FUNDAMENTALS OF ARTIFICIAL INTELLIGENCE

Lecture 6: Search and planning

Ola Ringdahl

UMEÅ UNIVERSITY

SEARCH

R&N Chapter 3

UMEÅ UNIVERSITY

PROBLEM SOLVING

- Rational agents need to perform sequences of actions in order to achieve goals.
- Possible to use look-up table telling the agent what to do in every circumstance, but:
  - Difficult to build and all contingencies must be anticipated
- More general approach: agent has knowledge of the world and how its actions affect it
- This is the general task of **problem solving** and is typically performed by **searching** through an internally modelled space of world states

PROBLEM SOLVING TASK

- Given:
  - **An initial state** of the world
  - A set of possible actions or **operators** that can be performed.
  - A **goal test** that can be applied to a single state of the world to determine if it is a goal state.
- Find:
  - A **solution** in form of a **path** of states and operators that shows how to transform the initial state into one that satisfies the goal test.
  - The initial state and set of operators implicitly define a **state space** - *the set of all states reachable from the initial state by any sequence of actions*

MEASURING PERFORMANCE

- **Path cost**: a function that assigns a cost to a path, typically by summing the cost of the individual operators in the path. Usually want to find minimum cost solution (*optimal*).
- **Search cost**: The computational time and space (memory) required to find the solution.
- Usually a trade-off between path cost and search cost
  - Must find the best solution in the time that is available.

SEARCH STRATEGIES

- Easiest way to implement various search strategies is to maintain a *queue* of unexpanded search nodes.
- Different strategies result from different methods for inserting new nodes in the queue

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
     if nodes is empty then return failure
     node ← REMOVE-FRONT(nodes)
     if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
     nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

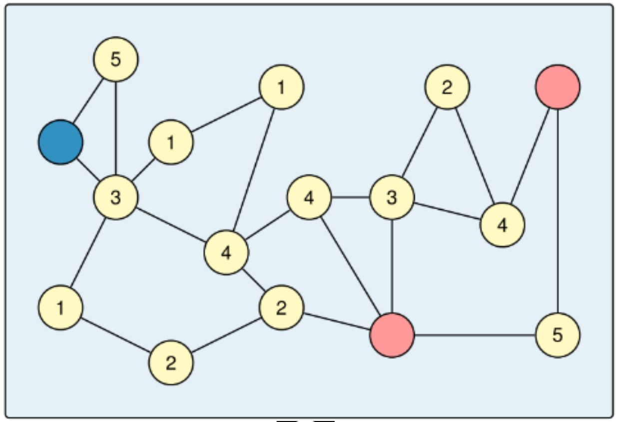
SEARCH STRATEGIES

Properties of a search strategy:

- **Completeness** - guaranteed to find a solution if there is one?
- **Optimality** - does it find the best solution?
- **Time complexity** - how long time to find a solution?
- **Space complexity** - how much memory is required during the search?

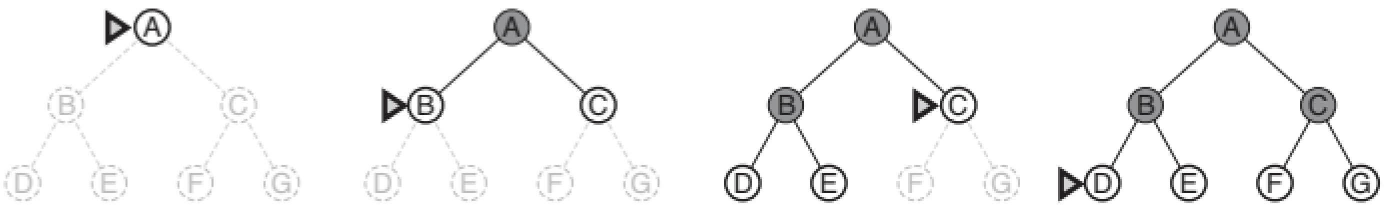
SEARCH ALGORITHMS

Uninformed search algorithms	Informed search algorithms
• Depth first • Breadth first • Uniform cost • Depth limited • Iterative deepening • Bidirectional	• Greedy best first • A* • Memory-bounded heuristic



BREADTH FIRST SEARCH

- Expands search nodes level by level, all nodes at level *d* are expanded before expanding nodes at level *d*+1



- Implemented by adding new nodes to the end of the queue (FIFO queue):

  GENERAL-SEARCH(problem, ENQUEUE-AT-END)

- **Complete** - eventually visits every node to a given depth
- **Optimal** - provided path cost is a nondecreasing function of the depth of the node (e.g. all actions have same cost) since nodes explored in depth order.

