Introduction to Multi-Agent Programming

4. Search algorithms and Path-finding

Uninformed & informed search, online search, ResQ Freiburg path planner

Alexander Kleiner, Bernhard Nebel

Contents

Problem-Solving Agents

→ Goal-based agents

Formulation: *goal* and *problem*

Given: *initial state*

Goal: To reach the specified goal (a state) through the *execution of appropriate actions.*

→ Search for a suitable action sequence and execute the actions

A Simple Problem-Solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  inputs: percept, a percept
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

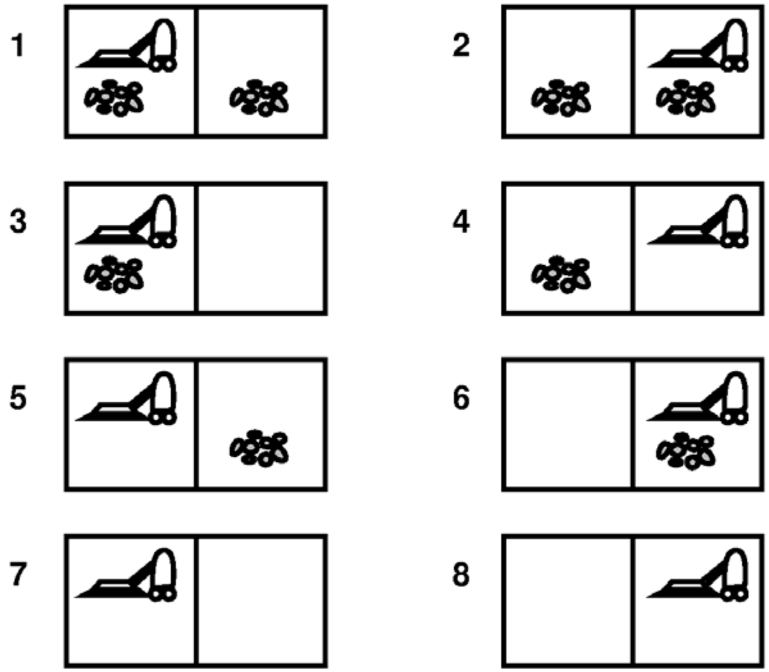
Problem Formulation

- Goal formulation
  World states with certain properties
- Definition of the state space
  important: only the relevant aspects → abstraction
- Definition of the actions that can change the world state
- Determination of the search cost (search costs, offline costs) and the execution costs (path costs, online costs)

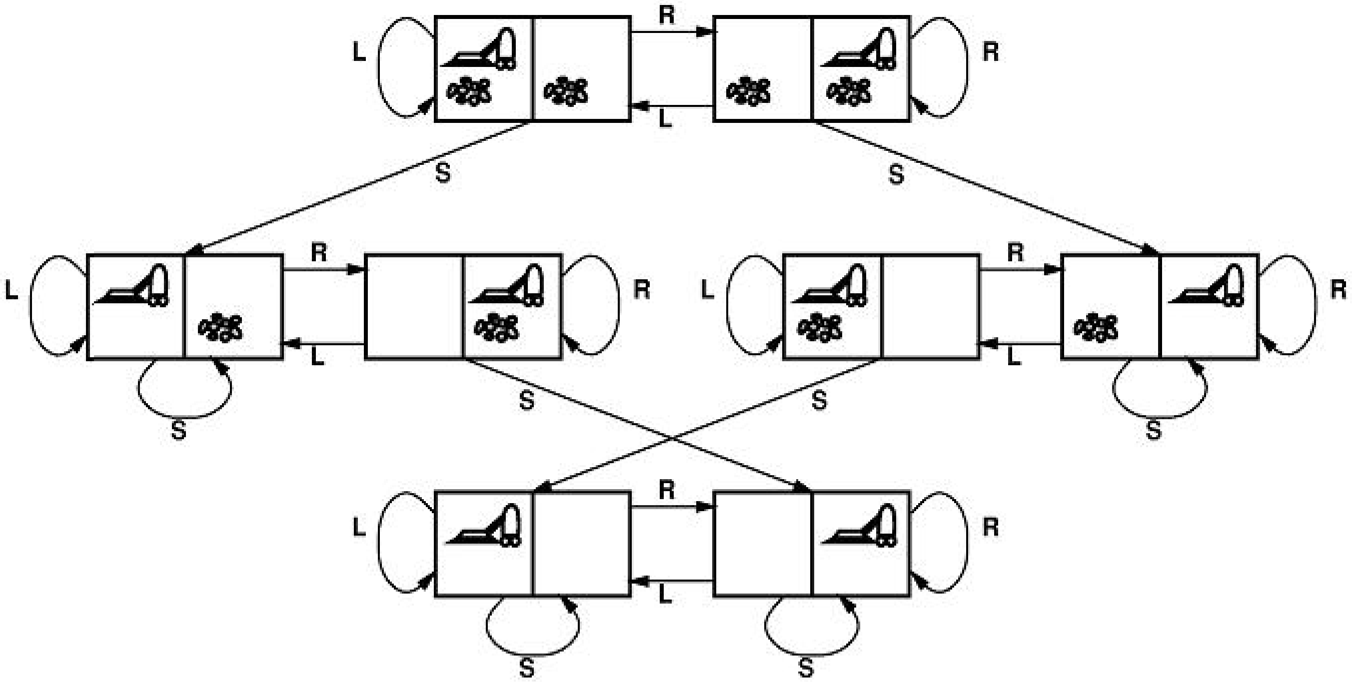
Note: The type of problem formulation can have a big influence on the difficulty of finding a solution.

Problem Formulation for the Vacuum Cleaner World

- World state space:
  2 positions, dirt or no dirt
  → 8 world states
- Successor function (Actions):
  Left (L), Right (R), or Suck (S)
- Goal state:
  no dirt in the rooms
- Path costs:
  one unit per action



The Vacuum Cleaner State Space



States for the search: The world states 1-8.

Example: Missionaries and Cannibals

Informal problem description:

- Three missionaries and three cannibals are on one side of a river that they wish to cross.
- A boat is available that can hold at most two people and at least one.
- You must never leave a group of missionaries outnumbered by cannibals on the same bank.

→ Find an action sequence that brings everyone safely to the opposite bank.

Formalization of the M&C Problem

State space: triple (x,y,z) with $0 \leq x,y,z \leq 3$, where x,y, and z represent the number of missionaries, cannibals and boats currently on the original bank.

Initial State: (3,3,1)

Successor function: From each state, either bring one missionary, one cannibal, two missionaries, two cannibals, or one of each type to the other bank.

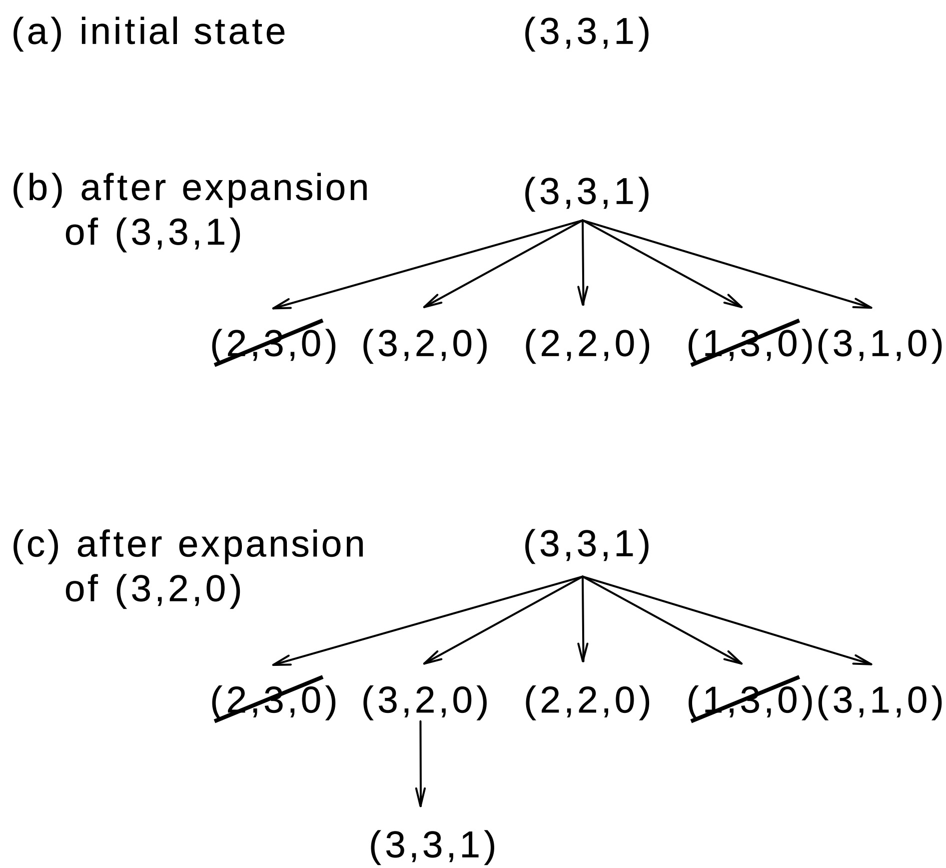
Note: Not all states are attainable (e.g., (0,0,1)), and some are illegal.

