Artificial Intelligence

Instructor: Kietikul Jearanaitanakij

Department of Computer Engineering

King Mongkut’s Institute of Technology Ladkrabang

Lecture 4
Heuristic & Adversarial Search

- Informed (Heuristic) Search Strategies
- Optimal Decisions in Games
- Alpha-Beta Pruning

Informed (Heuristic) Search Strategies

- The general approach we consider is called best-first search in which a node is selected for expansion based on an **evaluation function**, **f(n)**.

- **Heuristic function h(n)** provides an informed way to guess which neighbor of a node will lead to a goal. We usually form f(n) by combine h(n) with another cost function.

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

  Where h(n) is nonnegative and h(n) = 0 is n is a goal state.

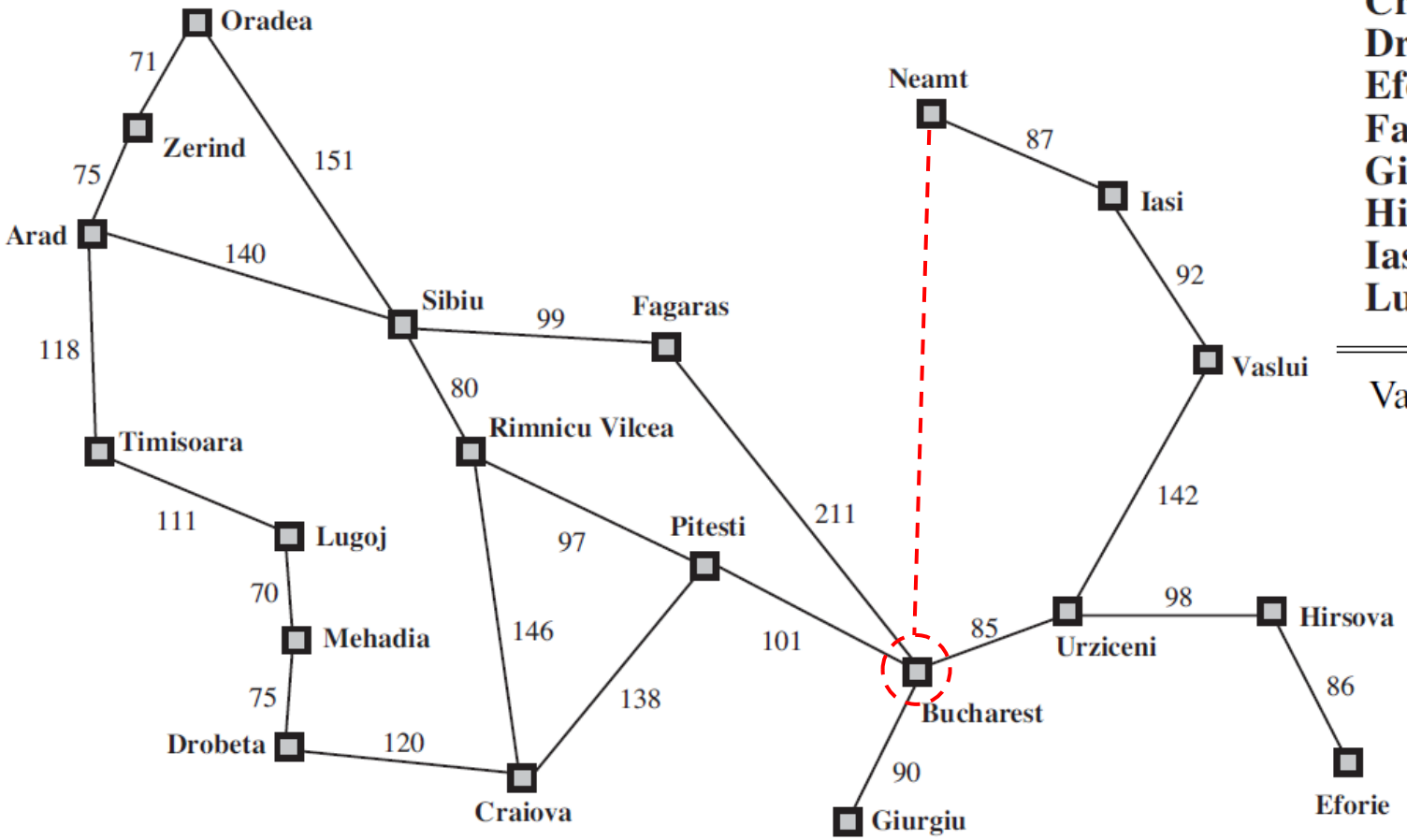
- The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first.
- The implementation of best-first graph search is identical to that for uniform-cost search.

Informed (Heuristic) Search Strategies

1. **Greedy best-first search**

Greedy best-first search tries to expand the node that is closest to the goal. This is likely lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, f(n) = h(n).

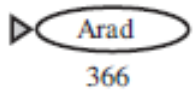
Let's consider the route-finding problems in Romania. We will use the straight-line distance heuristic, which we will call h_{SLD}. If the goal is Bucharest, we need to know the straight-line distances from any city to Bucharest



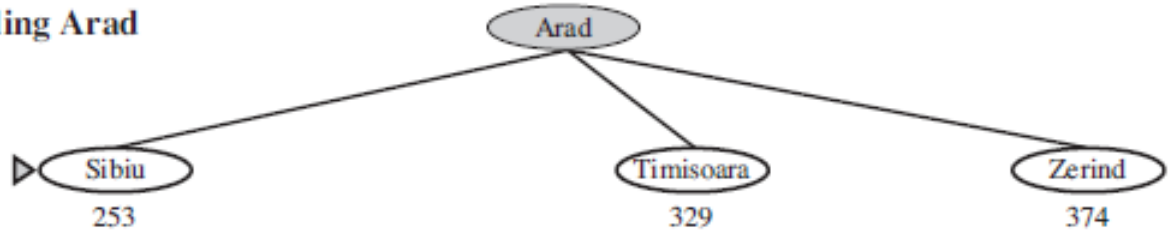
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