BREADTH FIRST COMPLEXITY

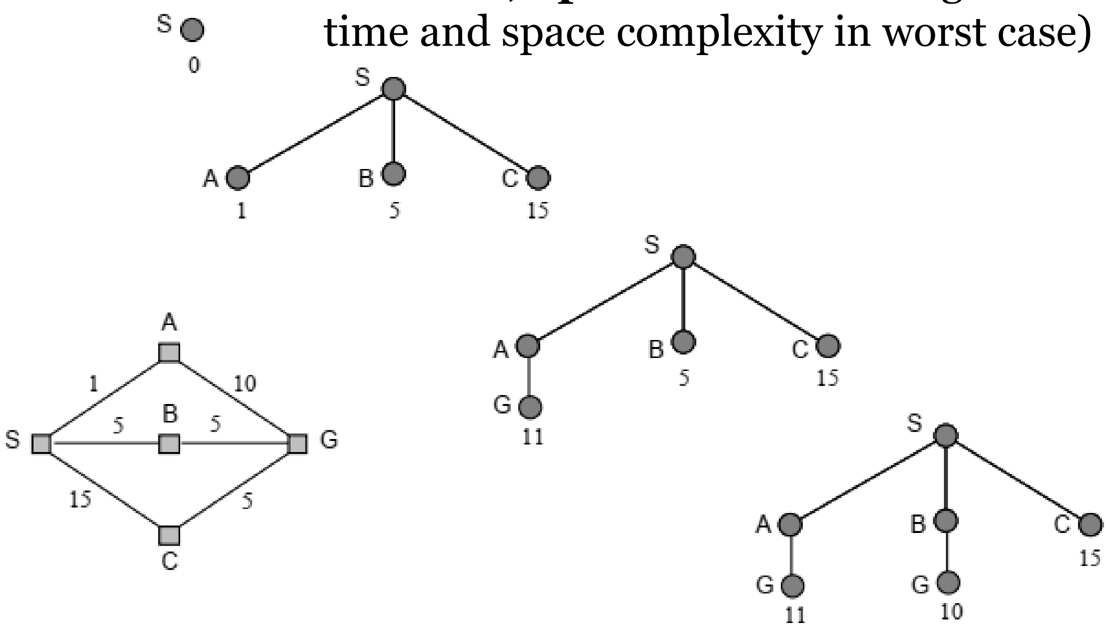
- Assume there are an average of b successors to each node, called the **branching factor**.
- To find a solution path of length d we must explore
$$1 + b + b^2 + b^3 + \cdots + b^d = O(bd)$$
- Need b^{d} nodes in memory to store leaves in the queue

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

UNIFORM COST SEARCH

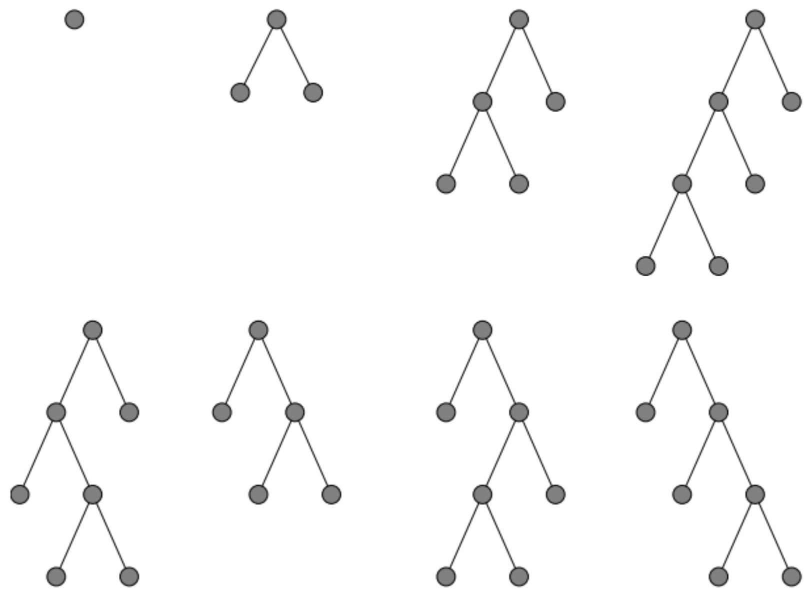
- Like breadth-first except **always expand node of lowest path cost** instead of least depth (i.e. sort new queue by path cost).
- Do not recognize goal until it is the least-cost node on the queue (i.e. when the goal node is selected for expansion).
  - Therefore, **optimal** with non-negative step costs (also means higher time and space complexity in worst case)



(a) (b)

DEPTH FIRST SEARCH

- Always expand node at deepest level of the tree, i.e. one of the most recently generated nodes. When hit a dead-end, backtrack to last choice.



- Implemented by adding new nodes to front of the queue (LIFO):
  GENERAL-SEARCH(problem, ENQUEUE-AT-FRONT)

UMEÅ UNIVERSITY

DEPTH-FIRST PROPERTIES

- **Not complete** - might get lost following infinite path.
- **Not optimal -** can find deeper solution before shallower ones explored.
- Worst case **time complexity $O(b^d)$** - need to explore entire tree.
  - May find a solution quickly before exploring the whole space.
- **Space complexity** is **$O(b^m)$** where m is maximum depth of the tree
  - The queue just contains a single path from the root to a leaf node along with remaining sibling nodes for each node along the path.
- Can add a **depth limit** l to prevent exploring nodes beyond a given depth.
  - Prevents infinite path. Still **incomplete** if no solution within depth limit

ITERATIVE DEEPENING

- Conduct a series of depth-limited searches, increasing depth-limit each time.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

- Most nodes are at the bottom level, so only adds constant time
- Depth-first:

$$1 + b + b^2 + \cdots + b^{d-2} + b^{d-1} + b^d$$

- Iterative deepening:

