

I Artificial Intelligence

II Problem Solving

3. Solving Problems by Searching

4. Search in Complex Environments

5. Adversarial Search and Games

6. Constraint Satisfaction Problems

III Knowledge, Reasoning, Planning

IV Uncertain Knowledge and Reasoning

V Machine Learning

VI Communicating, Perceiving, and Acting

VII Conclusions

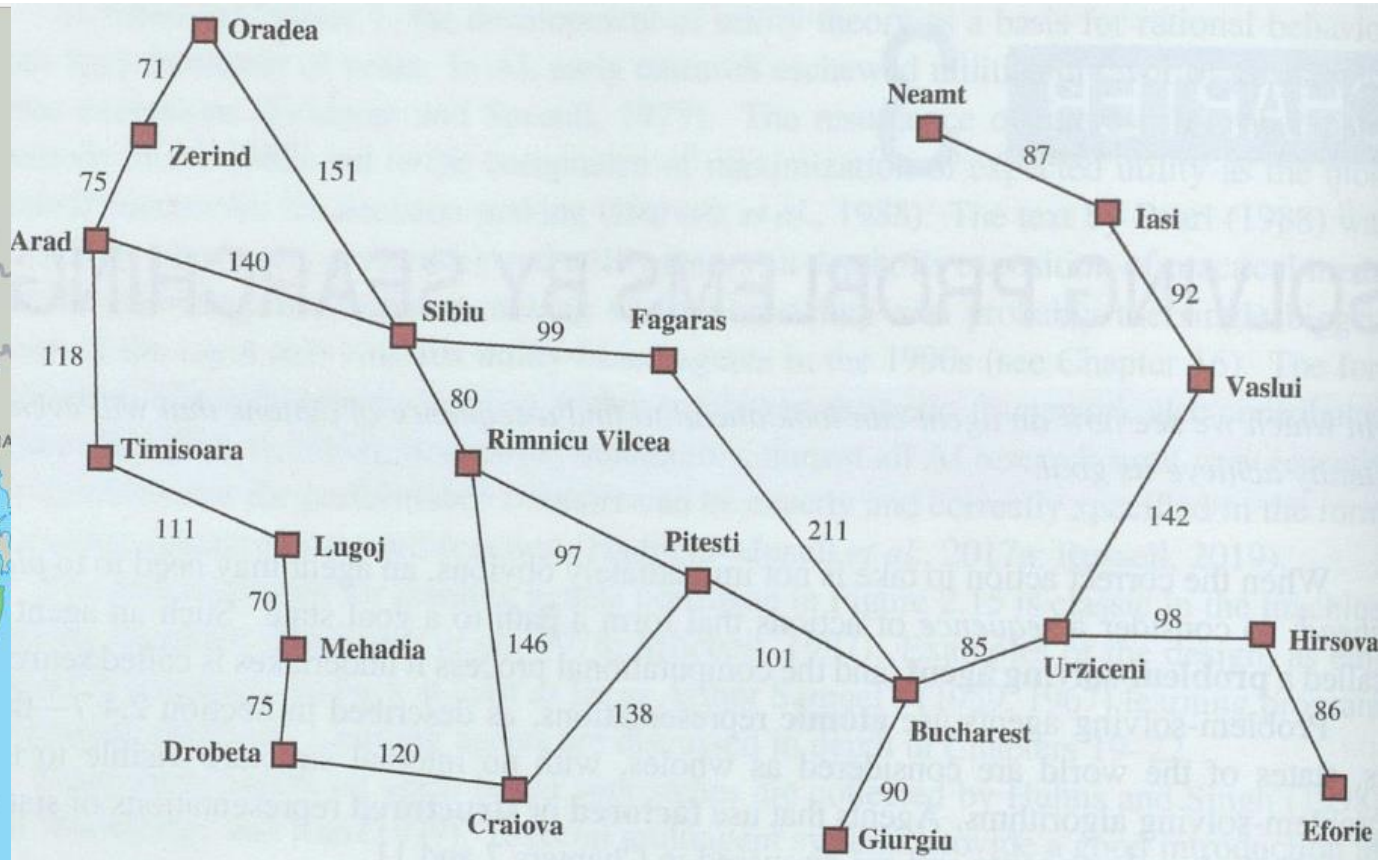


- Problem descriptions
- Agent context
- Definition
- Examples
- Tree- vs. graph-search
- Uninformed search strategies
- Breadth-first and depth-first search (normal and iterative)
- Bidirectional Search
- Informed (Heuristic) search (greedy, A^*)
- Heuristic functions



Frank Puppe

- Imagine, you are in an unfamiliar country in some city and want to travel to another city.
- But you have background knowledge (e.g. a map of Romania; to go from Arad to Bucharest)



- **Goal formulation:** Reaching Bucharest
- **Problem formulation:** Description of states (current city) and actions (travel to an adjacent city)
- **Search:** Simulate sequences of actions to find the best one
- **Execution:** Execute the actions

Reflection:

- Atomic state representation
- Fully observable, deterministic, known environment
- Hierarchical approach: Executing an action is another problem solving process (e.g. by driving a car from one city to the next city)
 - Many abstraction layers: Finding the right level of abstraction might be difficult in itself



- **State space:** Set of possible states in the environment
- **Initial state** [e.g. Arad]
- One or more **goal states** [e.g. Bucharest]
- **Actions** [e.g. $\text{ACTIONS}(\text{Arad}) = \{\text{ToSibiu}, \text{ToTimisoara}, \text{ToZerind}\}$]
- **Transition model** [e.g. $\text{RESULT}(\text{Arad}, \text{ToSibiu}) = \text{Sibiu}$]
- **Action cost function:** $c(s, a, s') = x$ [e.g. $c(\text{Arad}, \text{ToSibiu}, \text{Sibiu}) = 140$]
- **Path** = sequence of actions
- **Solution** = path from initial state to goal state
- **Optimal solution** = has lowest path costs among all solutions
- State space can be represented as graph

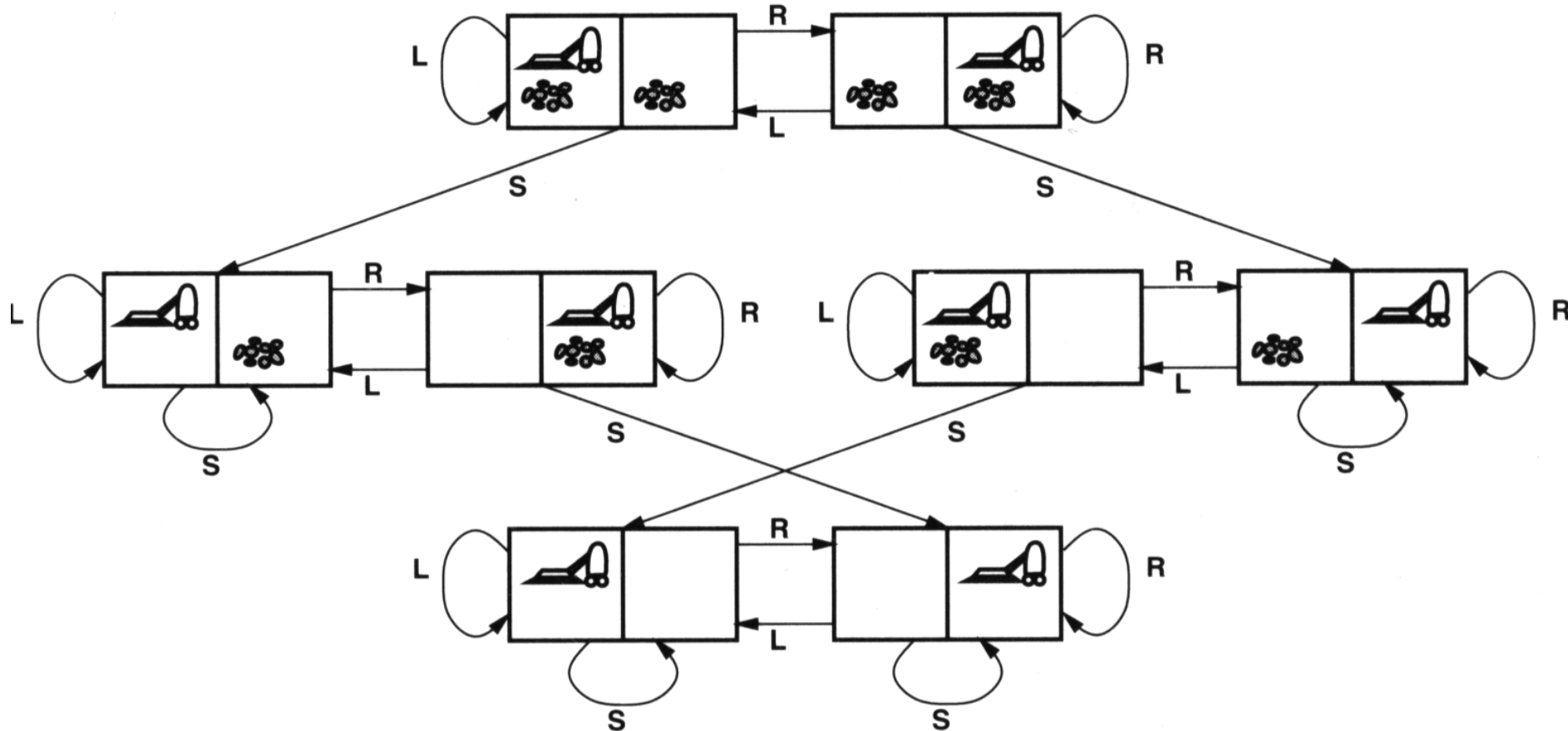


- Vacuum cleaner
- Sliding tile puzzle (eg. 8-puzzle)
- Mathematical problem
- 8-queen-problem



Frank Puppe

- **States:** Two cells with or without dirt and agent in it
- **Initial state:** Any state
- **Actions:** (1) Suck (S), (2) MoveLeft (L), (3) MoveRight (R)
- **Transition model:** (see right)
- **Goal states:** Every cell is clean
- **Action cost:** Each action costs 1



- **States:** Location of all tiles
- **Initial state:** Any state
- **Actions:** Move blank tile (1) Up, (2) Down, (3) Left (L), (4) Right (R)
- **Transition model:** Switch blank and adjacent tile in state, if possible.
- **Goal state:** s. right (tiles in a certain order)
- **Action cost:** Each action costs 1

5	4	
6	1	8
7	3	2

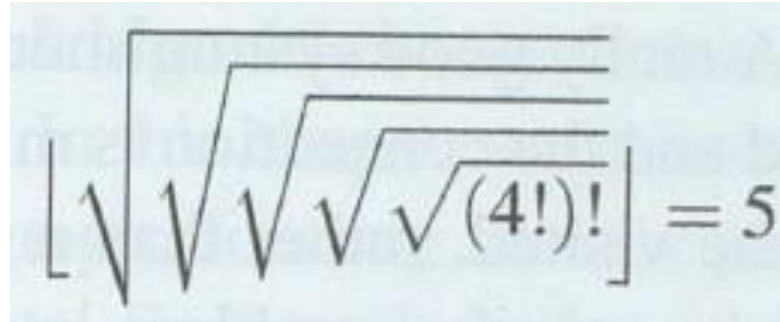
Start State

1	2	3
8		4
7	6	5

Goal State



Don Knuth conjectured in 1964, that starting with the number 4, a sequence of square root, floor and factorial operations can reach any desired positive integer, e.g.


