Artificial Intelligence (AI)

CCS-3880 – 3rd Semester 2023

CO3: Heuristic Search

Dr. Abdullah Alshanqiti

Heuristic Search

CO3: Design problem solving algorithms using heuristic searches.
CO4: Investigate the performance of local, global, and optimized search methods.

Outline - Heuristic Search

- Greedy Best First Search,
- A* Algorithm
- Local Search Algorithms
  - Hill-climbing search
  - Gradient Descent
  - Simulated annealing (suited for either local or global search)
- Global Search Algorithms (it will be covered next week)
  - Genetic Algorithm

Heuristic Search Strategies

Informed (Heuristic) search strategy:

- One that uses problem-specific knowledge beyond the definition of the problem itself
- It can find solutions more efficiently than can an uninformed strategy.

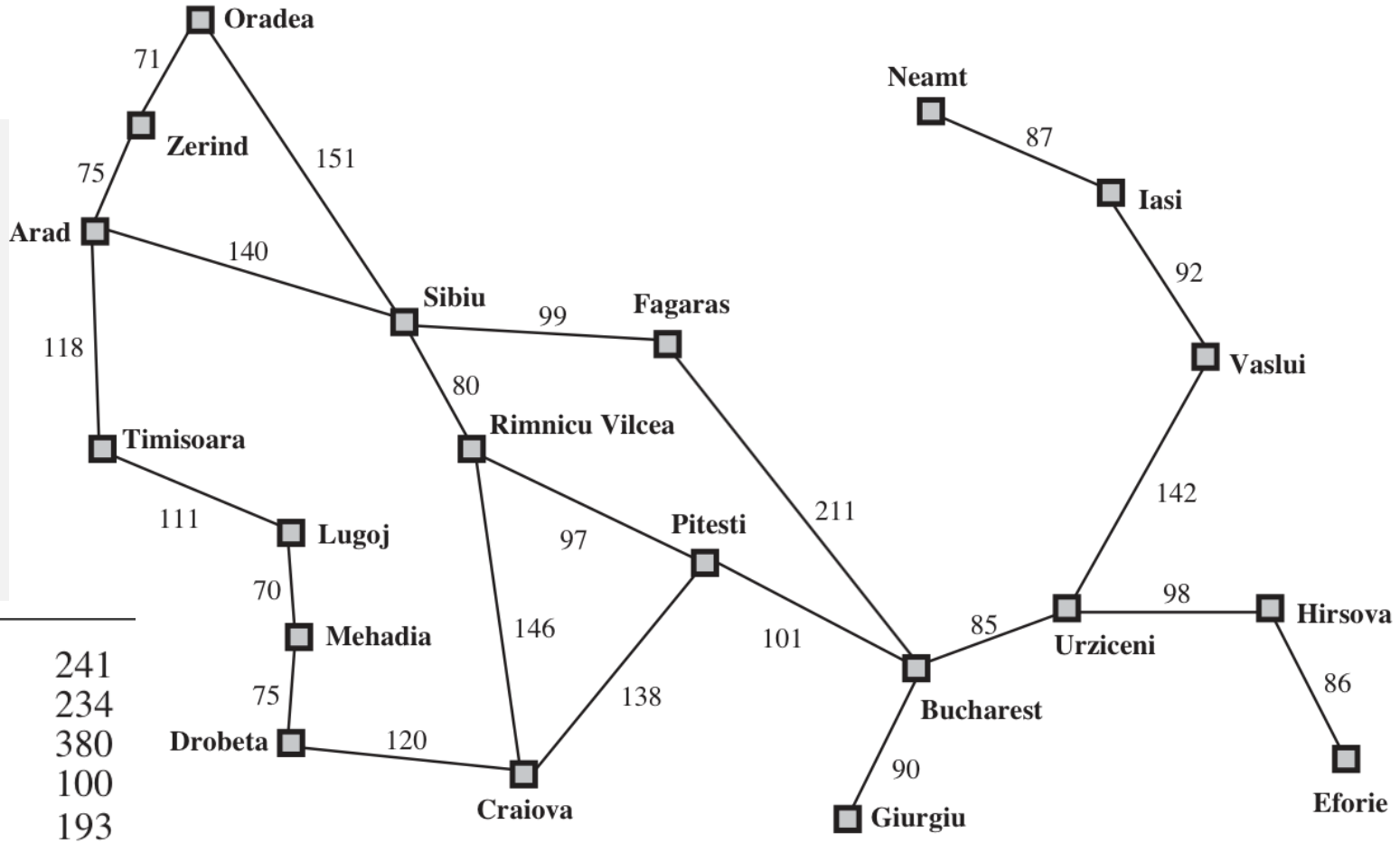
Most algorithms include a heuristic function, denoted by h(n),

