SOLVING PROBLEMS BY SEARCHING

This chapter describes one kind of goal-based agent called a **problem-solving agent.**

PROBLEM-SOLVING AGENTS

We will consider the problem of designing goal-based agents in *fully observable*, *deterministic*, *discrete*, *known* environments.

- Under these assumptions, the solution in any problem is a, *fixed* sequence of actions.
- The process of looking for a sequence of actions that reaches the goal is called *search*.
- A search algorithm takes a problem as input and returns a solution in the form of an action sequence.
- Notice that while the agent is executing the solution sequence it ignores its percepts when choosing an action because it knows in advance what they will be (open-loop system).

A problem can be defined formally by five components:

1- Initial state.
2- Set of actions.
3- Transition model (the result of each action)

- **State space** = initial state + set of actions + transition model.
- State space can be formed by **graph** in which the nodes are states and the links between nodes are actions.
- A **path** in the slate space is a sequence of states connected by a sequence of actions.

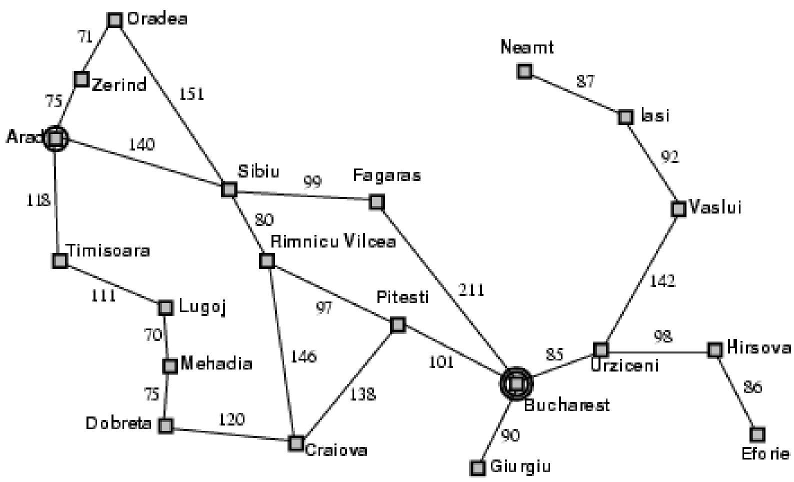
4- The **goal test:** determines which state is a goal.
5- **A path cost function**: that assigns a numeric cost to each path.

- A **solution** to a problem is an action sequence that leads from the initial state to a goal state.
- An **optimal solution** has the lowest path cost among all solutions.

Problems examples:

1- Rout-finding problem: travel from *Arad* city to *Bucharest* city with a minimum cost.

Initial state
Arad city
Actions
Go from one city to another
Transition model
If you go from city A to city B, you end up in city B
Goal state
Bucharest city
Path cost
Sum of edge costs



2- Puzzles

States

Locations of tiles

8-puzzle: 8!= 181,440 states

15-puzzle:15!= 1.3 trillion states

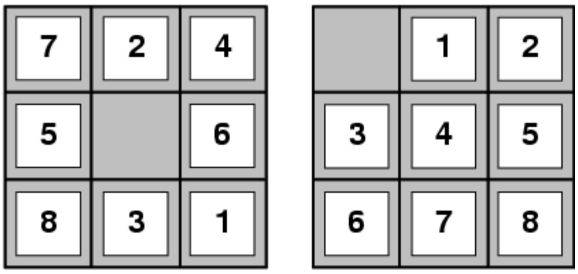
24-puzzle:24!= 10^{25} states

Actions

Move blank: left, right, up, down

Path cost

1 per move



Start State   Goal State

3- The **traveling salesperson problem (TSP)** is a touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour.

Tree Search

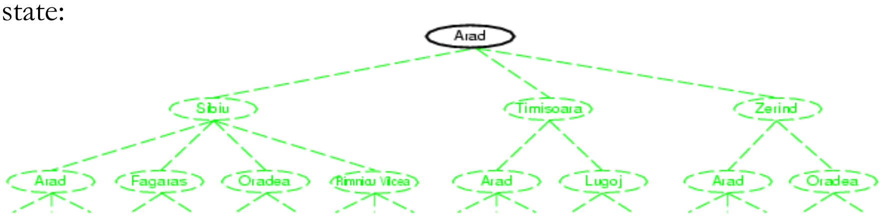
The possible action sequences starting at the initial state form a search **tree** with the initial state at the root; the branches are actions and the **nodes** correspond to states in the state space of the problem.