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

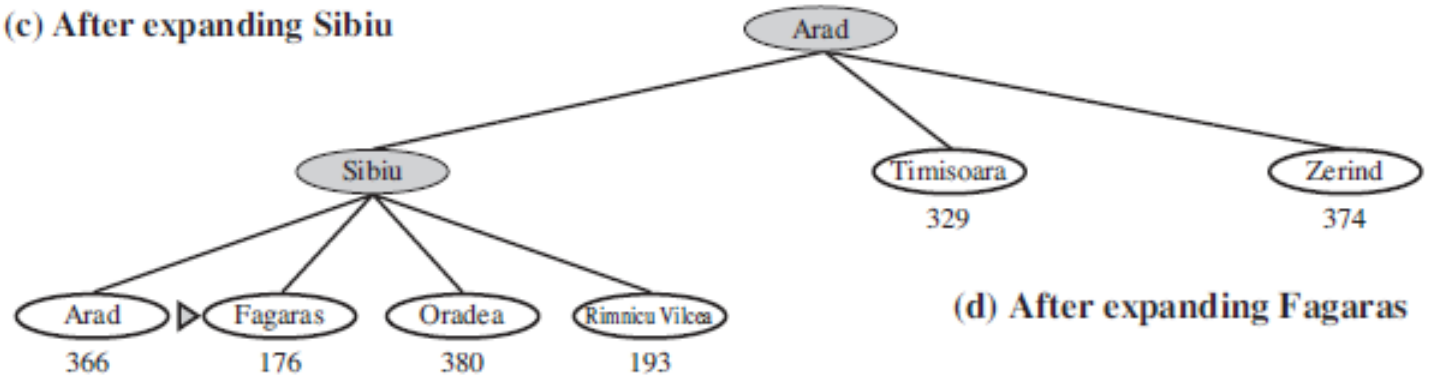
(a) The initial state



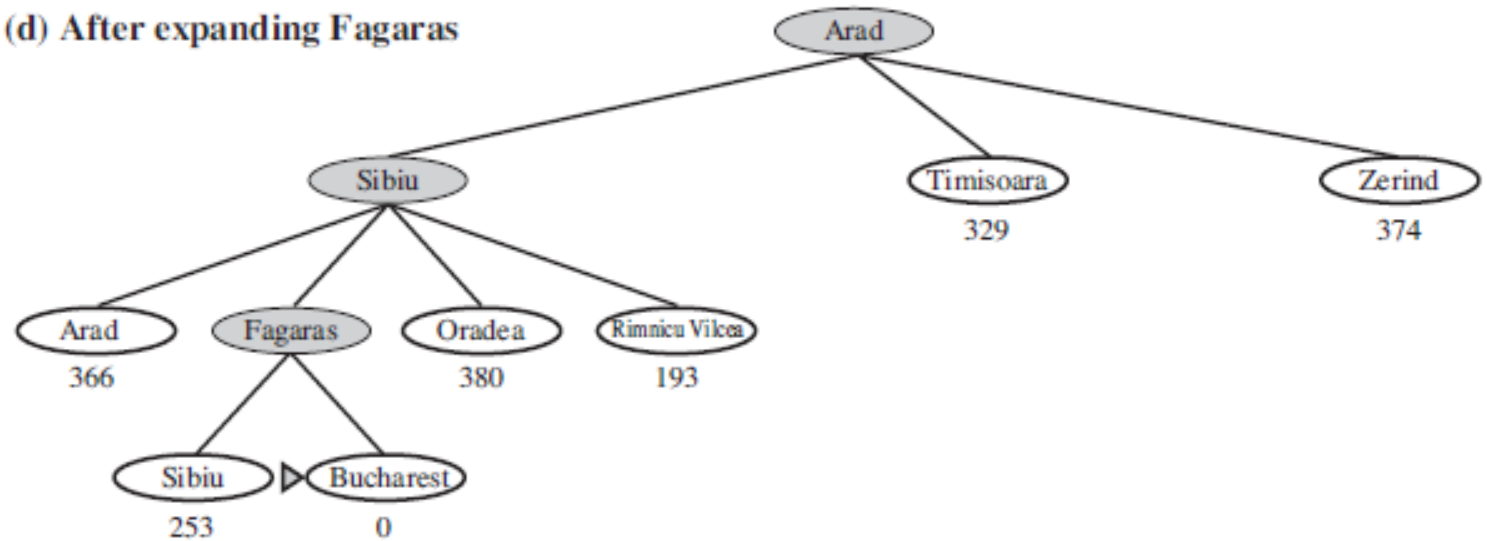
(b) After expanding Arad



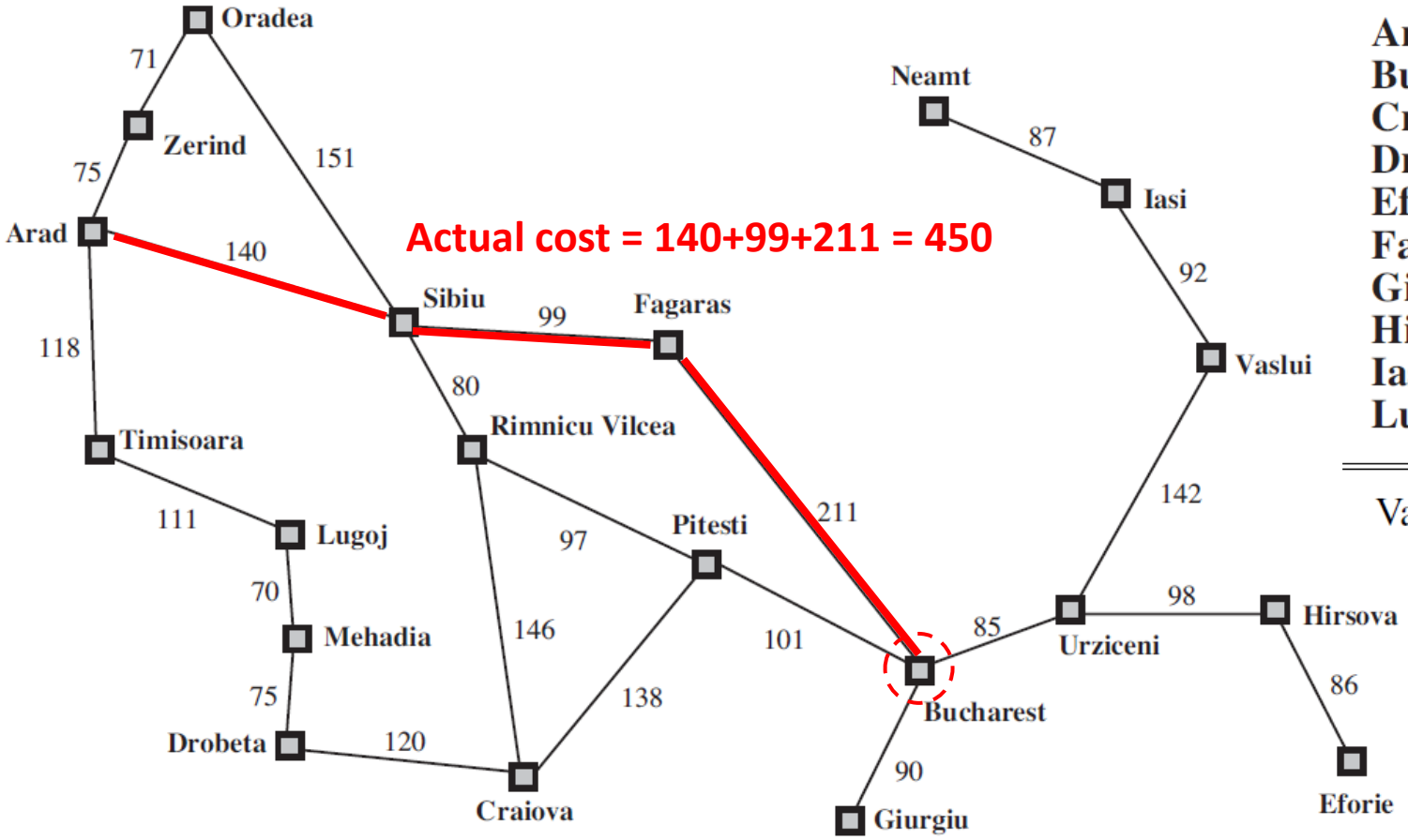
(c) After expanding Sibiu



(d) After expanding Fagaras



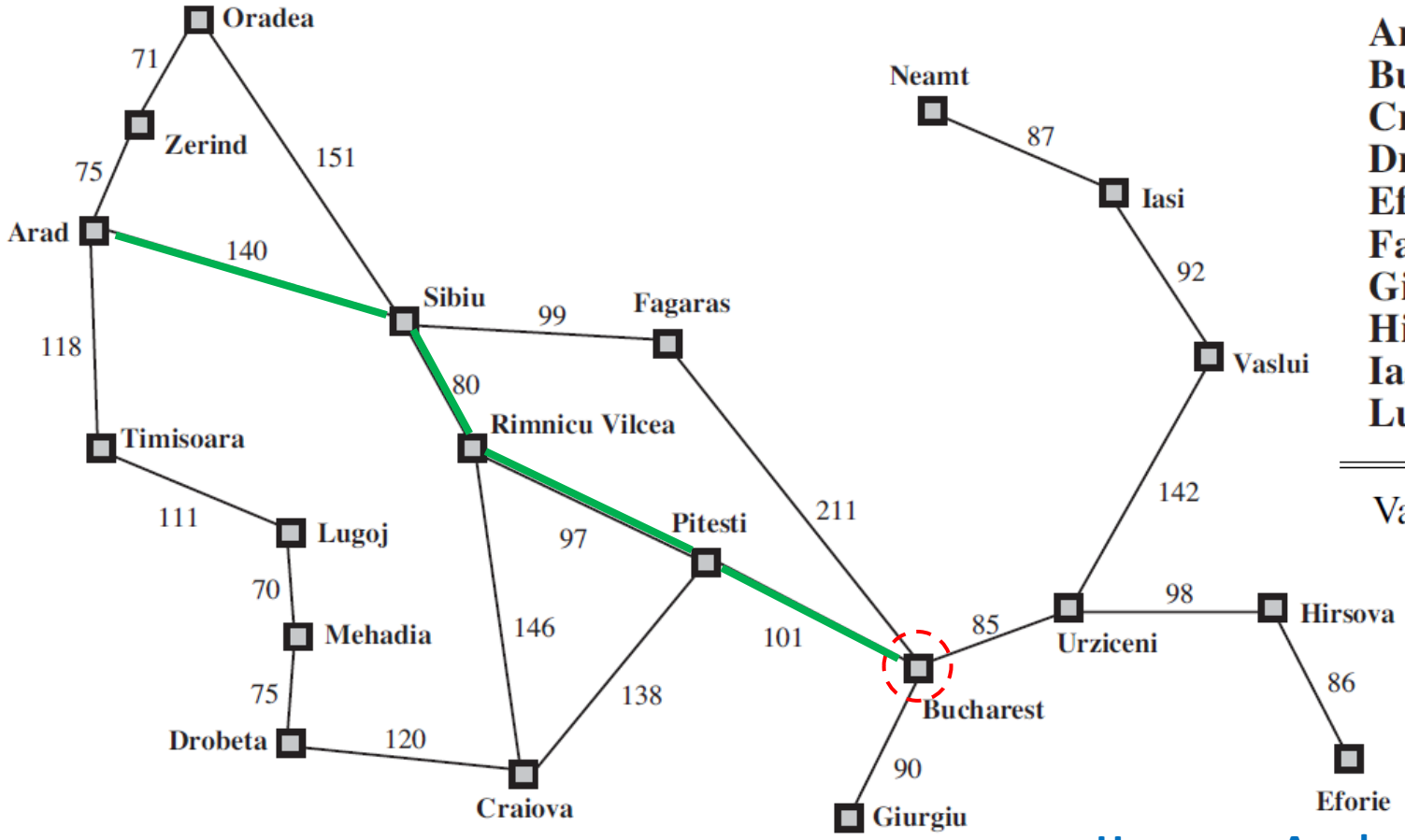
Searching from Arad to Bucharest by using greedy best-first search



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.

Searching from Arad to Bucharest by using greedy best-first search

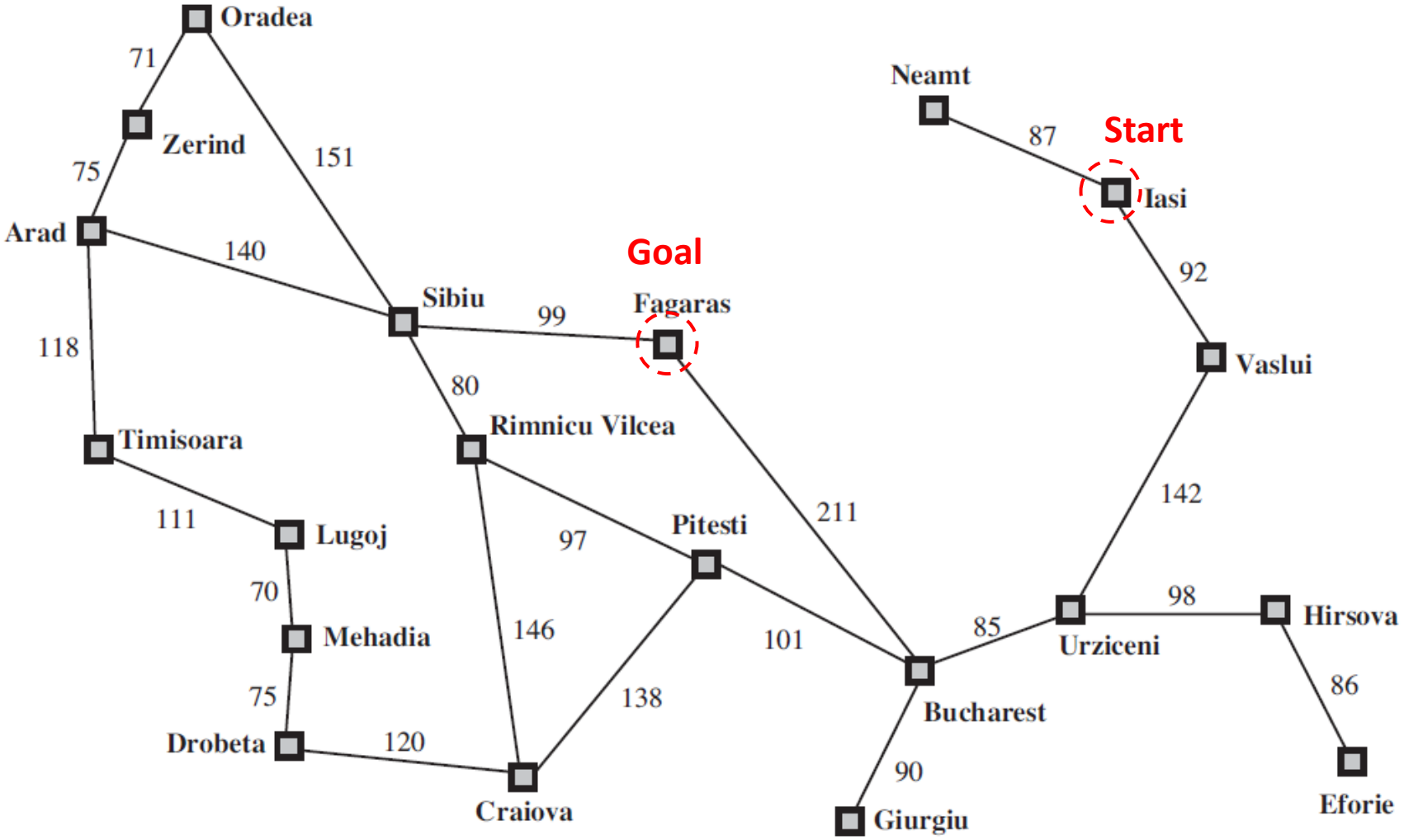


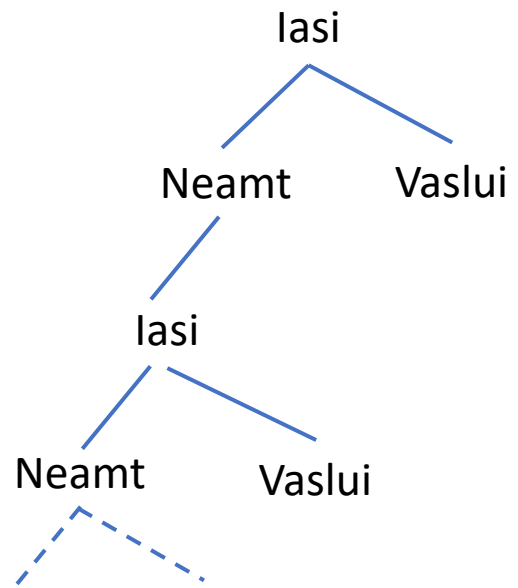
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

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

However, Arad -> Sibui -> Rimnicu -> Pitesti -> Bucharest takes lower actual cost : 140 + 80 + 97 + 101 = 418 ! (Not optimal)

Greedy best-first search is also incomplete.





Infinite loop

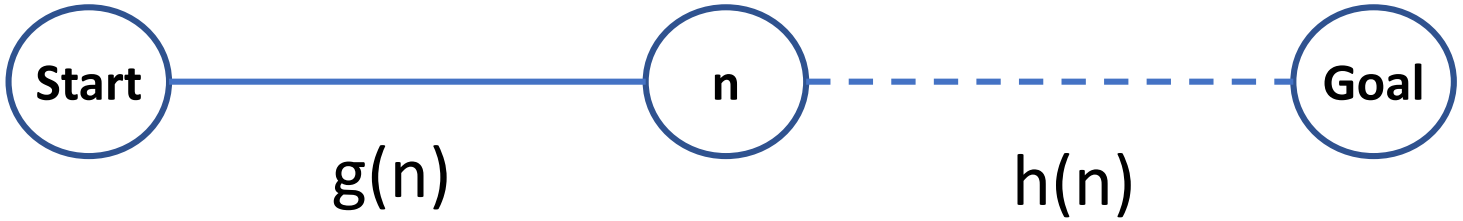
Time & Space Complexities = $O(b^m)$
Where m is the max depth of the search space.

Informed (Heuristic) Search Strategies

2. **A* search**

The most widely known form of best-first search is called A∗ search (pronounced “A-star search”). It evaluates nodes by combining g(n), the path cost to reach the node n, and h(n), the cost to get from the node n to the goal:

$$f(n) = g(n) + h(n) .$$



f(n) = estimated cost of the cheapest solution through n .