where h(n) = estimated cost of the cheapest path from the state at node *n* to a goal state.

For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the **Straight-Line Distance (SDL)** function.

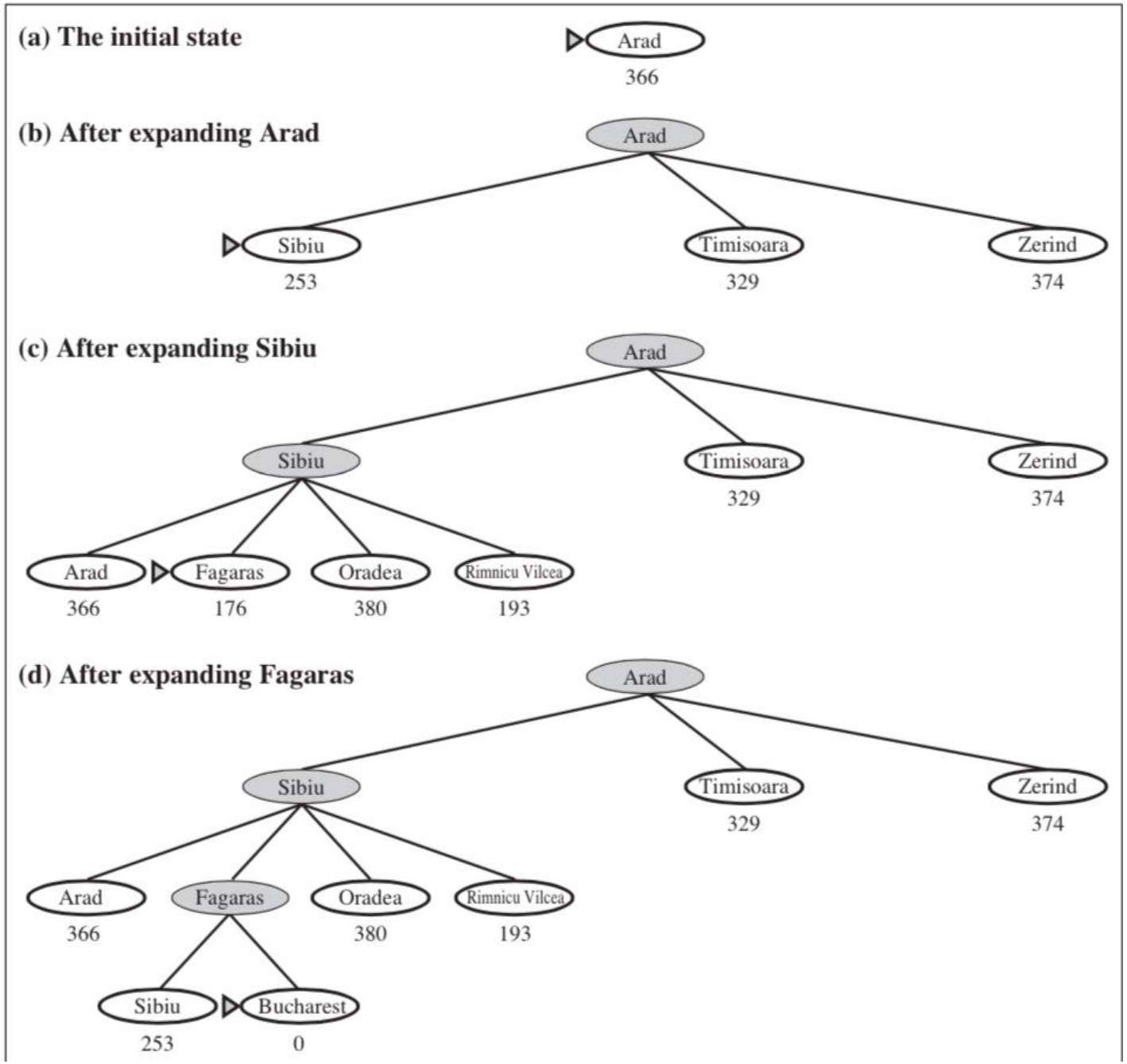
Greedy best-first search

Greedy best-first search tries to expand the node that is closest to the goal by using the **straight- line distance** heuristic h_{SLD}. If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in Figure below.



Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Values of h_{SLD}—straight-line distances to Bucharest.



Greedy best-first search

- The left Figure shows the progress of a greedy best-first search using h_{SLD}
- It is not optimal as the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.
- This shows why the algorithm is called "**greedy**"—at each step it tries to get as close to the goal as it can.