$$\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \rfloor = 5$$

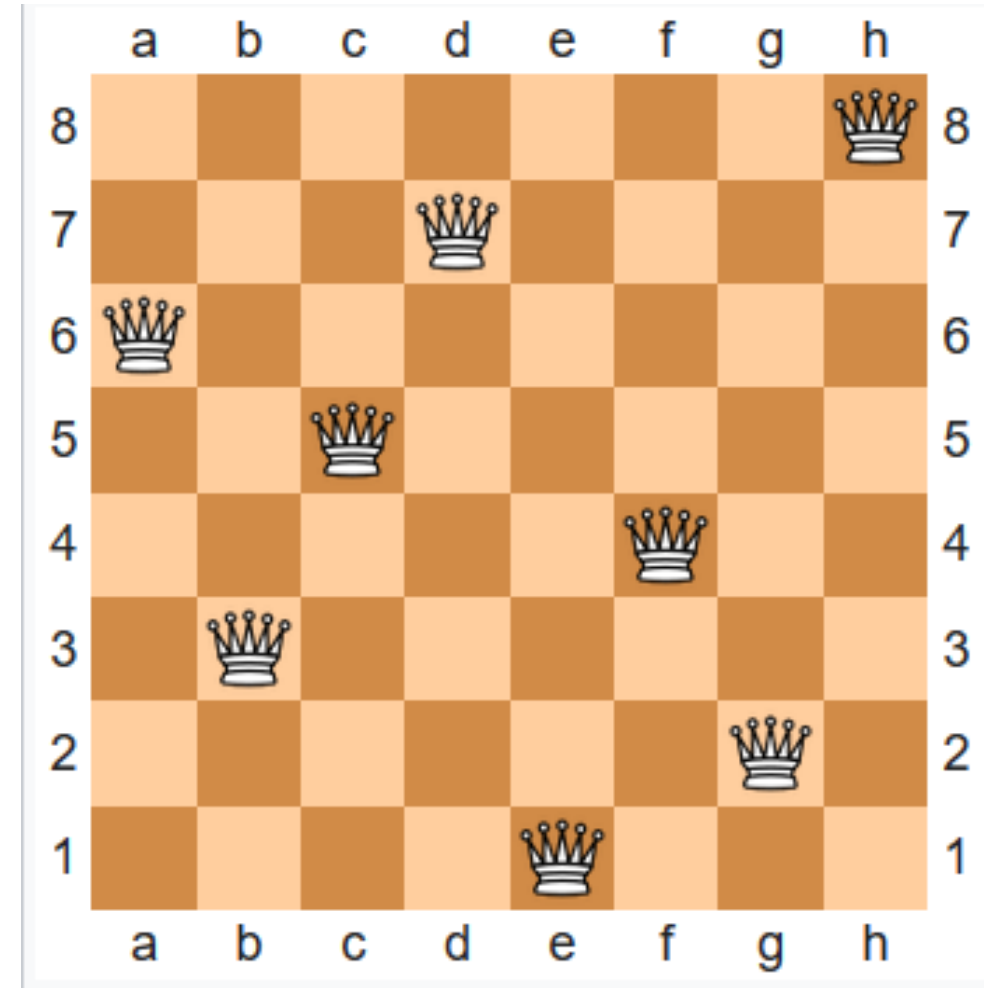
- **States:** Positive real numbers
- **Initial state:** 4
- **Actions:** Apply (1) square root, (2) floor, (3) factorial operation (for integers only)
- **Transition model:** As given by the mathematical definitions of the operations.
- **Goal state:** The desired positive number
- **Action cost:** Each action costs 1



- **States:** Any arrangement of 0 to 8 queens on board
- **Initial state:** No queen on board
- **Actions:** Add a queen to an empty square
- **Transition model:** Return the board with a queen added to the specified square
- **Goal states:** 8 queens are on board, none attacked
- **Action cost:** -

Better formulation:

- **States:** All possible arrangements of 0 to 8 queens one per column in the leftmost columns, with no queen attacking another
- **Actions:** Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.



picture from wikipedia



- **Route-finding problems:** Travel with traffic dependant delays, routing video streams in computer networks, military operations planning, train or airline travel-planning systems
- Touring problem (set of locations must be visited, like travelling salesman problem)
- Robot navigation
- VLSI layout problem
- Automatic assembly sequencing
- ...

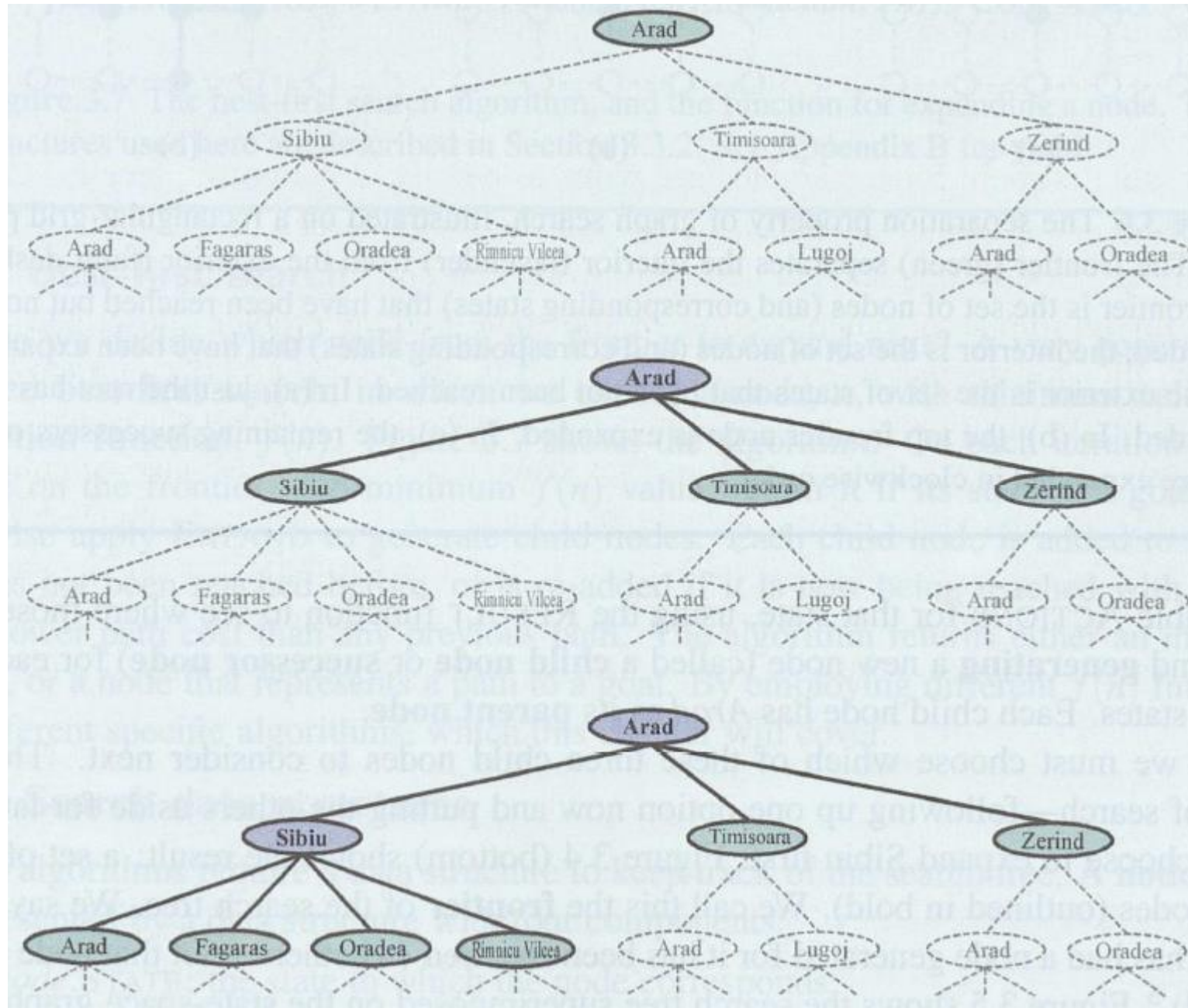


Frank Puppe

- Start with initial state (start node)
- Expand a node according its possible actions, adding a child node (successor node)
- Look for paths leading to the goal
- **Search frontier:** Active nodes to be investigated
 - Nodes above are explored
 - Nodes below are unexplored
- Three kinds of queues for frontier:
 - **Priority queue**
 - **FIFO** queue (First in, First out)
 - **LIFO** queue (Last in, First out)



Frank Puppe



- **Tree Search** may explore redundant paths, but needs only little memory.
- **Graph search** avoids redundant paths (see next slide) by memorizing typically all explored nodes
- Trade-off between time and space complexity.

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set
```



Goal: If there are multiple paths to a node, avoid checking the node multiple times

➤ In the worst case, this might lead to infinite loops.

Three methods with increasing amount of memory:

- Do not return to state from which you came
- Avoid cycles in a path: Check, whether new node is contained in path to that node
- Avoid generating a node more than once: Check, whether a new node was already explored (needs storage of all explored nodes)



- Expand always the node n with minimal costs on its path so far computed by the function $f(n)$.
- Reached: a table containing every explored node with the minimal costs so far.
- Data structure of node:
 - node.State
 - node.Parent
 - node.Action (action applied to parent's state to generate the node)
 - node.Path-Cost (total cost from initial state to node)

```

function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
    node  $\leftarrow$  NODE(STATE=problem.INITIAL)
    frontier  $\leftarrow$  a priority queue ordered by f, with node as an element
    reached  $\leftarrow$  a lookup table, with one entry with key problem.INITIAL and value node
    while not IS-EMPTY(frontier) do
        node  $\leftarrow$  POP(frontier)
        if problem.IS-GOAL(node.STATE) then return node
        for each child in EXPAND(problem, node) do
            s  $\leftarrow$  child.STATE
            if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
                reached[s]  $\leftarrow$  child
                add child to frontier
    return failure

function EXPAND(problem, node) yields nodes
    s  $\leftarrow$  node.STATE
    for each action in problem.ACTIONS(s) do
        s'  $\leftarrow$  problem.RESULT(s, action)
        cost  $\leftarrow$  node.PATH-COST + problem.ACTION-COST(s, action, s')
        yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
    