Tree search Algorithm

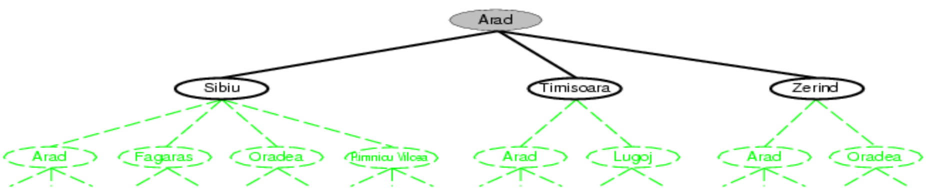
- Let's begin at the start node and **expand** it by making a list of all possible successor states.
- Maintain a **fringe** or a list of unexpanded states.
- At each step, pick a state from the fringe to expand.
- Keep going until you reach the goal state.
- Try to expand as few states as possible.

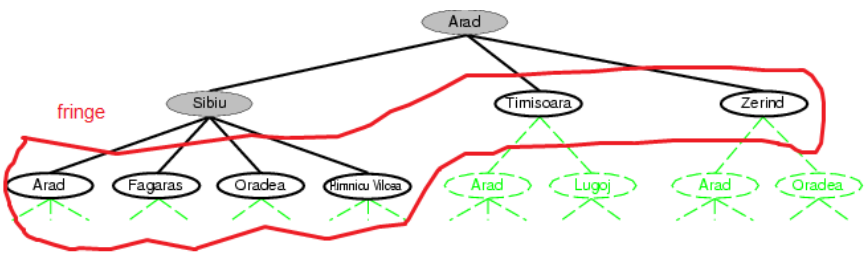
A **search strategy** is defined by picking the order of node expansion.

initial state:



expansion:





Search Algorithms evaluation

We can evaluate an algorithm's performance in four ways:

1- Completeness: Is the algorithm guaranteed to find a solution when there is one?
2- Optimality: Does the strategy find the optimal solution?
3- Time complexity: How long does it take to find a solution?
4- Space complexity: How much memory is needed to perform the search?

Complexity is expressed in terms of three quantities:

- The **branching factor (b):** maximum number of successors of any node.

- The **depth (d):** the number of steps along the path from the root to the goal.

- The maximum length **(m)** of any path in the state space.

Search Strategies:

Search strategies can be classified into:

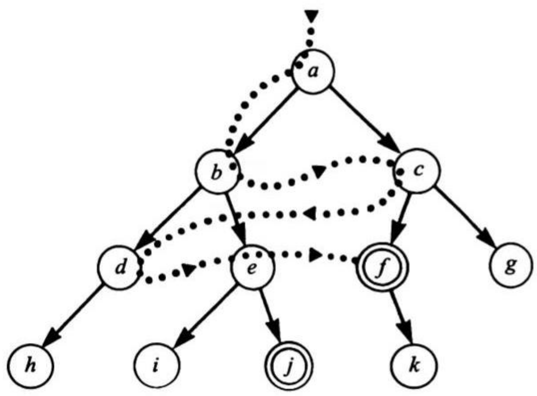
- **Uninformed** (*blind search*) and
- **Informed** (*heuristic*) search.

The first one has no additional information about states beyond what provided in problem definition.

Uninformed search strategies

1- Breadth-first search

Algorithm: Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on.



Implementation: by using a FIFO queue for the fringes. Thus, new nodes go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.

Properties:
- Complete if *b* and *d* are finites.
- Optimal if all steps have the same cost.
- Time and space complexity is $O(d^b)$ (bad feature).

2- Uniform-cost search (*Dijkstra's Algorithm*)

Algorithm: expands the node *n* with the *lowest path cost g(n)*.

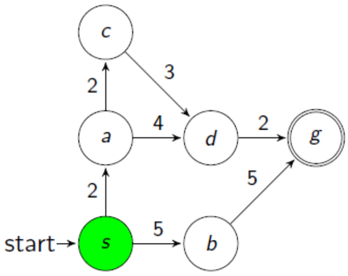
* *g(n)* is the cost from the root to the node *n*.