8-Puzzle: example

initial state:

8		7
6	5	4
3	2	1

goal state:

1	2	3
4	5	6
7	8	

What to use as heuristic function?

- Number of displaced squares:

  $h_1($  $) = 7$

- Sum of city-block (L1) distances between all squares and their final positions:

  $h_2($  $) = 21$

Notice that $h_2(s) \geq h_1(s)$

Properties of greedy search

- **Complete**? No–it can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt → …
  - ***Complete*** in finite space with repeated-state checking
- **Time**? $O(b^m)$, but a good heuristic can give dramatic improvement
- **Space**? $O(b^m)$—keeps all nodes in memory
- **Optimal**? No

With a good heuristic function, however, the complexity can be reduced substantially.

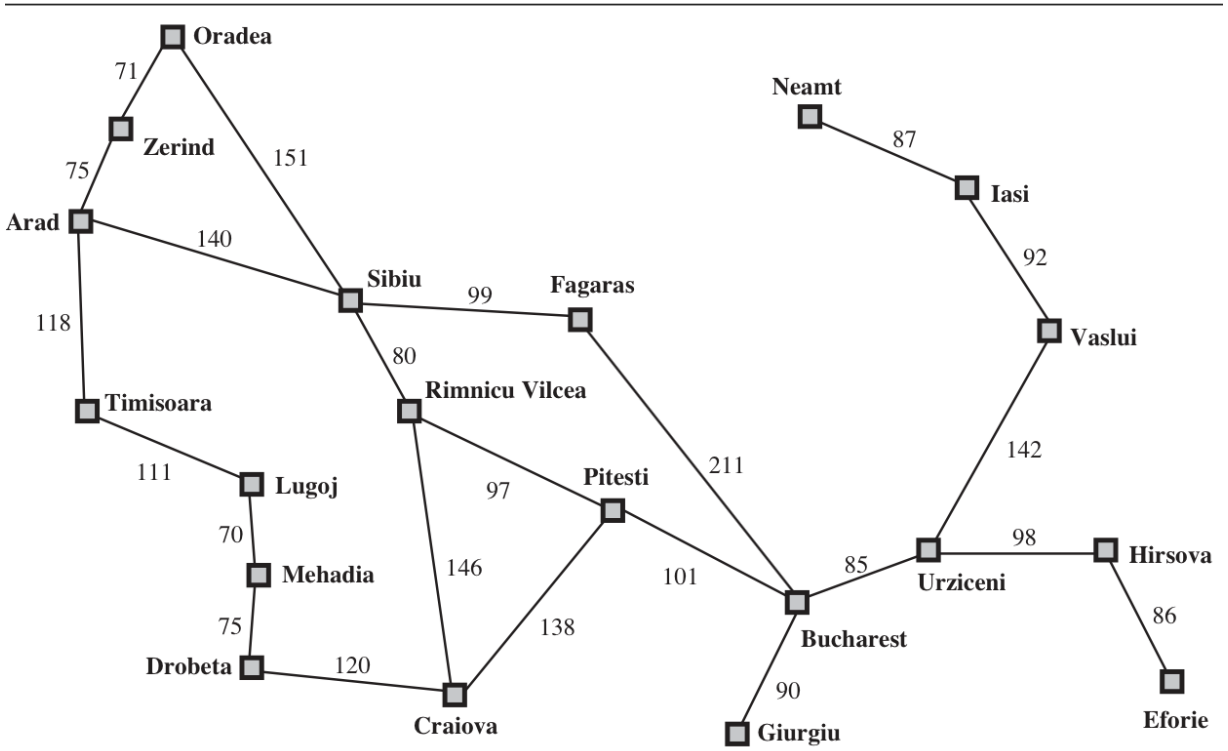
A* Search

Idea: avoid expanding paths that are already expensive.

