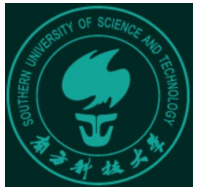
The choice of f determines the search strategy.



Best-first Search Methods:

Greedy Best-first Search

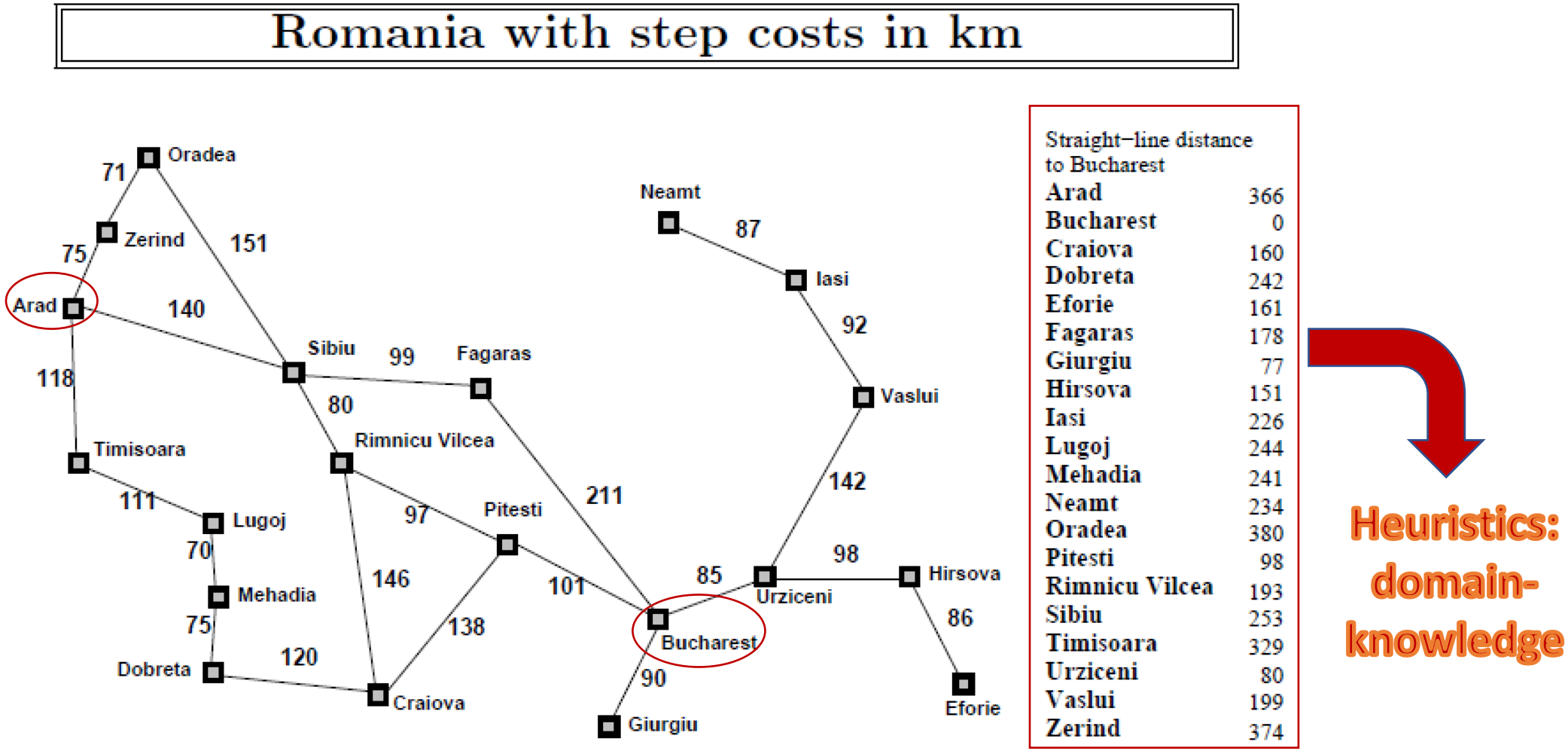
Core Idea



- Expand node n that has the minimal $f(n) = h(n)$.
- Interpretation: Expand the node that seems closest to the goal.

- A generalization of UCS, i.e., use of $f(n)$ instead of the path cost from the start to node n.

Greedy Search: Searching Illustration

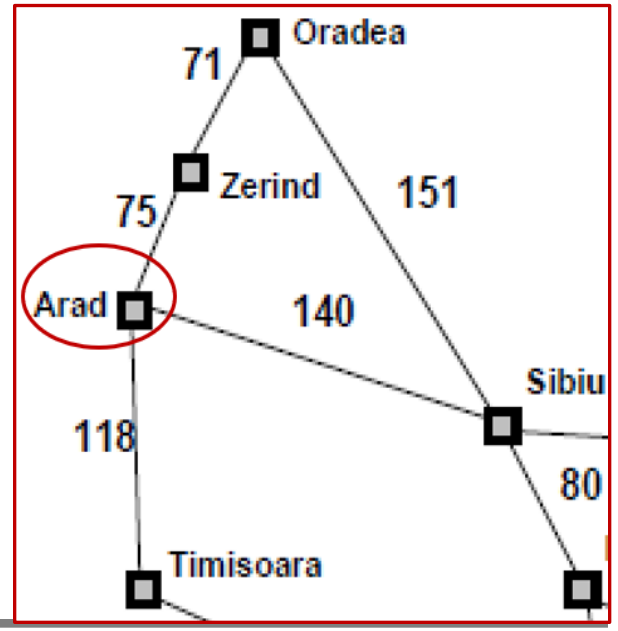


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Heuristics: domain-knowledge