* this algorithm equals breadth-first search if *g(n)=1* for all *n*.

Implementation: fringe is a queue ordered by path cost (priority queue).

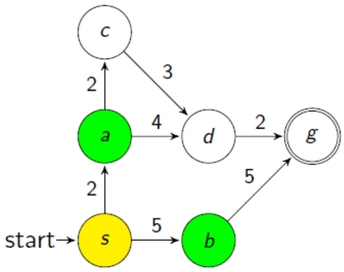
Q:

path	cost
$\langle s \rangle$	0



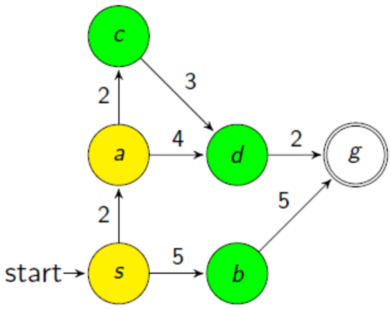
Q:

path	cost
$\langle a, s \rangle$	2
$\langle b, s \rangle$	5



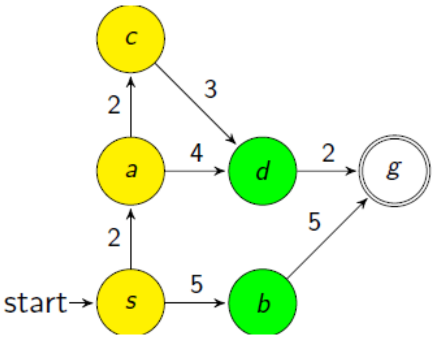
Q:

state	cost
$\langle c, a, s \rangle$	4
$\langle b, s \rangle$	5
$\langle d, a, s \rangle$	6



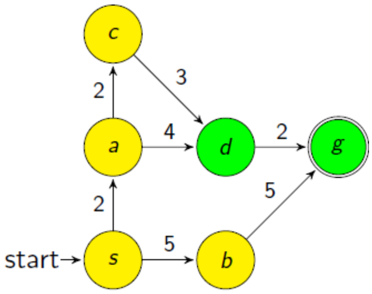
Q:

state	cost
$\langle b, s \rangle$	5
$\langle d, a, s \rangle$	6
$\langle d, c, a, s \rangle$	7



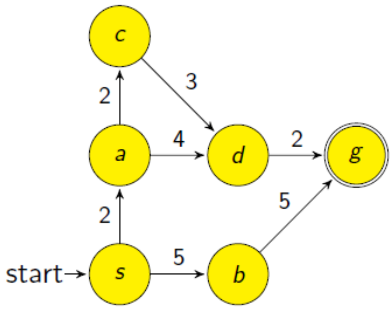
Q:

state	cost
$\langle d, a, s \rangle$	6
$\langle d, c, a, s \rangle$	7
$\langle g, b, s \rangle$	10



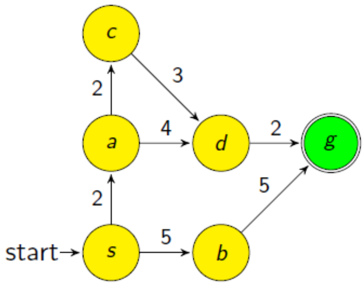
Q:

state	cost
$\langle d, c, a, s \rangle$	7
$\langle g, d, a, s \rangle$	8
$\langle g, b, s \rangle$	10



Q:

state	cost
$\langle g, d, a, s \rangle$	8
$\langle g, d, c, a, s \rangle$	9
$\langle g, b, s \rangle$	10

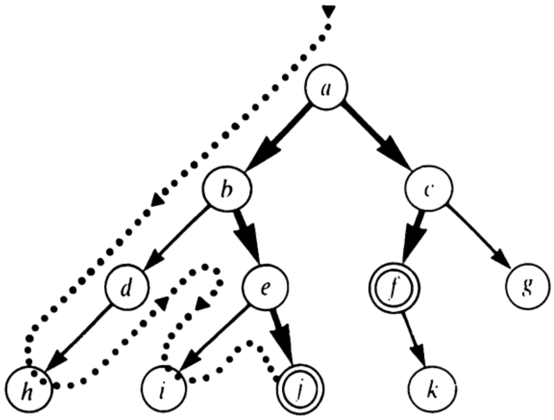


Properties:

- Complete: is guaranteed provided the cost of every step exceeds some small positive constant *e*.
- Optimal: yes.
- Complexity: Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d. Instead, let C^* be the cost of the optimal solution, and assume that every action costs at least e. Then the algorithm's worst-case time and space complexity is $O(b^{C^*/e})$

3- Depth-first search

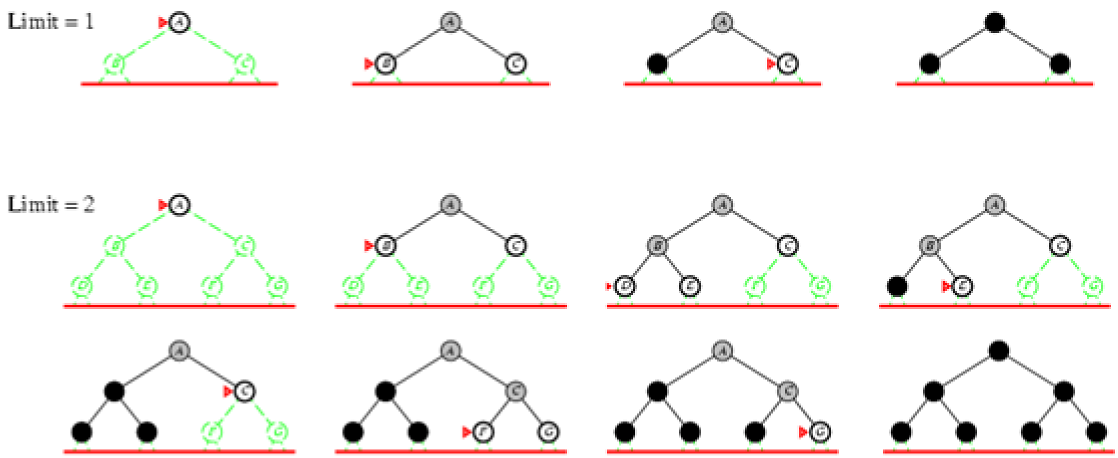
Algorithm: expands the ***deepest*** node in the current fringe of the search tree.
Implementation: uses a LIFO stack.

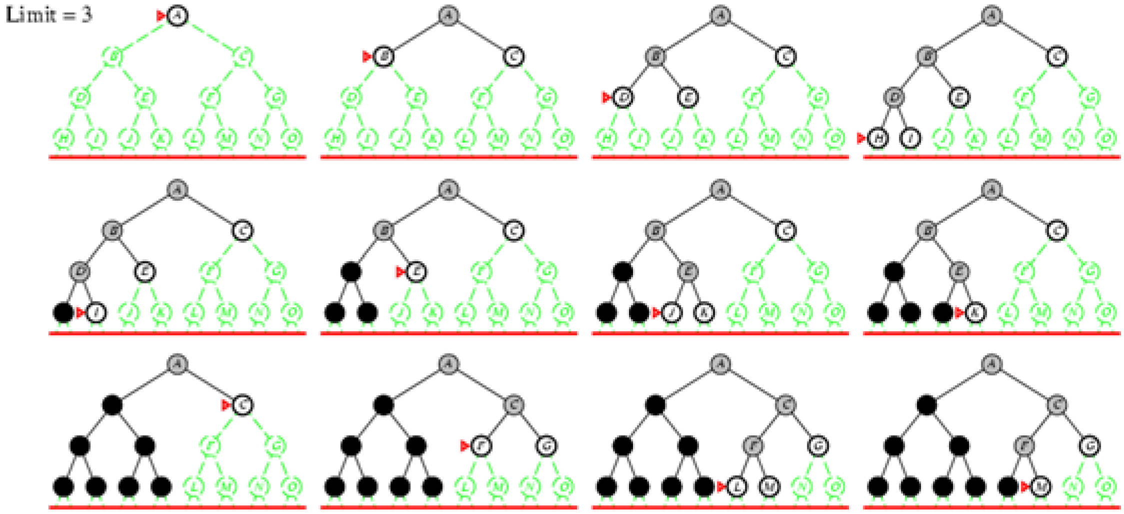


Properties:
- Complete: Fails in infinite-depth spaces
- Optimal: No – returns the first solution it finds
- Time: Could be the time to reach a solution at maximum depth *m*: $O(b^m)$. Terrible if *m* is much larger than *d*
- Space: *O*(*bm*), i.e., linear space! (good feature).