Goal State: (0,0,0)

Path Costs: 1 unit per crossing

General Search

From the initial state, produce all successive states step by step → search tree.



Implementing Search Algorithms

Data structure for nodes in the search tree:

State: state in the state space

Node: Containing a state, pointer to predecessor, depth, and path cost, action

Depth: number of steps along the path from the initial state

Path Cost: Cost of the path from the initial state to the node

Fringe: Memory for storing expanded nodes. For example, s stack or a queue

General functions to implement:

Make-Node(state): Creates a node from a state

Goal-Test(state): Returns true if state is a goal state

Successor-Fn(state): Implements the successor function, i.e. expands a set of new nodes given all actions applicable in the state

Cost(state,action): Returns the cost for executing action in state

Insert(node, fringe): Inserts a new node into the fringe

Remove-First(fringe): Returns the first node from the fringe

General Tree-Search Procedure

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
     if EMPTY?(fringe) then return failure
     node ← REMOVE-FIRST(fringe)
     if GOAL-TEST[problem] applied to STATE[node] succeeds
        then return SOLUTION(node)
     fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes

  successors ← the empty set
  for each ⟨action, result⟩ in SUCCESSOR-FN[problem](STATE[node]) do
     s ← a new NODE                                                  ┐
     STATE[s] ← result                                               │
     PARENT-NODE[s] ← node                                           │ Make-
     ACTION[s] ← action                                              │ Node
     PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)     │
     DEPTH[s] ← DEPTH[node] + 1                                      ┘
     add s to successors
  return successors
```

Search Strategies

Uninformed or blind searches:

No information on the length or cost of a path to the solution.

- breadth-first search, uniform cost search, depth-first search,
- depth-limited search, Iterative deepening search, and
- bi-directional search.

In contrast: informed or heuristic approaches

Criteria for Search Strategies

Completeness:

Is the strategy guaranteed to find a solution when there is one?

Time Complexity:

How long does it take to find a solution?

Space Complexity:

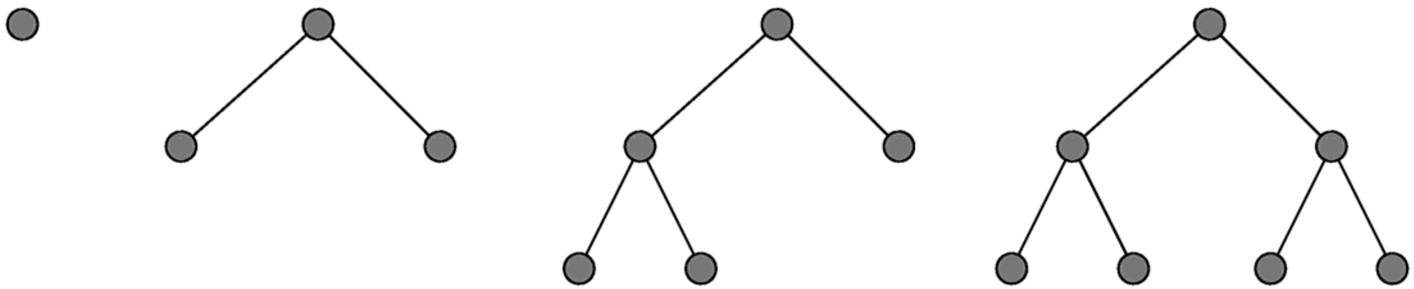
How much memory does the search require?

Optimality:

Does the strategy find the best solution (with the lowest path cost)?

Breadth-First Search (1)

Nodes are expanded in the order they were produced . *fringe* = Enqueue-at-end*()* (LIFO)*.*



- Always finds the shallowest goal state first.
- Completeness.
- The solution is optimal, provided the path cost is a non-decreasing function of the depth of the node (e.g., when every action has identical, non-negative costs).

Breadth-First Search (2)

The costs, however, are very high. Let *b* be the maximal branching factor and *d* the depth of a solution path. Then the maximal number of nodes expanded is

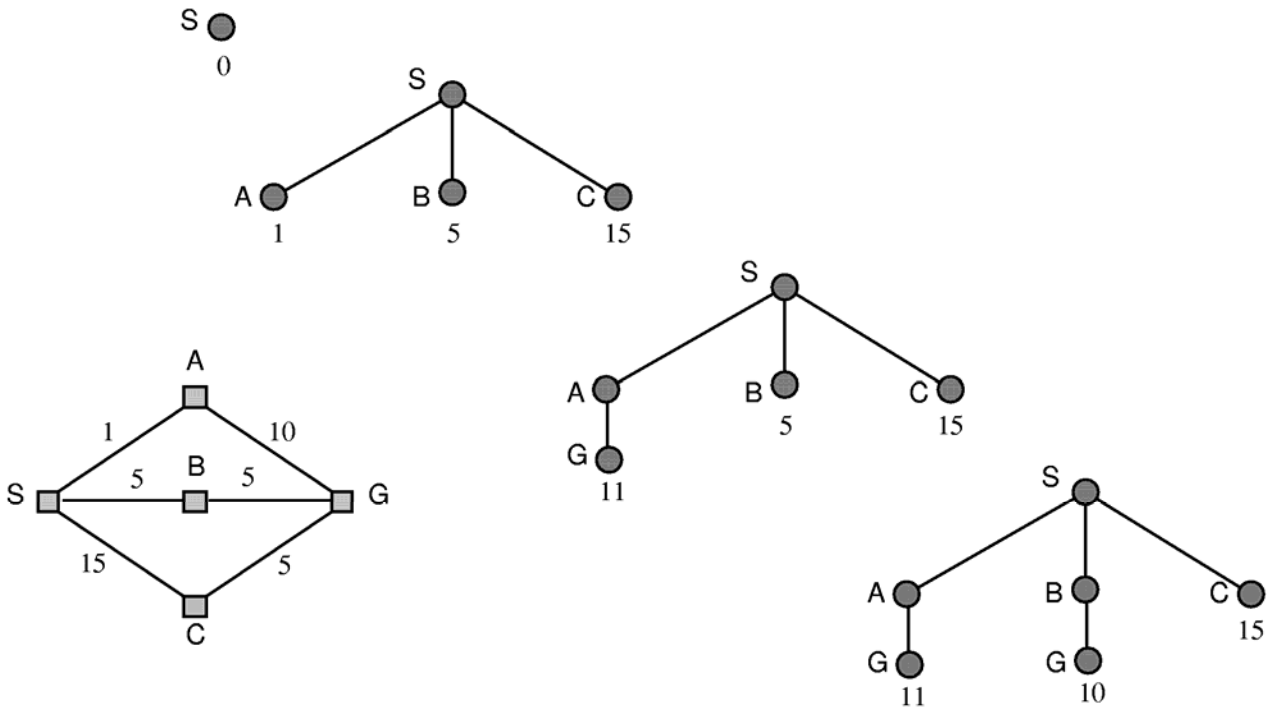
$$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) \in O(b^{d+1})$$

Example: $b = 10$, 10,000 nodes/second, 1,000 bytes/node:

Depth	Nodes	Time	Memory
2	1,100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

Uniform Cost Search

Modification of breadth-first search to always expand the node with the lowest-cost *g(n)*.



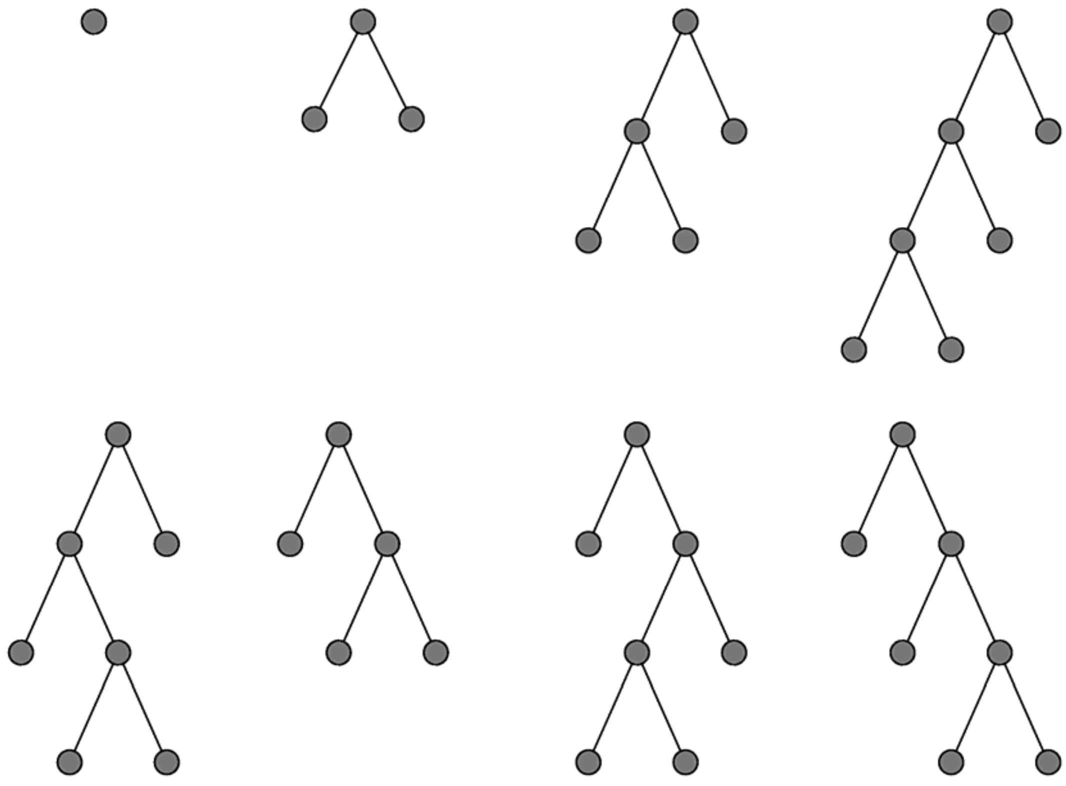
(a) (b)

Always finds the cheapest solution, given that *g(successor(n))* >= *g(n)* for all n.

Depth-First Search

Always expands an unexpanded node at the greatest depth
fringe = Enqueue-at-front (FIFO).

Example (Nodes at depth 3 are assumed to have no successors):



Iterative Deepening Search (1)

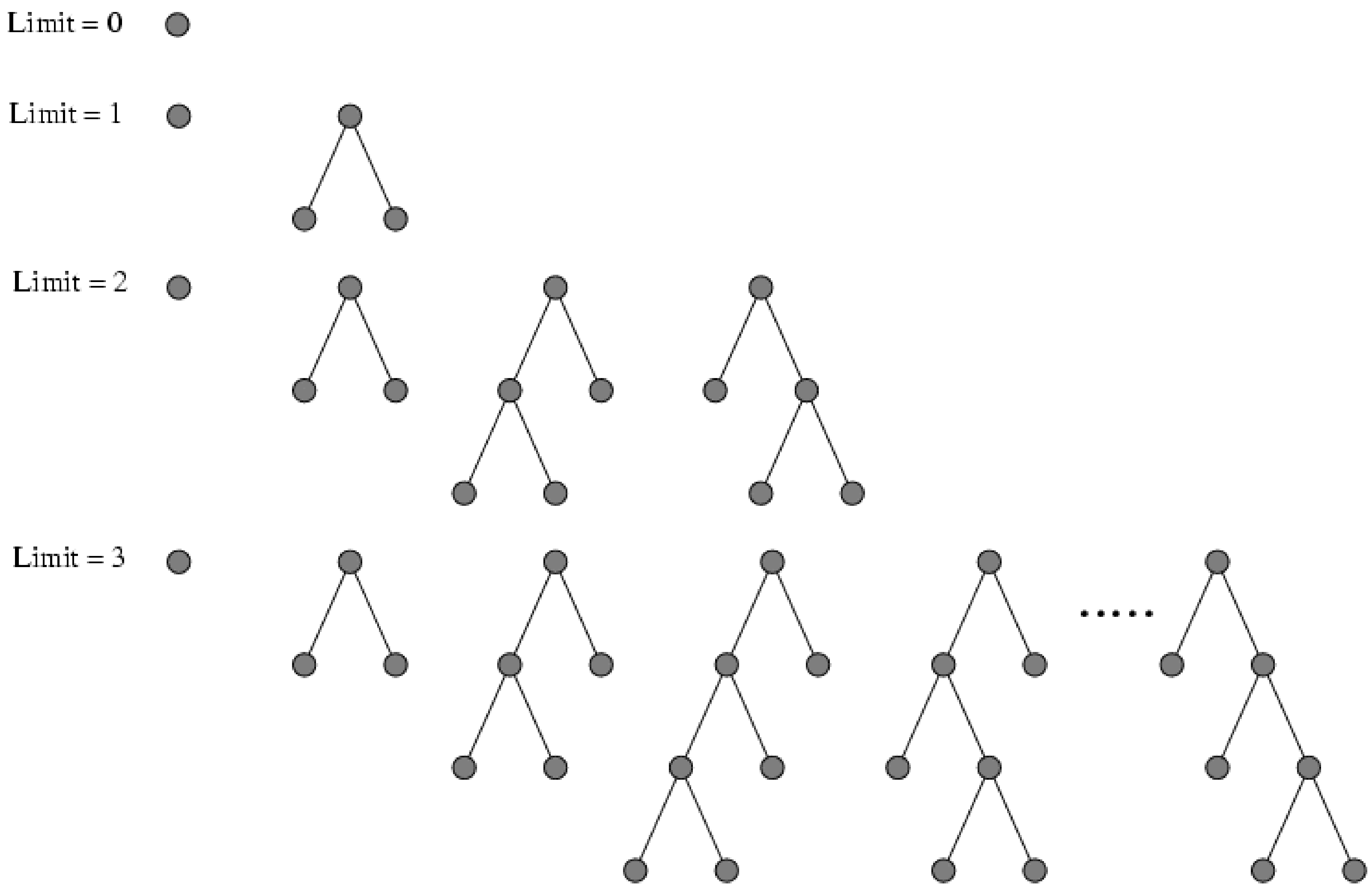
- Combines depth- and breadth-first searches
- Optimal and complete like breadth-first search, but requires less memory

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
   inputs: problem, a problem

   for depth ← 0 to ∞ do
      if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
   end
   return failure
```

Iterative Deepening Search (2)

Example



Iterative Deepening Search (3)

Number of expansions

Iterative Deepening Search	$(d)b + (d-1)b^2 + ... + 3b^{d-2} + 2b^{d-1} + 1b^d$
Breadth-First-Search	$b + b^2 + ... + b^{d-1} + b^d + b^{d+1} - b$