```



Frank Puppe

- **Completeness:** If a solution exist, it is found, otherwise failure is reported.
- **Cost optimality:** Is the path to the solution the best one, i.e. with the lowest costs?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

Time and space complexity are usually reported in O-notation.



- Uninformed = not using any knowledge about the problem
- Systematic, exhaustive search strategies

Graph-Search | Tree-Search

- | | | |
|--|---|---|
| • Breadth-first search | x | |
| • Uniform-cost search (Dijkstra's algorithm) | x | |
| • Depth-first search | | x |
| • Depth-limited search | | x |
| • Iterative deepening search | | x |
| • Bidirectional search | x | |



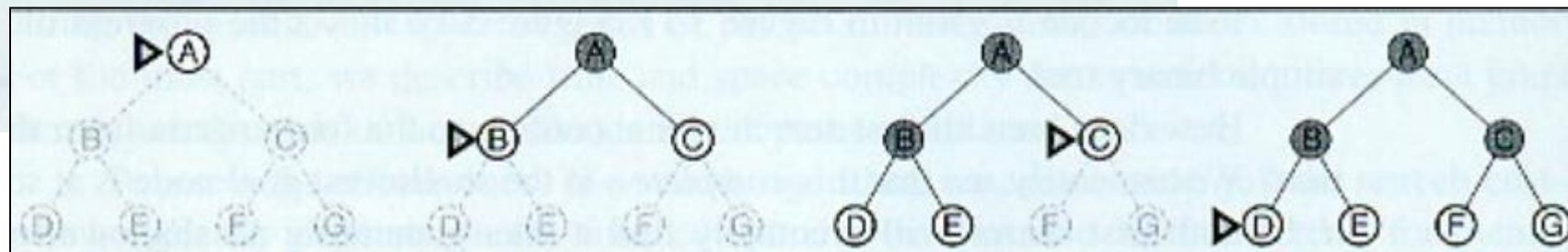
```

function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node  $\leftarrow$  NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier  $\leftarrow$  a FIFO queue, with node as an element
  reached  $\leftarrow$  {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node  $\leftarrow$  POP(frontier)
    for each child in EXPAND(problem, node) do
      s  $\leftarrow$  child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
  
```

Each level is explored completely before the next level (FIFO queue)

- **Completeness:** yes
- **Cost optimality:** yes
- **Time complexity:** $O(b^d)$
- **Space complexity:** $O(b^d)$

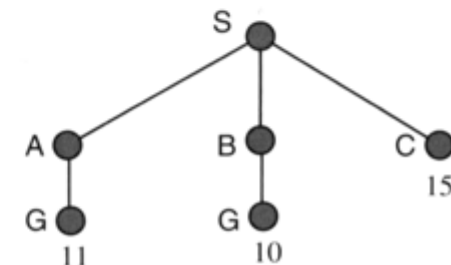
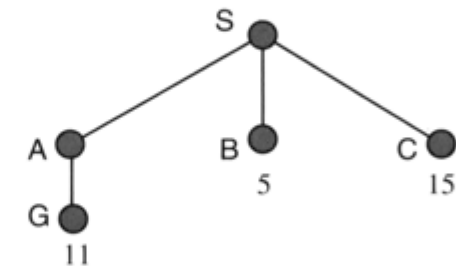
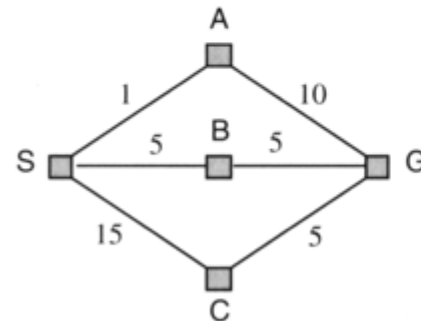
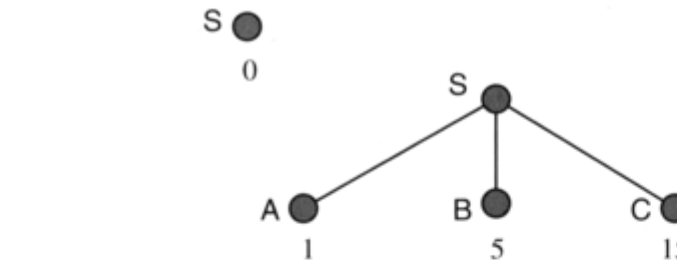
b = branching factor, d = depth



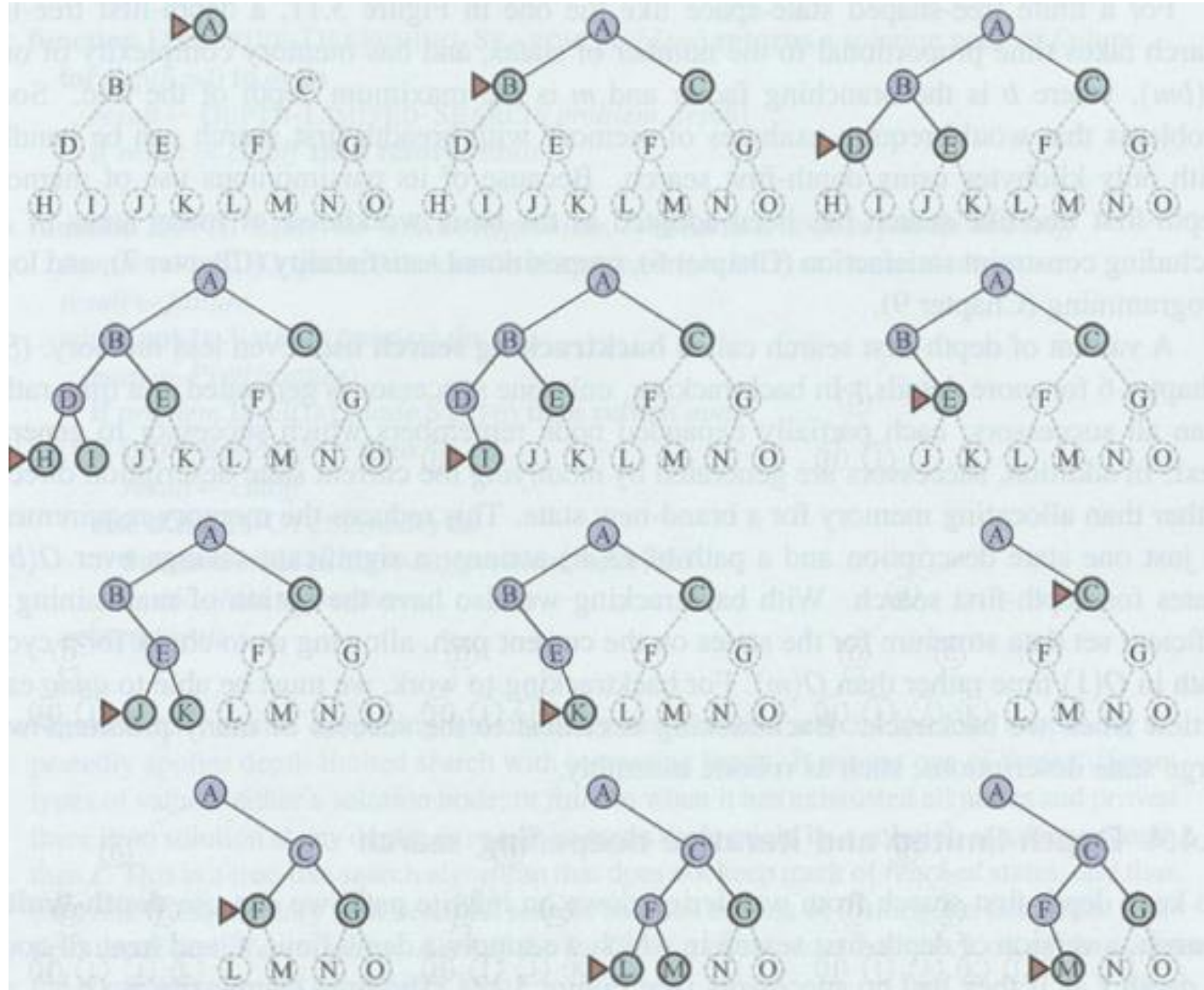
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
return BEST-FIRST-SEARCH(*problem*, PATH-COST)

- If path-Cost is depth in search-tree, uniform-Cost-Search is equivalent to Breadth-first search.
- For finding the shortest path, uniform-cost can use the distance between two cities as path-cost, thus always exploring the path with the least total costs from the starting point.
- Small example for searching the shortest path from S to G via A, B or C (s. right)
- Less efficient, if they are many and very small distances.

- **Completeness:** yes
- **Cost optimality:** yes
- **Time complexity:** (more than) $O(b^d)$
- **Space complexity:** $O(b^d)$



- Like breadth-first search, but with LIFO queue instead of FIFO queue:
- The successors of the last inserted node are explored first
- **Completeness:** no
- **Cost optimality:** no
- **Time complexity:** $O(b^d)$
- **Space complexity:** $O(b \cdot d)$



Frank Puppe

Standard Search Algorithm!

- Combines advantages of breath-first and depth-first search
- repeats search of upper levels to avoid storing all states
- Depth-limited-Search with increasing depth
- Cycle-control

