AI基础

Lecture 3： Problem Solving: Informed Search & Search in Complex Environments

Bin Yang

School of Data Science and Engineering

byang@dase.ecnu.edu.cn

[Some slides adapted from Nikita Kitaev, UCB]

Lecture 2 ILOs

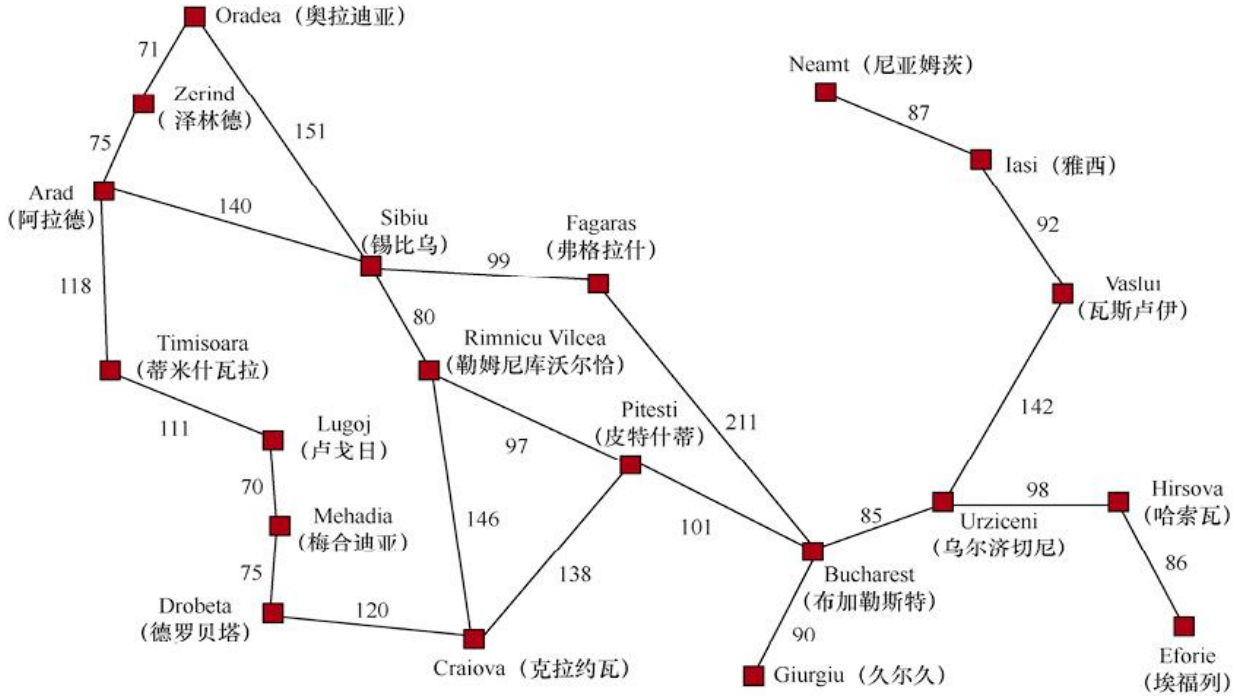
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

- Problem-solving agent
  - What is a problem-solving agent
  - Search problems and solutions
    - State space, initial state, goal states, action, transition model, action cost function
    - A solution: a path from the initial state to a goal state
- Example problems
  - Standardized problems
  - Real-world problems
- Search algorithms
  - Uninformed search vs. informed search
  - Performance measure: completeness, optimality, time complexity, space complexity
  - Best-First-Search: Which node from the frontier to expand next? Node with minimum f(n).
- Uninformed search
  - Breath-First-Search
  - Uniform cost search/Dijkstra's algorithm
  - Bidirectional search
  - Depth-First-Search
  - Iterative deepening search

Lecture 3 ILOs

- Informed Search Strategies
  - Heuristic function
  - Greedy best-first search
  - A* search, cost optimality, admissibility
  - Memory bounded search, weighted A* search
  - Design heuristics functions
- Search in complex environments
  - Local search
    - Hill climbing
    - Simulated annealing
    - Local beam search
    - Evolutionary search

Some concepts



- Arad to Bucharest
  - Known: with map
    - Three roads leaving Arad
  - Unknown: no map
- Informed search
  - The agent can estimate how “far” a state is from the goal
  - Among all three neighboring states of Arad, Sibiu is the closest to Bucharest
- Uninformed search
  - The agent has no idea about how “far” a state is from the goal

Outline

- Informed Search
  - Greedy best-first search
  - A* search
  - Admissible heuristics
  - Weighted A* search
  - Design heuristics functions

- Search in complex environment
  - Local search and optimization problem

Informed Search Strategies

- Uses domain-specific hints about the location of goals
  - Find solutions more efficiently than an uninformed strategy
- Heuristic function h(n)
  - h(n) returns estimated cost of the cheapest path from the state at node n to a goal state

- Example
  - Route-finding problems
  - Estimate the distance from the current state to a goal by computing the **straight-line distance** on the map between the two points.

Best-first search

- Which node from the frontier to expand next?
  - We choose a node n with the minimum value of evaluation function f(n)

- Each iteration
  - Choose a node on the frontier with minimum f(n) to expand/ stop
    - Priority queue
  - The node has not been reached before, or re-added

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Greedy best-first search

- Greedy best-first search is a special form of best-first search
  - f(n)=h(n)
- Greedy best-first search expands first the node with the lowest heuristic function h(n) value
  - The node that appears to be the closes to the goal