Example: $b = 10$, $d = 5$

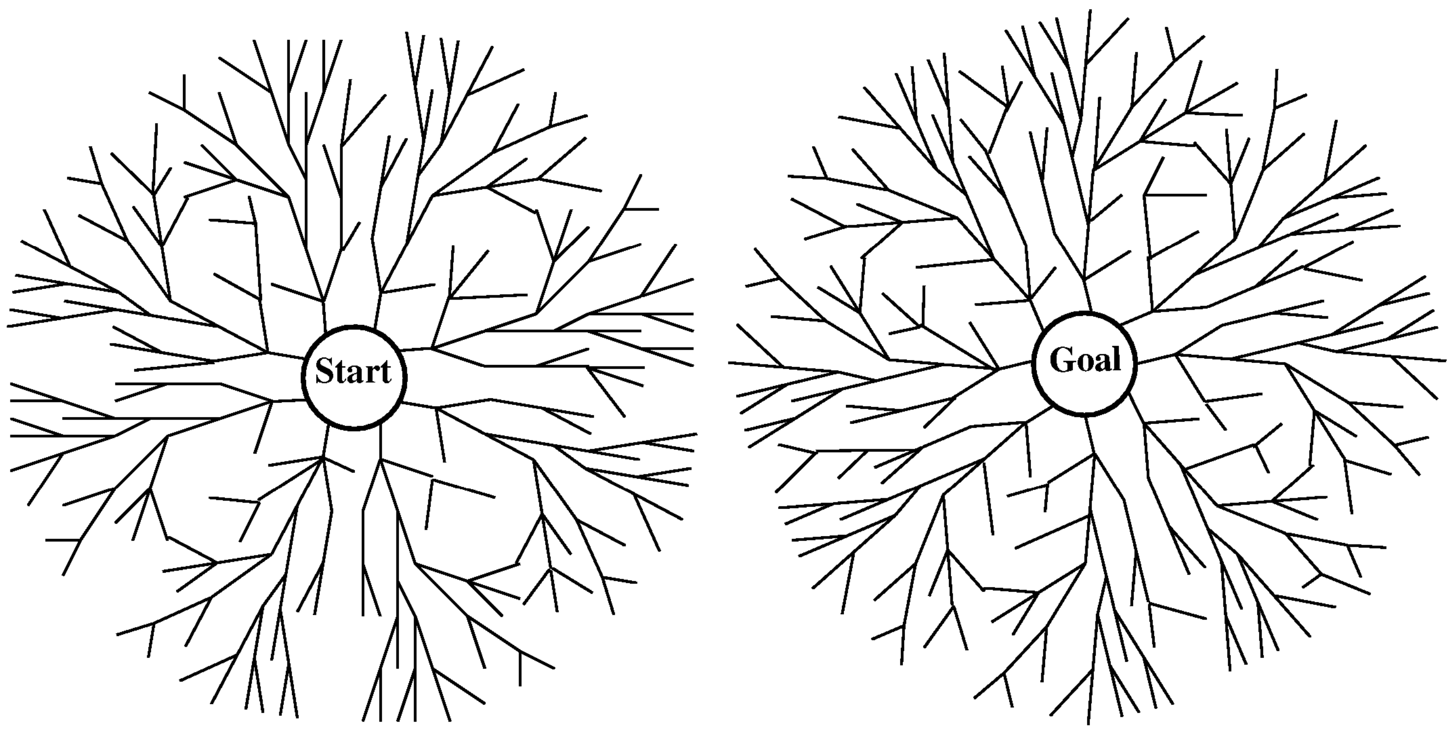
Breadth-First-Search	10 + 100 + 1,000 + 10,000 + 999,990 = 1,111,100
Iterative Deepening Search	50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

For b = 10, only 11% of the nodes expanded by breadth-first-search are generated, so that the memory requirement is considerably lower.

Time complexity: $O(b^d)$ Memory complexity: $O(b \cdot d)$

→ *Iterative deepening in general is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.*

Bidirectional Search



As long as forwards and backwards searches are symmetric, search times of $O(2 \cdot b^{d/2}) = O(b^{d/2})$ can be obtained.

E.g., for b=10, d=6, instead of 111111 only 2222 nodes!

Comparison of Search Strategies

Time complexity, space complexity, optimality, completeness

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

b branching factor

d depth of solution,

m maximum depth of the search tree,

l depth limit,

C* cost of the optimal solution,

ϵ minimal cost of an action

Superscripts:

a) b is finite

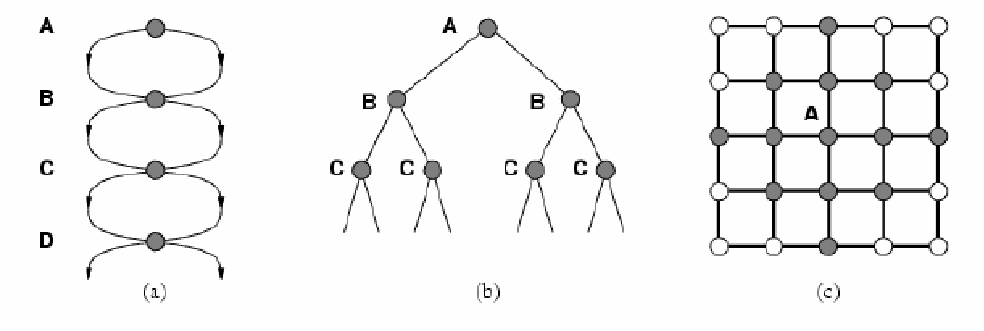
b) if step costs not less than ϵ

c) if step costs are all identical

d) if both directions use breadth-first search

Problems With Repeated States

- Tree search ignores what happens if nodes are repeatedly visited
  - For example, if actions lead back to already visited states
  - Consider path planning on a grid
- Repeated states may lead to a large (exponential) overhead



- (a) State space with d+1 states, were d is the depth
- (b) The corresponding search tree which has 2^d nodes corresponding to the two possible paths!
- (c) Possible paths leading to A

Graph Search

- Add a *closed* list to the tree search algorithm
- Ignore newly expanded state if already in *closed* list
- *Closed list* can be implemented as hash table
- Potential problems
  - Needs a lot of memory
  - Can ignore better solutions if a node is visited first on a suboptimal path (e.g. IDS is not optimal anymore)

Best-First Search

Search procedures differ in the way they determine the next node to expand.

Uninformed Search: Rigid procedure with no knowledge of the cost of a given node to the goal.

Informed Search: Knowledge of the cost of a given node to the goal is in the form of an *evaluation function* f or h, which assigns a real number to each node.

Best-First Search: Search procedure that expands the node with the "best" f- or h-value.

General Algorithm

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
   inputs: problem, a problem
           Eval-Fn, an evaluation function