Evaluation function f(n) = g(n) + h(n), i.e., the algorithm

combines best-first and uniform cost search

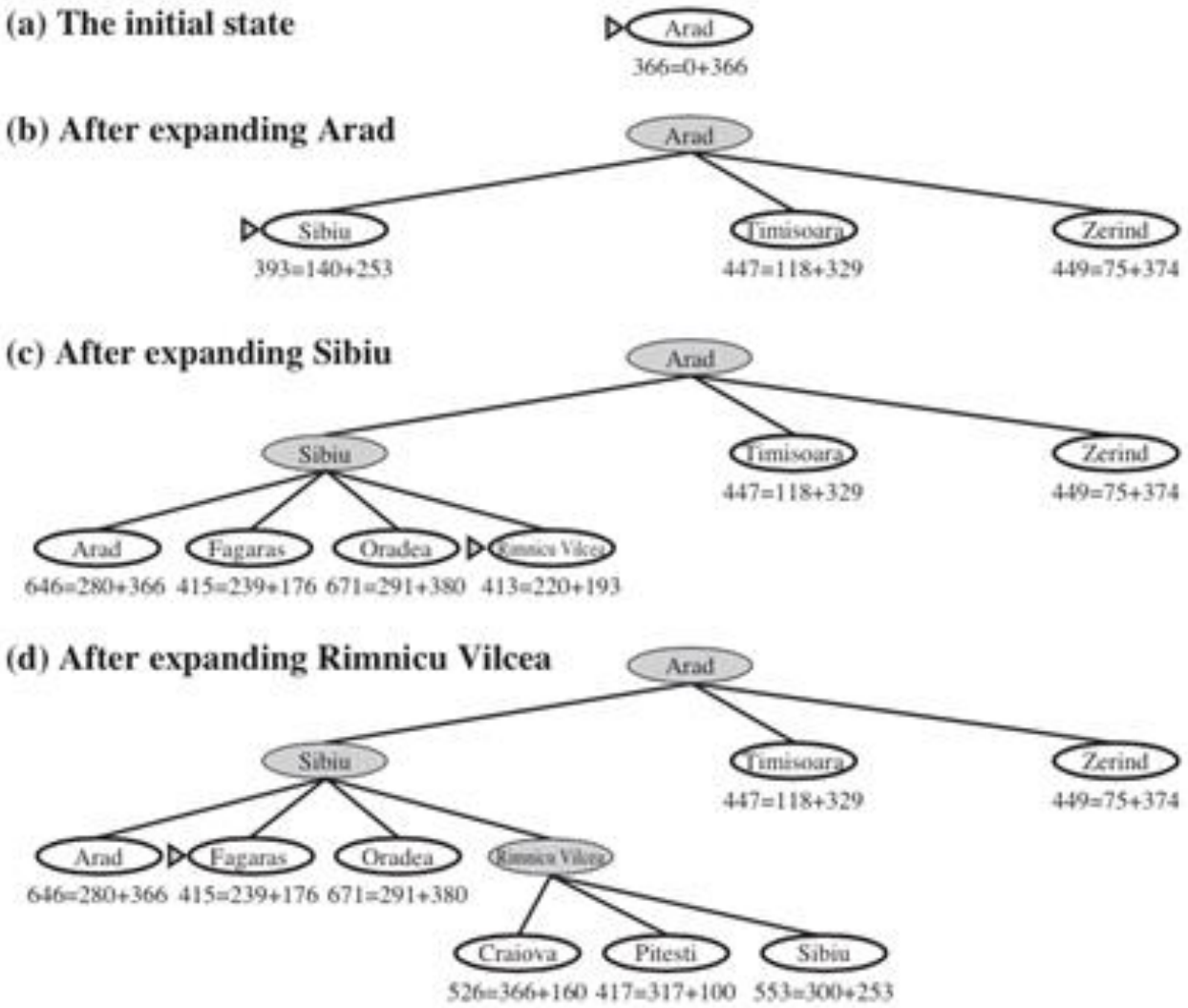
- g(n) = cost so far to reach n (true path cost)
- h(n) = estimated cost to goal from n
- f(n) = estimated total cost of path through n to goal