Greedy best-first Search

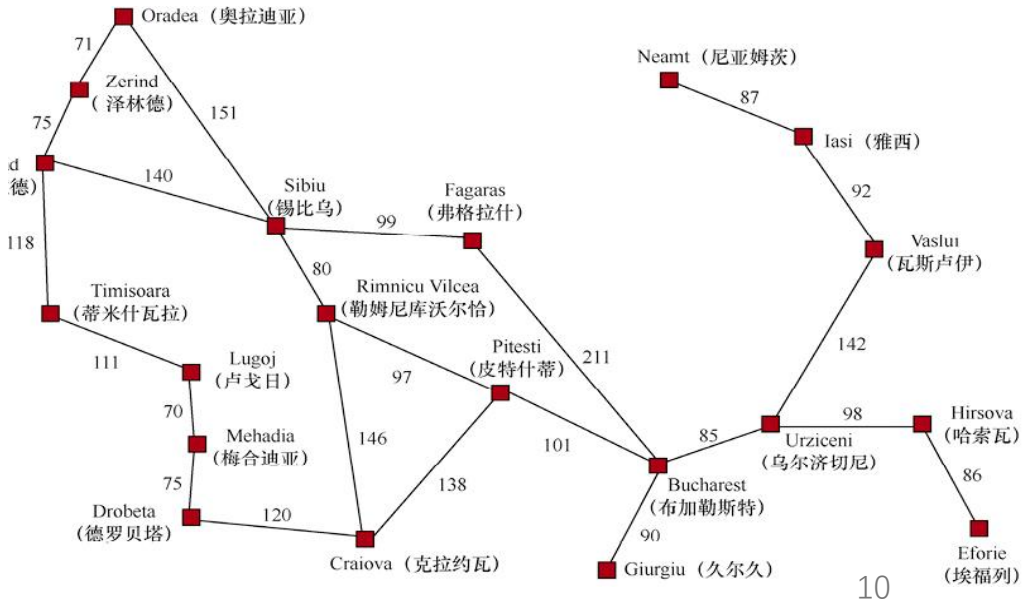
Example: route-finding in Romania

- Heuristic function uses the Straight-Line Distance $h_{SLD}()$
  - It takes a certain amount of world knowledge that straight-line distance is correlated with actual road distances
  - h_{SLD}(Sibiu)=253
  - More formal requirements will follow later

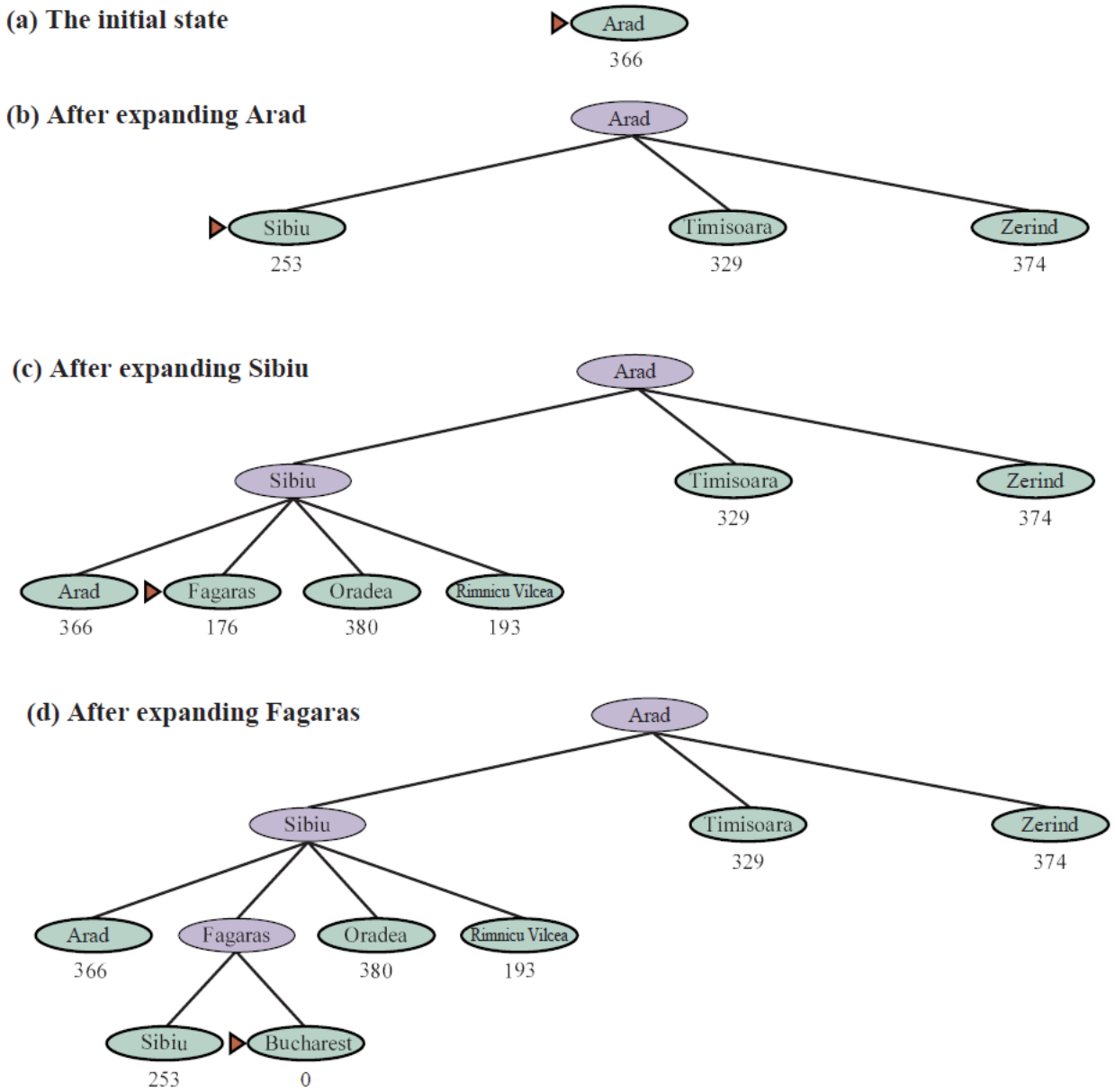
$h_{SLD}()$

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

Figure 3.16 Values of h_{SLD}—straight-line distances to Bucharest.