   Queueing-Fn ← a function that orders nodes by EVAL-FN
   return GENERAL-SEARCH(problem, Queueing-Fn)
```

When *h* is always correct, we do not need to search!

Greedy Search

A possible way to judge the "worth" of a node is to estimate its distance to the goal.

h(n) = estimated distance from n to the goal

The only real condition is that $h(n) = 0$ if n is a goal.

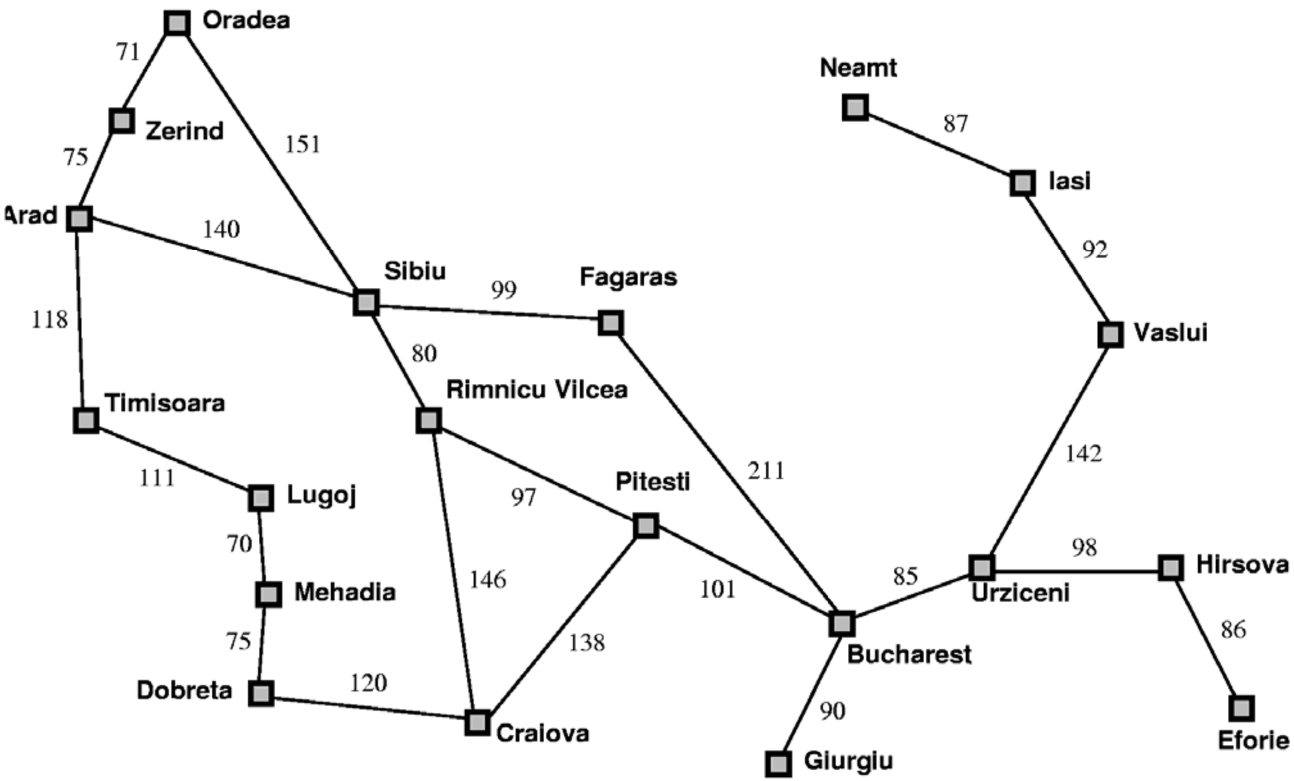
A best-first search with this function is called a *greedy search.*

The evaluation function h in greedy searches is also called a *heuristic* function or simply a *heuristic.*

→In all cases, the heuristic is *problem-specific* and *focuses* the search!

Route-finding problem: h = straight-line distance between two locations.

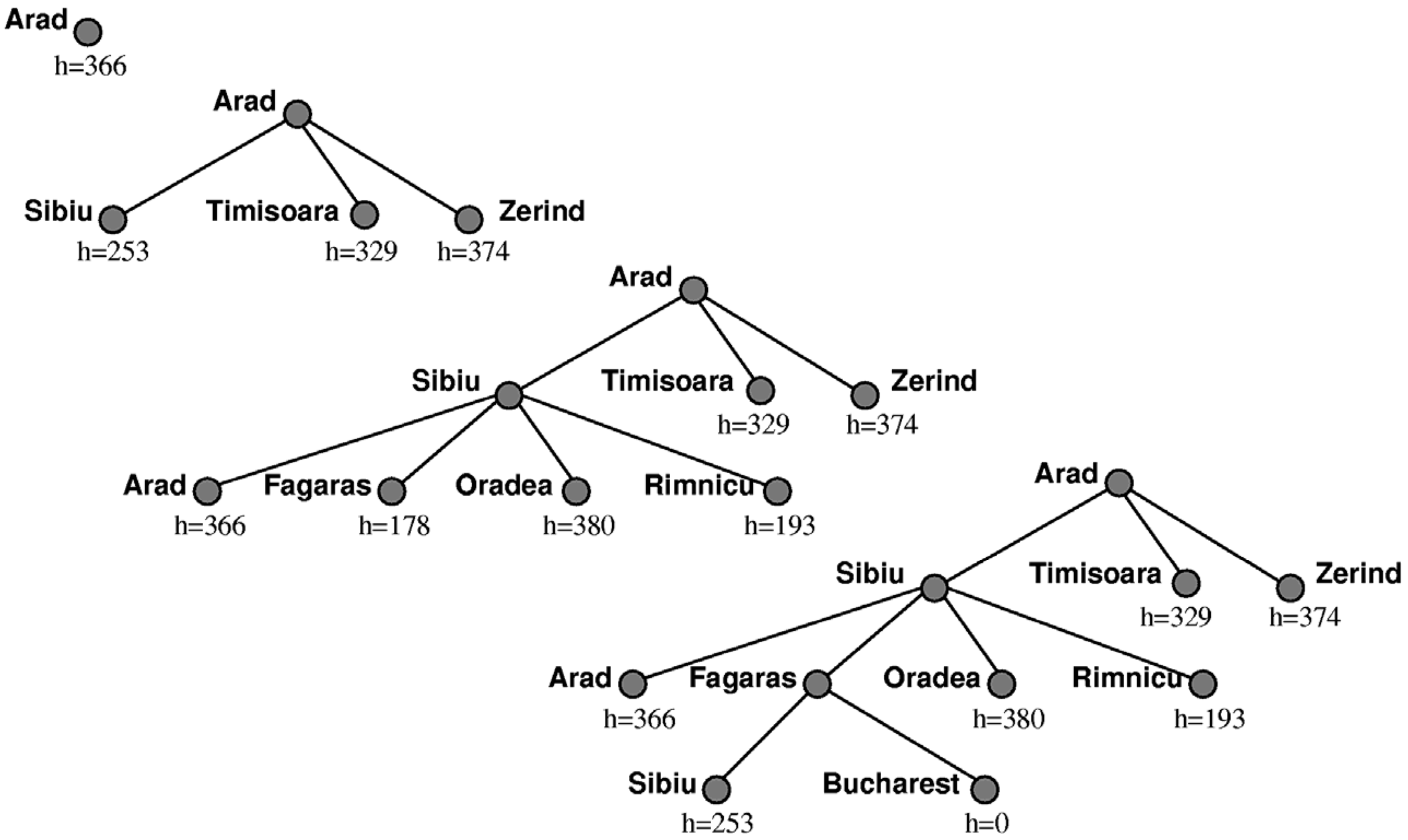
Greedy Search Example



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 from *Arad* to *Bucharest*



A*: Minimization of the estimated path costs

A* combines the greedy search with the uniform-cost-search, i.e. taking costs into account.

$g(n)$ = actual cost from the initial state to n.

$h(n)$ = estimated cost from n to the next goal.

$f(n) = g(n) + h(n)$, the estimated cost of the cheapest solution through n.

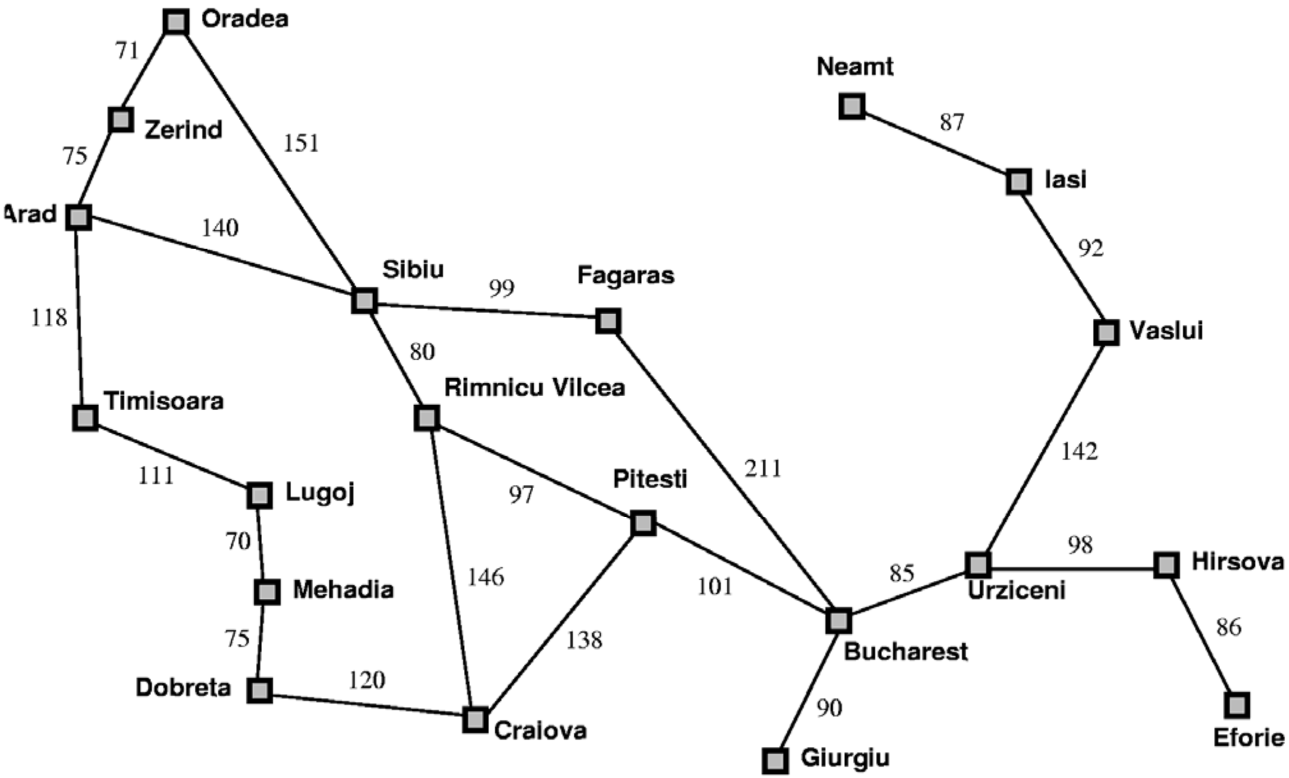
Let h*(n) be the true cost of the optimal path from n to the next goal.

h is *admissible* if the following holds for all n :

$$h(n) \leq h^*(n)$$

We require that for optimality of A*, h is admissible (straight-line distance is admissible).

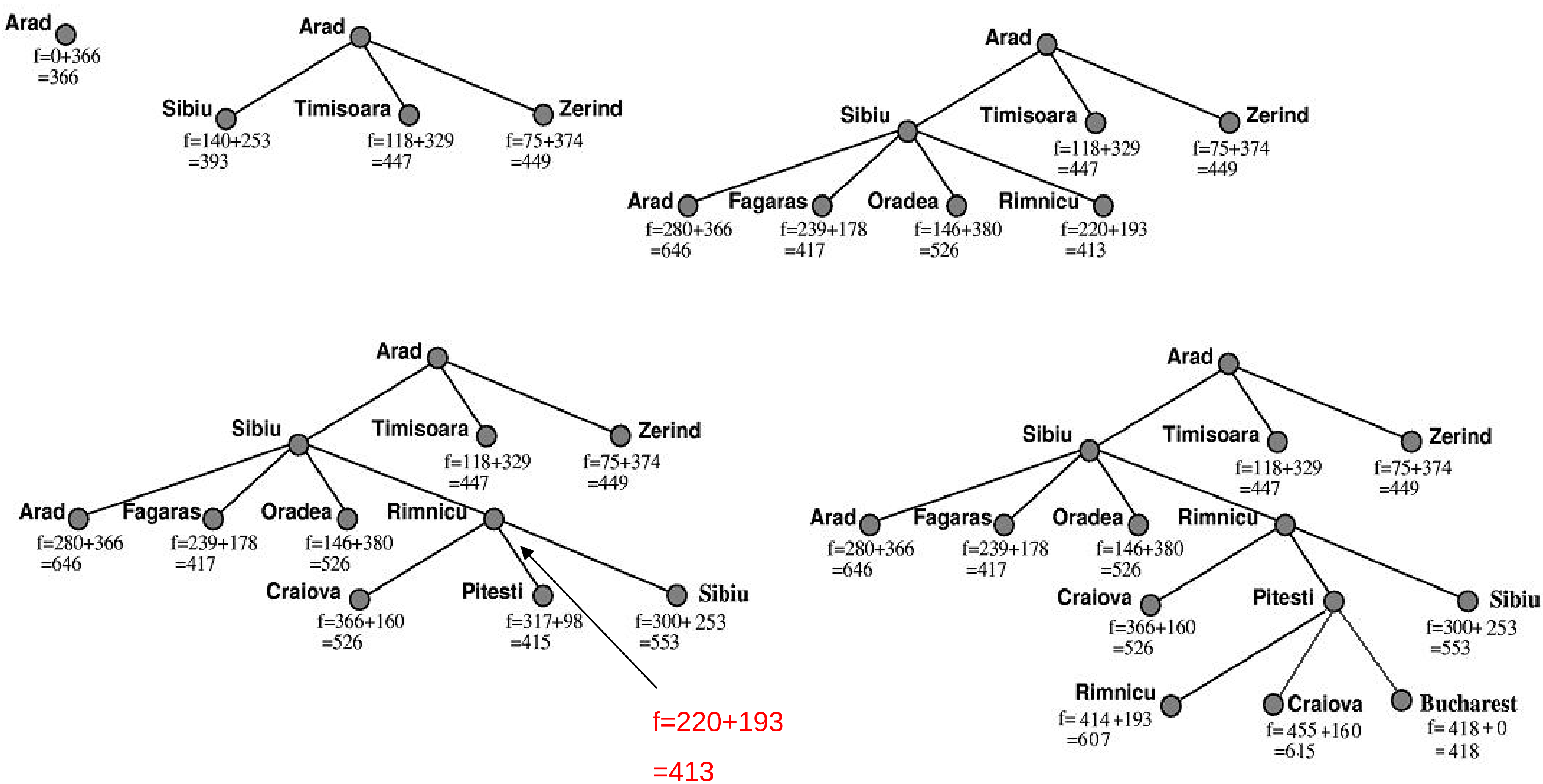