```

function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result

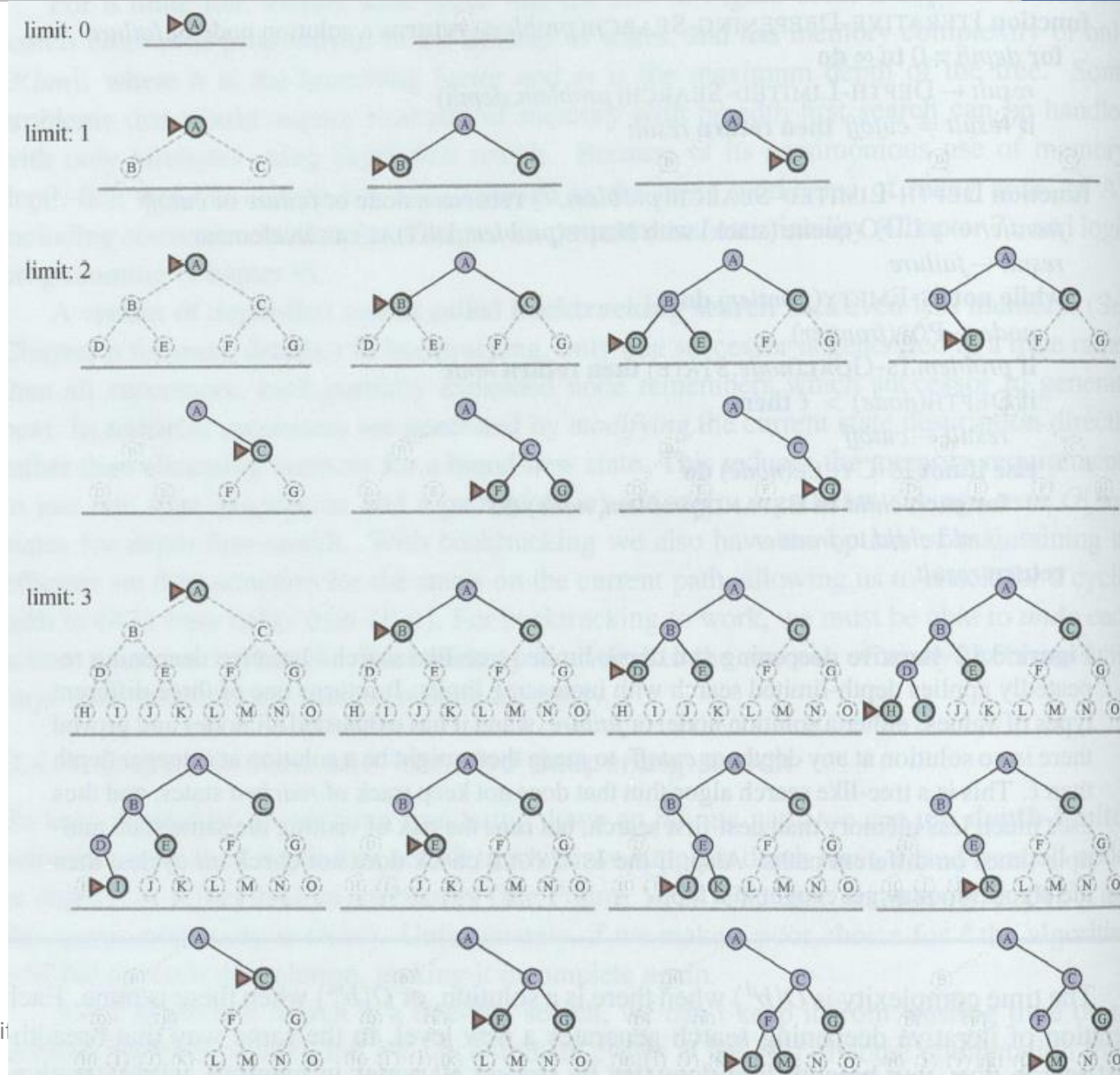
function DEPTH-LIMITED-SEARCH(problem,  $\ell$ ) returns a node or failure or cutoff
  frontier  $\leftarrow$  a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result  $\leftarrow$  failure
  while not IS-EMPTY(frontier) do
    node  $\leftarrow$  POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) >  $\ell$  then
      result  $\leftarrow$  cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
  
```

- **Completeness:** yes
- **Cost optimality:** yes
- **Time complexity:** $O(b^d)$
- **Space complexity:** $O(b \cdot d)$

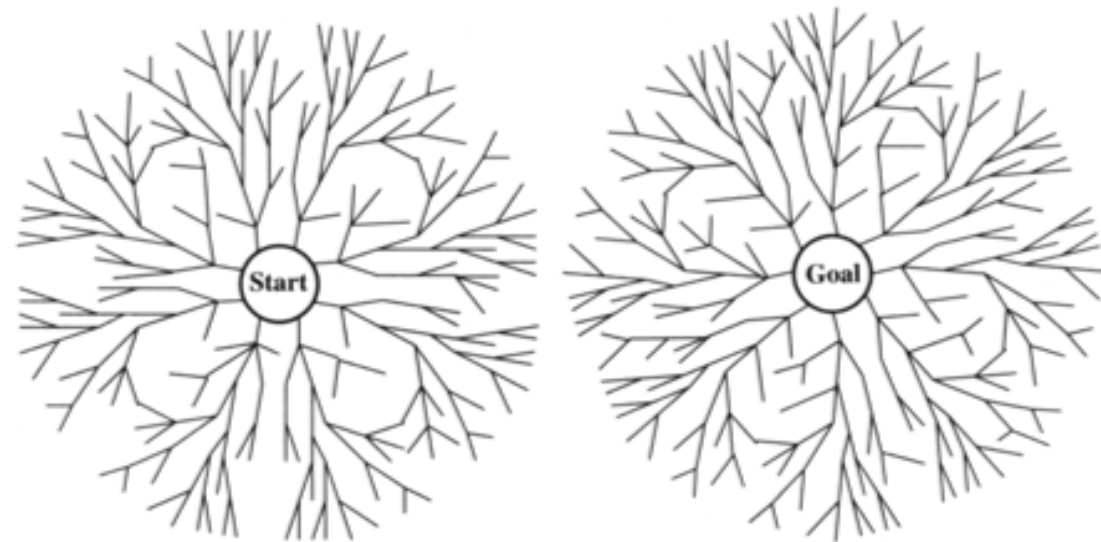


Standard Search Algorithm!

- Each level is fully explored, before the next level is investigated
- With a branching factor of $b \geq 2$, there are more nodes on the last level than on all levels beneath
- Thus, the time complexity increases only by a factor smaller than 2, thus still being $O(b^d)$



- Idea: Searching simultaneously forward from the start node and backward from the goal node.
- Requires graph-based search procedure (e.g. uniform-cost search oder breadth-first search)
- Time and space complexity: $O(b^{d/2})$



Frank Puppe

```

function BIBF-SEARCH( $problem_F, f_F, problem_B, f_B$ ) returns a solution node, or failure
     $node_F \leftarrow \text{NODE}(problem_F.INITIAL)$  // Node for a start state
     $node_B \leftarrow \text{NODE}(problem_B.INITIAL)$  // Node for a goal state
     $frontier_F \leftarrow$  a priority queue ordered by  $f_F$ , with  $node_F$  as an element
     $frontier_B \leftarrow$  a priority queue ordered by  $f_B$ , with  $node_B$  as an element
     $reached_F \leftarrow$  a lookup table, with one key  $node_F.STATE$  and value  $node_F$ 
     $reached_B \leftarrow$  a lookup table, with one key  $node_B.STATE$  and value  $node_B$ 
     $solution \leftarrow failure$ 
    while not TERMINATED( $solution, frontier_F, frontier_B$ ) do
        if  $f_F(\text{TOP}(frontier_F)) < f_B(\text{TOP}(frontier_B))$  then
             $solution \leftarrow \text{PROCEED}(F, problem_F, frontier_F, reached_F, reached_B, solution)$ 
        else  $solution \leftarrow \text{PROCEED}(B, problem_B, frontier_B, reached_B, reached_F, solution)$ 
    return solution
    
```

```

function PROCEED( $dir, problem, frontier, reached, reached_2, solution$ ) returns a solution
    // Expand node on frontier; check against the other frontier in  $reached_2$ .
    // The variable "dir" is the direction: either F for forward or B for backward.
     $node \leftarrow \text{POP}(frontier)$ 
    for each child in EXPAND( $problem, node$ ) do
         $s \leftarrow child.STATE$ 
        if  $s$  not in  $reached$  or  $\text{PATH-COST}(child) < \text{PATH-COST}(reached[s])$  then
             $reached[s] \leftarrow child$ 
            add child to frontier
            if  $s$  is in  $reached_2$  then
                 $solution_2 \leftarrow \text{JOIN-NODES}(dir, child, reached_2[s])$ 
                if  $\text{PATH-COST}(solution_2) < \text{PATH-COST}(solution)$  then
                     $solution \leftarrow solution_2$ 
    return solution
    