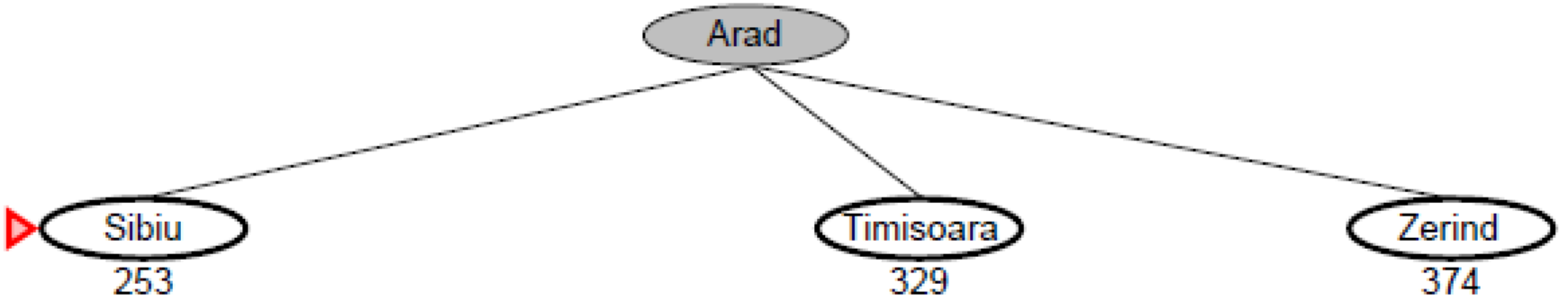
Greedy Search: Searching Illustration

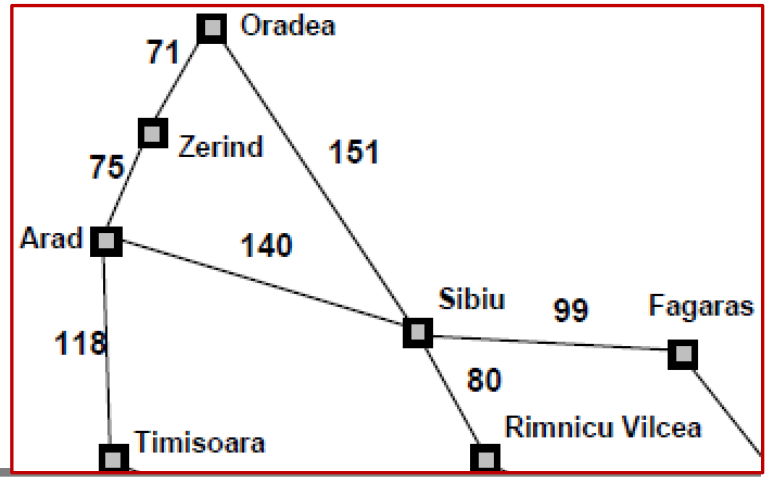
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Greedy Search: Searching Illustration

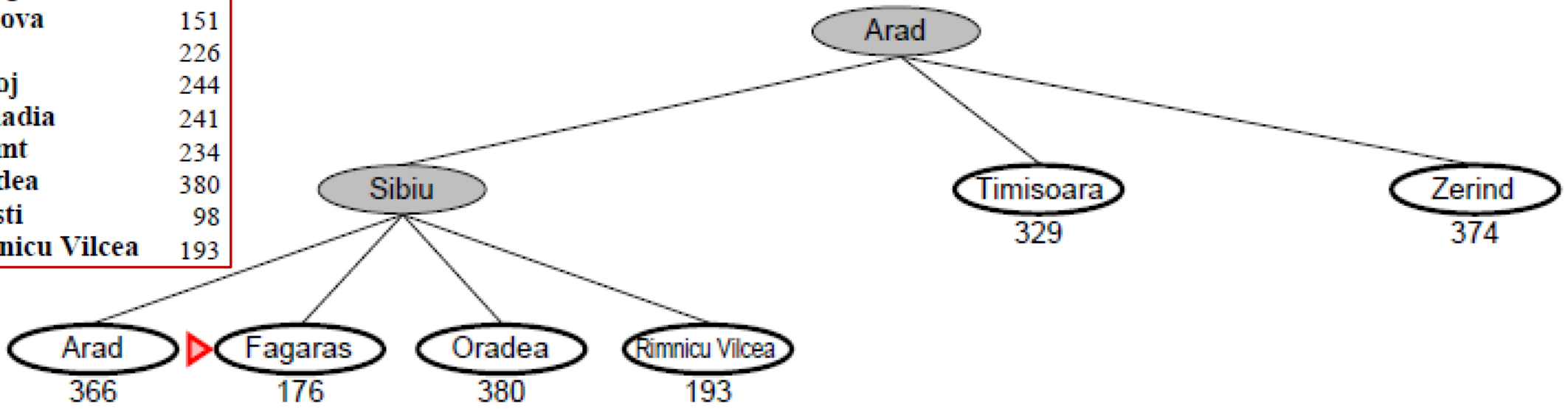
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