A* Search Example



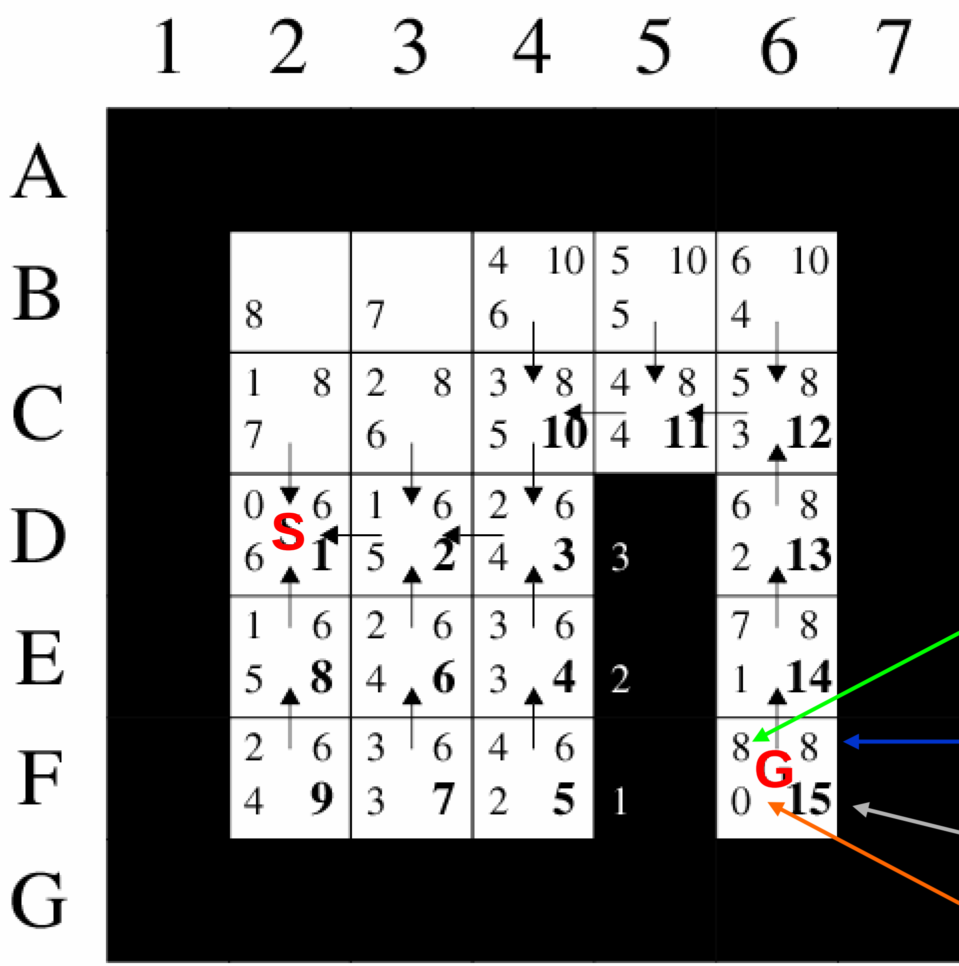
Straight–line distance to Bucharest

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* Search from *Arad* to *Bucharest*



A* Grid World Example



Courtesy of Sven Koenig

S: Start state

G: Goal state

→: Parent pointer in the A* search tree

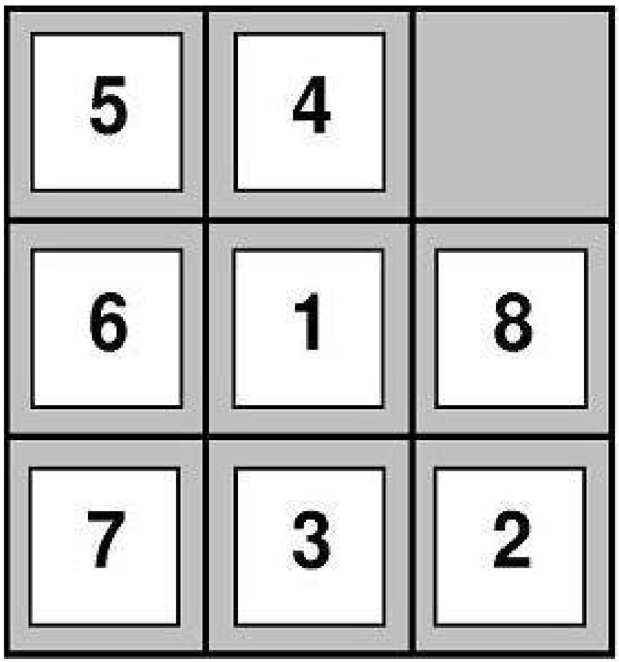
g(s): Accumulated path cost

f(s)=g(s)+h(s)

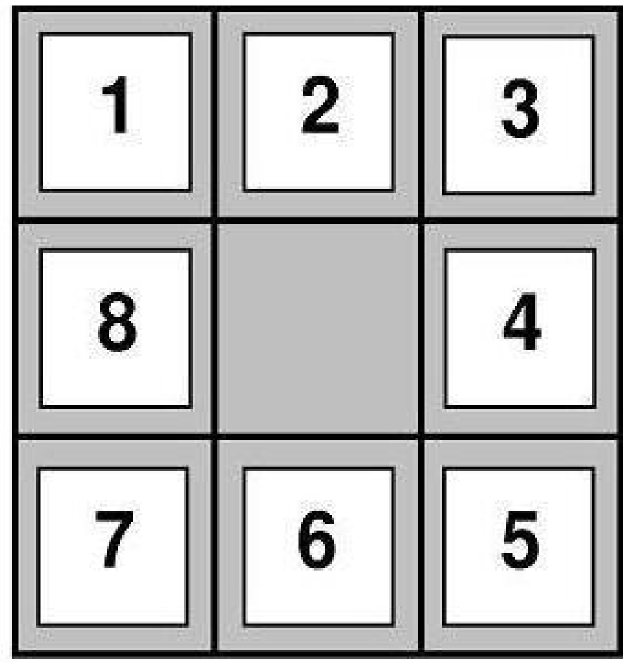
Time of expansion

h(s): Manhattan distance

Heuristic Function Example



Start State



Goal State

h_1 = the number of tiles in the wrong position

h_2 = the sum of the distances of the tiles from their goal positions (*Manhatten distance)*

Empirical Evaluation

- *d* = distance from goal
- Average over 100 instances

	Search Cost			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	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	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

A* Implementation Details

- How to code A* efficiently?
- Costly operations are:
  - Insert & lookup an element in the closed list
  - Insert element & get minimal element (f-value) from open list
- The closed list can efficiently be implemented as a hash set
- The open list is typically implemented as a priority queue, e.g. as
  - Fibonacci heap, binomial heap, k-level bucket, etc.
  - binary-heap with O(log n) is normally sufficient
- Hint: see *priority queue implementation in the "Java Collection Framework"*

Online search