```

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes ¹	Yes ^{1,2}	No	No	Yes ¹	Yes ^{1,4}
Optimal cost?	Yes ³	Yes	No	No	Yes ³	Yes ^{3,4}
Time	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: ¹ complete if b is finite, and the state space either has a solution or is finite. ² complete if all action costs are $\geq \epsilon > 0$; ³ cost-optimal if action costs are all identical; ⁴ if both directions are breadth-first or uniform-cost.

- $O(b^{1+\lceil C^*/\epsilon \rceil})$ is the precise cost estimation depending on ϵ , the smallest distance (cost). We have approximated it with $O(b^d)$, but it can be even higher.



- **Idea: Always expand the „best“ node (see „best-first search“ above)!**
- Needs heuristic knowledge about a node n : „ $h(n)$ “
- Two variants:
 - Expand the node, who is next to the goal: $f(n) = h(n)$
 - Expand the node, whose path to the goal is supposed to be the shortest: $f(n) = g(n) + h(n)$

with

$f(n)$ = value function of node n

$h(n)$ = heuristic cost estimate from node n to goal node

$g(n)$ = real path cost from initial node to node n

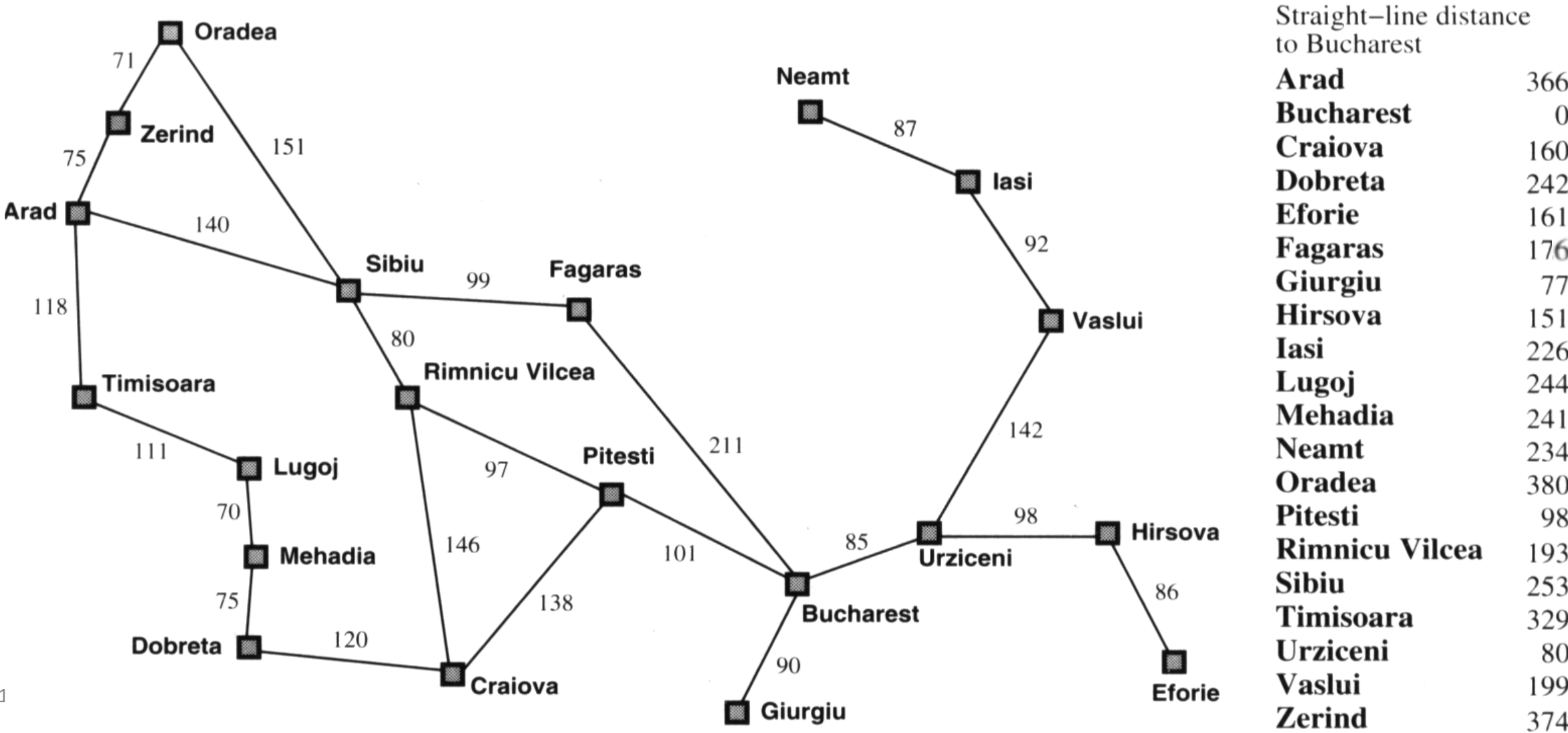


- The key heuristic is an estimate for each node of the costs to reach a goal state on the cheapest path
- The smaller the estimate, the better the node
- It is denoted as heuristic function or $h(n)$
- Example: In route finding, such an estimate is the straight-line distance between two nodes (current node and goal node) on a map.



Greedy Best-first Search (1)

- $f(n) = h(n)$
- Example: Arad (366) → Sibiu (253) → Fagaras (176) → Bucharest (0)



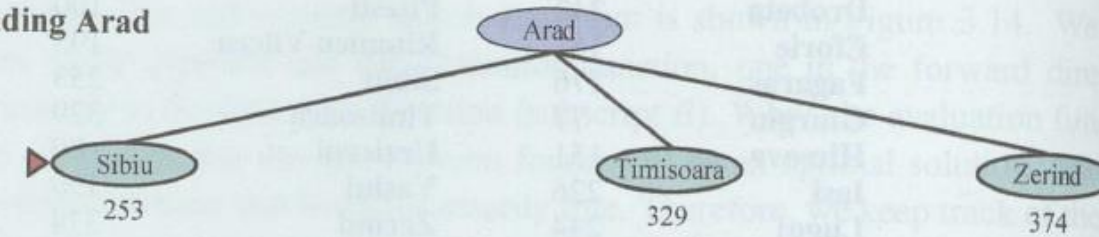
Greedy Best-first Search (2)

- Greedy search keeps a list of the search frontier and always selects the node with the lowest estimated costs to the goal.
- It is not optimal
- It finds a solution, if a path to the solution exists (if the solution space is finite)
- In the worst case, it checks every vertex V , i.e. it has a time and space complexity of $O(|V|)$ (equivalent to $O(b^d)$)
- Critic1: Add straight line distance to next city to straight line distance from that city to Bucharest for a better estimate.
- Critic2: Add real path costs to a city to estimate to the goal ($\rightarrow A^*$ search)

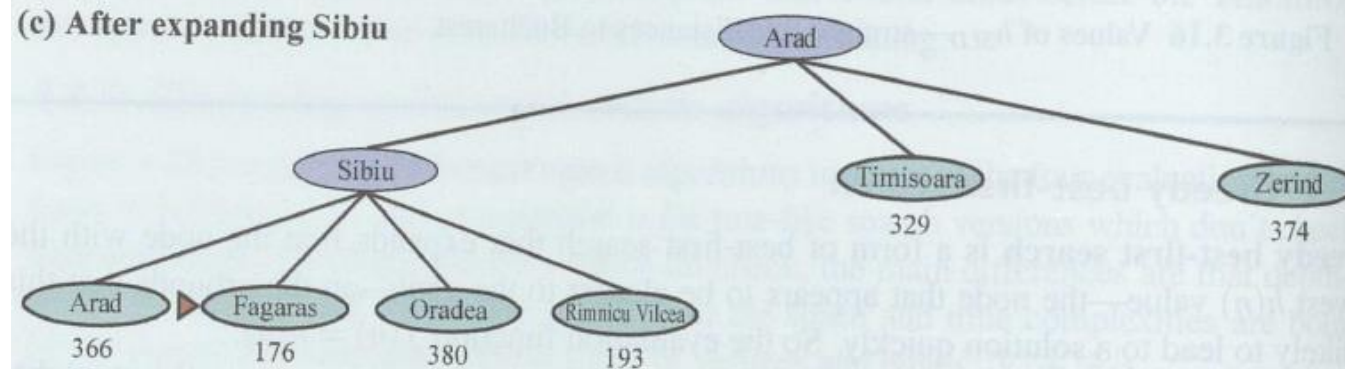
(a) The initial state



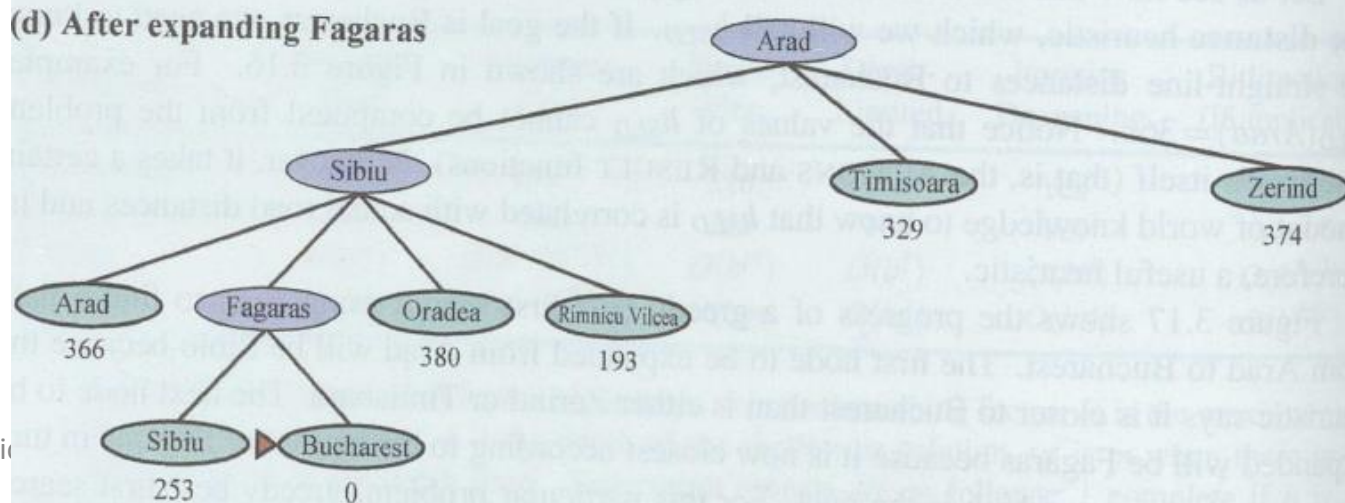
(b) After expanding Arad



(c) After expanding Sibiu



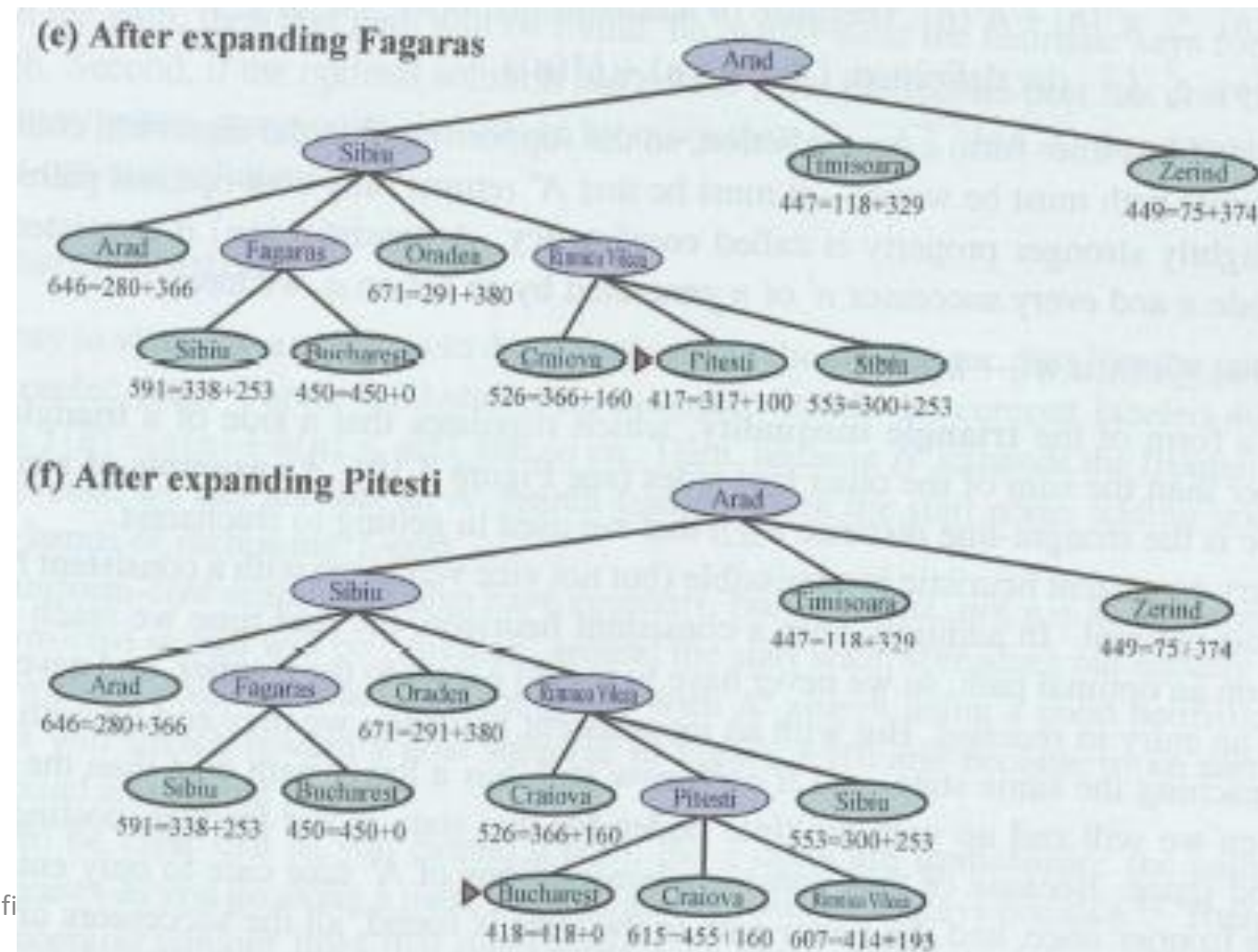
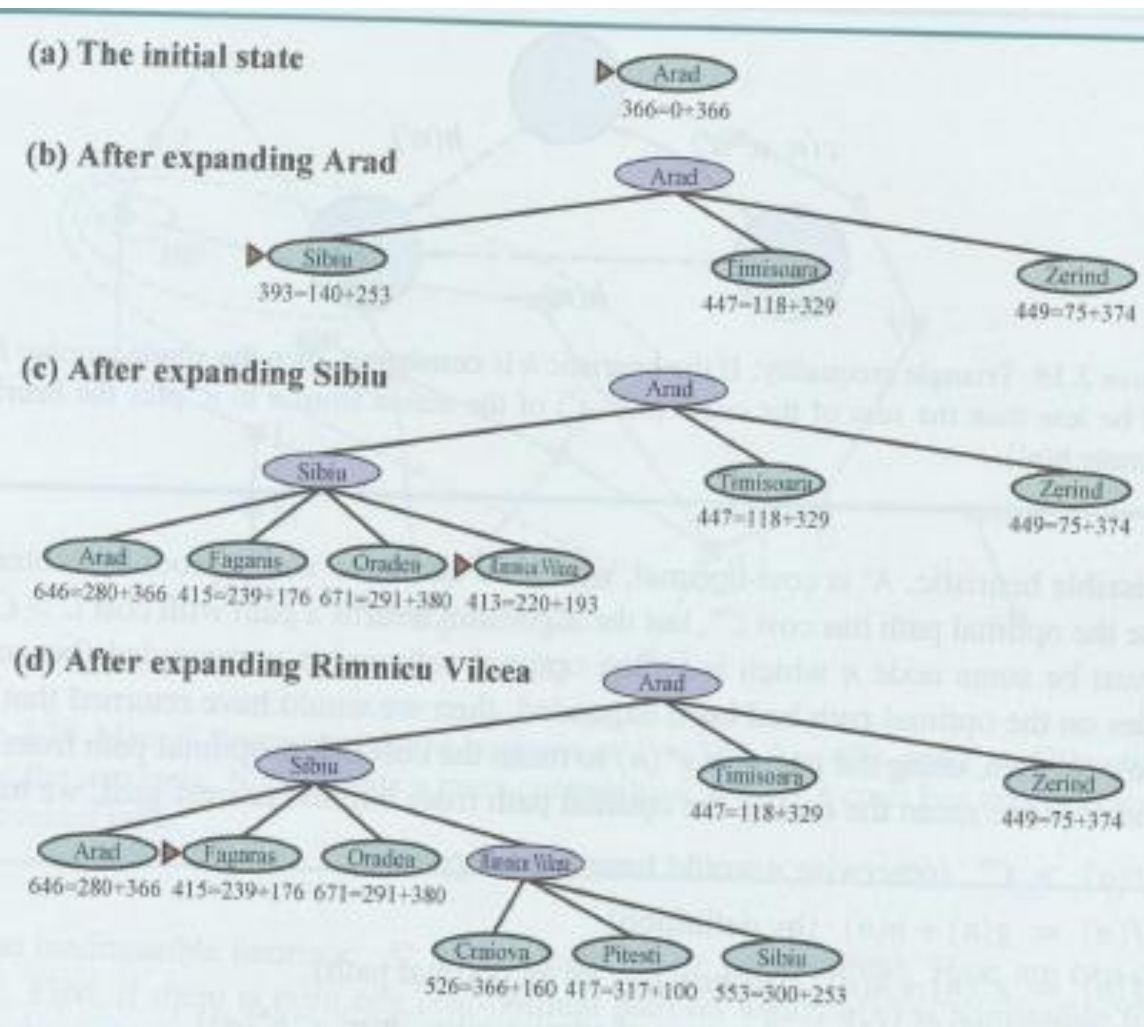
(d) After expanding Fagaras



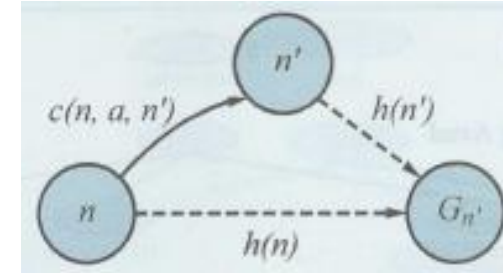
- Reduce Space Complexity:
 - Remember only the best path (hill climbing)
 - Remember several best paths (beam search)
- Treated in next chapter



- $f(n) = g(n) + h(n)$
- Most common informed search algorithm
- Example: In each step, the node with the least f-value from all frontier-nodes is expanded.