Conditions for optimality of A* search

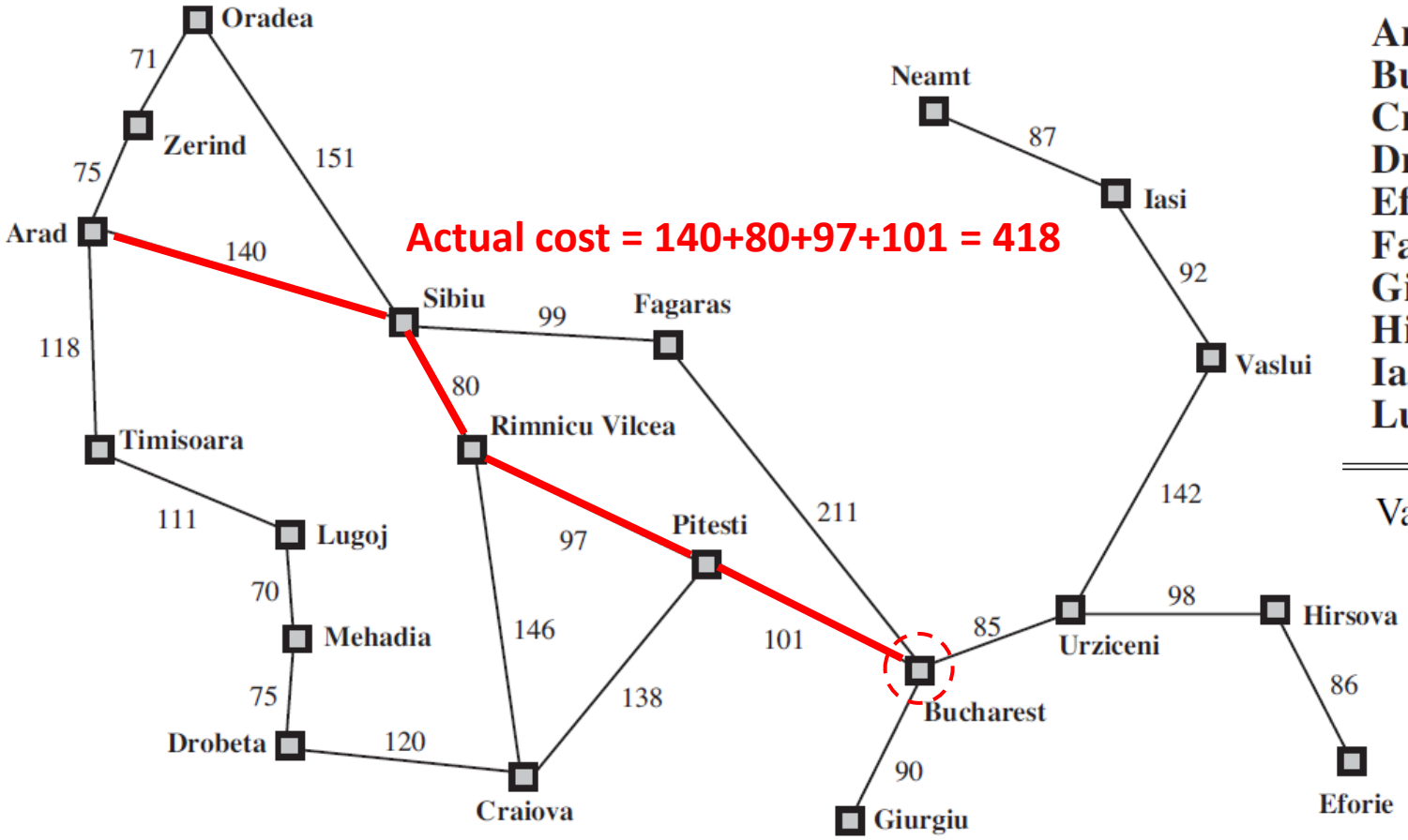
1. **h(n) must be an admissible heuristic.**

   An admissible heuristic is one that never overestimates the cost to reach the goal. An obvious example of an admissible heuristic is the straight-line distance h_{SLD}. Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate.

2. **h(n) must be consistency (monotonicity).**

   A heuristic h(n) is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

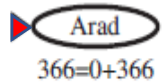
Searching from Arad to Bucharest by using A* search



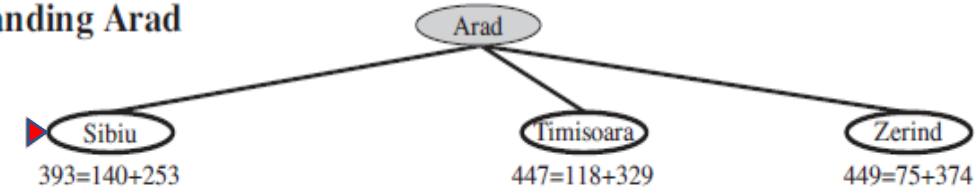
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