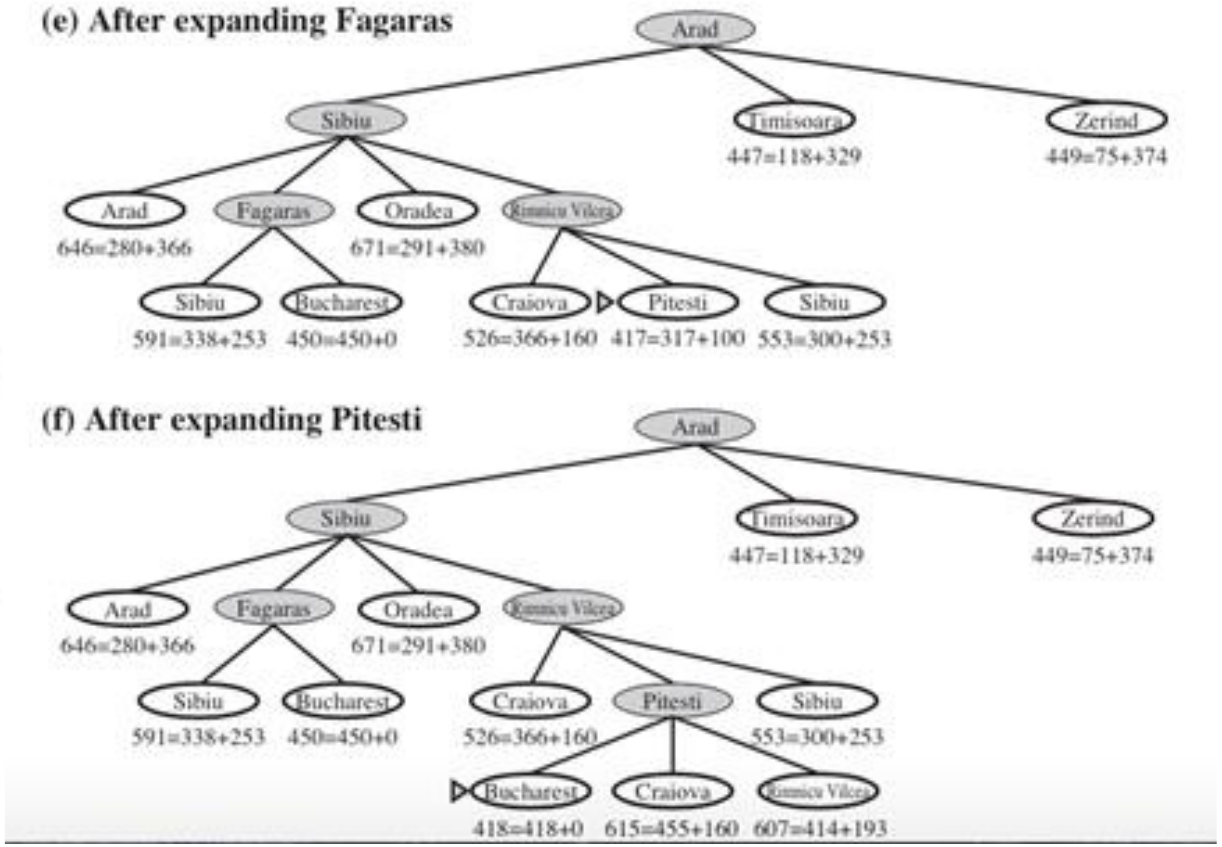


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

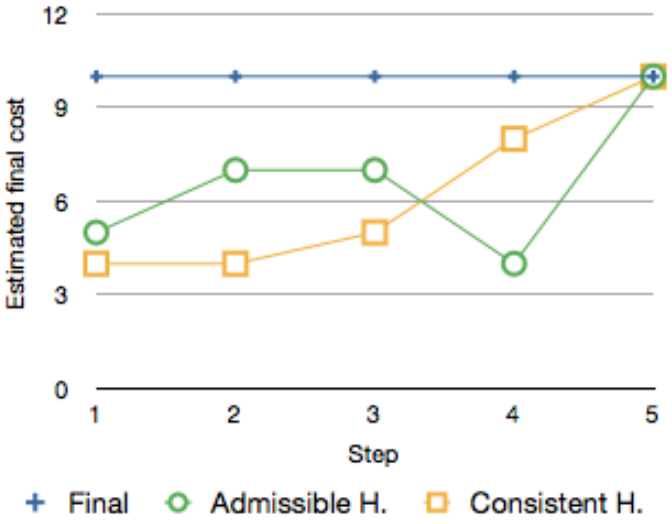
Values of h_{SLD}—straight-line distances to Bucharest.

Following figure explain it's execution using the information given in previous two figures.





Admissible heuristic vs. Consistent heuristic



Admissible heuristic

Heuristic function h is optimistic or admissible iff it never overestimates, i.e., its value is never greater than the true cost needed to reach the goal for every node (or state *S*): ∀s ∈ S. h(s) ≤ h∗(s), where h∗(s) is the true path cost of reaching the goal state from state *S*.

Consistent heuristics

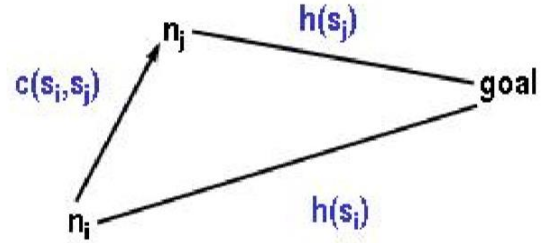
It is necessarily optimistic.

The heuristic cost is moving from one entry to the next cannot decrease by more than the arc cost between the states.

h(s_i)=0, if n_i is a goal.

h(s_i) ≤ h(s_j) + c ($s_i + s_j$) for s_j a child of s_i

The best way of visualizing the consistency condition is as a "triangle inequality", that is, one side of the triangle is less than or equal the sum of the other two side



Heuristic h is consistent or monotone iff: ∀(s2,c) ∈ succ(s1). h(s1) ≤ h(s2) + c

Properties of A*

- **Complete**? Yes, because it accounts for path costs
- **Time**? $O(b^{\varepsilon m})$—where $\varepsilon = (h^* - h)/h^*$ is the relative error in h, and h^* is the true cost
  - If h = 0, then ε = 1 and we get uniform-cost search
  - If $h = h^*$, then it is perfect, and we find the solution immediately
- **Space**? $O(b^m)$—it keeps all nodes in memory
- **Optimal**? Yes, but provided the heuristic h is an admissible (or optimistic)

If the heuristics is not optimistic, the search might bypass the optimal path because it seems more expensive than it really is

Issues in A* search

- Exponential complexity of A* with respect to depth of goal often makes it impractical in finding an optimal solution.
- Computation time is not the main drawback of A*, as it keeps all generated nodes in memory (as do all GRAPH-SEARCH algorithms), and usually runs out of space long before it runs out of time.
- For this reason, A∗ is not practical for many large-scale problems.

Beyond Classical Search

Simplifying assumptions of the previous study, and thereby getting closer to the real world.