- If function h is optimistic and doesn't overestimate the costs to the goal, it is „**admissible**“
- If function h fulfils the triangle inequility, $h(n) \leq c(n, a, n') + h(n')$, it is „**consistent**“ i.e.
- consistent \rightarrow admissable
- Warranty: if A* selects a (goal) node for expansion, it has found the best path to that node
- With an inadmissible heuristic, A* may be not cost-optimal.
 - Degree of overestimation of the heuristic relevant.



Proof by contradiction (for consistent heuristics, for admissible heuristics similar):

1. If $h(n)$ is consistent, then the f -values on a path never decrease
2. If A* has selected a node for expansion, then it has found the optimal path to that node
3. If a goal node is chosen for expansion, the optimal goal node and its path has been found

ad1: Follows from consistency: $f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$

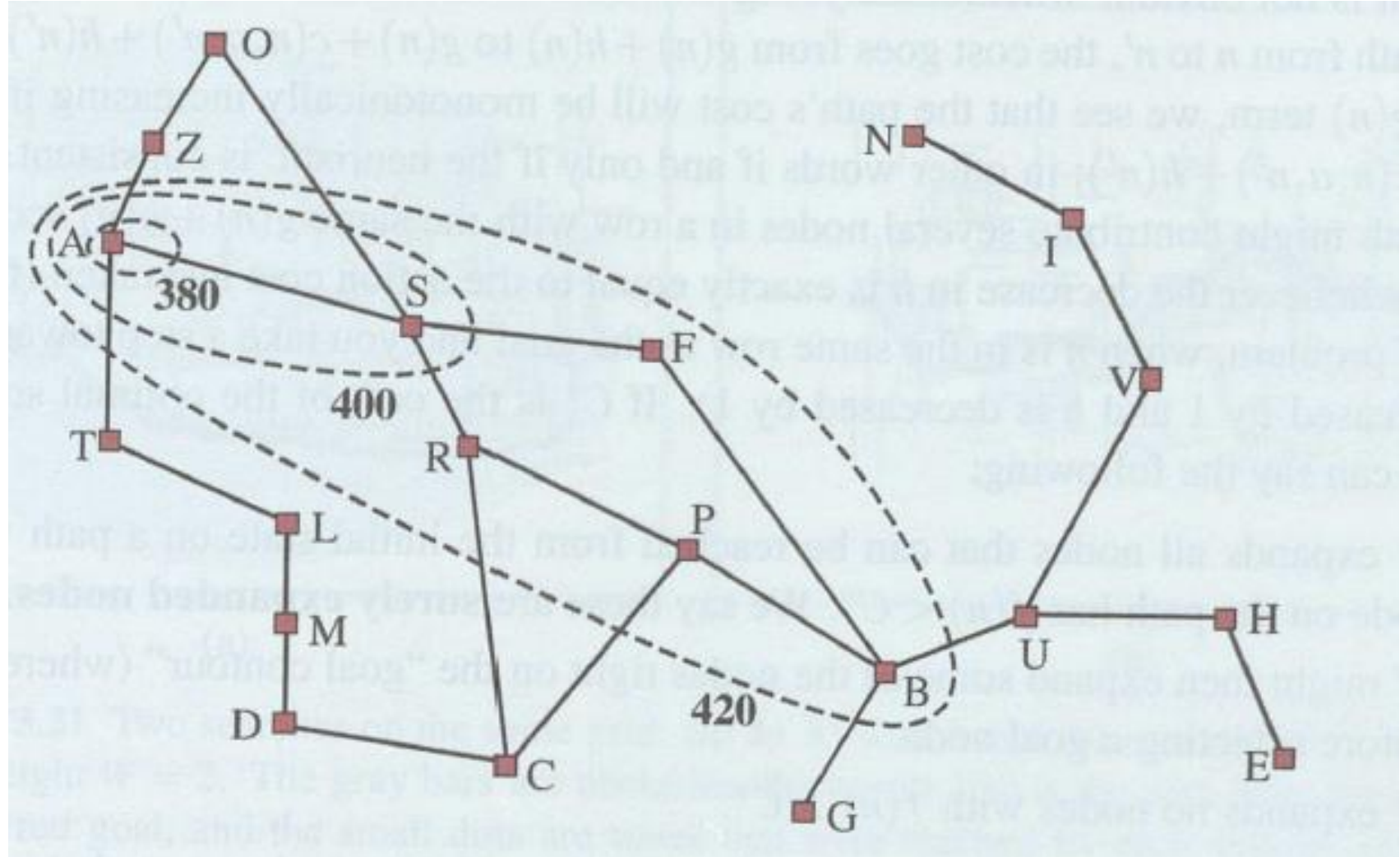
ad2: Otherwise a node n' exist on the path to n with smaller f -costs (because of 1). This node would have been expanded before n (because of A*-algorithm) \rightarrow contradiction

ad3: Goal nodes expanded after the first goal node must have f -costs equal or higher to the first goal node. For all goal nodes, $h(n) = 0$ and $f(n) = g(n)$. Therefore their g -costs (path costs) are higher or equal to the first goal node



A* cuts the search space by so call search-contours corresponding to f-values:

- Three search contours with $f = 380$, 400 , and 420 .
- A* increases the f-value in every step and explores only nodes within the contour.



- With an inadmissible heuristic, A* may be not cost-optimal, but more efficient.
 - Degree of overestimation of the heuristic relevant.
- Example: Detour index:
 - A road engineer might estimate the true length of a path between A and B as the straight line distance multiplied by a detour index (e.g. 1.3).
- **Weighted A***: $f(n) = g(n) + W * h(n)$ for some $W > 1$
- If the optimal solution has path costs C^* , it might yield a solution with costs $W * C^*$
 - In praxis much more closer to C^* than to $W * C^*$
- Generalization $f(n) = g(n) + W * h(n)$ with different values for W :
 - $W = 1$: A* search
 - $W = 0$: Uniform-Cost search
 - $W = \infty$: Greedy best-first search
 - $1 < W < \infty$: Weighted A* search (somewhat greedy)



- Main problem of A^* is exponential memory requirement
- Simple Idea: **Beam search**: Keep only the k best nodes (highest f -value) discarding the rest
 - might be efficient, but has no guarantee about the solution
- Other techniques for reduction are based on similar ideas as iterative deepening search: Recomputing some nodes to save memory
- Three approaches:
 - **IDA***: Iterative-deepening A^* search:
 - **RBFS**: Recursive best-first search
 - **SMA***: Simplified memory bounded A^* - use all available memory