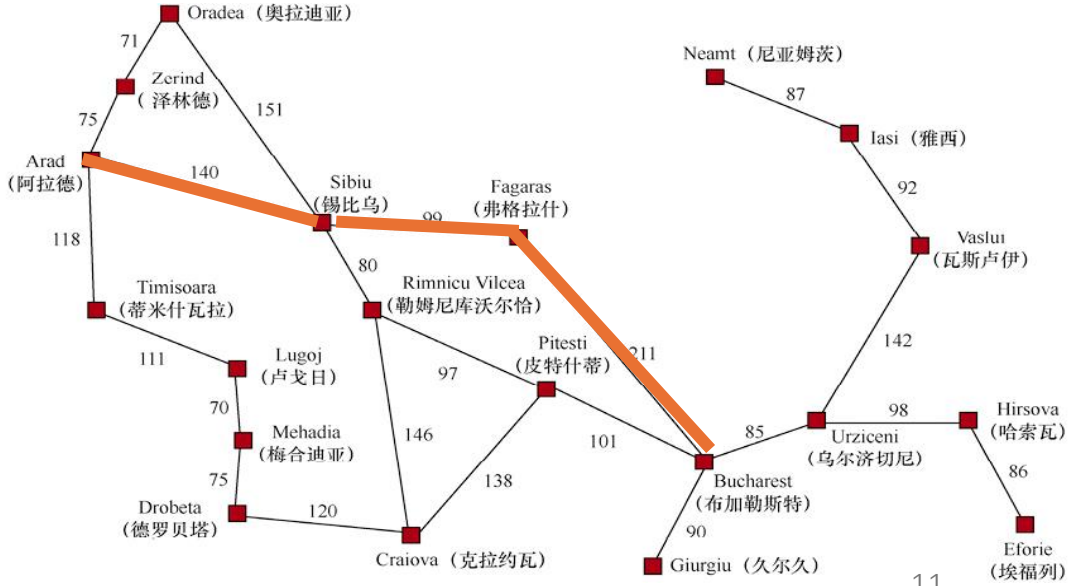


Greed best-first search with $h_{SLD}()$



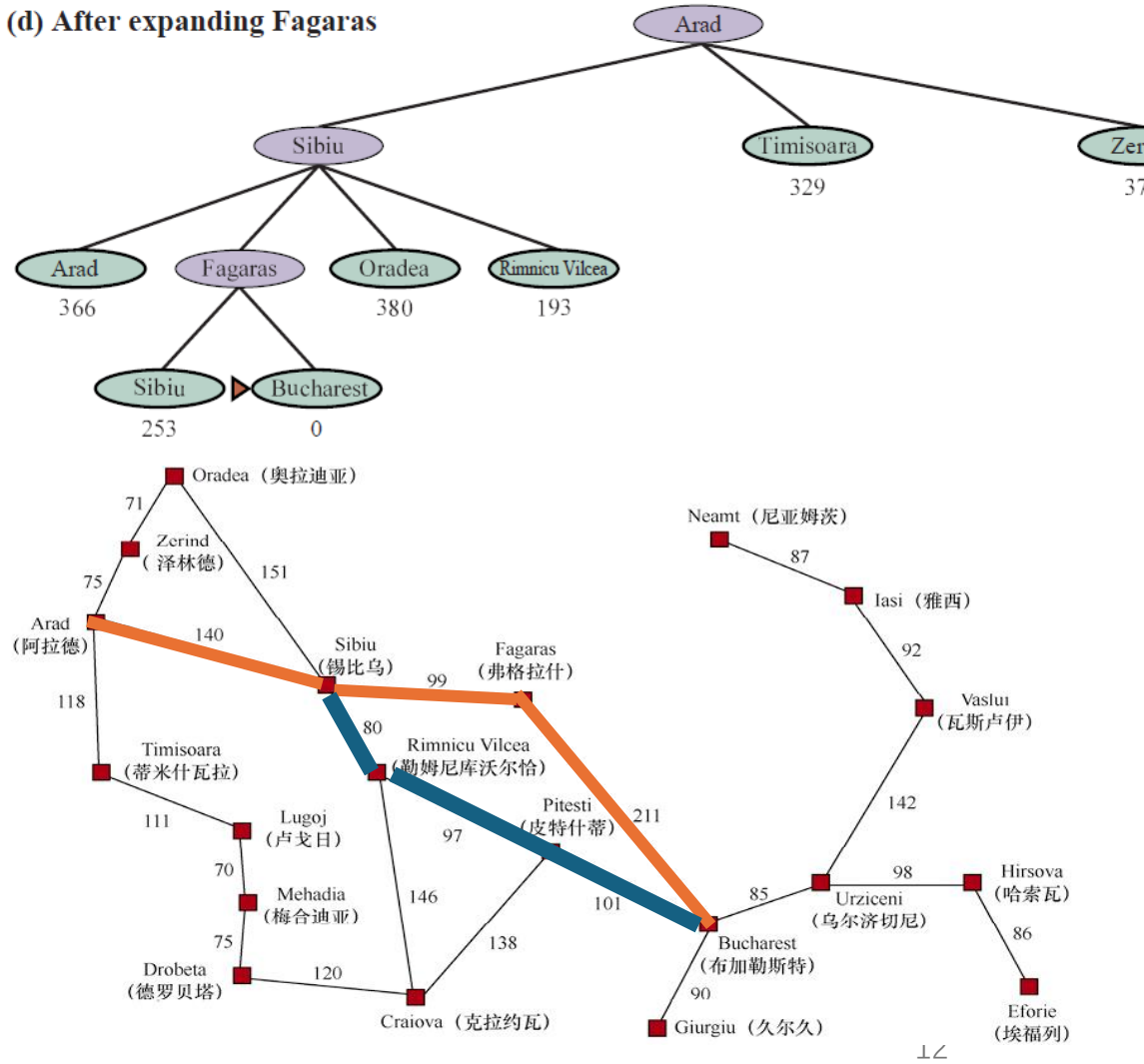
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

re 3.16 Values of h_{SLD}—straight-line distances to Bucharest.