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

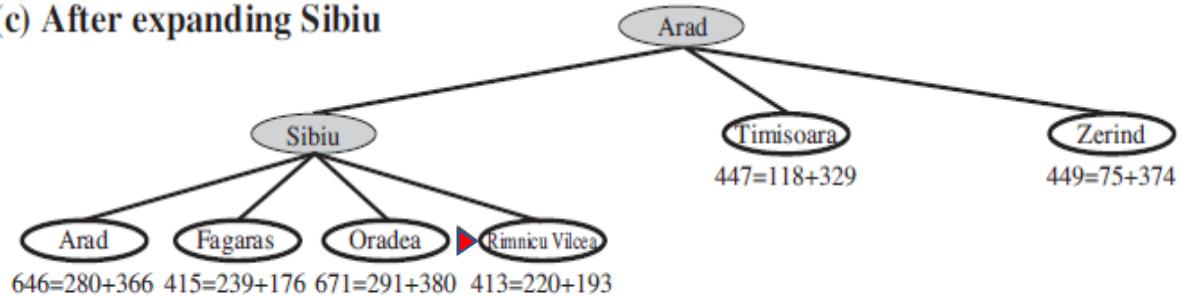
(a) The initial state



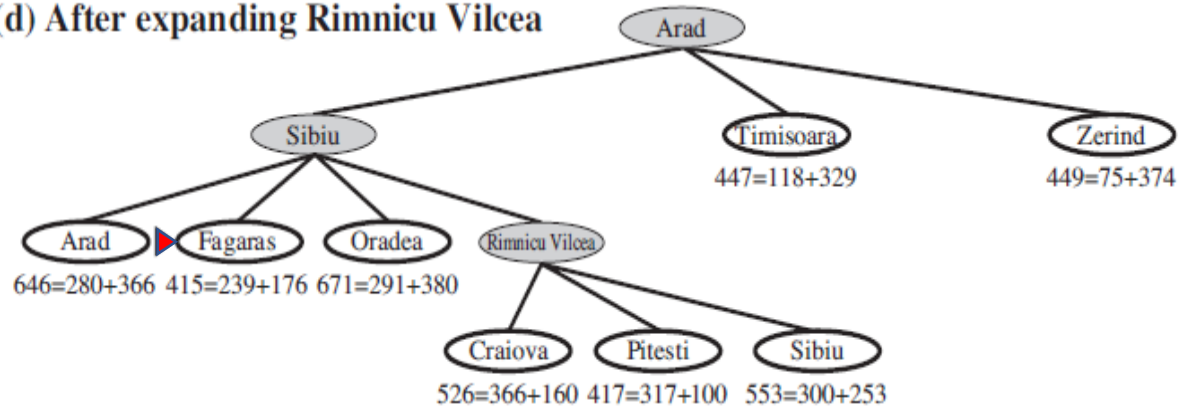
(b) After expanding Arad



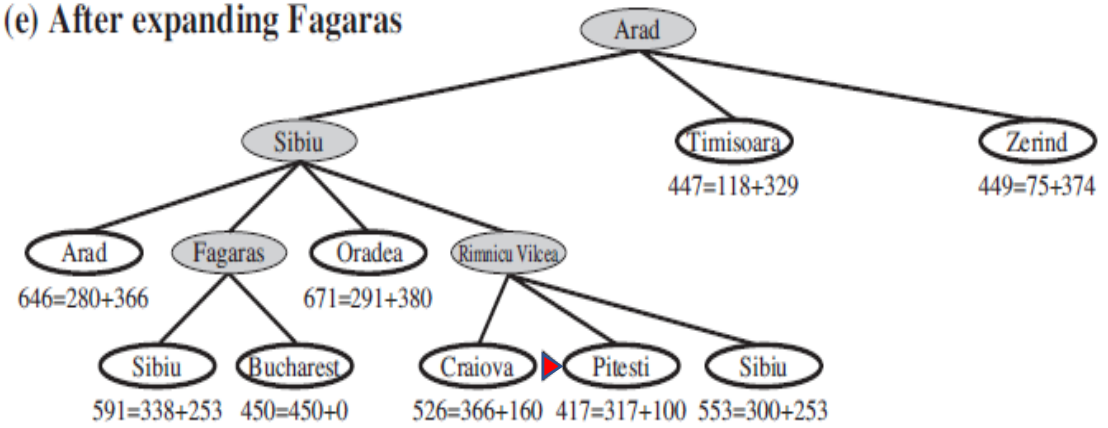
(c) After expanding Sibiu



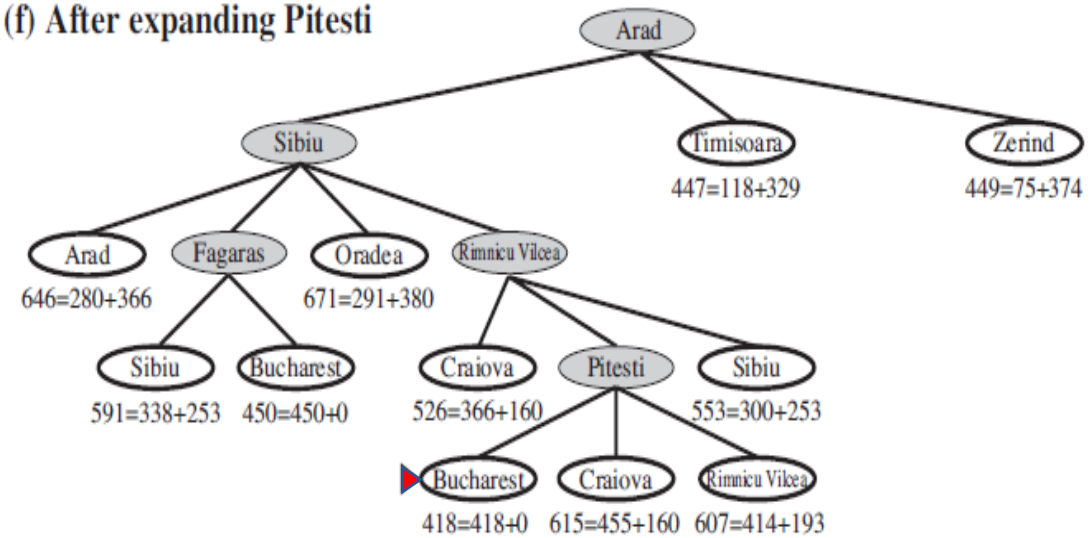
(d) After expanding Rimnicu Vilcea

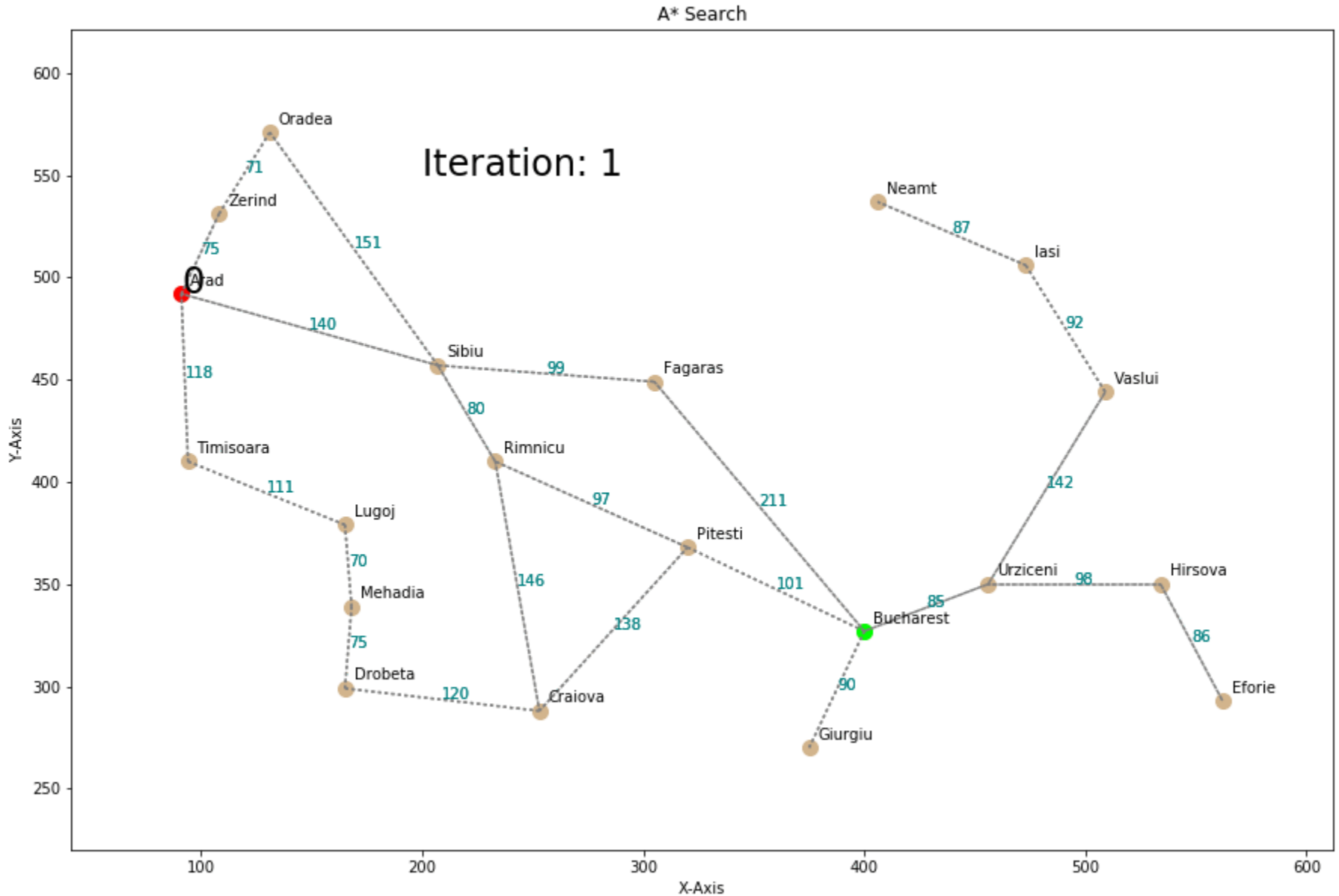


(e) After expanding Fagaras



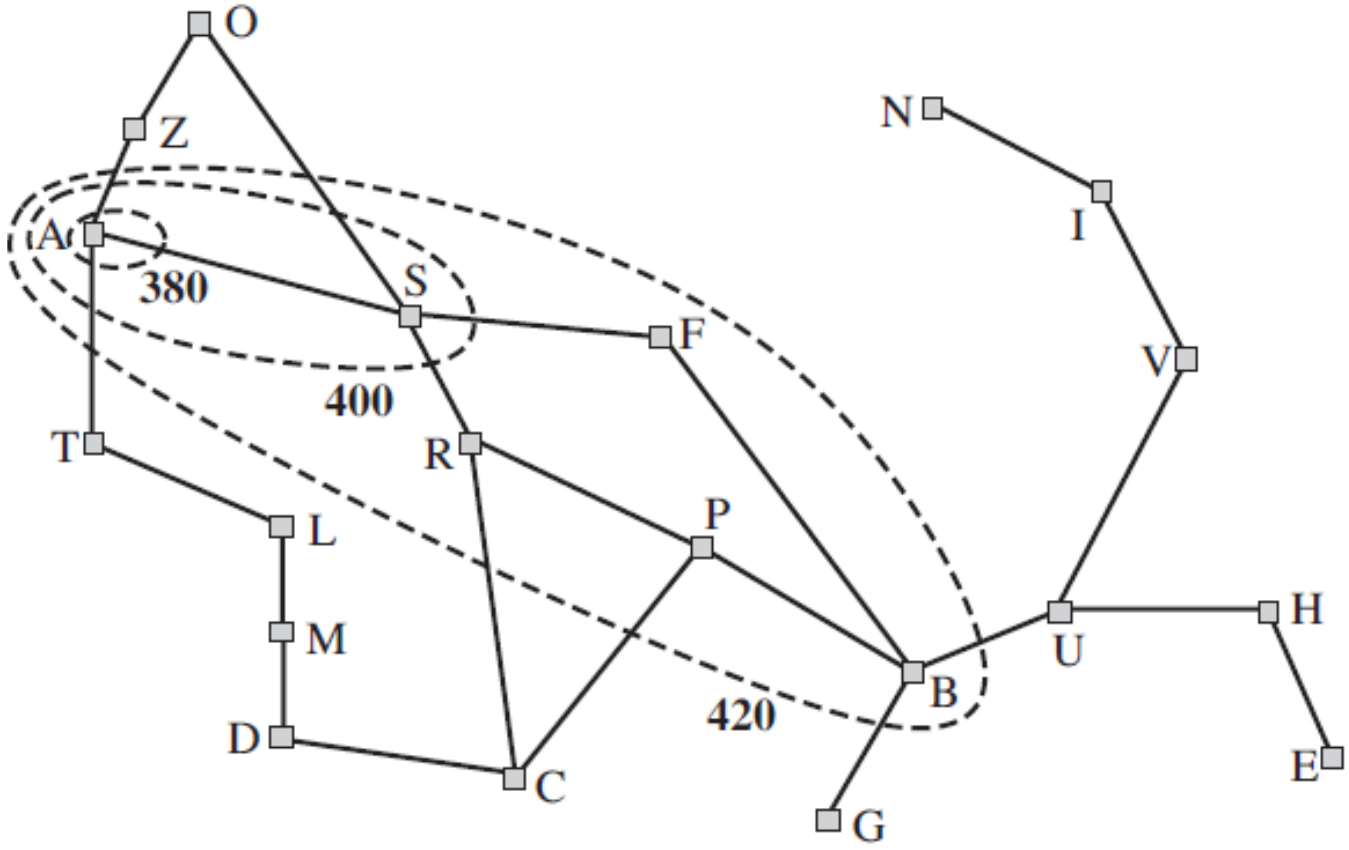
(f) After expanding Pitesti





http://teleported.in/posts/ai-search-algorithms/

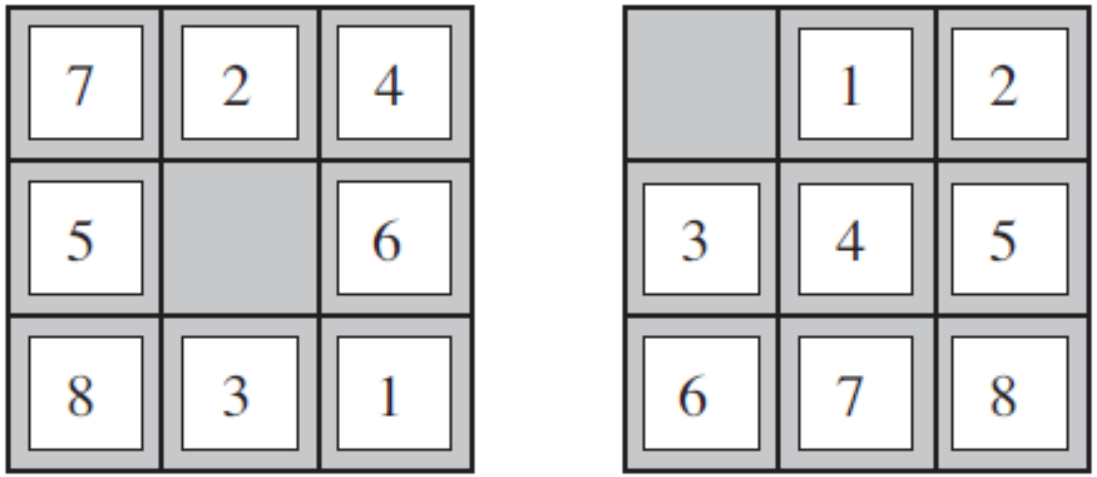
Contours at f = 380, f = 400, and f = 420, with Arad as the start state.
Nodes inside a given contour have f-costs less than or equal to the contour value.



A* search is complete and optimal. However, A* may not be suitable for large-scale problems since it has bad computation time and space complexity.

Examples of Heuristic Function

8-puzzle problem



A typical instance of the 8-puzzle. The solution is 26 steps long.

h1 = 8

h2 = 3 + 1 + 2 + 2 + 3 + 2 + 2 + 3 = 18

h1 = the number of misplaced tiles. h1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.

h2 = the sum of the distances of the tiles from their goal positions. Manhattan (city block) distance is the sum of the horizontal and vertical distances.

	Search Cost (nodes generated)			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	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	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

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

h_2 is better than h_1, and is far better than using iterative deepening search.

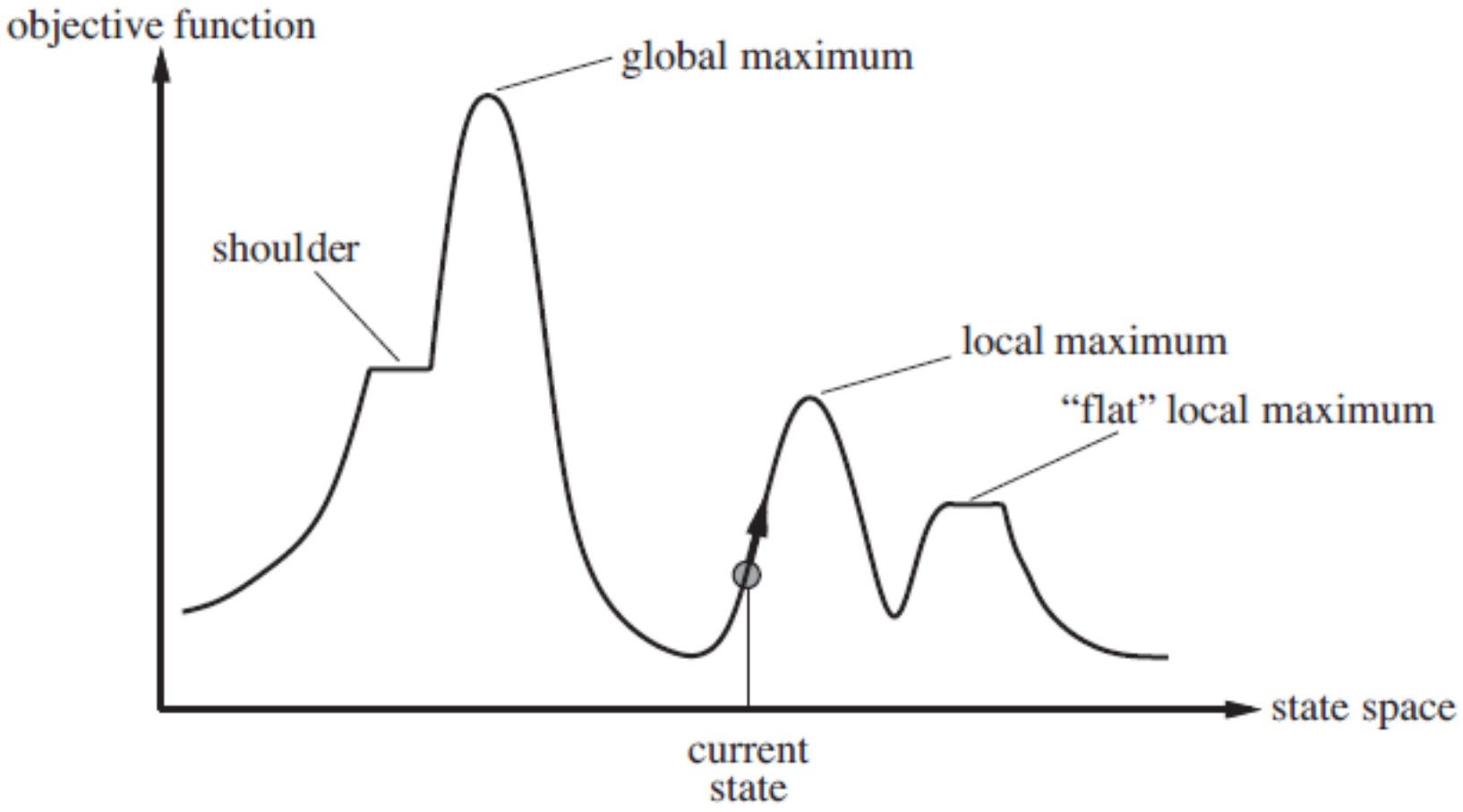
Local Search Algorithms and Optimization Problems

- The search algorithms that we have seen so far are designed to explore search spaces systematically.

- In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem, what matters is the final configuration of queens, not the order in which they are added. Other examples are IC design, factory-floor layout, job scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.

- If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all.
- Local search algorithms operate using a single current node (rather than multiple paths) and generally move only to neighbors of that node.
- Although local search algorithms are not systematic, they have two key advantages:
  1. They use very little memory—usually a constant amount; and
  2. They can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
- 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.

State-space landscape



A one-dimensional state-space landscape. The aim is to find the global maximum.