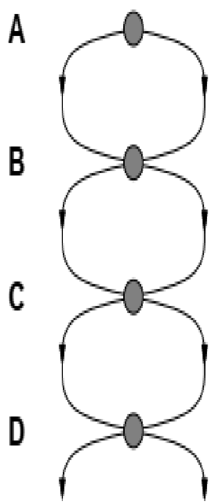
$$(d+1)1 + db + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

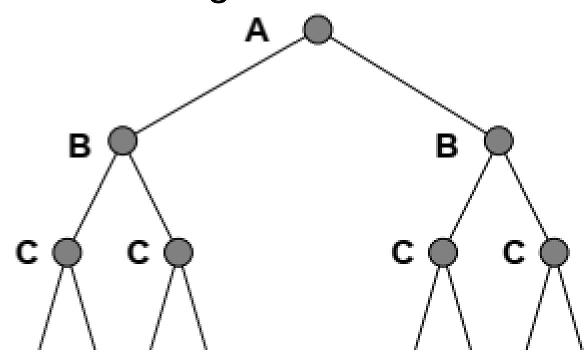
- **Time complexity** is still **O(*b*^{d})**
- **Space complexity O(*b*^{m})**

UMEÅ UNIVERSITY

AVOIDING REPEATED STATES

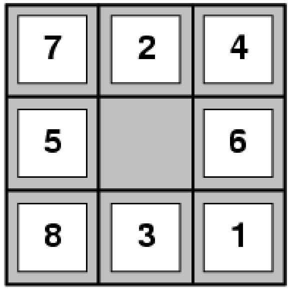
- Basic search methods may repeatedly search the same state if it can be reached via multiple paths.
- Three methods for reducing repeated work in order of effectiveness and computational overhead:
    - Do not follow self-loops (remove successors back to the same state).
    - Do no create paths with cycles (remove successors already on the path back to the root). O(*d*) overhead.
    - Do not generate any state that was already generated. Requires storing all generated states (O(b^d) space) and searching them (usually using a hash-table for efficiency).



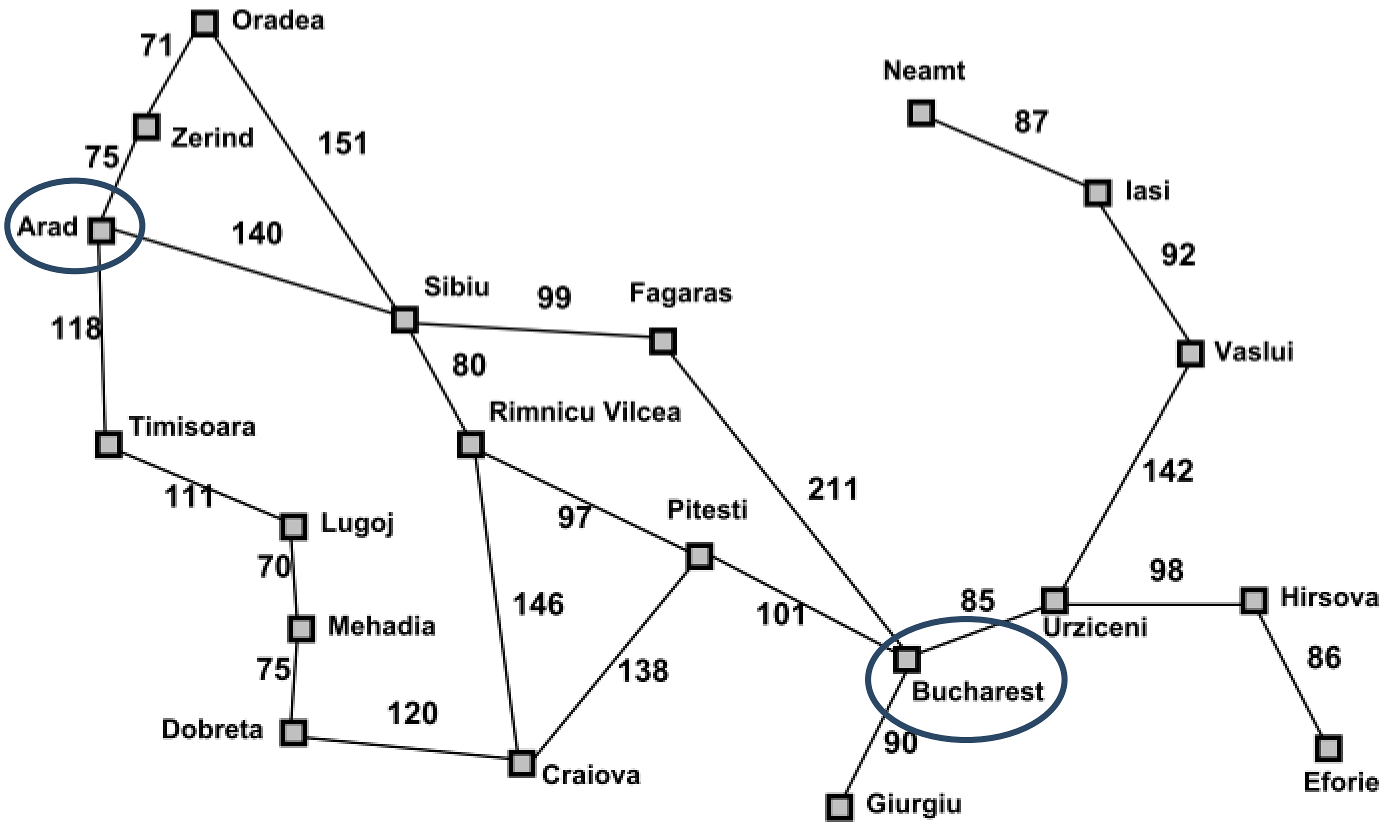


INFORMED SEARCH

- Use a heuristic function to estimate the cost of getting from the current state to a goal state.
- Reasonable heuristic functions can only be constructed using domain specific knowledge; i.e. they need to be informed.
- Examples:
  - When searching on a road map, use straight line distance.
  - Eight puzzle: no. of misplaced tiles
  - Sorting: how many pairs are in incorrect order?
- Note: these are not necessarily good heuristic functions.