- Intelligent agents usually don‘t know the state space (e.g. street map) exactly in advance
  - True travel costs are experienced during execution
- Planning and plan execution are interleaved
- Example: RoboCup Rescue
  - The map is known, but roads might be blocked from building collapses
  - Limited drivability of roads depending on traffic volume
- Important issue: How to reduce computational cost of repeated A* searches!

Online search

- Incremental heuristic search
  - Repeated planning of the complete path from current state to goal
  - Planning under the free-space assumption
  - Optimized versions reuse information from previous planning episodes:
    - Focused Dynamic A* (D*) [Stenz95]
      - Used by DARPA and NASA
    - D* Lite [Koenig et al. 02]
      - Similar as D* but a bit easier to implement (claim)
  - In particular, these methods reuse closed list entries from previous searches
  - All Entries that have been compromised by weight updates (from observation) are adjusted accordingly
- Real-Time Heuristic search
  - Repeated planning with limited look-ahead (agent centered search)
  - Solutions are suboptimal but faster to compute
  - Updated of heuristic values of visited states
    - Learning Real-Time A* (LRTA*) [Korf90]
    - Real-Time Adaptive A* (RTAA*) [Koenig06]

Real-Time Adaptive A* (RTAA*)

- Executes A* plan with limited lookahead
- Learns better informed heuristic H(s) from experience (initially h(s), e.g. Euclidian distance)
- Lookahed defines trade-off between optimality and computational cost