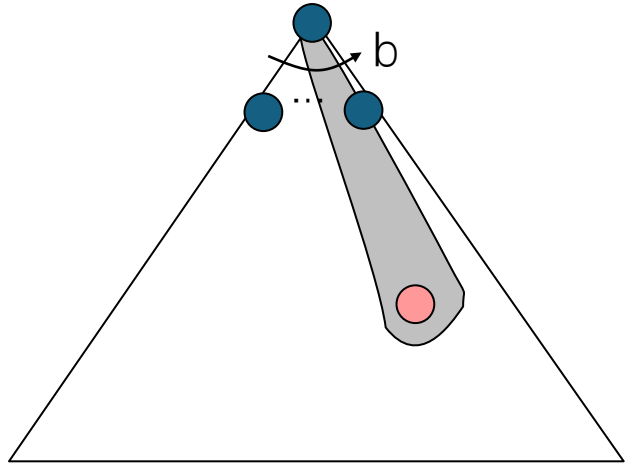
Greedy best-first search properties

- Complete in finite state spaces
  - Incomplete in infinite state spaces
- Cost optimality
  - No. 99+211=310<80+97+101=278
- Time and space complexity
  - O(|V|)

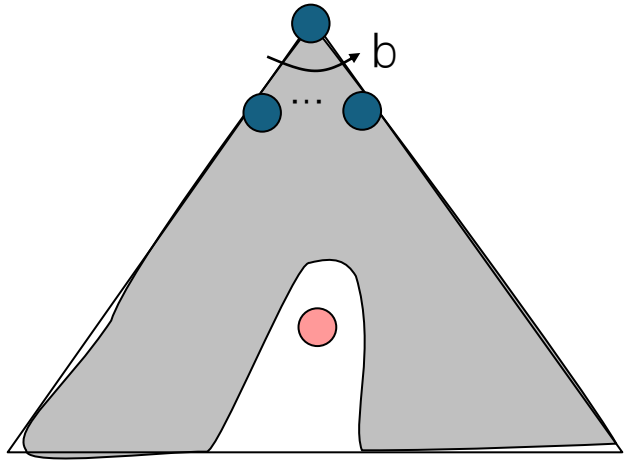


Greedy best-first Search

- Strategy: expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state



- A common case:
  - Best-first takes you straight to the (wrong) goal



- Worst-case: like a badly-guided DFS

Outline

- Informed Search
  - Greedy best-first search
  - A* search
  - Admissible heuristics
  - Weighted A* search
  - Design heuristics functions
- Search in complex environment
  - Local search and optimization problem

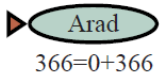
A* Search

A* Search

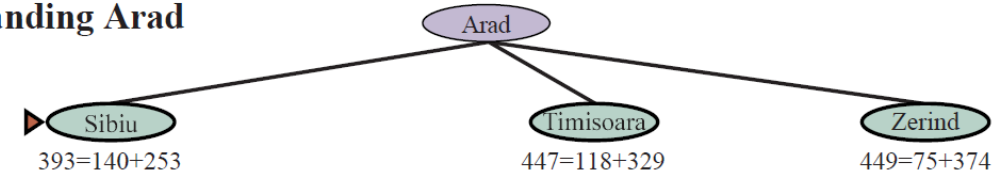
- Best-first search with a new evaluation function f(n)
- f(n)=g(n)+h(n)

- g(n) is the path cost from the initial state to node n
  - Based on the action costs
- h(n) is the estimated cost of the shortest path from n to a goal state
  - Based on the $h_{SLD}()$
- f(n) estimated cost of the best path that continues from n to a goal