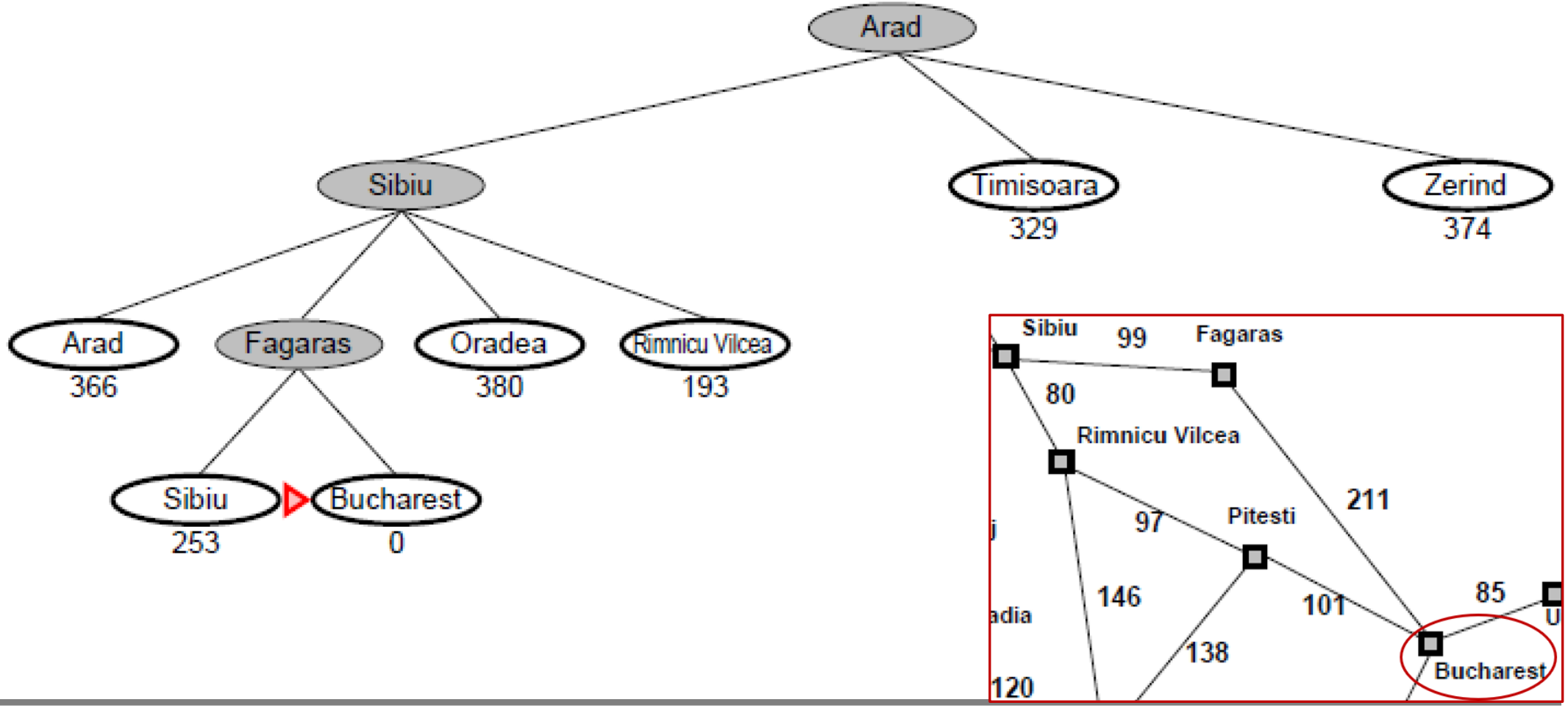


Greedy Search: Searching Illustration

Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193



Greedy Search: Searching Illustration



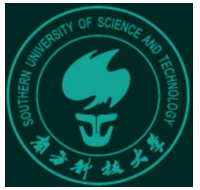
PF Metrics

- b – maximum branching factor of the search tree.
- d – depth of the least-cost solution.
- m – maximum depth of the state space.

- Complete? No, can stuck in loops.
  - E.g. with Oradea as goal, Iasi → Neamt → Iasi → Neamt → ⋯

  Complete in finite space with repeated-state checking.
- Optimal? No.
- Time? $O(b^m)$, but a good heuristic can give drastic improvement.
- Space? $O(b^m)$, keep all nodes in memory.

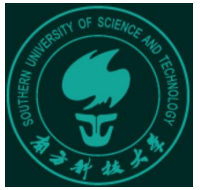
➢ Memory requirement is the biggest handicaps.



Best-first Search Methods:

A* Search

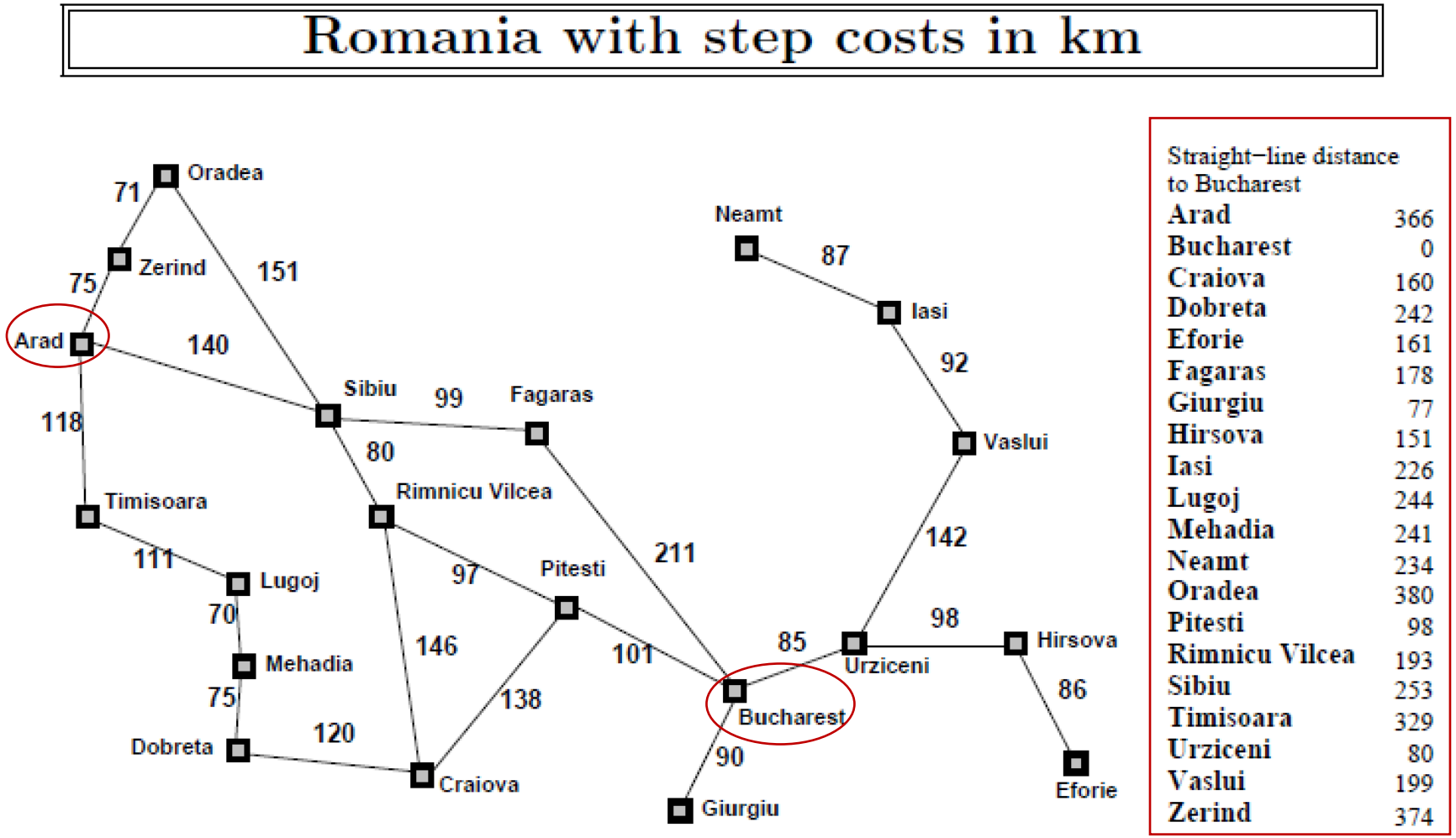
Core Idea



- Idea: avoid expanding paths that are already expensive.
- Expand the node n that has the minimal $f(n) = h(n) + g(n)$
  - $g(n)$: cost so far to reach n.
  - $h(n)$: estimated cost to goal from n.
  - ➢ $f(n)$: estimated total cost of path through n to goal.

➢ A generalization of UCS as well (i.e., use a different $f(n)$).