4- **Iterative deepening depth-first search**

Algorithm: call depth-first search but it gradually increasing the deep limit—first 0, then 1, then 2, and so on—until a goal is found.
* Iterative deepening combines the benefits of depth-first and breadth-first search.
* Iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.





Properties:

- Complete: Yes

- Optimal: Yes, if step cost = 1

- Time: $db + (d\text{-}1)\,b^2 + \ldots + (1)b = O(bd)$.

- Space: $O(bd)$.

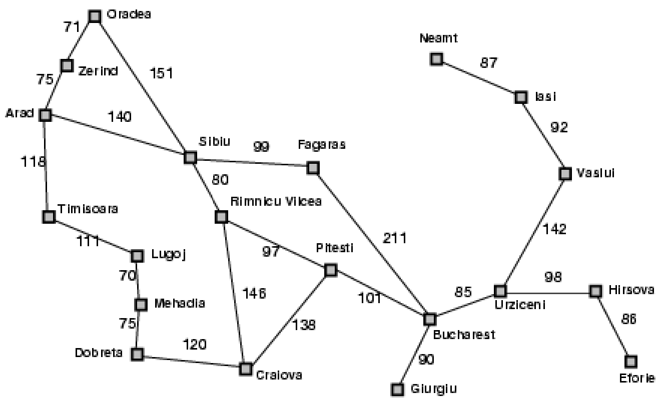
Informal search strategies

Informed search strategy uses knowledge beyond the definition of the problem itself. It can find solutions more efficiently than can an uninformed strategy. Such a strategies uses a *heuristic* function, ***h(n)*** to select the next node. ***h(n)*** is the *estimated* cost of the cheapest path from the state at node *n* to a goal state.

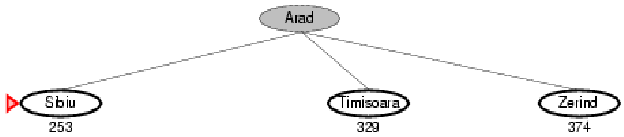
1- Greedy best-first search

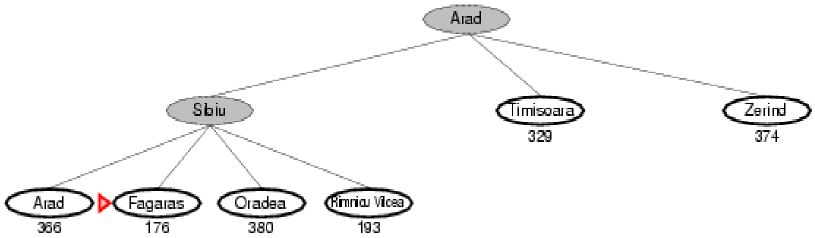
Algorithm: expand the node that has the lowest value of the heuristic function ***h(n)***.
* it is not optimal but efficient search.

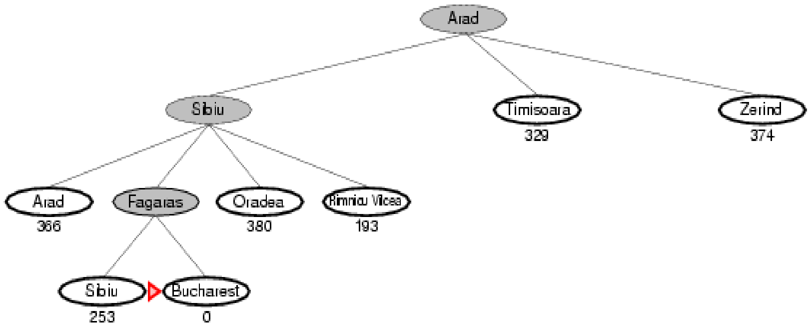
In the following example, the heuristic functions ***h(n)*** is ***straight line distance*** between the node *n* and the goal.



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374







Properties:

- Complete: No. Consider the problem of getting from *Iasi* to *Fagaras*. The heuristic suggests that *Neamt* be expanded first because it is closest to *Fagaras*, but it is a dead end.