```
while (s_curr ∉ GOAL)
        astar(lookahead);
        if (s' = FAILURE) then
                return FAILURE;
        for all s ∈ CLOSED do
                H(s) :=
g(s')+h(s')-g(s);
        end;
        execute(plan);
end;
return SUCCESS;
```

s': last state expanded during previous A* search

Real-Time Adaptive A* (RTAA*)
Example

After first A* planning with lookahead until s':

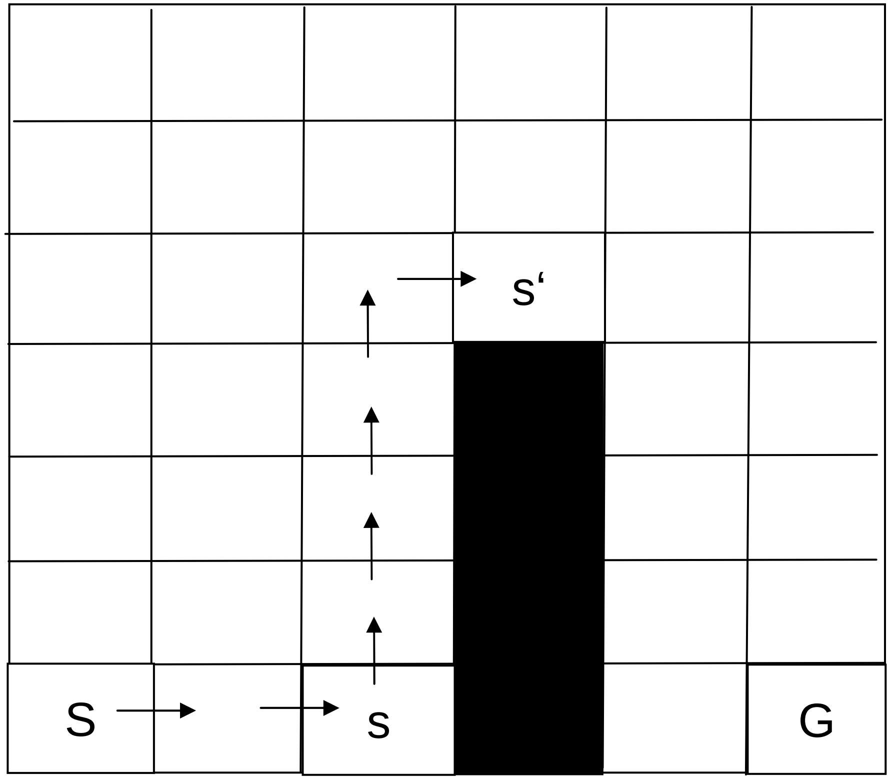
g(s‘)=7, h(s‘)=6, f(s‘)=13

g(s)=2, h(s)=3

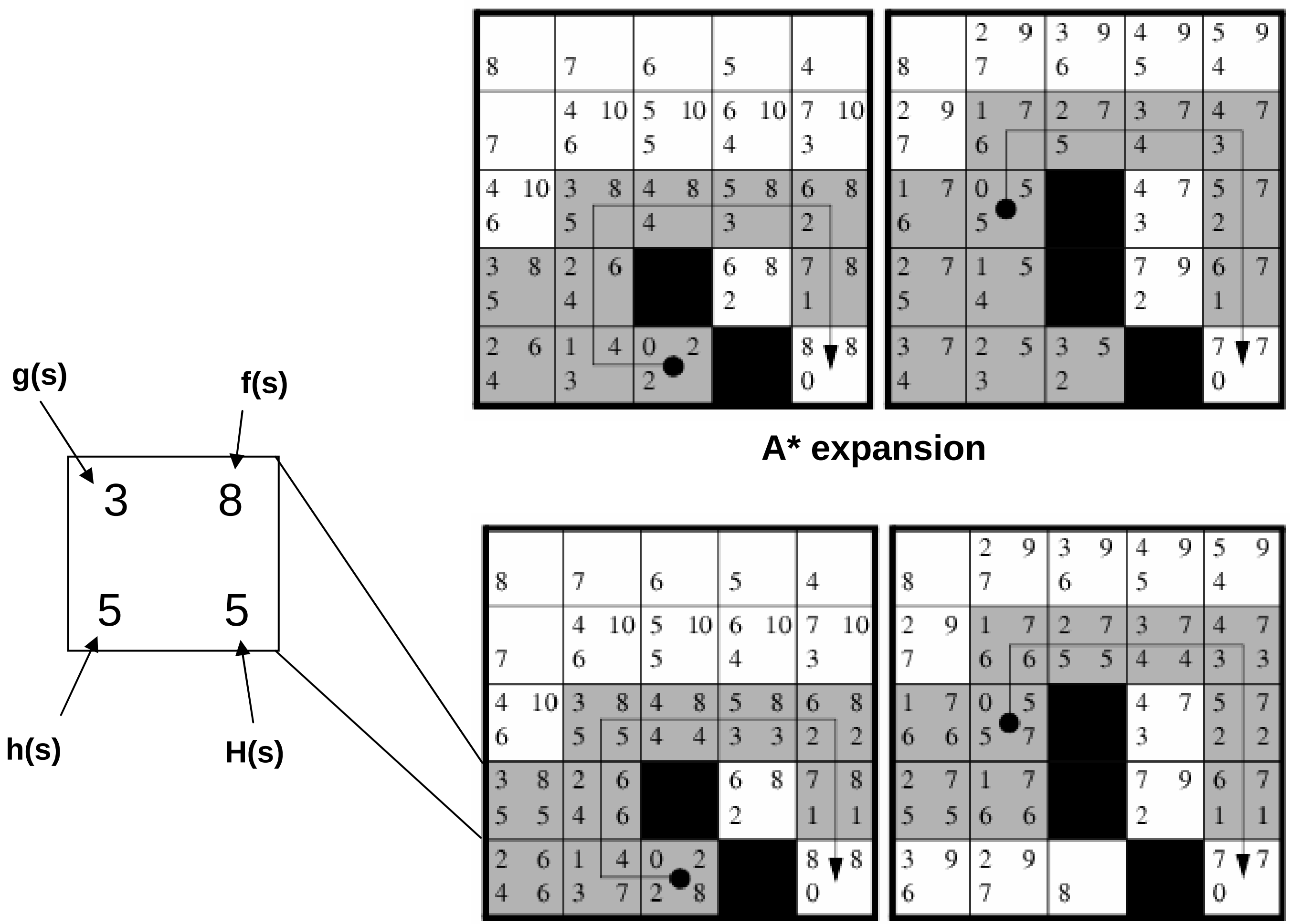
Update of each element in CLOSED list, e.g.:

H(s) = g(s‘) + h(s‘) – g(s)

H(s) = 7 + 6 - 2 = 11



Real-Time Adaptive A* (RTAA*)
A* vs. RTAA*



A* expansion

RTAA* expansion (inf. Lookahead)

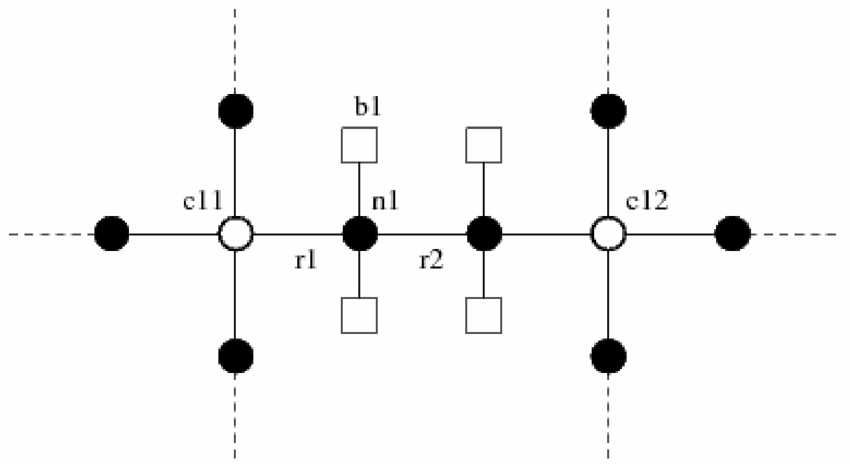
Case Study: ResQ Freiburg path planner

Requirements

- Rescue domain has some special features:
  - Interleaving between planning and execution is within large time cycles
  - Roads can be merged into "longroads"
- Planner is not used only for path finding, also for task assignment
  - For example, prefer high utility goals with low path costs
  - Hence, planner is frequently called for different goals
- Our decision: Dijkstra graph expansion on longroads

Case Study: ResQ Freiburg path planner
Longroads

- RoboCup Rescue maps consist of buildings, nodes, and roads.
  - Buildings are directly connected to nodes
  - Roads are inter-connected by crossings
- For efficient path planning, one can extract a graph of longroads that basically consists of road segments that are connected by crossings



Case Study: ResQ Freiburg path planner

Approach

- Reduction of street network to longroad network
- Caching of planning queries (useful if same queries are repeated)
- Each agent computes two Dijkstra graphs, one for each nearby longroad node
- Selection of optimal path by considering all 4 possible plans
- Dijkstra graphs are recomputed after each perception update (either via direct sensing or communication)
- Additional features:
  - Parameter for favoring unknown roads (for exploration)
  - Two more Dijkstra graphs for sampled time cost (allows time prediction)

Case Study: ResQ Freiburg path planner

Dijkstra‘s Algorithm (1)

Single Source Shortest Path, i.e. finds the shortest path from a single node to all other nodes

Worst case runtime $O(|E| \log |V|)$, assuming E>V, where E is the set of edges and V the set of vertices

- Requires efficient priority queue

Case Study: ResQ Freiburg path planner
Dijkstra‘s Algorithm (2)

Graph expansion

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:           // Initializations
        dist[v] := infinity               // Unknown distance function from source to v
        previous[v] := undefined          // Previous node in optimal path from source
    dist[source] := 0                     // Distance from source to source
    Q := the set of all nodes in Graph    // All nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                 // The main loop
        u := node in Q with smallest dist[]
        remove u from Q
        for each neighbor v of u:         // where v has not yet been removed from Q.
            alt := dist[u] + dist_between(u, v)       // be careful in 1st step - dist[u] is infinity yet
            if alt < dist[v]              // Relax (u,v)
                dist[v] := alt
                previous[v] := u
    return previous[]
```