A*: Searching Illustration

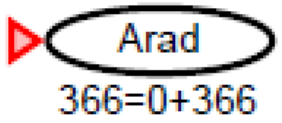


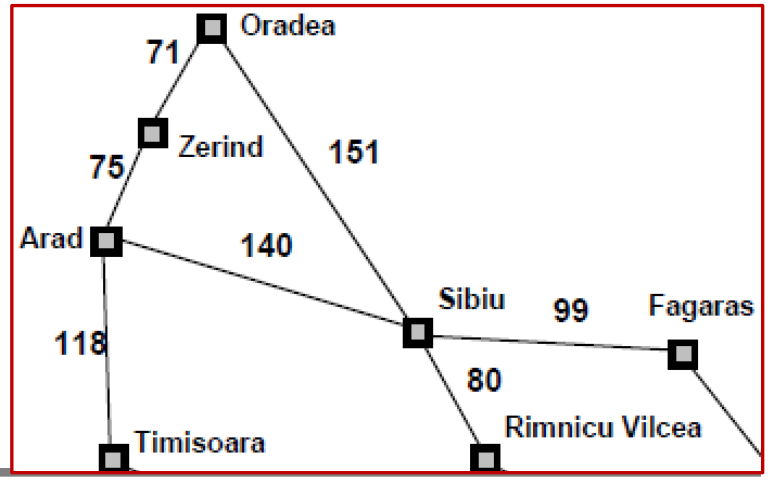
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Heuristics: domain-knowledge

A*: Searching Illustration

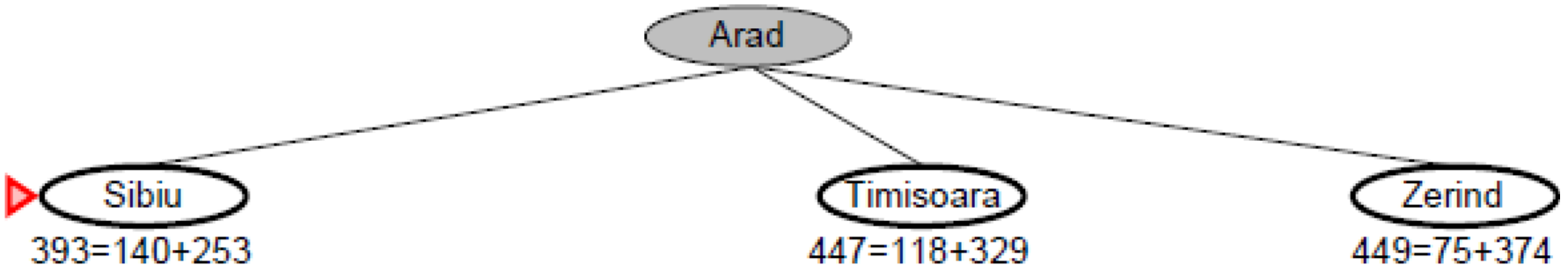
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