ROMANIA WITH STEP COSTS IN KM



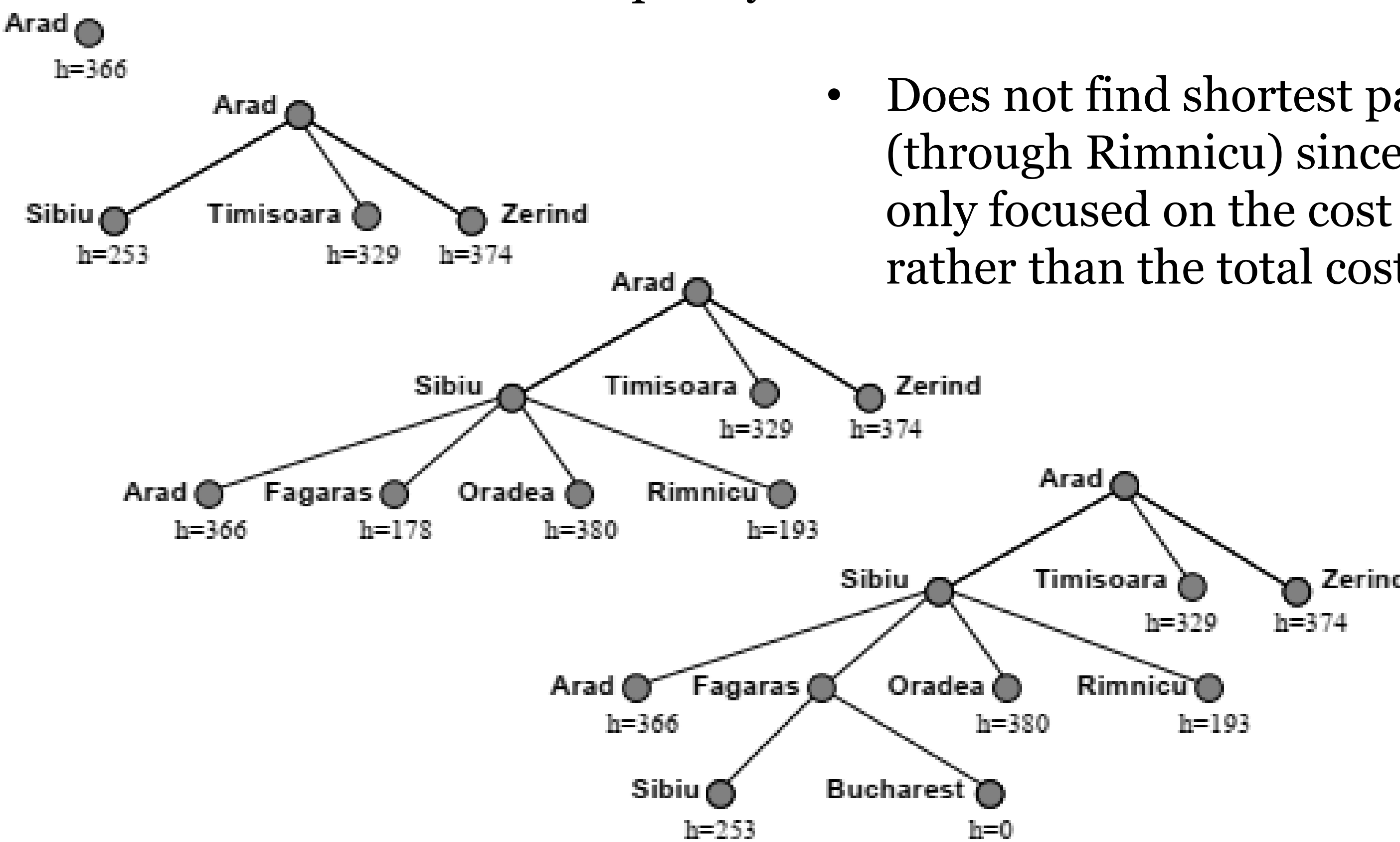
Straight−line distance to Bucharest

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

UMEÅ UNIVERSITY

GREEDY BEST-FIRST SEARCH

- Tries to expand the node closest to the goal – hoping that it leads to a solution quickly



- Does not find shortest path to goal (through Rimnicu) since it is only focused on the cost remaining rather than the total cost.