Pseudo code taken from Wikipedia

Extracting path to target

```
S := empty sequence
u := target
while defined previous[u]
    insert u at the beginning of S
    u := previous[u]
```

Pseudo code taken from Wikipedia

Summary

- Before an agent can start searching for solutions, it must formulate a goal and then use that goal to formulate a problem.
- A problem consists of five parts: The state space, initial situation, actions, goal test, and path costs. A path from an initial state to a goal state is a solution.
- A general search algorithm can be used to solve any problem. Specific variants of the algorithm can use different search strategies.
- Search algorithms are judged on the basis of completeness, optimality, time complexity, and space complexity.
- Heuristics focus the search
- Best-first search expands the node with the highest worth (defined by any measure) first.
- With the minimization of the evaluated costs to the goal h we obtain a greedy search.
- The minimization of $f(n) = g(n) + h(n)$ combines uniform and greedy searches. When $h(n)$ is admissible, i.e., h^* is never overestimated, we obtain the A* search, which is complete and optimal.
- Online search provides method that are computationally more efficient when planning and plan execution are tightly coupled

Literature

- On my homepage:
  - A. Kleiner, M. Brenner, T. BrÃ¤uer, C. Dornhege, M. Göbelbecker, M. Luber, J. Prediger, J. Stückler, and B. Nebel **Successful Search and Rescue in Simulated Disaster Areas** *Robocup 2005: Robot Soccer World Cup IX* pp. 323-334, 2005
- Homepage of Tony Stentz:
  - A. Stentz **The focussed D* algorithm for real-time replanning** *Proc. of the Int. Join Conference on Artificial Intelligence*, p. 1652-1659, 1995.
- Homepage of Sven Koenig:
  - S. Koenig and X. Sun. **Comparing Real-Time and Incremental Heuristic Search for Real-Time Situated Agents** *Journal of Autonomous Agents and Multi-Agent Systems*, 2009
  - S. Koenig and M. Likhachev **Real-Time Adaptive A*** *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 281-288, 2006
  - S. Koenig and M. Likhachev. **Fast Replanning for Navigation in Unknown Terrain** *Transactions on Robotics*, 21, (3), 354-363, 2005.
- Harder to find, also explained in the AIMA book (2nd ed.):
  - R. Korf. **Real-time heuristic search**. *Artificial Intelligence*, 42(2-3):189-211, 1990.
- Demo search code in Java on the AIMA webpage *http://aima.cs.berkeley.edu/*