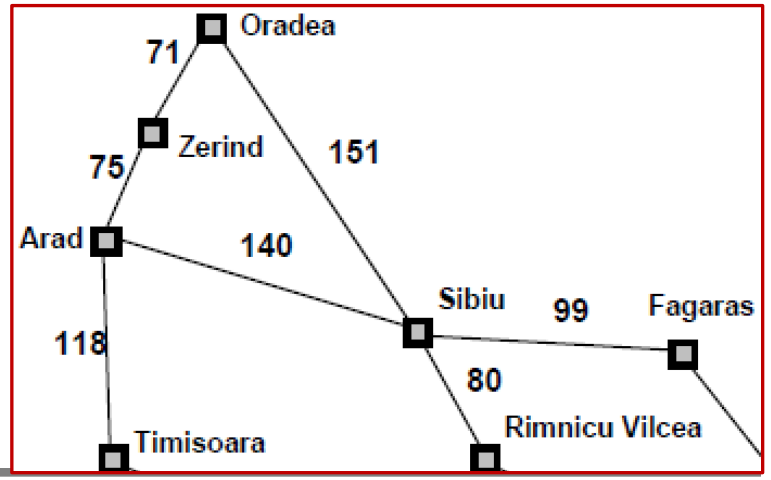




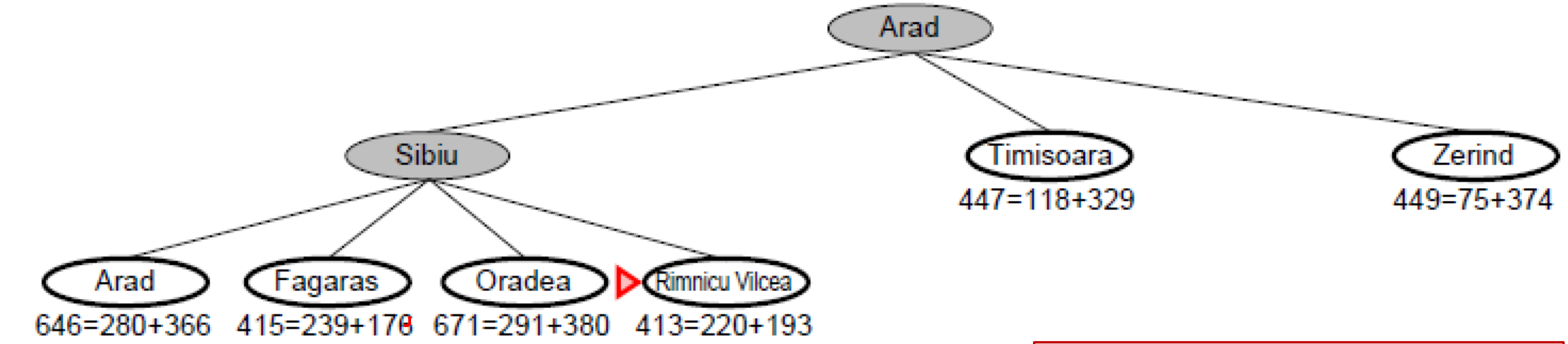
A*: Searching Illustration

Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

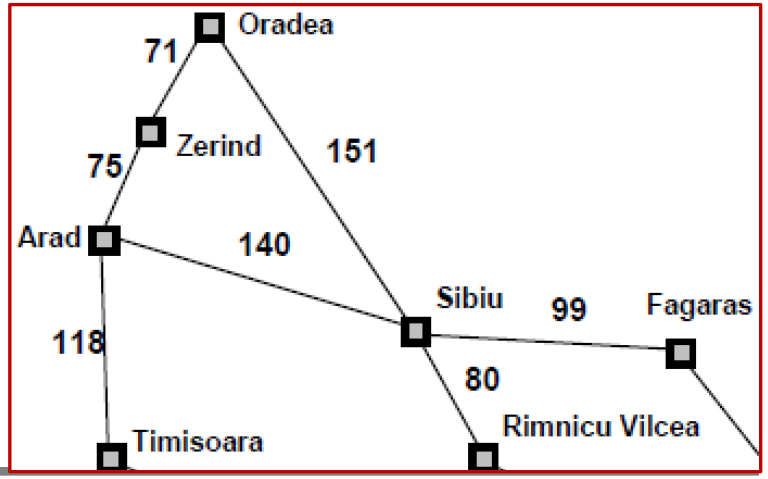




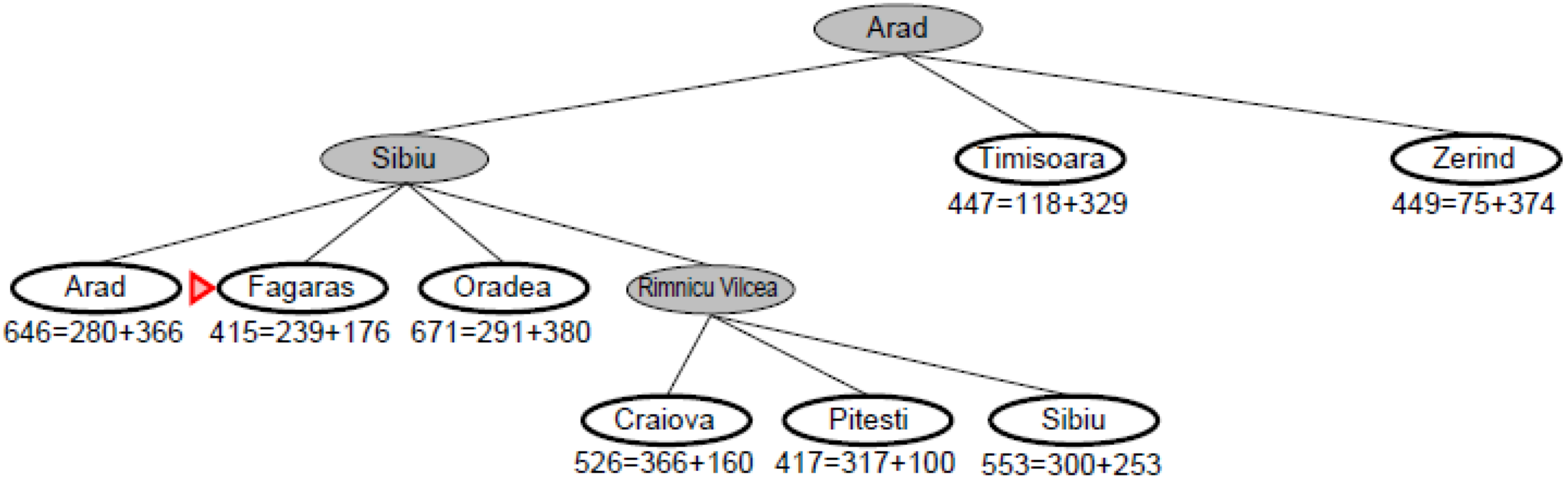
A*: Searching Illustration



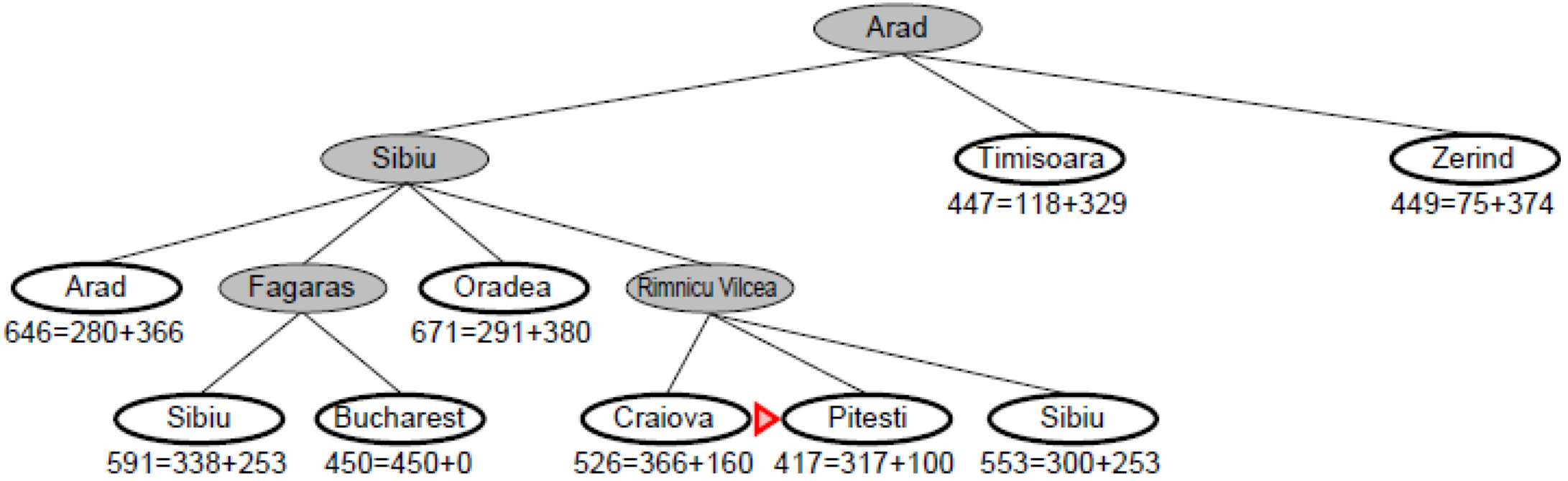
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193