Local search

- Hill Climbing
- Gradient Decent
- Simulated Annealing Search
- *and others ..*

Global search

- Genetic Algorithms

Local Search Algorithm

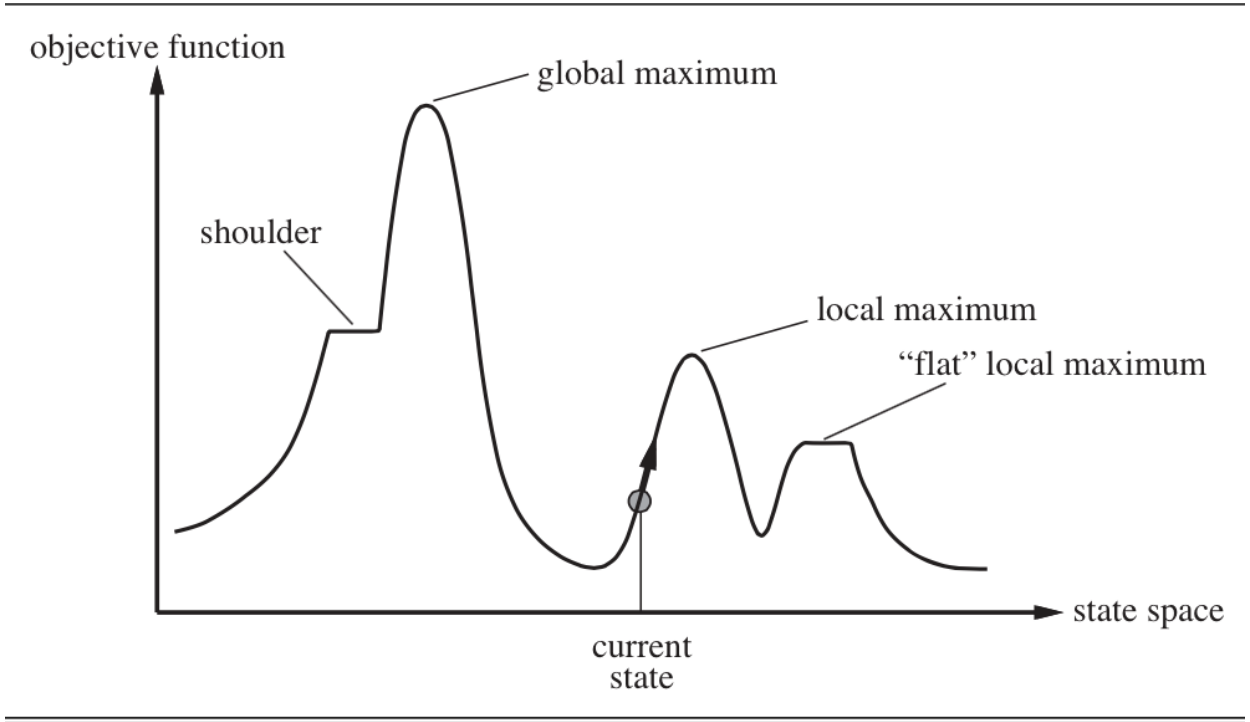
- Previously, state spaces were searched and recorded from start to end until a path to a solution is found, but here, the path to the goal is irrelevant.
  - For example, in 8-queens problem, what matters is the final configuration of queens, not the order in which they are added.

- If the path to the goal does not matter, then we are considering a different class of algorithms known as *Local search* algorithms.
  - It considers only current and its neighboring nodes.
  - Paths are not retained or in memory.
  - Two key advantages:
    - Use very little memory
    - Can find reasonable solutions in large or infinite state spaces.

Local Search Algorithm

To understand local search, **state-space** landscape is given in figure.

- If elevation (rising) corresponds to cost, then the aim is to find the lowest valley—a global minimum;
- if elevation corresponds to an objective function, then the aim is to find the highest peak—a global maximum.
- A **plateau (**hill**)** is a flat area: can be a flat local maximum or a **shoulder** (from which progress is possible).



Hill Climbing Algorithm

- Loop continues in the direction of increasing value—that is, uphill and terminates when it reaches a “peak” where no neighbor has a higher value.
- Record the current and its immediate neighboring node with the value of the objective function only.
- **Problems:**
  - Hill climbing is good for finding a Local Maximum, but It does not guarantee the best possible solution (Global Maximum).
  - A hill-climbing search might get lost on the plateau.
  - Depending on initial state, can get stuck in local maxima

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

Properties of Hill Climbing Algorithm

- Not complete
- Not optimal
- Time complexity O(m)
- Space complexity O(1)

Variants of Hill Climbing Search

Stochastic hill climbing:

- Difference is only random selection of neighboring nodes

Random-restart hill climbing:

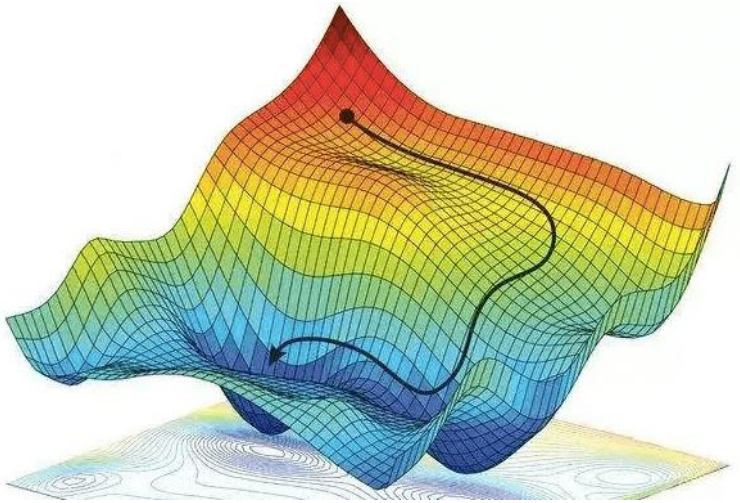
- It is also known as Shotgun hill climbing.
- It iteratively does hill-climbing, each time with a random initial condition .
- Best is kept if a new run of hill climbing produces a better than the stored state.
- Get rid of local maxima

Gradient descent:

- It is a specific kind of “hill climbing” algorithm.
- A main difference is that in hill climbing it maximizes a function, while gradient descent minimizes it.
- In hill climbing, one can look at all neighboring states and evaluate the cost function in each of them and then chose to move to the best neighboring state. While, in gradient descent we look at the slope of the local neighborhood and move in the direction with the steepest slope.

Gradient Descent

Assume we have some cost-function: $C(x_1,\ldots,x_n)$
and we want minimize over continuous variables X1,X2,..,Xn



1. Compute the *gradient :* $\frac{\partial}{\partial x_i} C(x_1,\ldots,x_n) \qquad \forall i$

2. Take a small step downhill in the direction of the gradient:

3. Check if $C(x_1,..,x'_i,..,x_n) < C(x_1,..,x_i,..,x_n)$

4. If true then accept move, if not reject. $x_i \to x'_i = x_i - \lambda \frac{\partial}{\partial x_i} C(x_1,\ldots,x_n) \qquad \forall i$

5. Repeat.

- Read it yourself for now, see e.g., https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0
- We will discuss it later in **Neural Network** after few weeks

Simulated Annealing Search

- A hill-climbing algorithm that never moves toward states with lower value is guaranteed to be incomplete, because it can get stuck on a local maximum.

- In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient.

- To try to combine hill climbing with a random walk in some way that yields both efficiency and completeness, **Simulated annealing** (heat and cool) is such an algorithm.

- Simulated-annealing solution is to start by shaking hard (i.e., at a high cost) and then gradually reduce the intensity of the shaking (i.e., lower the cost).

Simulated Annealing Search

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(*problem*.INITIAL-STATE)
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow schedule(t)$
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow next.\text{VALUE} - current.\text{VALUE}$
    if $\Delta E > 0$ **then** $current \leftarrow next$
    else $current \leftarrow next$ only with probability $e^{\Delta E/T}$

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of the temperature T as a function of time.

Simulated Annealing Search

- Instead of picking **the best move**, however, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.

- The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.

- E-25/10000= 0.997, e-25/25 = 0.36, e-25/20 = 0.28, e-25/10 = 0.08

Simulated Annealing Search

The acceptance probability function:

- The acceptance probability function takes in the old cost, new cost, and current temperature and spits out a number between 0 and 1, which is a sort of recommendation on whether to jump to the new solution. **For example:**
  - 1.0: definitely switch (the new solution is better)
  - 0.0: definitely stay but (the new solution is infinitely worse)
- Once the acceptance probability is calculated, it's compared to a randomly-generated number between 0 and 1. If the acceptance probability is larger than the random number, you're switching!

Contents for the next lecture

- Genetic Algorithms
- Constraint Satisfaction Problems
- Games as Search

Any questions?