Hill-climbing Search

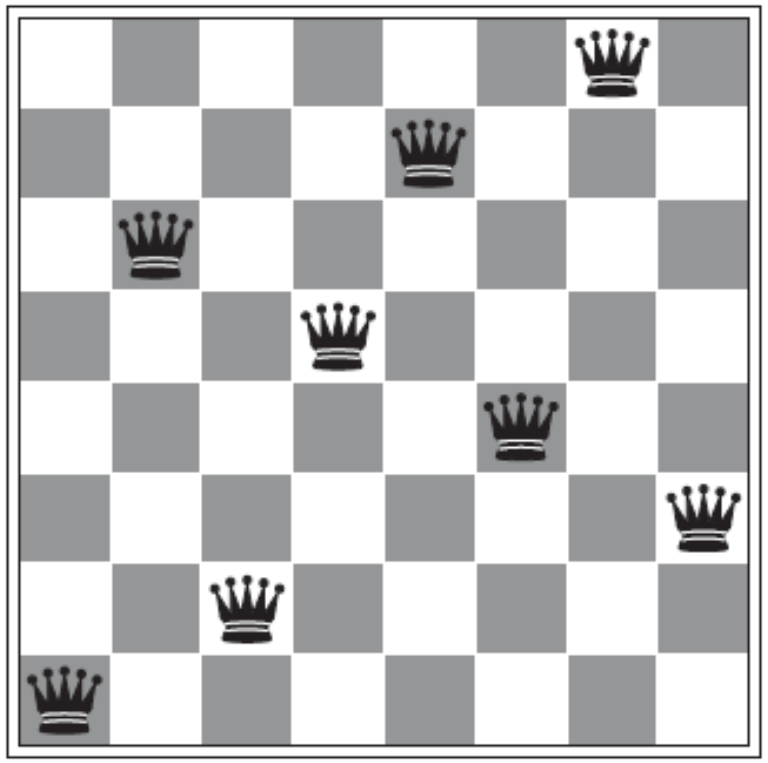
```
function HILL-CLIMBING(problem) returns a state that is a local maximum

    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
```

The hill-climbing search algorithm (steepest-ascent version) is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a “peak” where no neighbor has a higher value.

The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.

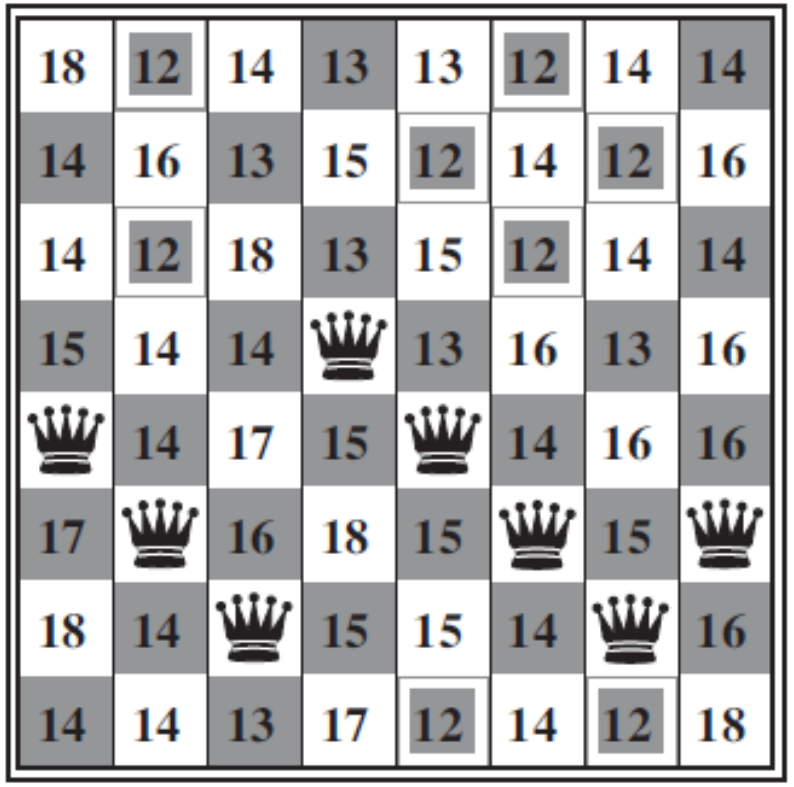
Example of using hill-climbing search



h = 1

- **A complete state** formulation: each state has 8 queens on the board, one per column.
- The **successors** of a state are all possible states generated by moving a single queen to another square in the same column (so each state has 8x7=56 successors).
- **Heuristic cost function** h is the number of pairs of queens that are attacking each other.

Example of using hill-climbing search

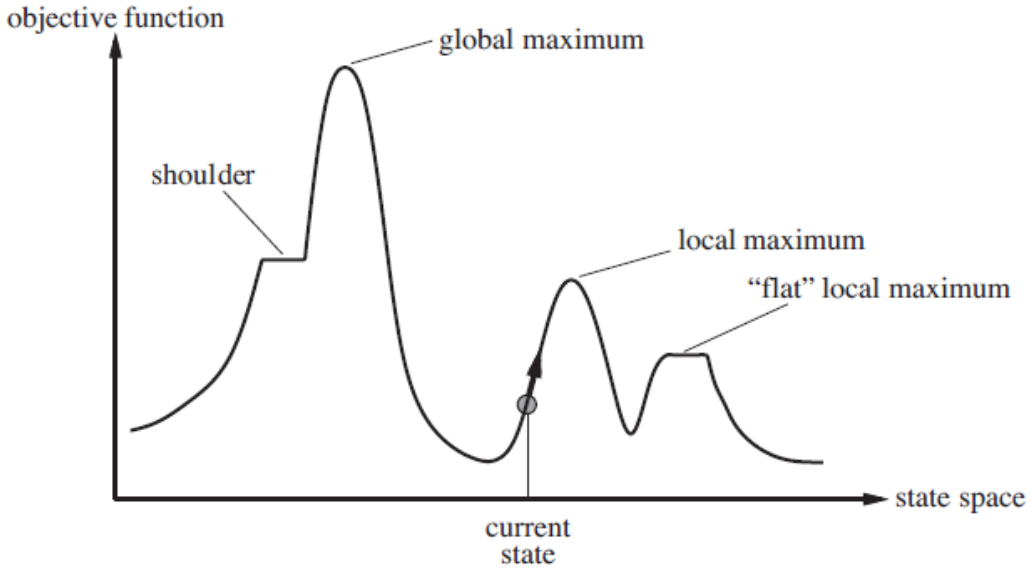


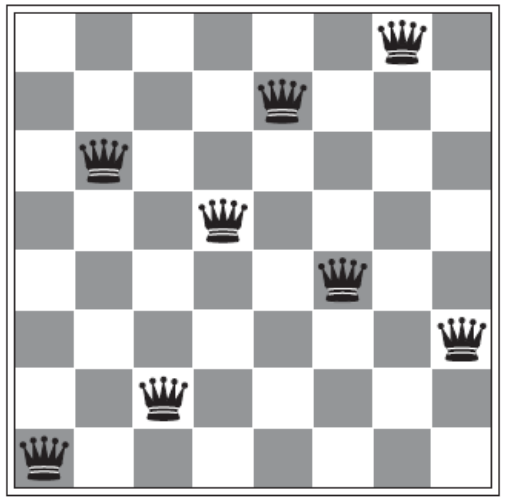
h = 17

- The figure shows the values of all its successors, with the best successors having h=12. (8 best successors which have h=12)
- Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.
- Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next.

Unfortunately, hill climbing often gets stuck for the following reasons:

1. **Local maxima**: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.

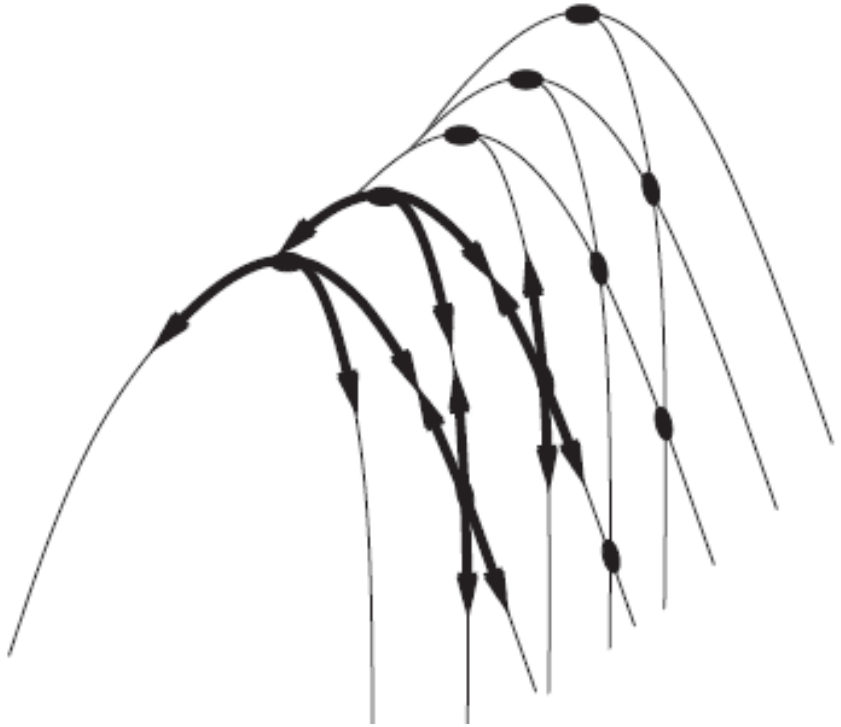




h = 1

This state is a local maximum (i.e., a local minimum for cost h); every move of a single queen makes the situation worse.

2. **Ridges**:



Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

3. **Plateaux**: A flat area of the state-space landscape. The highest neighbor may be slightly higher (different at 10^{th} decimal digit) than the current state. A hill-climbing search might get lost on the plateau.

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

Simulated Annealing

A hill-climbing algorithm that never makes “downhill” moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum.

Simulated annealing : Annealing is the process used to temper or harden metals (or glass) by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low energy crystalline state.

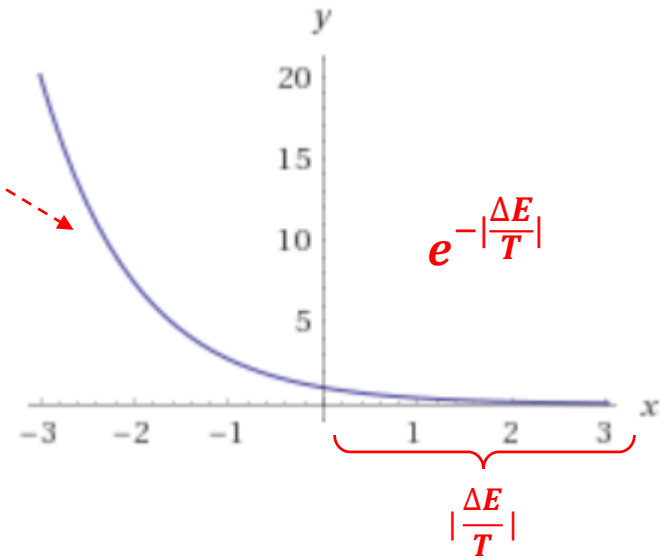
function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state

inputs: *problem*, a problem
schedule, a mapping from time to "temperature"

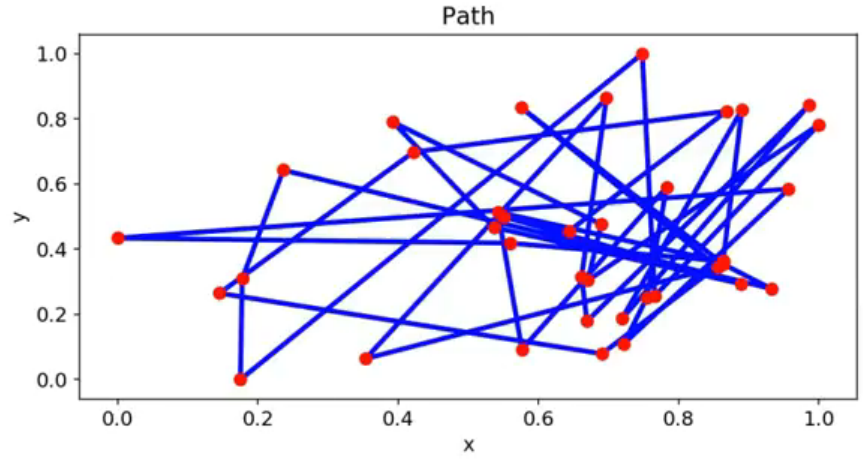
current ← MAKE-NODE(*problem*.INITIAL-STATE)
for $t = 1$ **to** ∞ **do**
  $T \leftarrow schedule(t)$
  if $T = 0$ **then return** *current*
  next ← a randomly selected successor of *current*
  $\Delta E \leftarrow next.\text{VALUE} - current.\text{VALUE}$
  if $\Delta E > 0$ **then** *current* ← *next*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