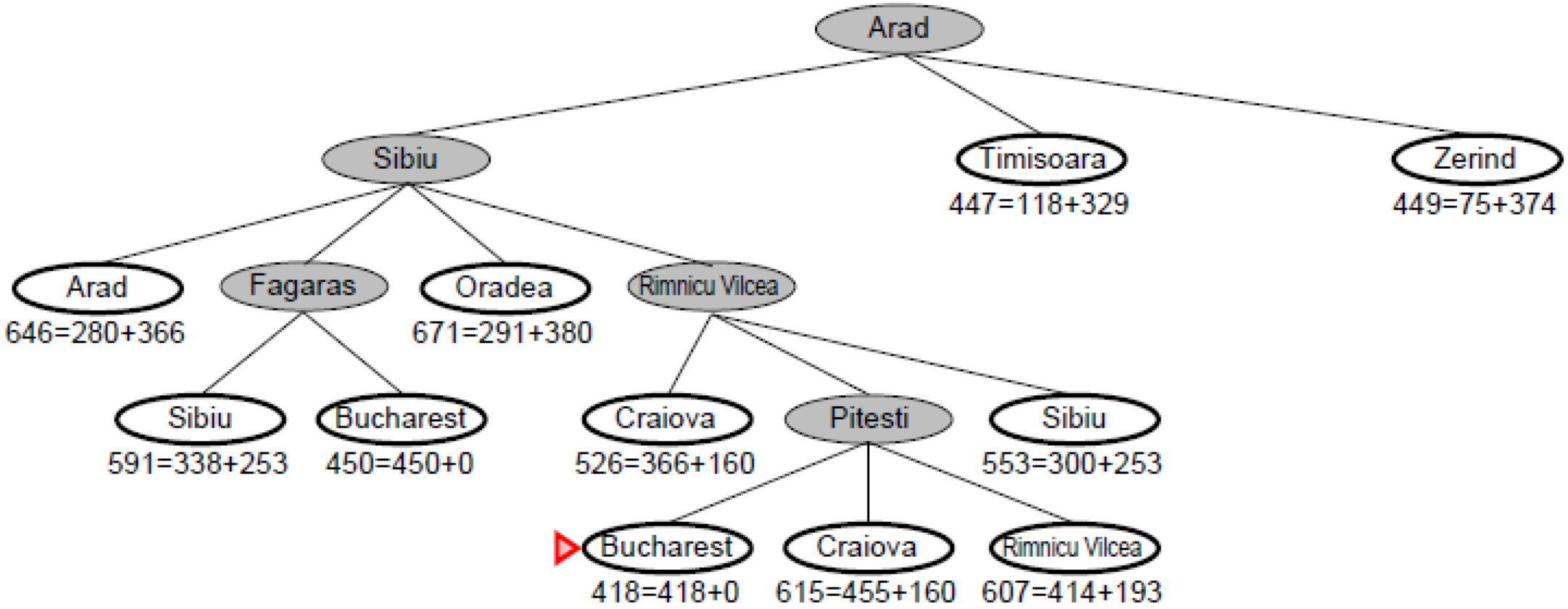
A*: Searching Illustration



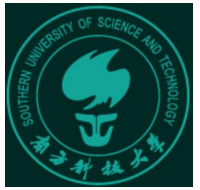
A*: Searching Illustration



A*: Searching Illustration



Pseudo-Code




```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

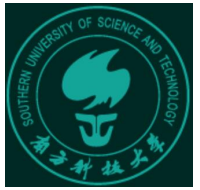
A*: PF Metrics

- b – maximum branching factor of the search tree.
- d – depth of the least-cost solution.
- m – maximum depth of the state space.

- Complete? Yes.
- Optimal? Yes*, if h is admissible.
- Time? $O(b^d)$.
- Space? $O(b^d)$, keep every node in memory.

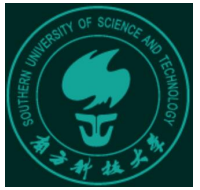
➢ If a solution exists, A* will find the best.

➢ Memory is the major problem for A*.



Recap

Recap: Best-first Search Methods



- Core idea: expand the path that seems most promising.
- Node n with lowest $f(n) \rightarrow$ the most promising.
- Usually h is a component of f.

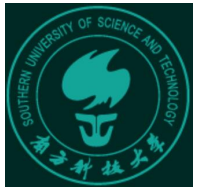
➢ Greedy: Incomplete & Not always optimal

➢ A*: Complete & Optimal

Recap: Difference on Evaluation Function

- The choice of f determines the search methods.
- Uniform-cost search: $f(n) = g(n)$.
- Greedy best-first search: $f(n) = h(n)$.
- A* search: $f(n) = g(n) + h(n)$.

- $g(n)$: the path cost from the start to node n.
- $h(n)$: the estimated cost to the goal.



Admissible Heuristic

Good Heuristic

A good heuristic can be powerful.

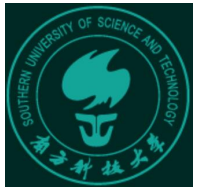
Only if it is of a ‘good quality’.

A good heuristic should be **admissible**!

Admissible Heuristic

- Heuristic function h is admissible: $h(n)$ never overestimates the cheapest cost from n to the goal.
- In formula, a heuristic h is admissible if $\forall n \rightarrow h(n) \leq h^*(n)$.
  - $h^*(n)$: the true cost from n to goal.
- Admissible heuristics are optimistic.

Admissible Heuristic: Examples



- Two extreme cases of admissible heuristics:
- (1) The trivial $h_0(n) = 0$: No help for searching ⇒ Not efficient.
- (2) The perfect $h^*(n) =$ the true cost from n to the goal: lead directly to the best path, but unknow.

Theorem: Optimality of A*

- A* with h is optimal if h is admissible.

  [Proof] **Notation**: S – start, G – goal, n – a node on optimal path, n' – non-optimal goal, and c^* – cost of optimal path.

  To show: A* always pick n over $n' \Leftrightarrow f(n) < f(n')$.

  Known: h is admissible $\Rightarrow h(n) < c^*(n, G)$.

  Deduction: ① $f(n') = g(n') + h(n') = g(n') + 0 > c^*$ (n' is the goal node & c^* the smallest).

  ② $f(n) = g(n) + h(n) < g(n) + c^*(n, G) = c^*(S, n) + c^*(n, G) = c^*$.

  Conclusion: combine ① &② $\Rightarrow f(n) < f(n')$ □

Admissible Heuristic: Examples

For Route Planning:

- $h_{SLD}(n)$: Admissible, since no solution path will ever shorter than straight-line connection.
- $h_{SLD}(n) + 20\%$: Not always admissible, since some may surpass the true distance to goal.

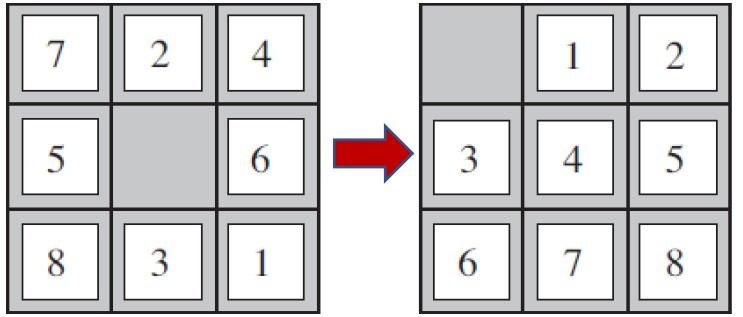
Admissible Heuristic: Examples

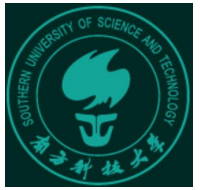
For Eight-puzzle:

- $h_{mis}(n) =$ #misplaced titles $\in [0,8]$: Admissible.
- $h_{1stp}(n) =$ #(1-step move) to reach goal configuration: Admissible.

➢ $h_{1stp}(n) \geq h_{mis}(n) \Rightarrow h_{1stp}(n)$ is 'better' than $h_{mis}(n)$.

$h_{mis}(n) =?$
$h_{1stp}(n) =?$



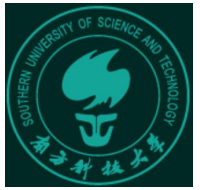


Recap

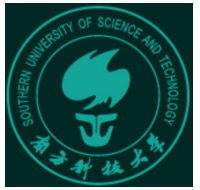
Recap: Heuristic Search

- Basic search uses No domain knowledge.
- Heuristic search uses domain knowledge by heuristic function.

- Good heuristics can drastically reduce search costs.
- Good heuristics should be admissible.

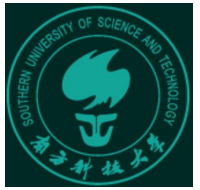


V. Further Studies on Heuristics



Searching Efficiency for Heuristics

Recall



- We focus on A* framework here.