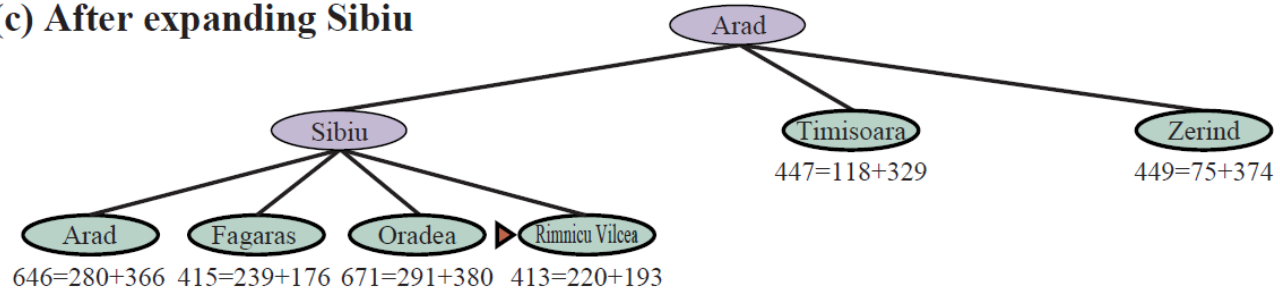
(a) The initial state



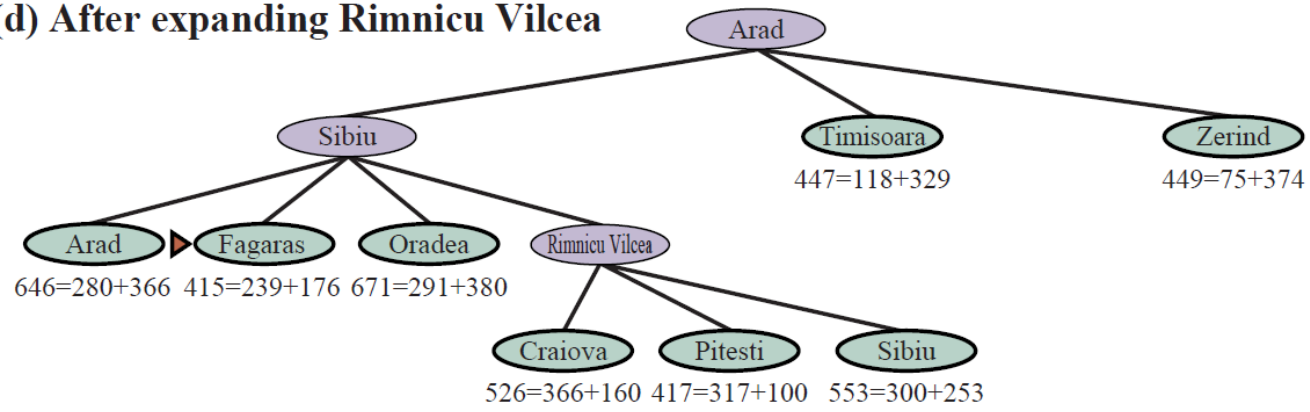
(b) After expanding Arad



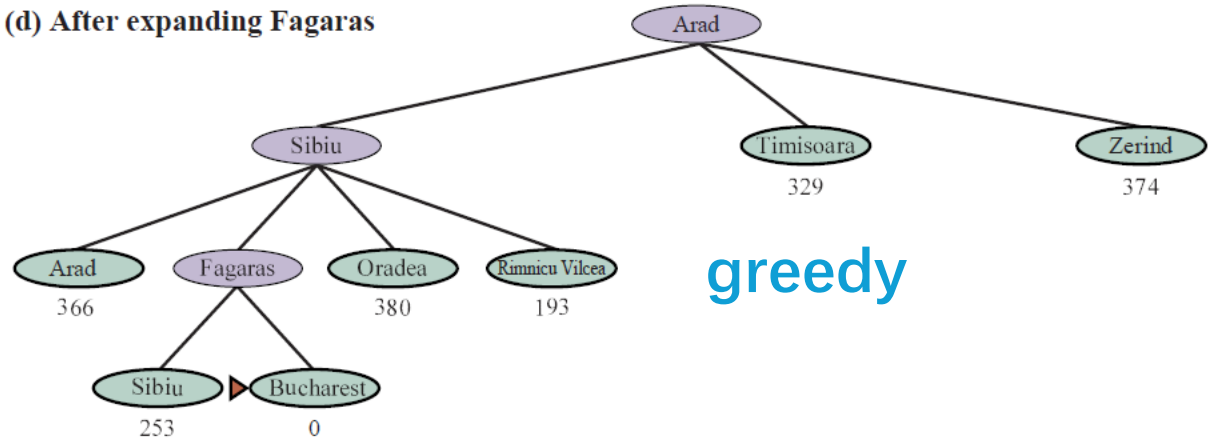
(c) After expanding Sibiu



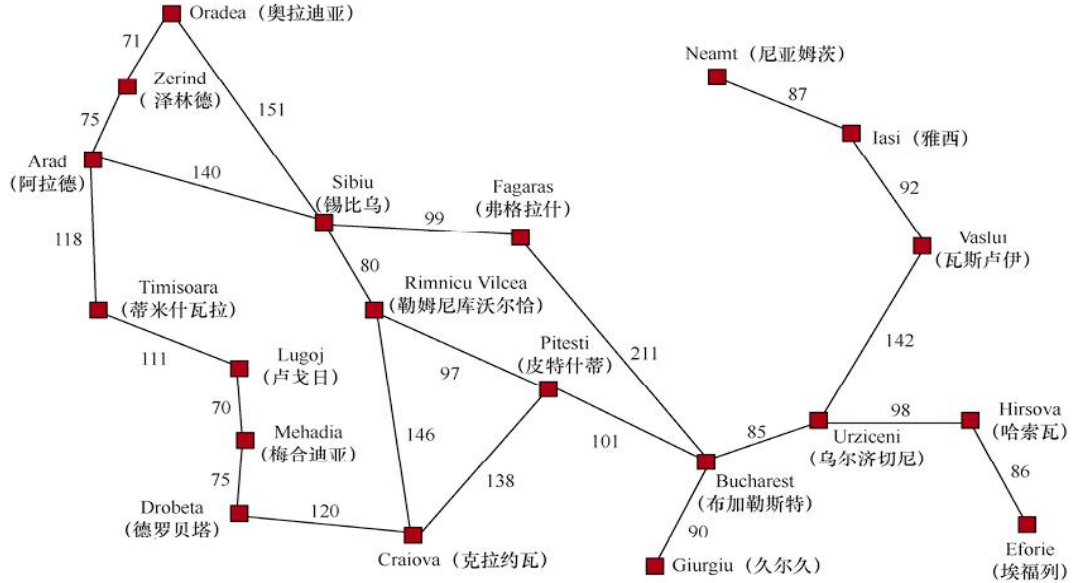
(d) After expanding Rimnicu Vilcea



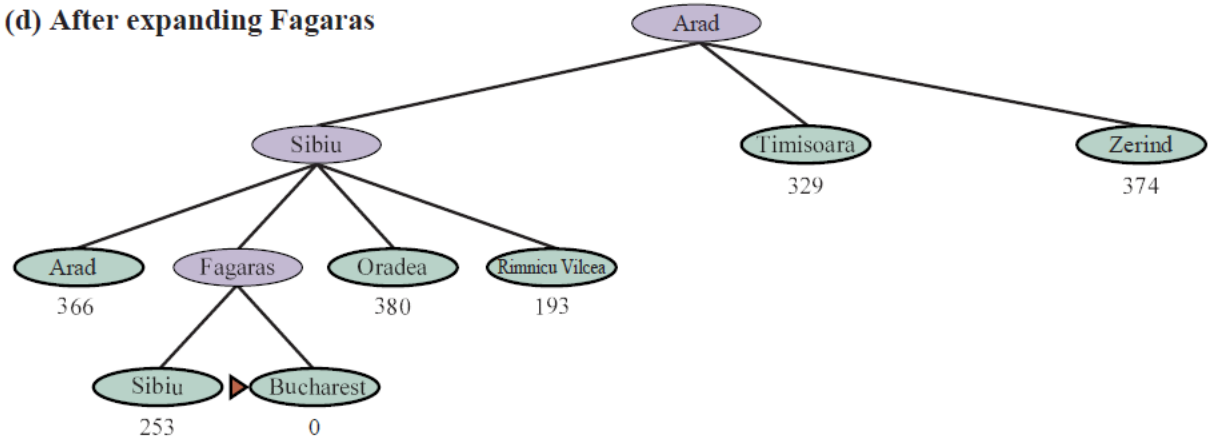
(d) After expanding Fagaras



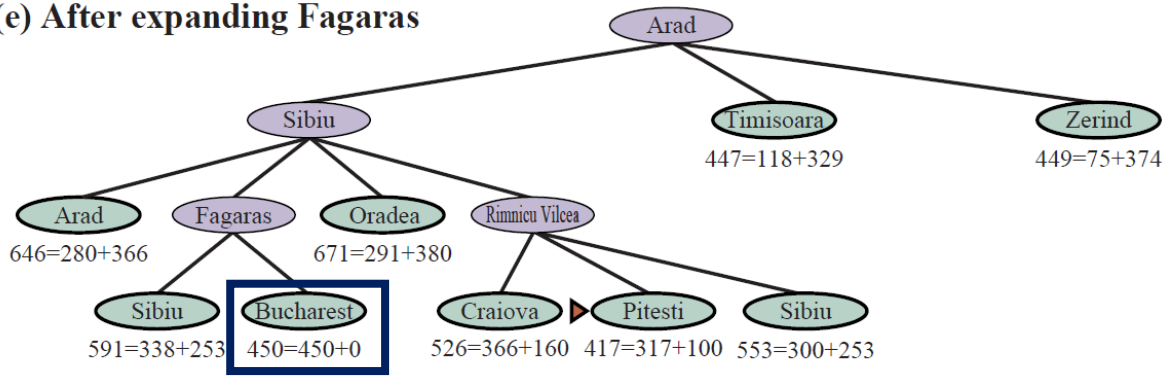
greedy

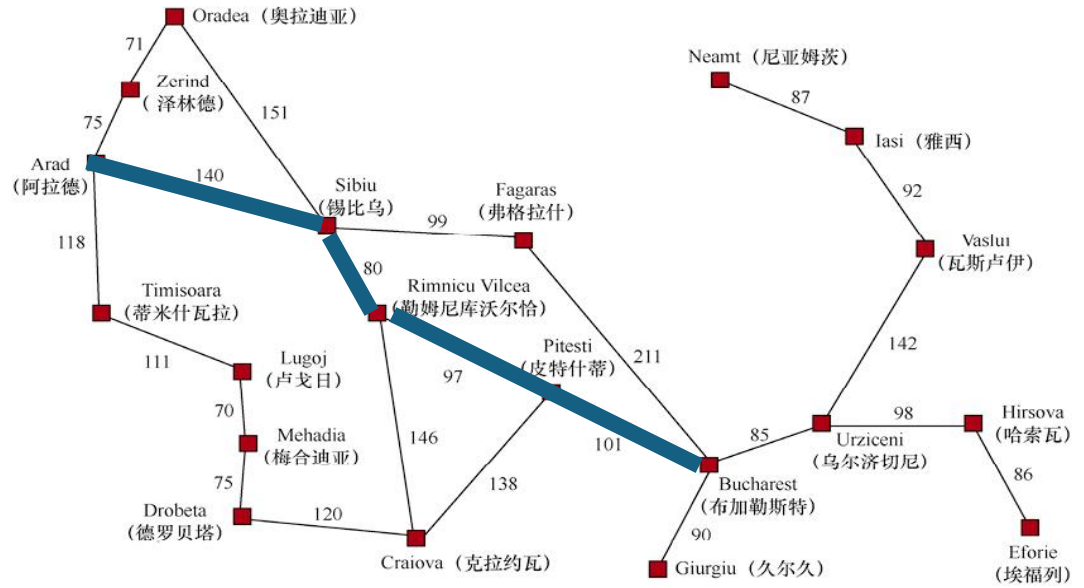


(d) After expanding Fagaras

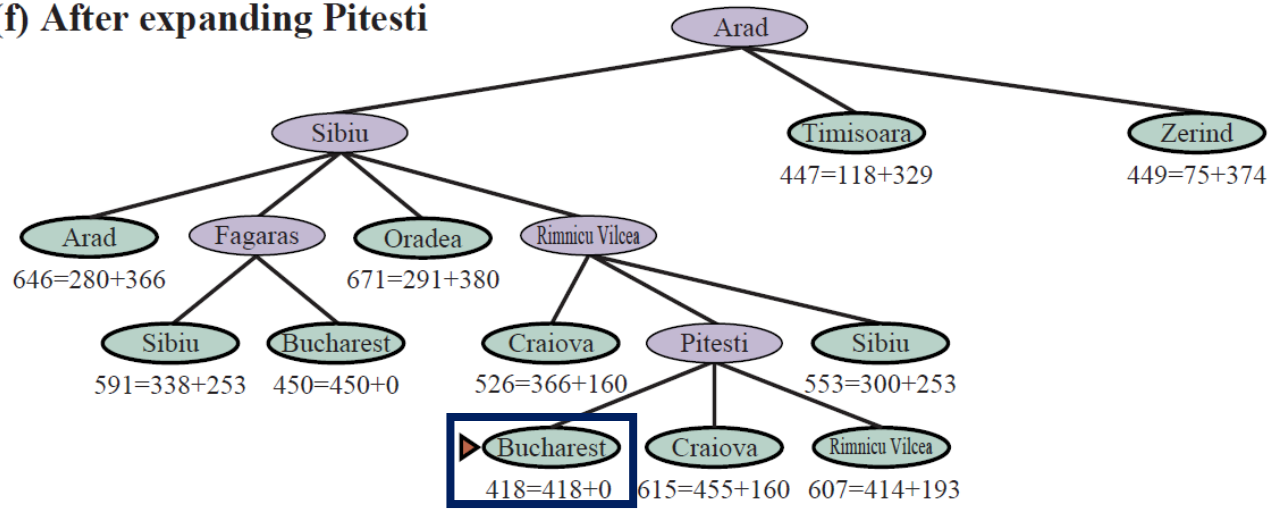


(e) After expanding Fagaras





(f) After expanding Pitesti



Outline

- Informed Search
  - Greedy best-first search
  - A* search
  - Admissible heuristics
  - Weighted A* search
  - Design heuristics functions
- Search in complex environment
  - Local search and optimization problem

A* properties

- Complete
- Cost optimality?
  - Depends on the heuristics function h(n)
  - Yes, when admissibility
    - Never overestimates the cost to reach a goal.
- Proof by contradiction
  - Suppose the optimal path has cost C*, but A* search returns a path with cost C>C*
  - Then, there must be some node n which is on the optimal path and is unexpanded
  - Because if all the nodes on the optimal path had been expanded, then A* would have returned the optimal solution already.
  - g*(n): the cost of the optimal path from the state to n
  - h*(n): the cost of the optimal path from n to the nearest goal