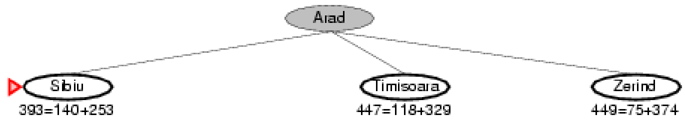
- Optimal: No. The path via *Sibiu* and *Fagaras* to *Bucharest* is 32 kilometers longer than the path through *Rimnicu Vilcea* and *Pitesti*.
- Time:
  * Worst case: $O(b^m)$
  * Best case: $O(bd)$, If $h(n)$ is 100% accurate
- Space:
  * Worst case: $O(b^m)$

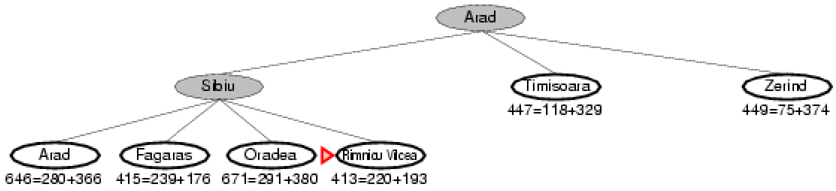
2- A* Search

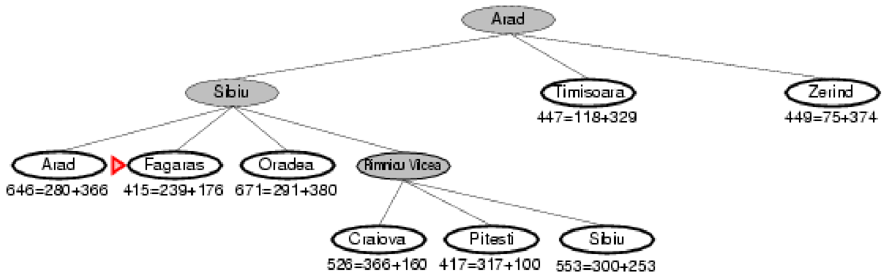
- *Algorithm:* expand the node that has the lowest value of the evaluation function *f(n)*: $f(n) = g(n) + h(n)$

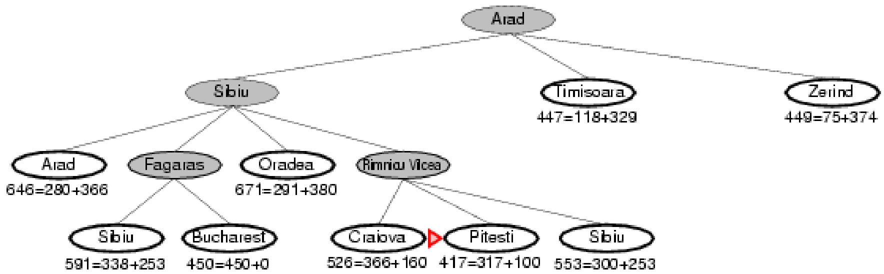
Where, g(n): cost so far to reach *n* (path cost), *h(n):* estimated cost from *n* to goal (heuristic).

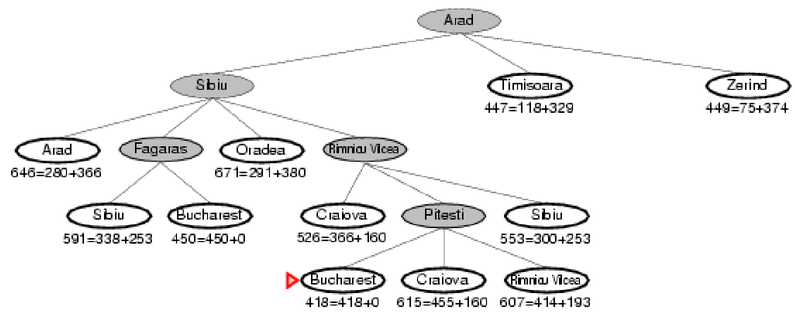
Example:











Conditions for optimality: Admissibility and consistency

We have one of the two conditions to make A* optimal:

1- A heuristic $h(n)$ should be **admissible** for every node *n*, $h(n) \leq h^*(n)$, where $h^*(n)$ is the ***true*** cost to reach the goal state from *n.*

- Example: straight line distance never overestimates the actual road distance.