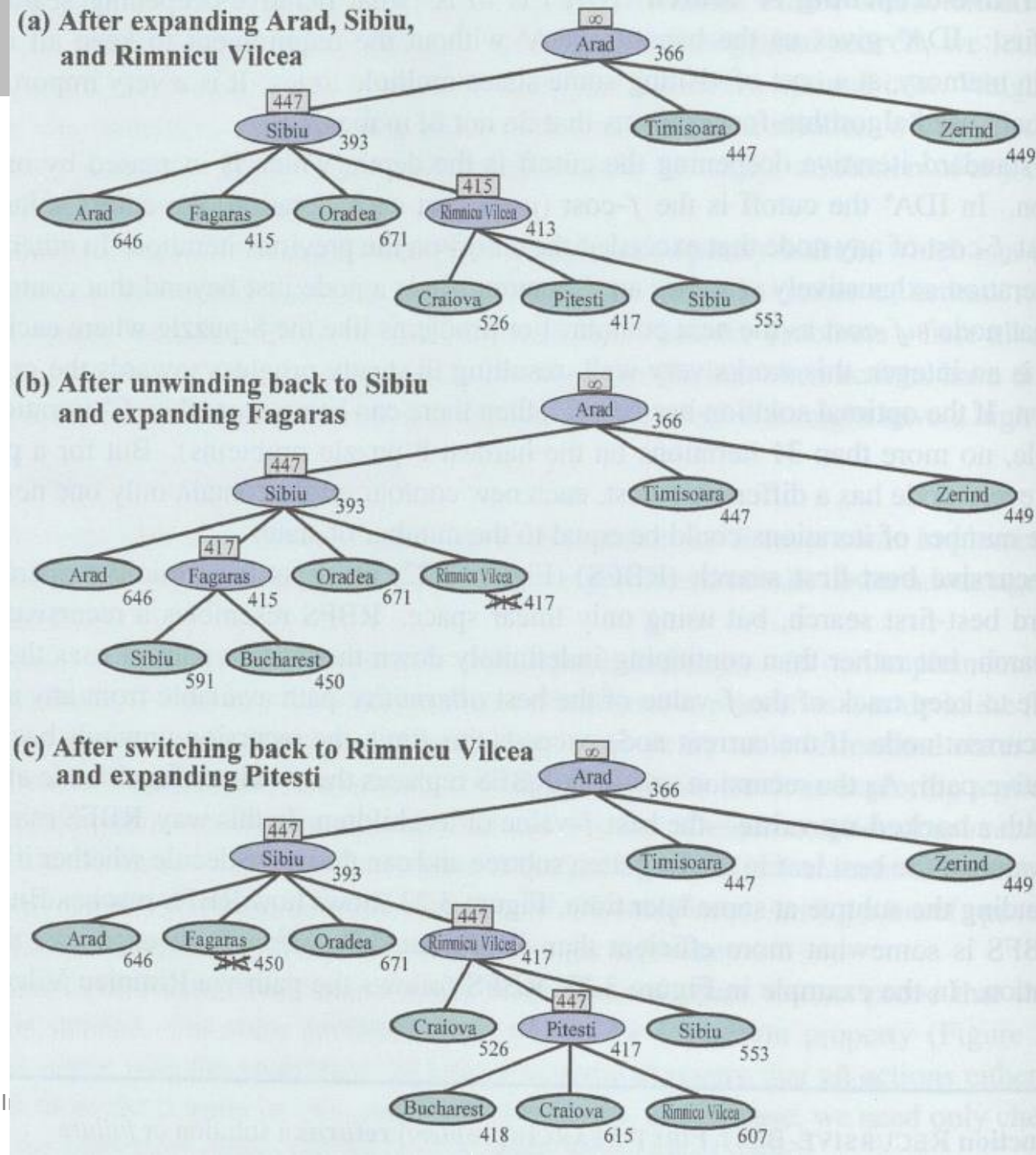
Same idea as interative deepening search: IDA* follows a path to a predefined f-cost boundary. If no solution is found, the boundary is increased and the search repeated.

- Advantage: It is not necessary to store all nodes in the search frontiers, since they are (re)generated.
- Disadvantage: No good heuristic for increasing the f-cost boundary
 - With minimal higher f-value in each iteration might be just one additional node computed → very inefficient

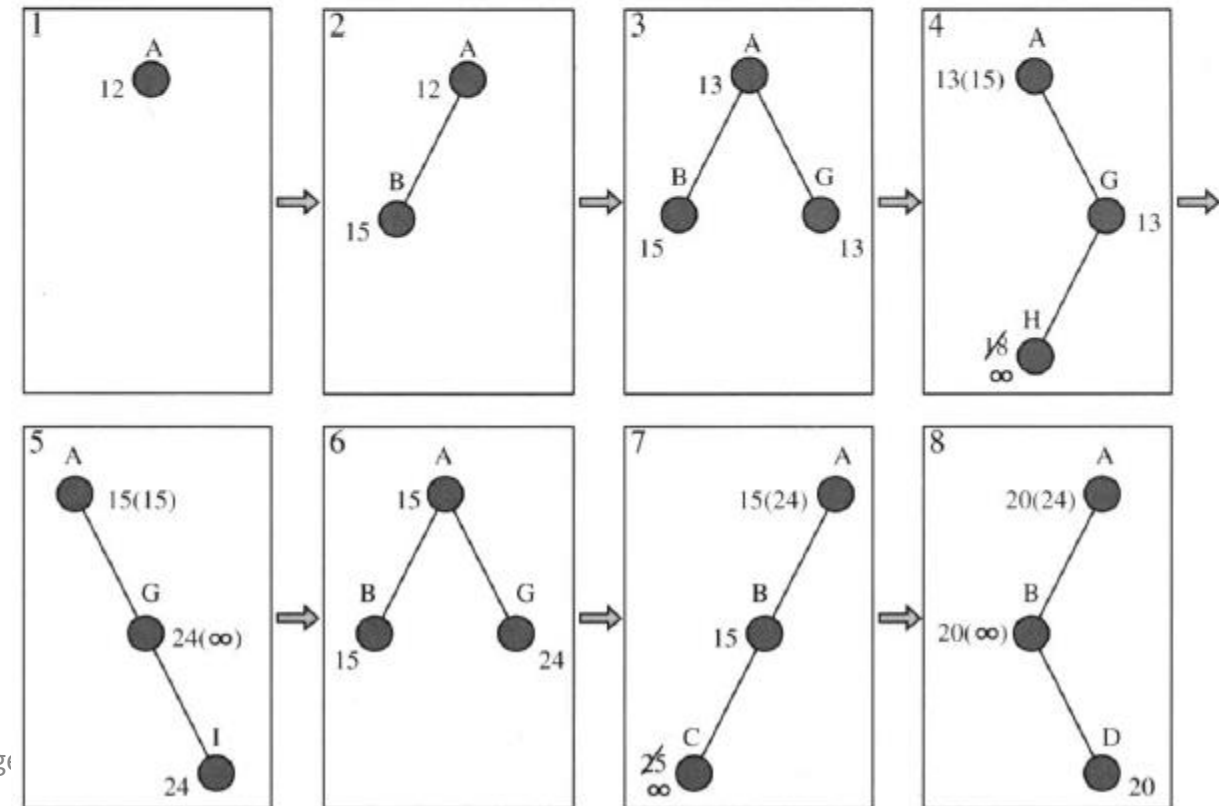
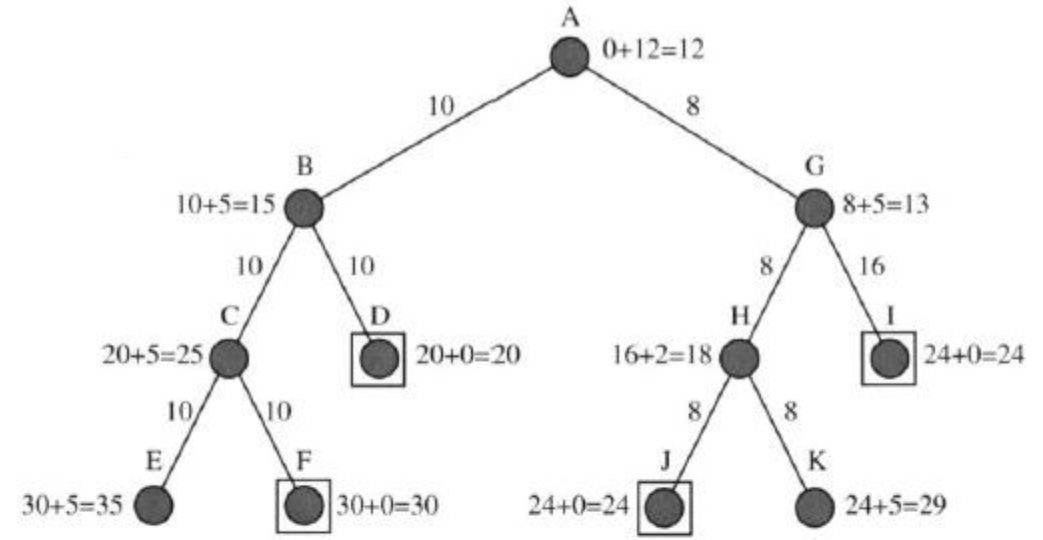


RBFS

- RBFS (recursive best-first search) follows just one path like depth-first-search until another node has a better f-value. Then it switches to the other path and remembers the f-value on the appropriate node in the new path.
- More efficient than IDA*, but does not really solve the underlying problem of increasing the f-boundary.



- IDA* remembers from one iteration to the next one only the f-cost boundary, while SMA* stores as many nodes as fitting into memory.
- Problem: What action if memory is full?
- Solution: Forget the node with the worst f-costs and store in predecessor node the f-costs of the forgotten node.
- Example with memory containing max. 3 nodes:
 - In step 3 the memory is full without a solution found, therefore node B is forgotten and its f-value (15) is stored in its predecessor A in step 4. Later in step 6, B is expanded again.



- Exploiting all available memory
- Avoiding multiple visits of nodes as long as memory is available
- Complete, if enough memory for the shortest solution path
- Optimal, if enough memory for the optimal solution path
- Optimal efficient, if enough memory for the full search graph (like A*)

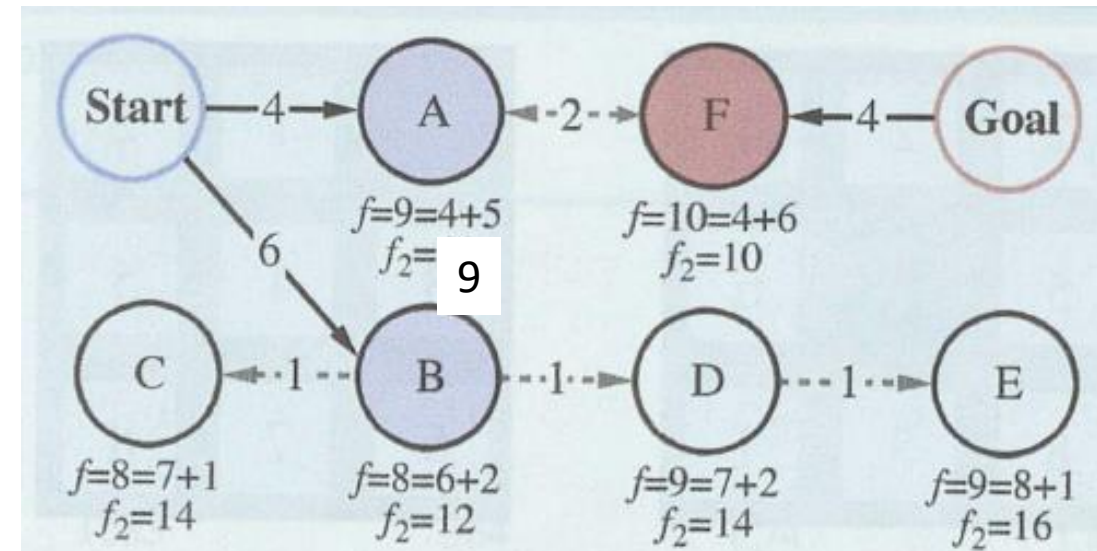
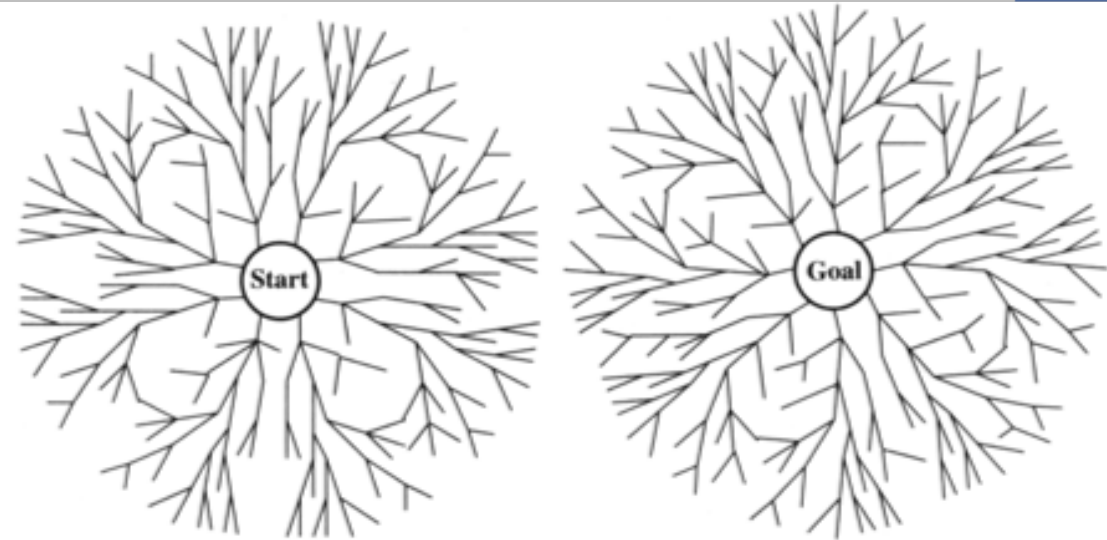


Frank Puppe

- Problem: It is possible, to guide A* in a way, that both frontiers searching SG (start to goal) and GS (goal to start) meet?
- Heuristic functions for SG and GS maybe different
- Key observation: Both optimal nodes on the search frontier meeting each other cannot have a g-value $>$ half of the optimal path ($C^*/2$)
- Bidirectional heuristic search uses an A* approach for both subproblems SG and GS and selects the best node from both search frontiers with an adapted f_2 -function:

$$f_2(n) = \max (2g(n), g(n) + h(n))$$

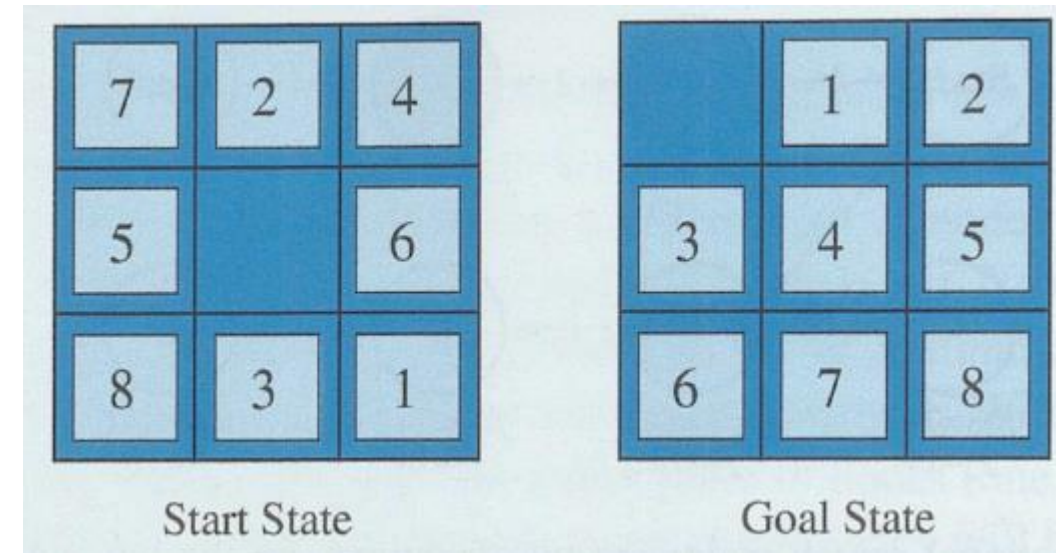
- e.g.: A: $f(n)=g(n)+h(n)$ i.e. $9=4+5$; $f_2(n)=\max(8,9)=9$
- Other bidirectional heuristic approaches possible



Example: Optimal Path: Start \rightarrow A \rightarrow F \rightarrow Goal
A & F have better f_2 -values than B, C, D, E (f_2 -values worse)



- Common heuristics for 8-puzzle:
 - h_1 = number of misplaced tiles (without blank)
 - h_2 = sum of the distances of the tiles from their goal positions
 - h_1 (example) = 8