Cost optimality proof

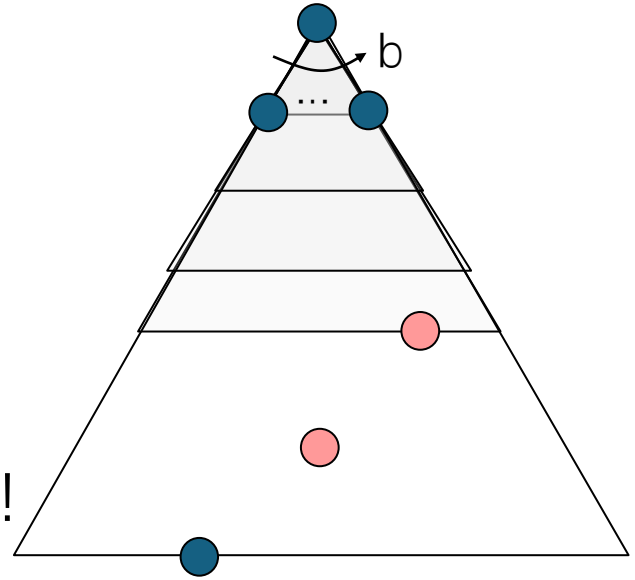
- f(n)>C* (due to the assumption, where some node n which is on the optimal path is unexpanded. If f(n)<=C*, n would have been expanded already)
- f(n)=g(n)+h(n) (by definition)
- f(n)=g*(n)+h(n) (because n is on an optimal path)
- f(n)<= g*(n)+h*(n) (because admissible heuristics, h(n)<=h*(n))
- f(n)<= C* (by definition, C*=g*(n)+h*(n))
- The first line and last line form a contradiction. So, the assumption that A* search returns a suboptimal path is wrong.

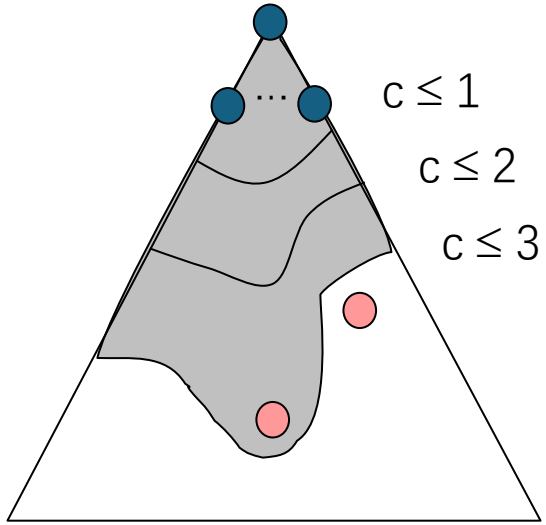
What about inadmissible heuristics

- A^* may or may not be cost-optimal
- When A^* with inadmissible heuristics is still cost-optimal?
  - If there is even one cost-optimal path on which h(n) is admissible for all nodes n on the path, then the path will be found.
    - No matter what the heuristic says for states off the path.
  - If optimal solution has cost C^*, and the second best has cost C_2, and if h(n) overestimates some costs, but never by more than C_2–C^*, then A^* is guaranteed to return cost-optimal solutions.

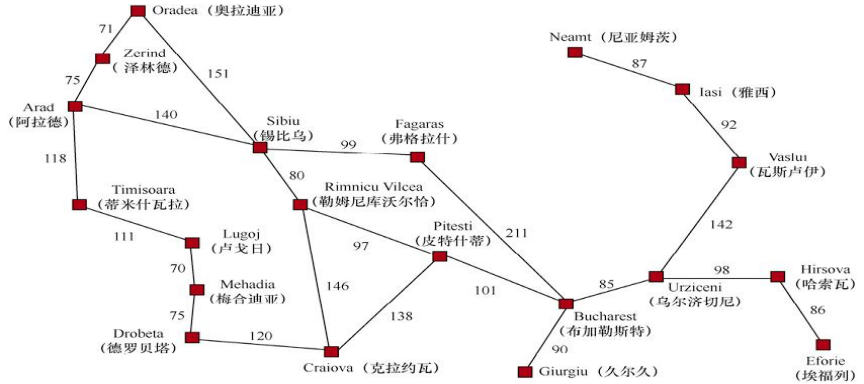
Search Contours

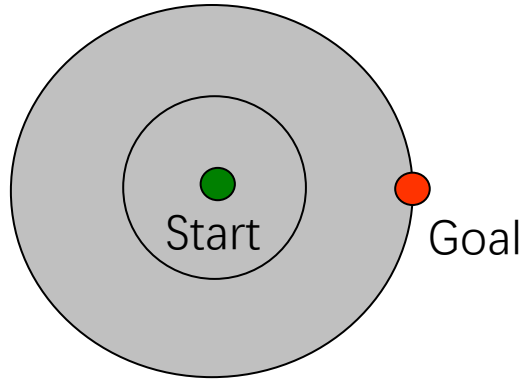
- UCS explores increasing cost contours
  - Cost contours based on g(n)





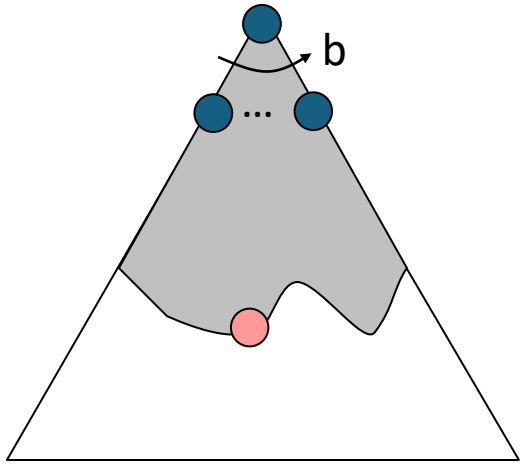
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every “direction”
  - No information about goal location
- We’ll fix that soon!