2- Heuristic $h(n)$ should be **consistent**, for every (x, y) nodes, $h(x) \leq h(y)+d(x,y)$, where $d(x,y)$ is the step cost between *x* and *y*. (Stronger condition)

For example: $h(Sibiu) < h(\text{Rimnicu Vikea}) + d(\text{Sibiu, Rimnicu Vikea})$

= 253 < 193+80

= 253 < 273

* If h is a consistent heuristics, then $f = g + h$ is non-decreasing along paths.
Hence, the values of *f* on the sequence of nodes expanded by A* is non-decreasing: the first path found to a node is also the optimal path) → no need to compare costs!

Properties:

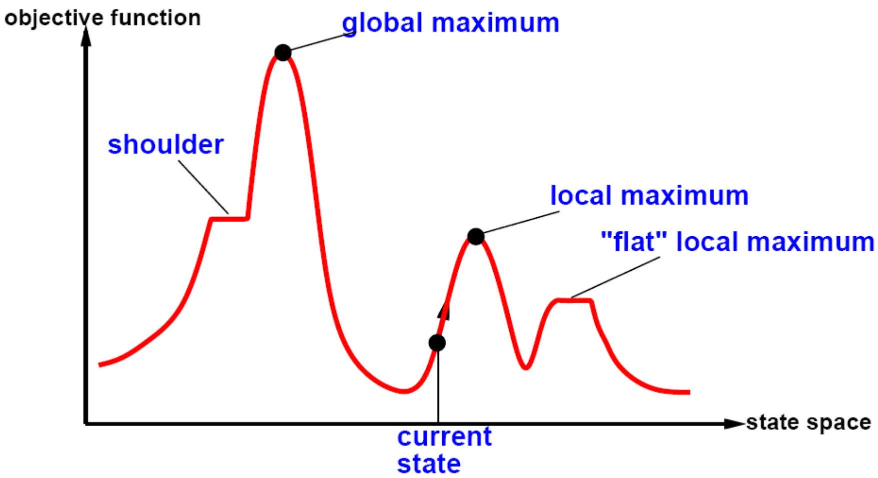
- Complete: Yes – unless there are infinitely many nodes with $f(n) \leq C^*$
- Optimal: Yes
- Time: Number of nodes for which $f(n) \leq C^*$ (exponential)
- Space: Exponential

Local Search Algorithms and Optimization problem

Local search algorithms operate using a single *current node* (rather than multiple paths) and generally move only to neighbors of that node. In such algorithms we don't have a start state, don't care about the path to a solution.

Local search algorithms are useful for solving pure **optimization problems**, in which the aim is to find the best state according to an **objective function**.

Objective function tells us about the quality of a possible solution, and we want to find a good solution by minimizing or maximizing the value of this function.



Hill-climbing search

Idea: keep a single "current" state and try to locally improve it.

Algorithm:

- Initialize *current* to starting state

- Loop:

  – Let *next* = highest-valued successor of *current*
  – If value(*next*) < value(*current*) return *current*
  – Else let *current* = *next*

Example: *n*-queens problem

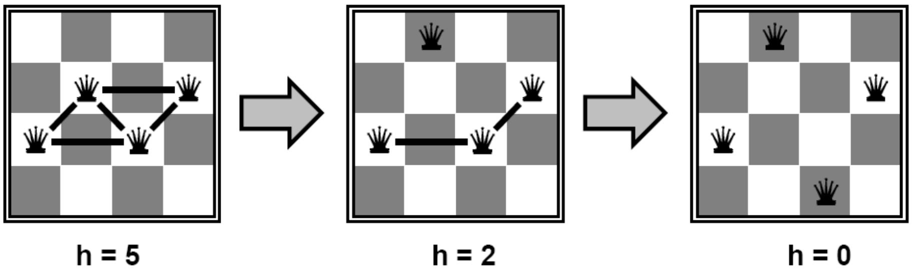
- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal

- *State space*: all possible *n*-queen configurations

- *Objective function*: number of pairwise conflicts

What's a possible local improvement strategy?

– Move one queen within its column to reduce conflicts.



Disadvantage:
 * Hill-climbing algorithms that reach the vicinity of a local maximum will reaches a point at which no progress is being made.
* A hill-climbing search might get lost on the flat local maximum or shoulder areas.

Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances.

Random-restart hill climbing conducts a series of hill-climbing searches from randomly generated initial states until a goal is found.