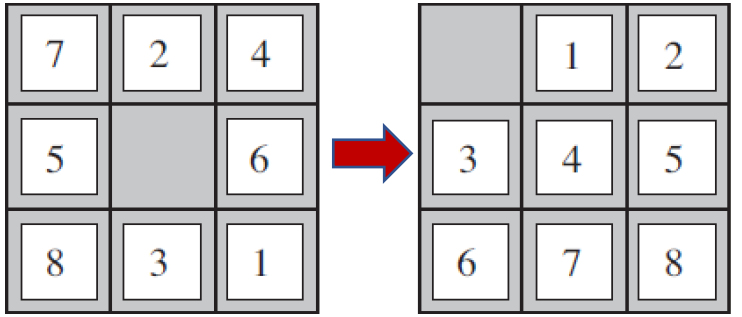
- In general, A* expands all nodes with $f(n) < c^*$

Recall: Heuristics for Eight-puzzle

- $h_{mis}(n) =$ #misplaced titles $\in [0,8]$: Admissible.
- $h_{1stp}(n) =$ #(1-step move) to reach goal configuration: Admissible.

➢ $h_{1stp}(n) \geq h_{mis}(n) \Rightarrow h_{1stp}(n)$ is 'better' than $h_{mis}(n)$.

What does 'better' mean?



Dominance

- For admissible h_1 and h_2, if $h_2(n) \geq h_1(n)$for $\forall n \Rightarrow h_2$ dominates h_1 and is more efficient for search.

- Theorem: For any admissible heuristics h_1 and h_2, $h(n) = \max\{h_1(n), h_2(n)\}$ is admissible and dominates both h_1 and h_2.

➢ 'Better' = dominance.

Effective Branching Factor b^*

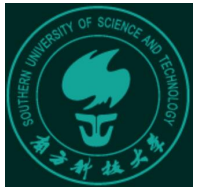
- For a solution from A* algorithm, calculate b^* satisfying:

$$N = b^* + (b^*)^2 + \cdots + (b^*)^d,$$

  - N: #nodes of the solution,
  - d: depth of the solution tree.
  - E.g., A* finds a solution at depth 5 using 52 nodes → $b^* = 1.92$.
- Good heuristics have b^* close to 1 ⇒ large problems solved at reasonable computational cost.

➢ b^* quantifies search efficiency of heuristics.

Verify: Search Cost & Factor b^*



- Aim: Compare h_1 and h_2 on searching efficiency.
- Setting: Generate 1200 random problems with $d = \{2, \cdots, 24\}$ and solve them with IDS and A* with h_1 & h_2.
- Note: IDA – a baseline.

Verify: Search Cost & Factor b^*

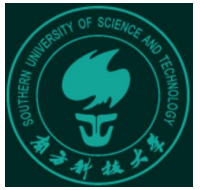
	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Verify: Search Cost & Factor b^*

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4						
6						
8						
10						
12						
14						
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

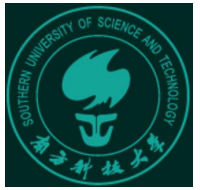
- h_2 is 'better' than h_1 regarding search efficiency.
- This goodness is reflected on smaller b^*.

- A* with h_2 performs much better than IDA.



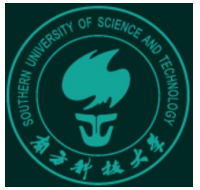
Generating Admissible Heuristics

Review



- We know for heuristic functions:
  - How to judge their admissibility, and
  - How to compare their goodness regarding searching efficiency.

- Question: How to invent such ‘good’ heuristics?



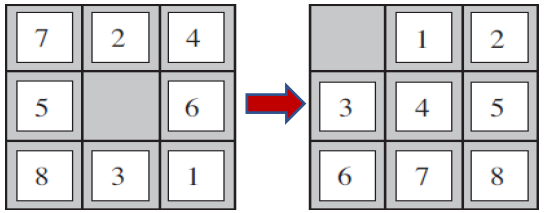
(1) Generating from Relaxed Problems

How Heuristics Invented?

For eight puzzle:

- Rule: A tile can only move to the adjacent empty square.
- Relaxed rules:
  - R1: A tile can move anywhere ⇒ $h_{mis}(n)$ = #(misplaced titles).
  - R2: A tile can move one step in any direction regardless of an occupied neighbour ⇒ $h_{1stp}(n)$ = #(1-step move) to reach goal.

  ➢ h_{mis} and h_{1stp} are admissible.
- Note: optimal solutions with R1, R2 are easier to find.



Relaxed Problem

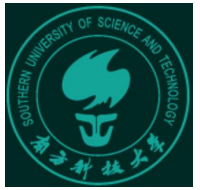
- Relaxed problem: a problem with fewer restrictions on the action.

➢ Eight-puzzles with C1 and C2 are relaxed problems.

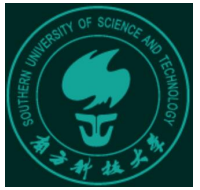
- Theorem: The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.

➢ No wonder h_{mis} and h_{1stp} are admissible.

Suggestion for Better Admissible h



- Question: Which one to choose from a collection of admissible heuristics $h_1, \cdots, h_m$ & none dominates any of the other?
- Answer: $h(n) = \max\{h_1(n), \cdots, h_m(n)\}$ dominates all the others.



(2) Generating from Sub-problems

Subproblem

- Subproblem: See Fig.1.
  - Task: get tiles 1, 2, 3 and 4 into their correct positions.
  - Relaxation: move them disregard the other tiles.
- Theory: cost*(subproblem)<cost*(8-puzzle).

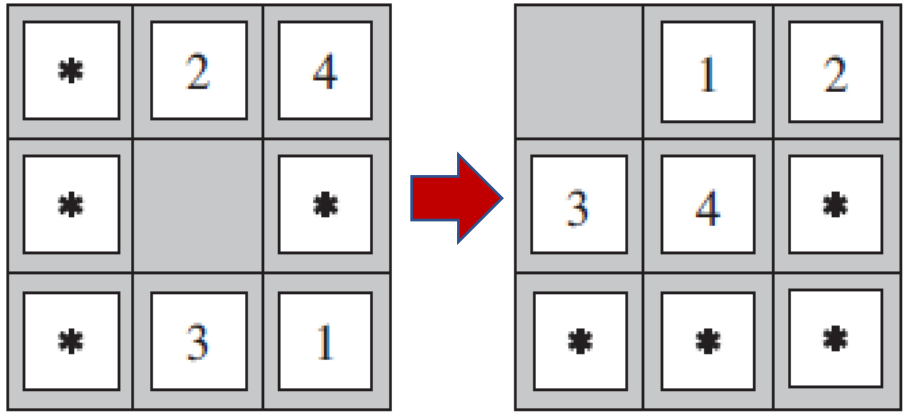
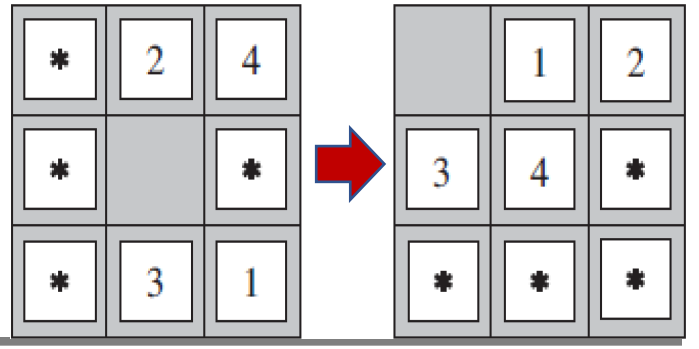


Fig.1. A subproblem of 8-puzzle.

Subproblem and Admissible Heuristic

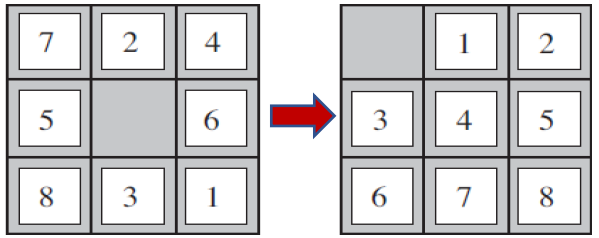