 - h_2 (example) = 18 (3+1+2+2+2+3+3+2)
- Is h_2 always better than h_1 ?
 - Yes, because both are admissable and $h_2 \geq h_1$



Example above:

- shortest solution: 26 actions
- $9!/2 = 181\,400$ reachable states
 - half of the states are not reachable

For comparison 16-puzzle:

- $16!/2 > 10$ trillion states (10^{12})



- Expanded nodes at depth d by A^* : $1 + b^* + (b^*)^2 + (b^*)^3 + \dots + (b^*)^d$
- Effect of A^* : reduction of branching factor from $O(b^d)$ to $O(b^{d-k})$; example for 8-puzzle:

d	Search Cost (nodes generated)			Effective Branching Factor		
	BFS	$A^*(h_1)$	$A^*(h_2)$	BFS	$A^*(h_1)$	$A^*(h_2)$
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36



- Generating heuristics
 - From relaxed problems
 - From subproblems (pattern database)
 - From landmarks
- Learning to search better (metalevel learning and learning heuristics from experience)



h_1 and h_2 can be viewed as exact distance measures for a simplified problem:

- h_1 , if a tile can be moved to every field
- h_2 , if a tile can be moved to any neighbor field

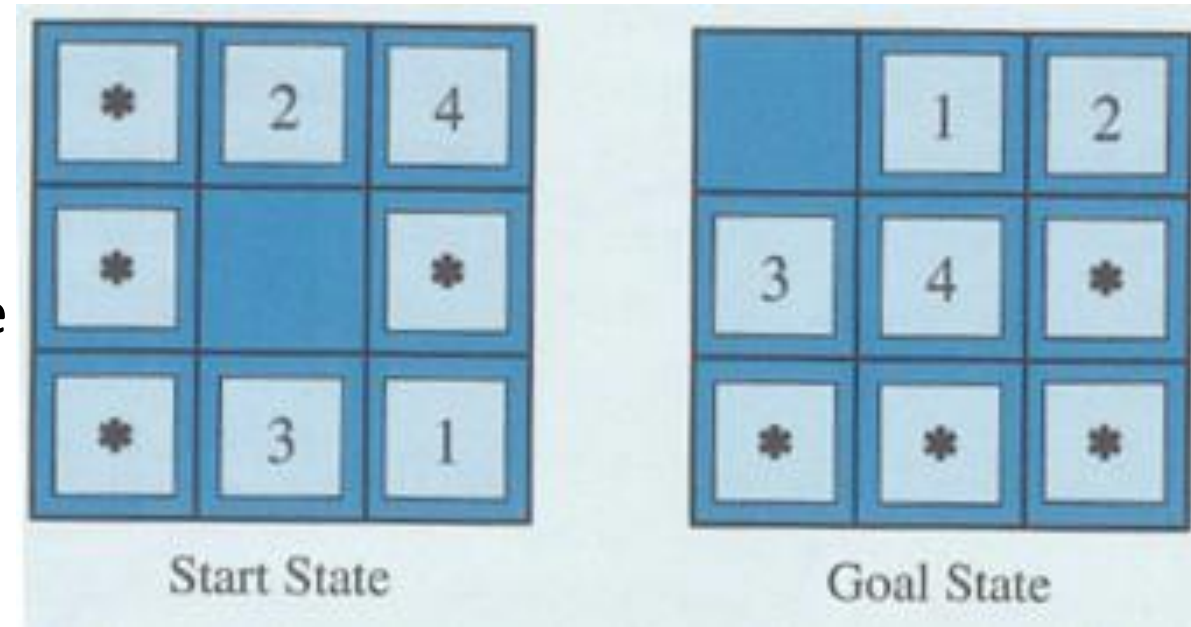
Often, the costs for an **exact solution for a simplified problem** is a good (admissible) heuristic for the original problem

Examples

- **8-puzzle standard rule:** A tile can move to another field, if it is adjacent and empty
 - **Relaxation** of adjacent and empty
- **Shortest path standard rule:** You have to use streets to move from one place to another
 - **Relaxation:** You can fly.



- The costs for solving a subproblem is a lower bound for the costs for solving the complete problem
 - Compute solutions for subproblems and use their costs as admissible heuristic, if the current state matches the subproblem
 - If for a current state several subproblems match, take the maximum



- There exist $9 \times 8 \times 7 \times 6 \times 5 = 15\,120$ patterns of four tiles and the blank.



- For estimating the shortest path to a goal, landmark points are useful:
 - We know the shortest path to the landmark (precomputed)
 - We compute the shortest path from the landmark to the goal
 - We add both values
- Problem: Overestimation (e.g. if the goal lies slightly before the landmark) → inadmissible
- Solution: Subtract both values → admissible
 - Useless, if goal is far from landmark
 - Take the maximum of all landmarks

$$h_{DH}(n) = \max_{L \in Landmarks} |C^*(n, L) - C^*(goal, L)|$$



- Instead from clever considerations, heuristics can be simply learned from experience:
 - Each state must be described with some features and a learning algorithms can learn a function with a correlation of these feature to the true costs (which needs to be known).
 - Any learning algorithm suitable
 - However: No guarantee, that the learned heuristic is admissable
 - Learning on a metalevel, e.g. when a state is not promising
 - Reinforcement Learning