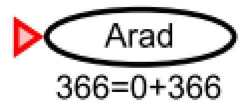
GREEDY BEST-FIRST PROPERTIES

- **Not complete -** may follow an infinite path.
  - Most reasonable heuristics will not cause this problem however.
- Worst case **time complexity** is still $\mathbf{O(b^m)}$ where m is the maximum depth.
  - A good heuristic will avoid the worst-case behavior for most problems.
- **Space-complexity** is also $\mathbf{O(b^m)}$ - must maintain a queue of all unexpanded states

A* SEARCH

- A* selects which node to expand next by using an evaluation function f.
- This function is a combination of the cost of getting to the node and the heuristic evaluation of the node: $f(n) = g(n) + h(n)$
- The value $f(n)$ is the *estimated* cost of the cheapest solution that goes through node n.
- Note: A* works for any search problem, but the heuristic function h is problem specific.

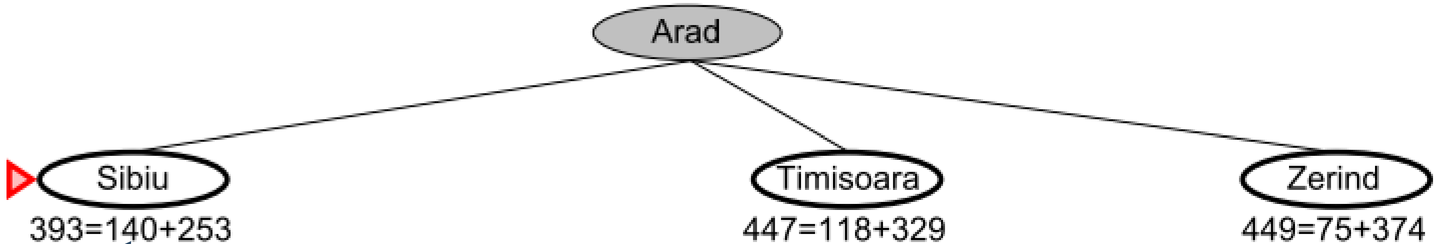
A* SEARCH EXAMPLE



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160

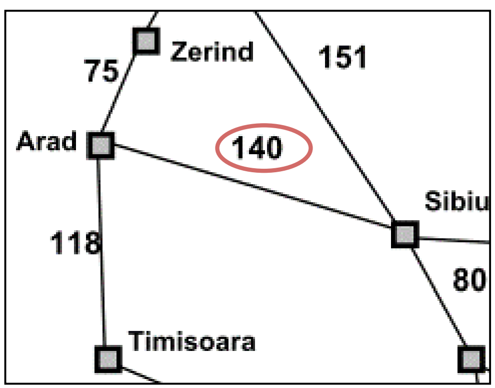
A* SEARCH EXAMPLE



$g(n)$ $h(n)$

Actual distance Arad - Sibu

Straight line distance Sibu - Bucharest

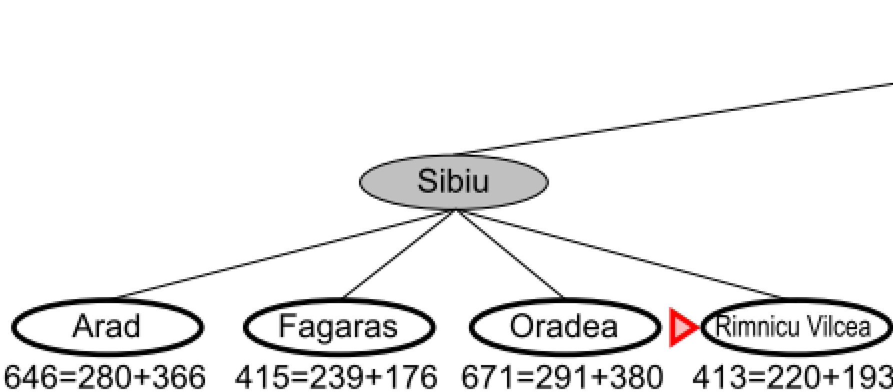


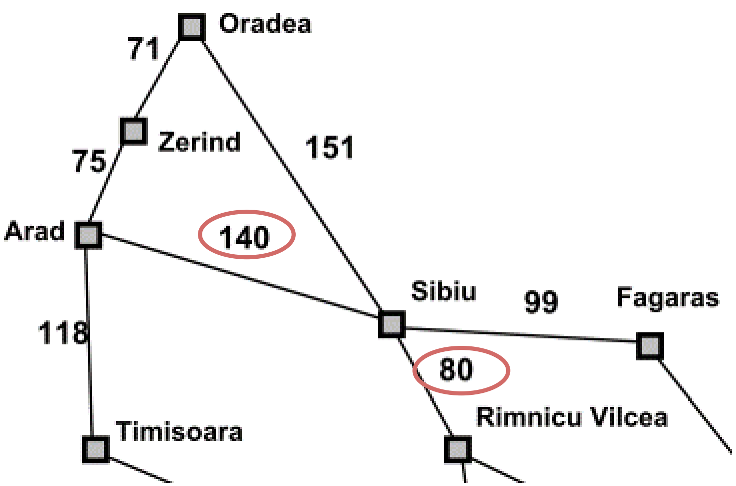
Straight−line distance to Bucharest

Sibiu	253

UMEÅ UNIVERSITY

A* SEARCH EXAMPLE

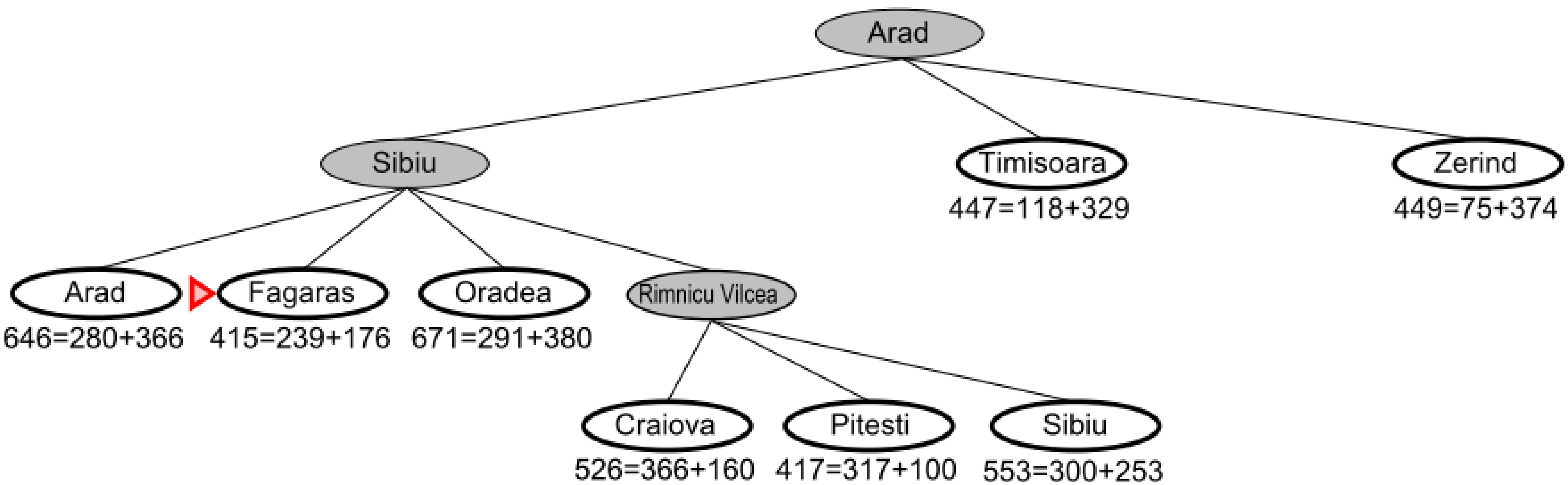




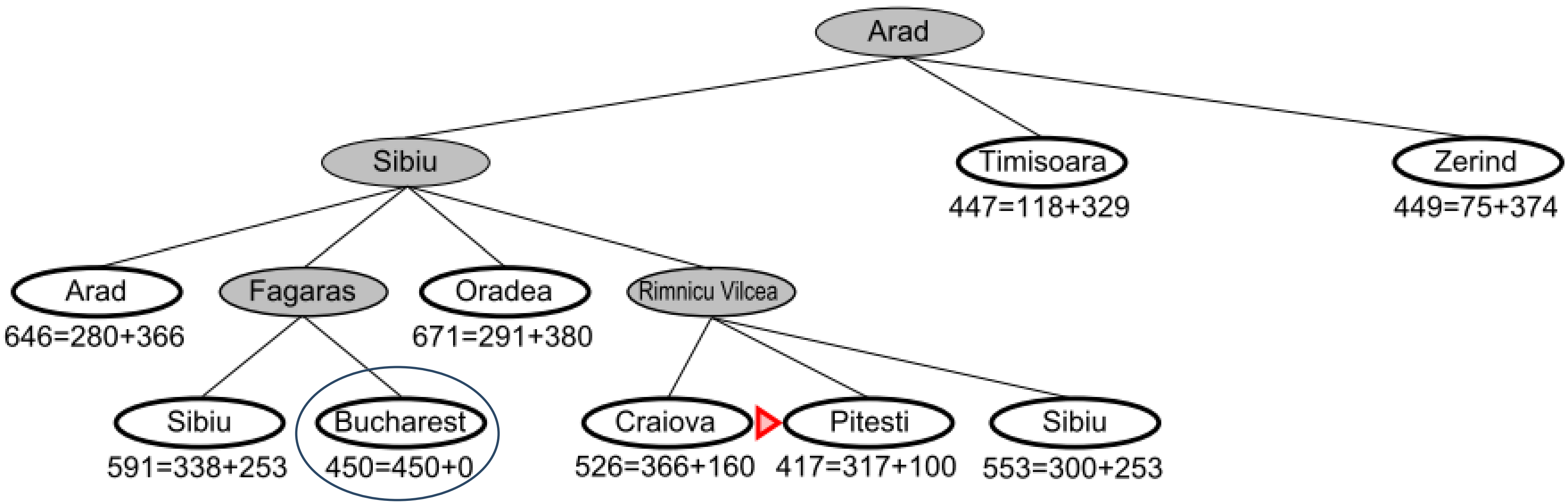
Straight−line distance
to Bucharest

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

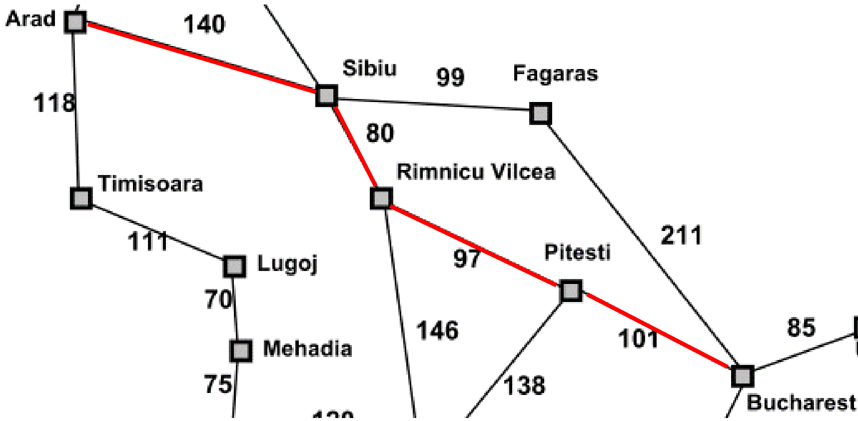
A* SEARCH EXAMPLE



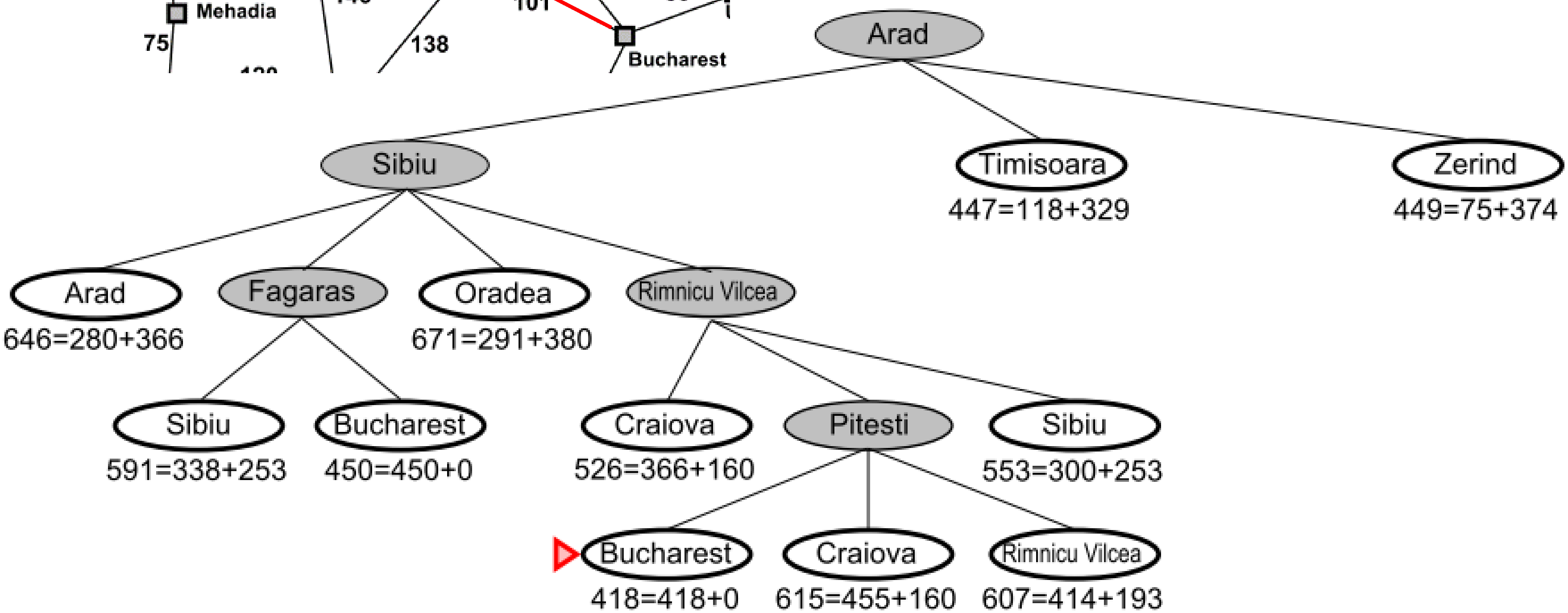
A* SEARCH EXAMPLE



Note that we reached Bucharest, but another path seems to be shorter, so we must check it out!



CH EXAMPLE



No path can be shorter than this!

A* OPTIMALITY

- A* tree search is optimal with any admissible heuristic.
  - Admissible = **never overestimates the cost**
- A* graph search is optimal with any consistent heuristic.
  - Consistent = stronger requirement than admissible (≈ taking a detour cannot be cheaper)
  - Most admissible heuristics are also consistent
- A* is optimally efficient for any consistent heuristic function h.
  - I.e. no other optimal strategy expands fewer nodes than A* when using the same h.
- A major drawback with A* (and other graph-based search algs) is that it runs out of memory for large problems
  - Generally runs out of memory before running out of time.