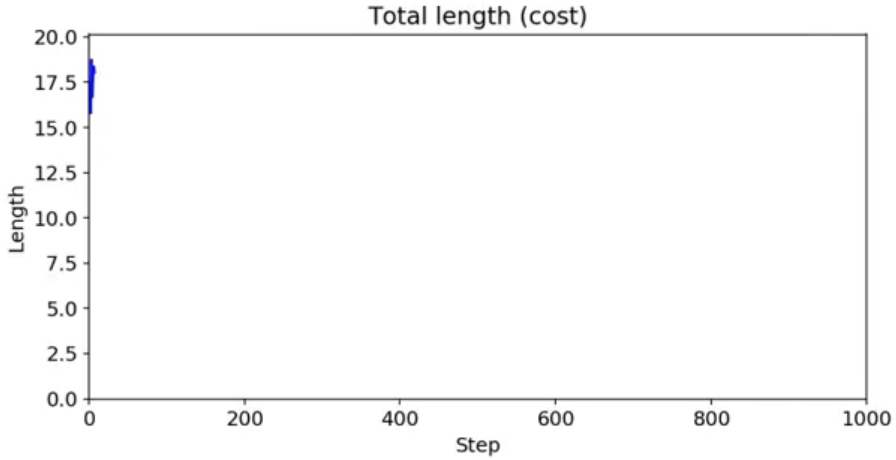
schedule

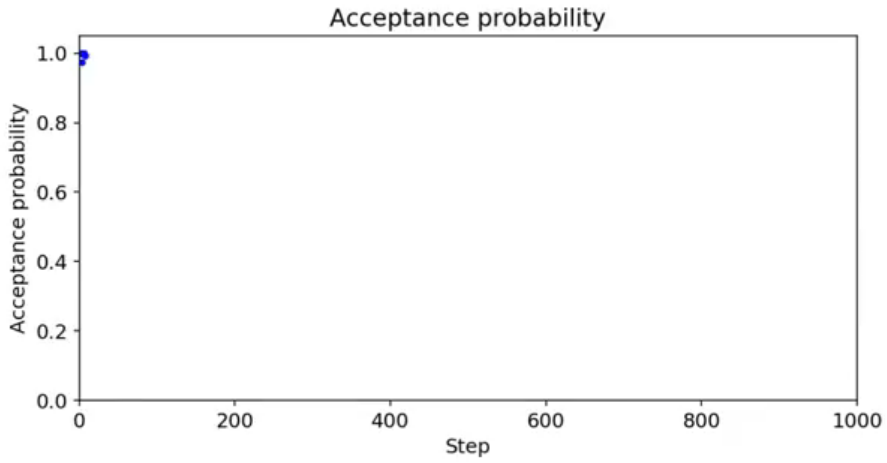
t (time)	T (Temperature)
1	T_MAX
2	T_MAX*0.98
3	T_MAX*(0.98^2)
4	T_MAX*(0.98^3)
…	…
t_MAX	Approach 0

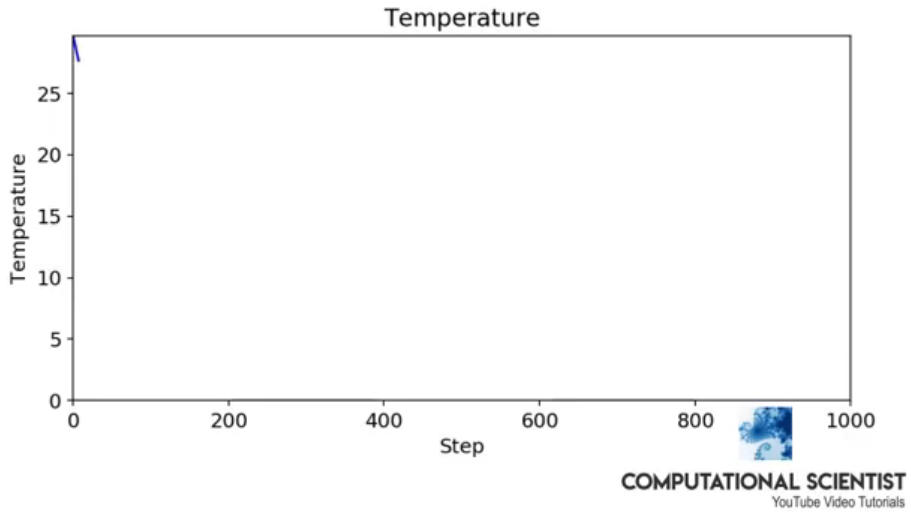


Simulated Annealing on TSP









Local Beam Search

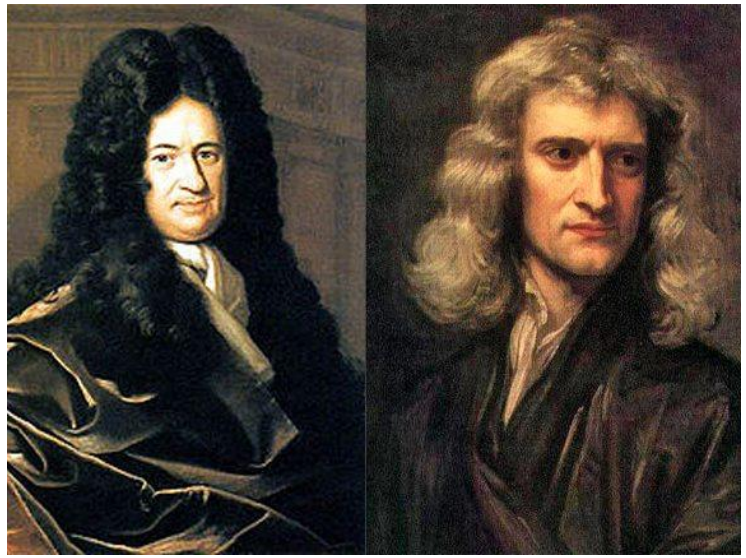
- The **local beam search** algorithm keeps track of k states rather than just one.
- It begins with k randomly generated states. At each step, all the successors of all k states are generated.
- If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.
- However, they quickly become concentrated in a small region of the state space.

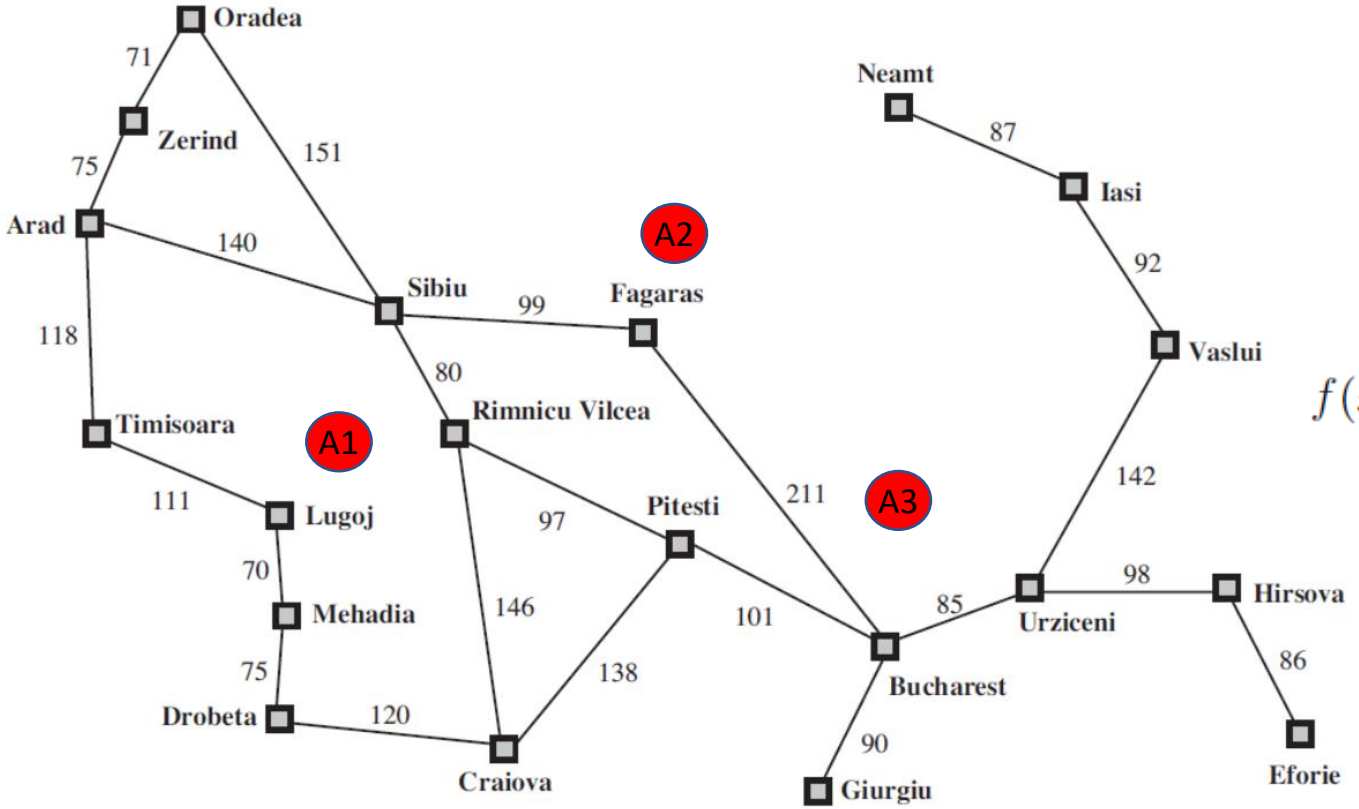
Local Search In Continuous Spaces

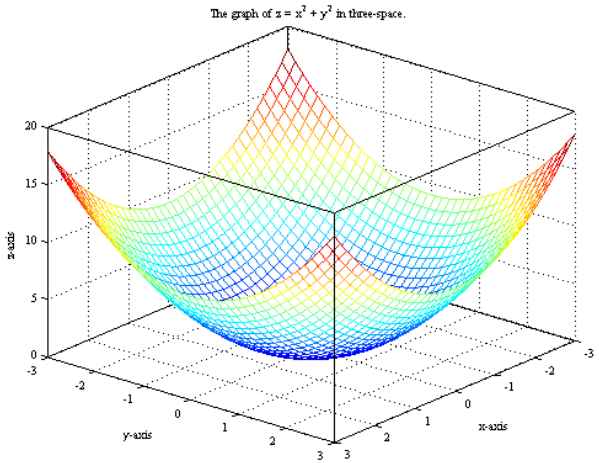
- None of the algorithms we have described (except for first-choice hill climbing and simulated annealing) can handle continuous state and action spaces, because they have infinite branching factors.
- Local search in continuous spaces is based on calculus (Newton & Leibniz)

Example:

- Suppose we want to place 3 new airports anywhere in Romania, such that the sum of squared distances from each city on the map to 3 airports is minimized.