- Admissible heuristic $h_{DB}(n)$ estimates the cost from n to the subproblem goal.
  - E.g. heuristic $h_{DB}^{(1,2,3,4)}$ is the cost to solve the 1-2-3-4 subproblem.
- Combination: $h_{DB}(n) = max\{h_{DB}^{(1,2,3,4)}(n), h_{DB}^{(2,3,4,5)}(n), \cdots\}$.
- Theorem: $h_{DB}(n)$ dominates $h_{1stp}(n)$.

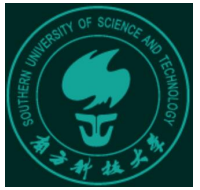


Disjoint Pattern Databases

- Question: Can heuristics from the 1-2-3-4 subproblem and the 5-6-7-8 added, since the two subproblems are not overlapped?
- Answer: No, since they always share some moves.

- Question: What if disregard these shared moves?
- Answer: $h_{DB}^{(1,2,3,4)}(n) + h_{DB}^{(5,6,7,8)}(n) \leq h_{DB}(n) \Rightarrow$ admissible heuristic.
  - Disjoint pattern database: see [6].





(3) Generating from Experiences

‘Experience’

For 8-puzzle problem:

- Solve many 8-puzzles to obtain many examples.
- Each example consists of a state from the solution path and the actual cost of the solution from that point.

➢ These examples are our ‘experience’ for this problem.

- Question: How to learn $h(n)$ from these experience?

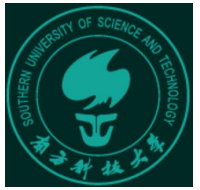
Learn Heuristics from Experience

- Question: What are good experience features?
- Answer: Relevant to predicting the state's cost to the goal, e.g.
  - $x_1(n)$: #(displaced tiles).
  - $x_2(n)$: #(pairs of adjacent tiles) that are not adjacent in the goal state.

- Question: How to learn h from the relevant experience features?
- Answer: (e.g.) Construct model as

$$h(n) = w_1 x_1(n) + w_2 x_2(n), \text{ where}$$

w_1, w_2 are model parameters to learn from training data by a learning method such as neural networks and decision trees.



Recap

Recap: Heuristic Function $h(n)$

- $h(n)$ estimates the cheapest cost from n to the goal.
  - (1) $h(n) = 0$ if n the goal node. (2) nonnegative. (3) problem-specific.
- Admissibility: $h(n)$ never overestimates cheapest cost from n to Goal.
- Effective Branching Factor b^* quantifies the search efficiency of $h(\cdot)$.
- Generate admissible heuristics:
  - (1) from relaxed problems.
  - (2) from sub-problem.
  - (3) from experience.

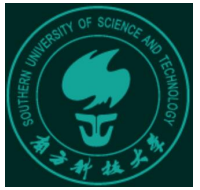
Summary

Basic Search Methods

Uniformed Search: Use no domain knowledge.

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

Recap: Search Methods



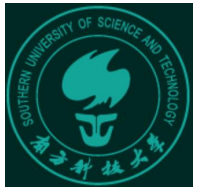
- **Uniformed Search**: Use no domain knowledge.
  - BFS, UCS, DFS, DLS,IDS
- **Informed Search**: Use a heuristic function that estimates how close a state is to the goal.
  - Greedy first-best search, A*

Reading Materials for Uninformed search

- [1] AI book Sec. 3.1 -- 3.4: http://pan.baidu.com/s/1gftL8Pl
- [2] Online course: https://courses.edx.org/courses/course-v1:ColumbiaX+CSMM.101x+2T2017/course/
- [3] Basic search algorithm: http://artint.info/html/ArtInt_52.html
- [4] AI-book codes: https://github.com/aimacode
- [5] Robert M. Lewis, Virginia Torczon and Michael W. Trosset, Direct Search Methods: Then and Now. Journal of Computational and Applied Mathematics 124(2000) 191-207

Reading Materials for Informed Search

- [1] AI book Sec.3.5 & 3.6: http://pan.baidu.com/s/1gftL8Pl
- [2] Online course: https://courses.edx.org/courses/course-v1:ColumbiaX+CSMM.101x+2T2017/course/
- [3] A*: http://www.cnblogs.com/0zcl/p/6242790.html
- [4] A*: http://www.360doc.com/content/16/1201/12/99071_610999046.shtml
- [5] A*: https://en.wikipedia.org/wiki/A*_search_algorithm
- [6] Richard E. Korf and Ariel Felner. Disjoint Pattern Database Heuristics. Artificial Intelligence 134(2002) 9-22.
- [7] Richard E Korf and Michael Reid and Stefan Edelkamp. Time Complexity of Iterative-deepening A*. Artificial Intelligence 129:5(2001).



To be continued