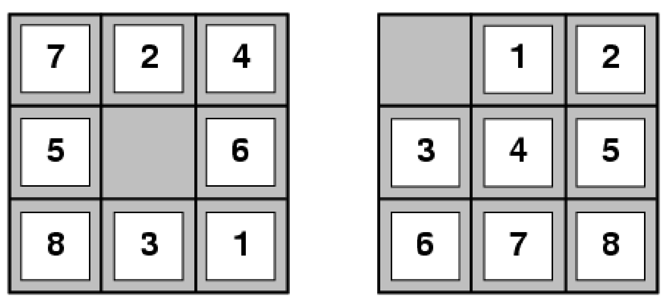
A* SPEED VS. ACCURACY

- If $h(n) = 0$ for all nodes n, A* is the same as Uniform Cost Search. It will always find an optimal path, but it may take a while.
  - ➢ Accurate
- If $h(n)$ is the exact cost of the cheapest path from n to a goal state, A* only searches the optimal path.
  - ➢ Fast and accurate (but unrealistic)
- If $h(n)$ overestimates the cost of getting to a goal state, A* will be fast, but will not always find the shortest path.
  - ➢ Fast, but **not** accurate
- We see that the performance of a heuristic search alg. depends on the quality of $h(n)$

A* PERFORMANCE

- Typical search costs (average number of nodes expanded) for 8-puzzle:
  - h_1: Number of tiles out of place
  - h_2: Manhattan distance (i.e., no. of squares from the desired location of each tile)

Depth	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6
12	3 644 035	227	73
24	-	39 135	1 641



Start State Goal State

SUMMARY SEARCH ALGORITHMS

- Implemented by maintaining a queue of unexpanded search nodes.
- Can be informed or uniformed
  - Informed use heuristics. Most known: A*
- Judged based on **completeness**, **optimality**, **time complexity**, and **space complexity**.
  - Complexity depends on branching factor *b* and the depth *d*
- Performance is a trade-off between path cost and search cost
- Performance of heuristic search algorithms depends on the quality of the heuristic

COMPARISON TABLE

	BFS	DFS	IDS	A*
Complete?	Yes	No	Yes	Yes
Optimal?	Yes[1]	No	Yes[1]	Yes[2]
Time	O(bd)	$O(b^d)$	$O(b^d)$	$O(b^d)$
Space	$O(b^d)$	$O(b^m)$	$O(b^m)$	$O(b^d)$

[1]if all actions have same cost
[2]with admissible heuristic.

Note! Time complexity of A* depends on the heuristics

CLASSICAL PLANNING
R&N CHAPTER 10

What is planning?

How to represent plans

Searching for plans

PLANNING

- **Planning** - devising a plan of action to achieve one's goals - is a central part of AI
  - Used for large logistics problems, operational planning, robotics, scheduling etc.
  - Bi-annual planning competition
  - Several international Conferences on Planning
- So far, we have looked at two ways to do planning:
  - search-based problem-solving agent (Ch. 3) – needs domain-depended heuristics
  - hybrid logical agent (Ch. 7) – suffer from combinatorial explosion for bigger problems
- Now we will combine the two to handle larger problems

CLASSICAL PLANNING

- In FOL we need to specify what changes *and* what stays the same as the result of an action -> the **frame problem**
- Classical planning focuses on problems where most things stays the same
  - We need another representation
  - Solution: **PDDL** - Planning Domain Definition Language, a subset of FOL

PDDL

- PDDL is derived from the STRIPS planning language.
  - More on STRIPS in upcoming lecture
- Contains:
  - Initial state and goal state.
  - A set of *Actions*(*s*) in terms of preconditions and effects.
  - **Closed world assumption**: Unmentioned state variables are assumed false.

$$\begin{aligned}
&Action(Fly(P_1, SFO, JFK),\\
&\quad \text{PRECOND:} At(P_1, SFO) \land Plane(P_1) \land Airport(SFO) \land Airport(JFK)\\
&\quad \text{EFFECT:} \neg At(P_1, SFO) \land At(P_1, JFK))
\end{aligned}$$

EXAMPLE: AIR CARGO TRANSPORT

- A classical transportation problem: Loading and unloading cargo and flying between different airports.

Actions: Load(cargo, plane, airport), Unload(cargo, plane, airport), Fly(plane, airport, airport)

Predicates: In(cargo, plane), At(cargo ∨ plane, airport)

<u>Example solution:</u>

Load(C1, P1, SFO), Fly(P1, SFO, JFK), Unload(C1, P1, JFK),

Load(C2, P2, JFK), Fly(P2, JFK, SFO), Unload(C2, P2, SFO).

PDDL

- Intended to be a standard for describing a certain *planning domain* (not a specific plan)
- Restricted language -> efficient algorithm
- Several implementations of PDDL exists
- Possible to construct software for very advanced plans, with many preconditions and effects for different actions
    - The designers of the PDDL solver does not need to know the domain in which the software is going to be used
    - Companies that need planning software can define their planning domains in PDDL, without aligning too hard to a specific implementation

STATE SPACE SEARCH

- Planning can be done by searching the sate space (defined by PDDL for example)
- Forward (progression):
  - state-space search considers actions that are *applicable* (could be *next* part of the plan leading up to the current goal state)
- Backward (regression):
  - state-space search considers actions that are *relevant* (could be the *last* part of the plan)
- Neither of them is efficient without good heuristics!
  - Can be derived automatically

SUMMARY PLANNING

- PDDL is a language for representing (simple) planning problems
  - Builds on FOL but more restricted
- State space search (forward or backward) can be used to search for a plan
- Real-world planning and scheduling (e.g. operations of spacecrafts, factories, and military campaigns) is more complex
  - Both the representation language and the way the planner interacts with the environment needs to be extended
  - Beyond the scope of this course

UMEÅ UNIVERSITY