$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

$$\nabla f = 0$$

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_i - x_c)$$

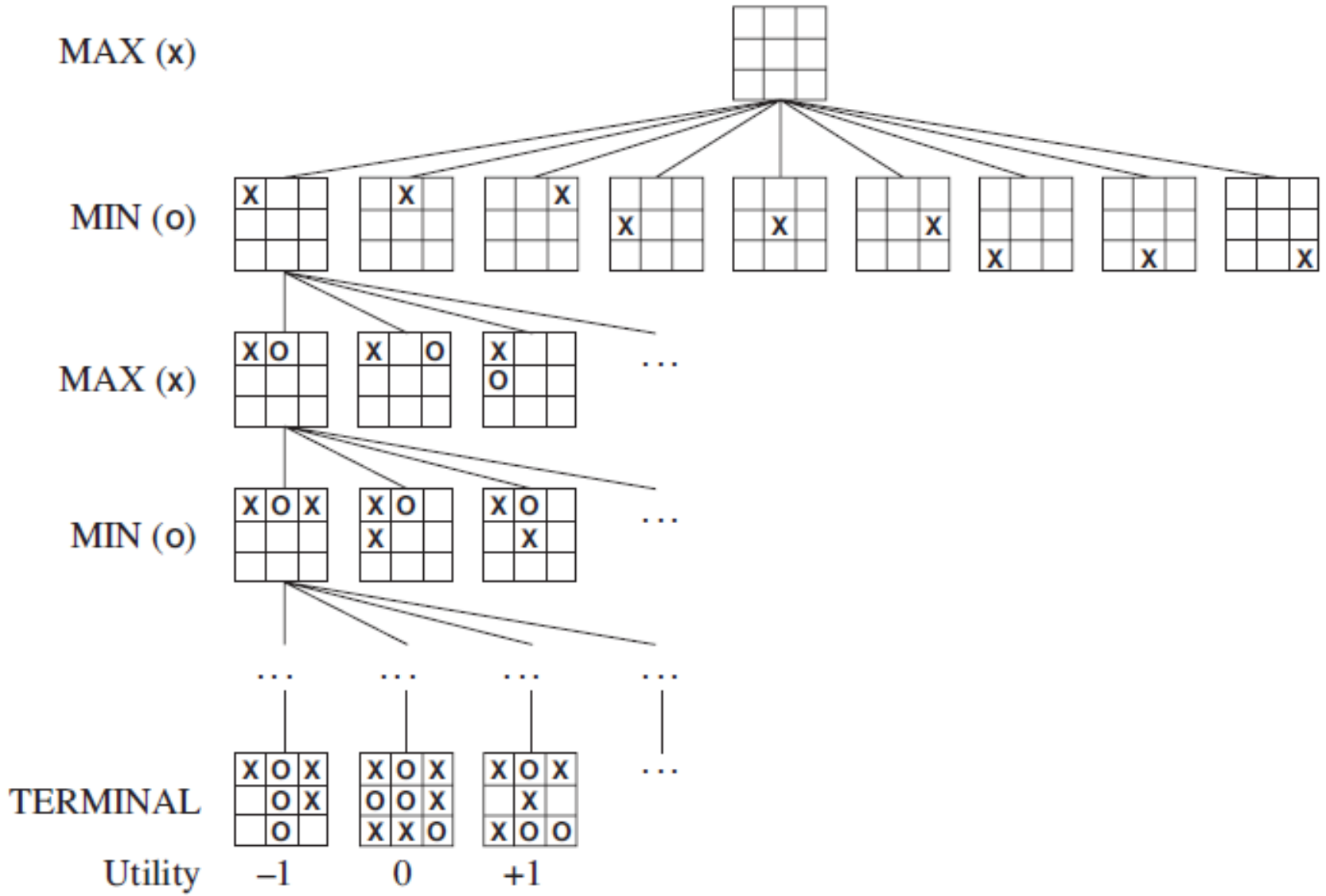
$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$$

MINIMAX Game

- We consider games with two players, whom we call **MAX** and **MIN** for reasons that will soon become obvious.
- MAX moves first, and then they take turns moving until the game is over.
- At the end of the game, points are awarded to the winning player and penalties are given to the loser.

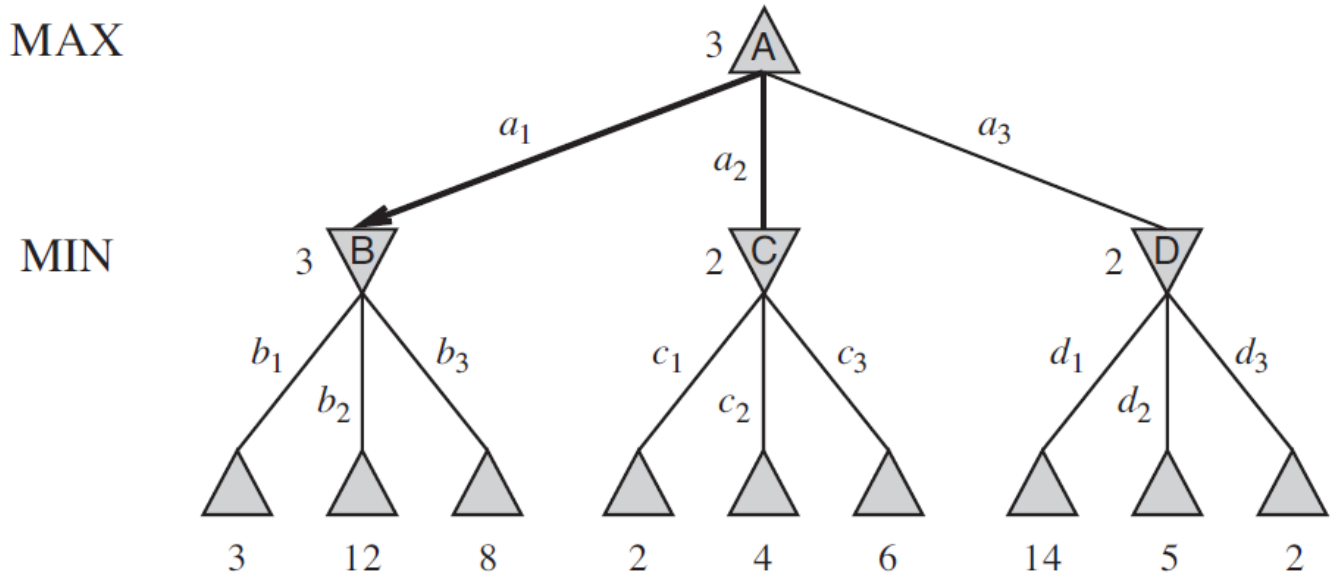
Example: **Tic-Tac-Toe**

- From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O.
- The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).


MAX (x)
MIN (o)
MAX (x)
MIN (o)
TERMINAL
Utility
–1
0
+1

Optimal Decisions in Games

Let's simplify the scenario of the two-ply game tree as follow:



- The Δ nodes are "MAX nodes," in which it is MAX's turn to move.
- The ∇ nodes are "MIN nodes."
- The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values.

- MAX's best move at the root is a1, because it leads to the state with the highest minimax value, and MIN's best reply is b1, because it leads to the state with the lowest minimax value.
- Given a game tree, the optimal strategy can be determined from the minimax value of each node, which we write as MINIMAX(n), i.e., the utility of node n.

We can derive the value of MINIMAX(s) of each state s by applying the following formula.

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Example:

$$\begin{aligned} MINIMAX(root) &= \max(\min(3,12,8), \min(2,4,6), \min(14,5,2)) \\ &= \max(3,2,2) \\ &= 3 \end{aligned}$$

- The minimax algorithm performs a complete depth-first exploration of the game tree.
- If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is $O(bm)$.

```
function MINIMAX-DECISION(state) returns an action
    return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s, a)))
    return v
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(s, a)))
    return v
```

Optimal decisions in multiplayer games

- Vector stores utility value of each player.
- Every player tries to maximize his/her utility.

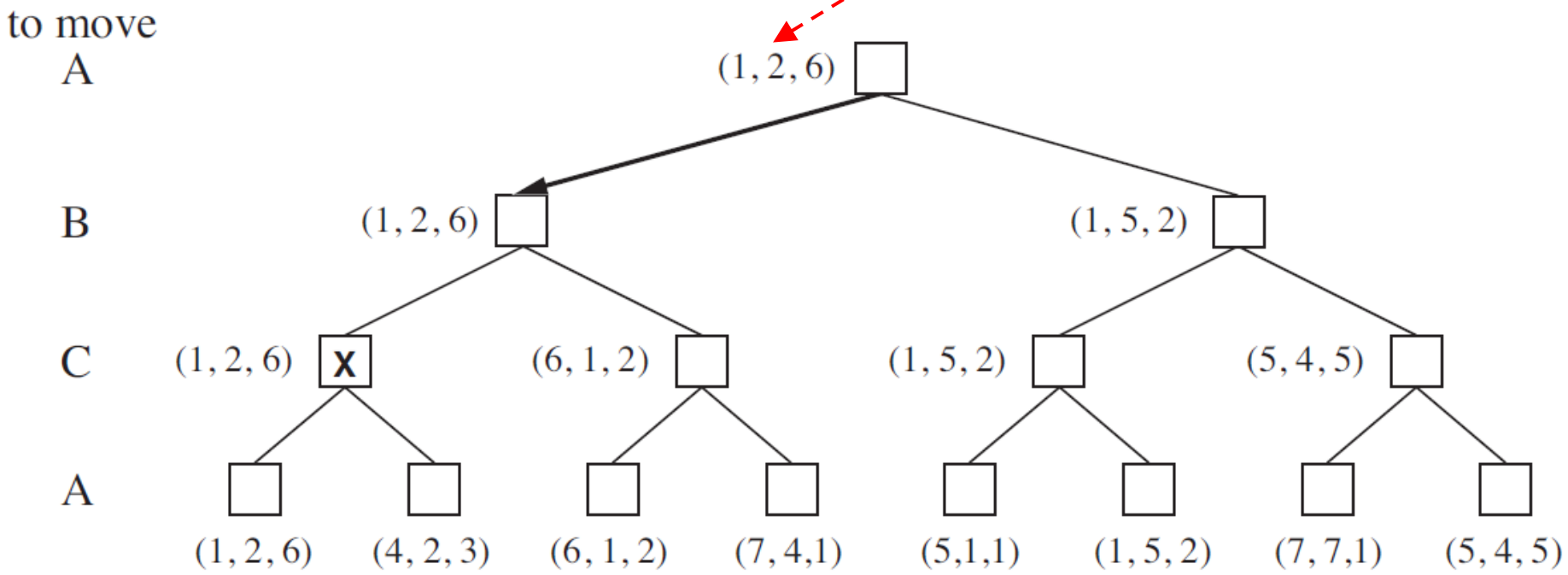


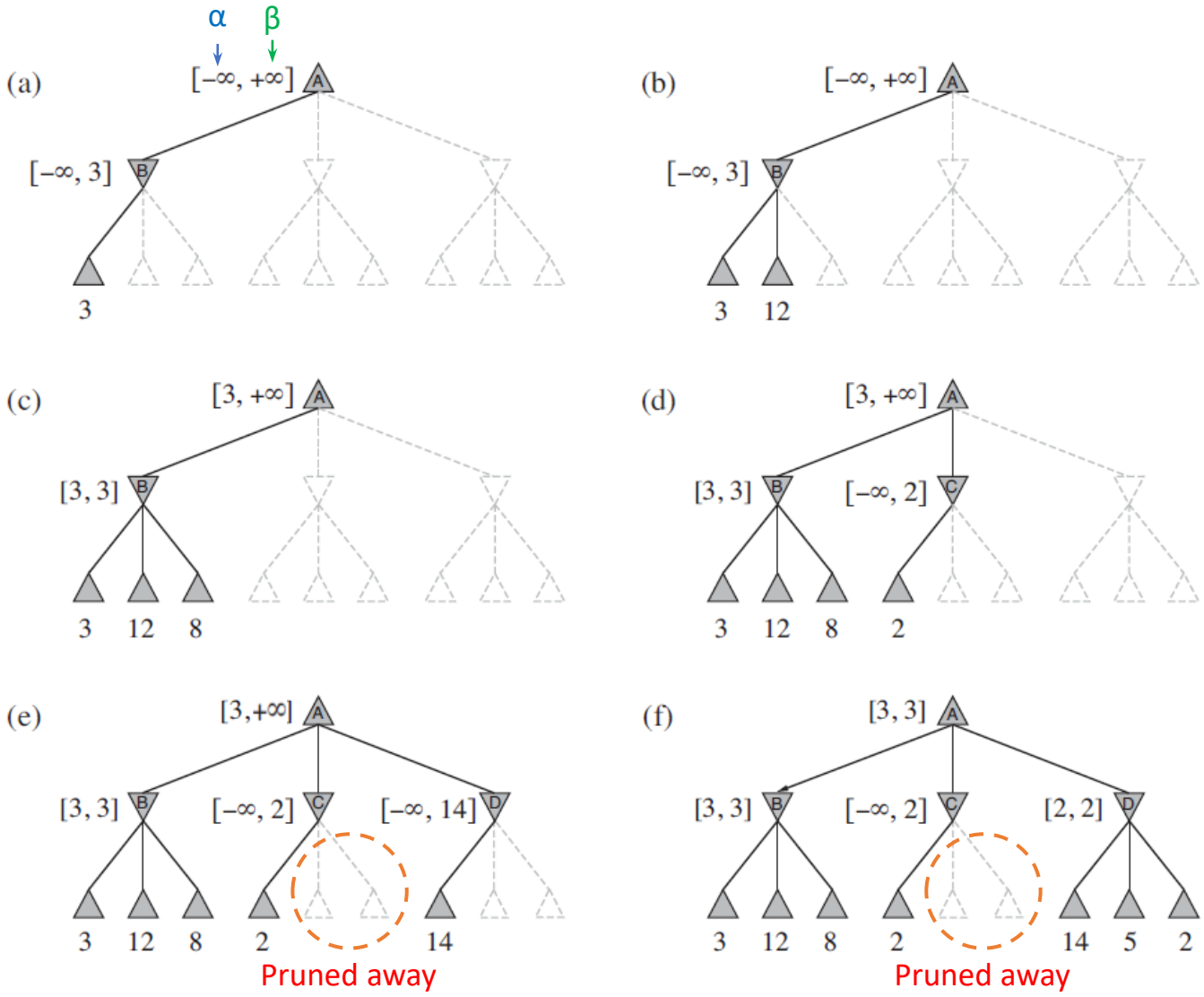
Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Alpha–Beta Pruning

- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.
- However, it is possible to compute the correct minimax decision without looking at every node in the game tree.

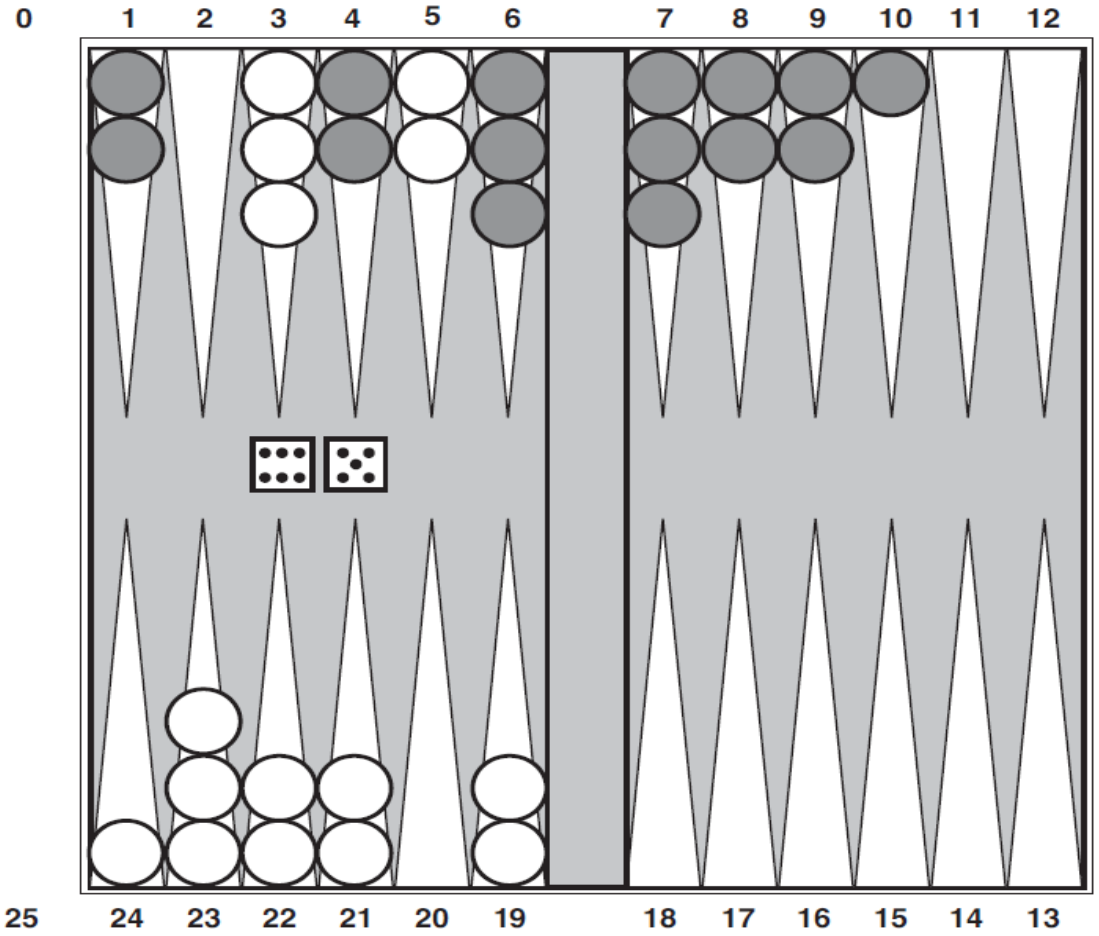
α: the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

β: the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

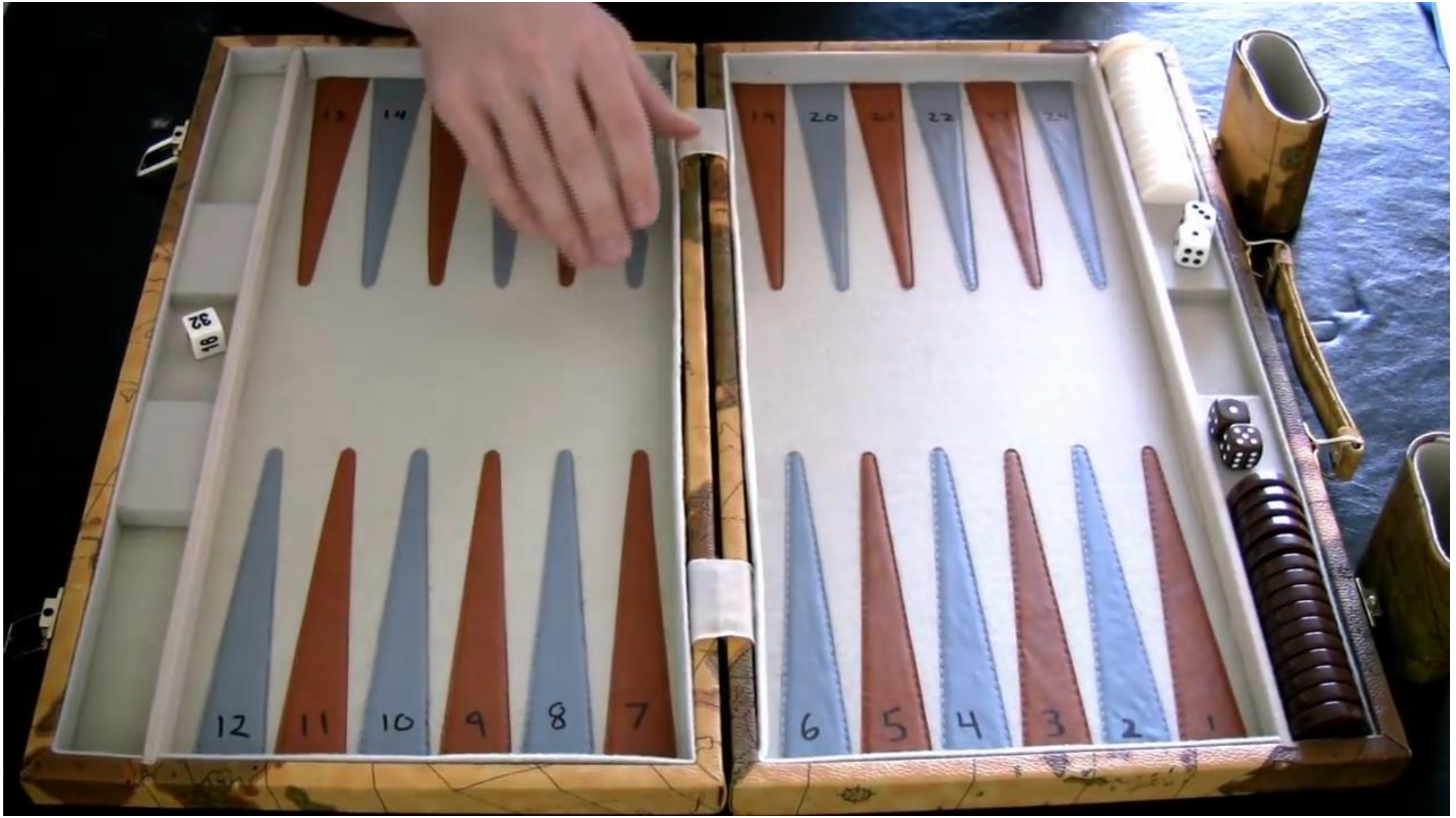


Stochastic Games

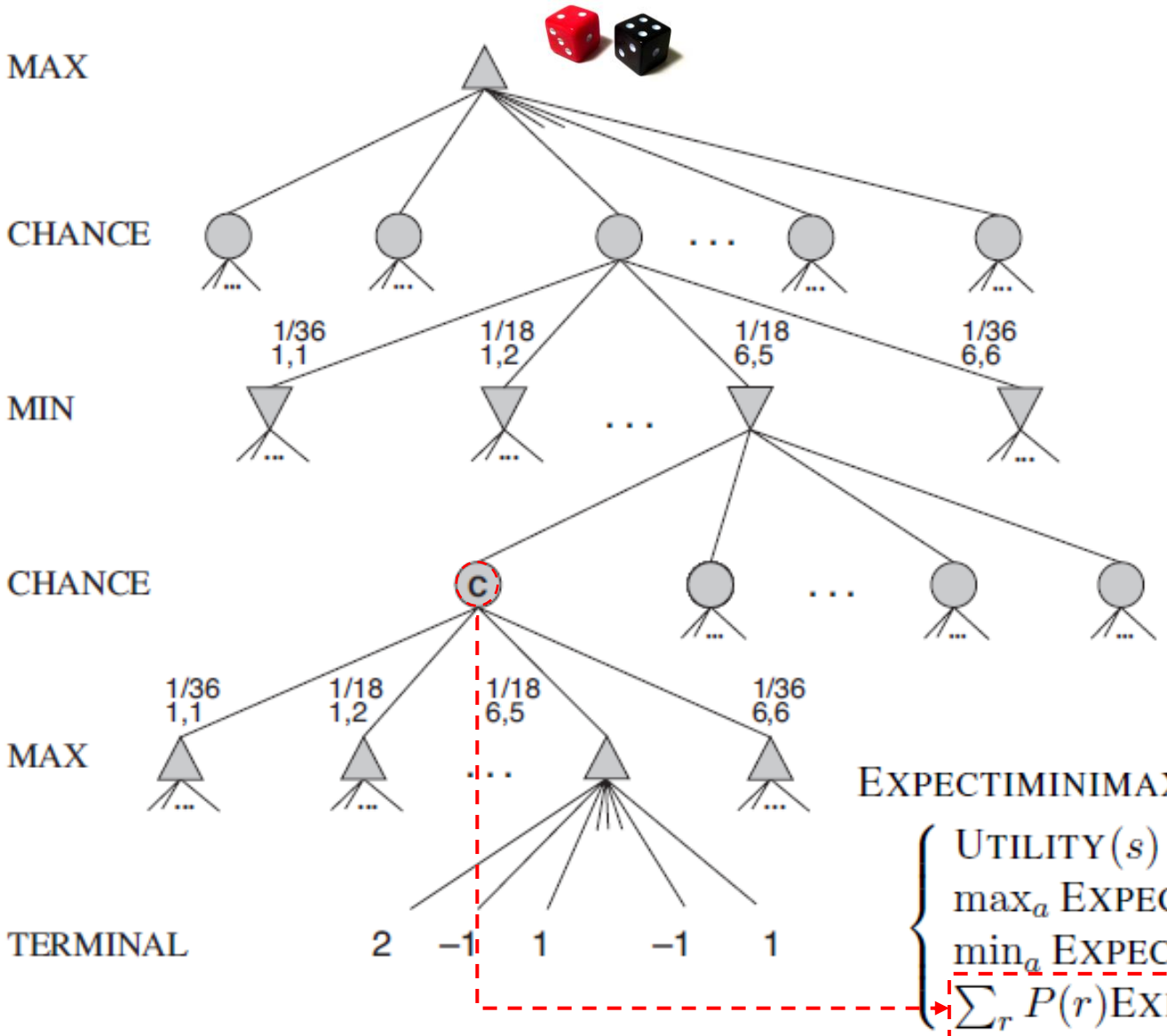
- Backgammon is a typical game that combines luck and skill.
- Dice are rolled at the beginning of a player's turn to determine the legal moves.
- The goal of the game is to move all one's pieces off the board.
- White moves clockwise toward 25, and Black moves counterclockwise toward 0.
- A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over.



Backgammon tutorial by Bucky: https://youtu.be/h0D0bQE_Lfc


12
11
10
9
8
7
6
5
4
3
2
1


11
10
9
7
5
4
3
2



- Although White knows what his or her own legal moves are, White does not know what Black is going to roll and thus does not know what Black's legal moves will be.
- That means White cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe.
- A CHANCE NODES game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes.

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}(\text{RESULT}(s,r)) & \text{if PLAYER}(s) = \text{CHANCE} \end{cases}$$

Where r represents a possible dice roll (or other chance event) and RESULT(s, r) is the same state as s, with the additional fact that the result of the dice roll is r.