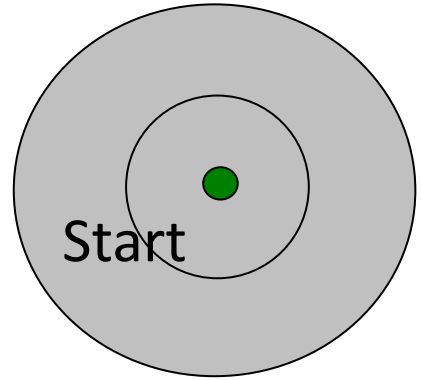




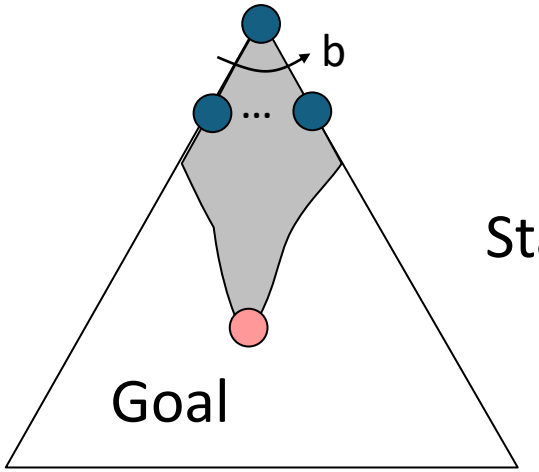
UCS vs A* Contours

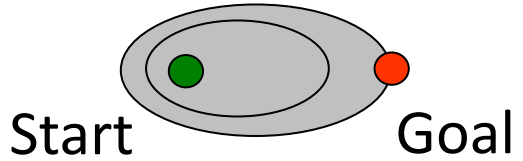
- Uniform-cost expands equally in all "directions" to ensure optimality
  - Cost contours based on g(n)
  - Contours are circular



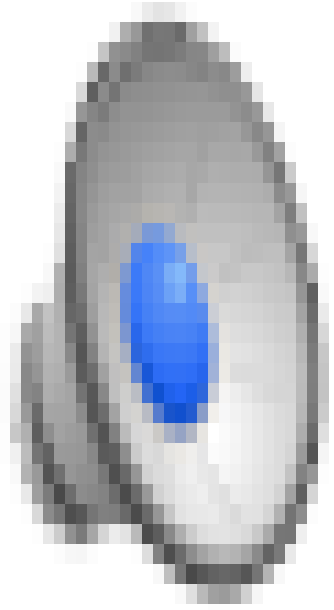


- A* expands mainly toward the goal, while also ensure optimality
  - Cost contours based on g(n)+h(n)
  - Contours will stretch toward a goal

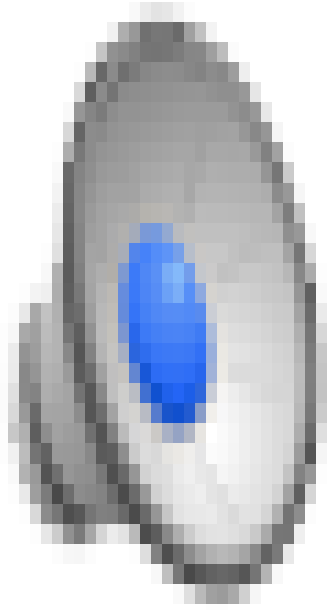




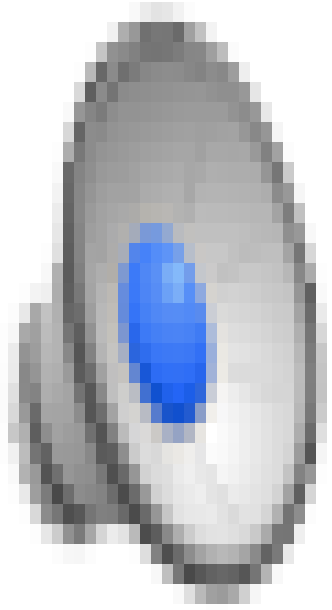
Video of Demo Contours (Empty) -- UCS



Video of Demo Contours (Empty) – A*



Video of Demo Contours (Empty) -- Greedy

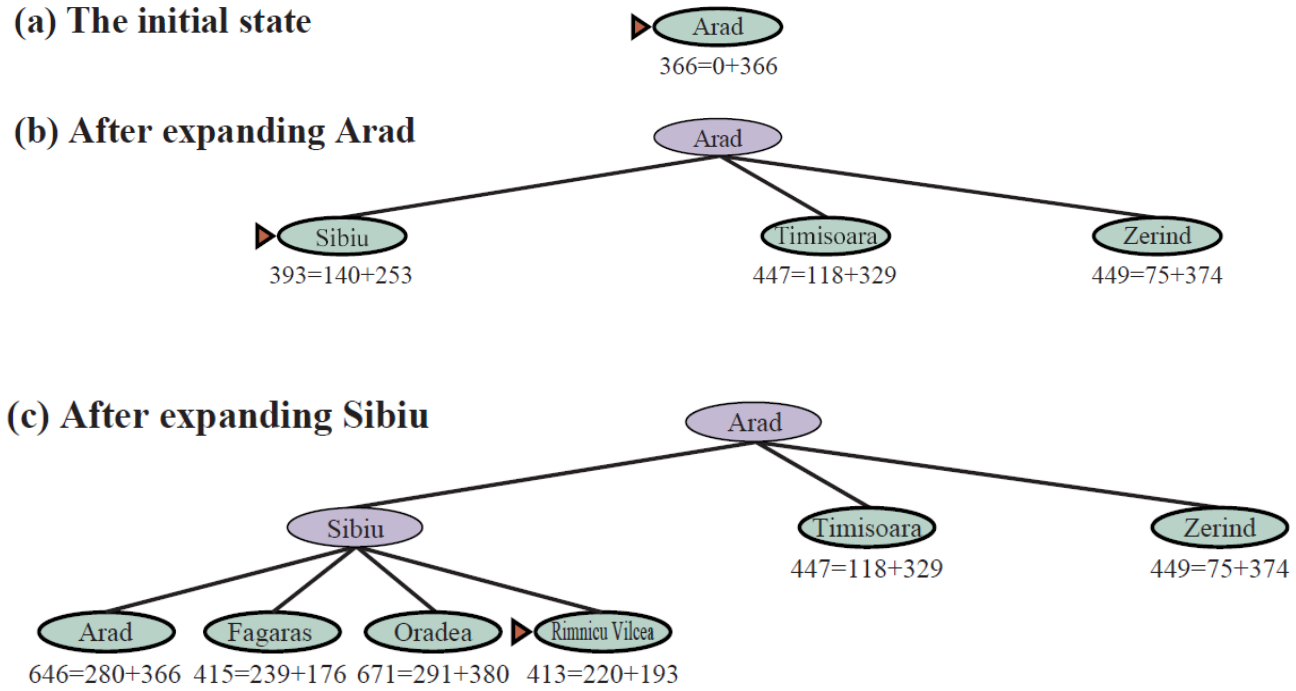


A* properties

- A* expands all nodes that can be reached from the initial state on a path where every node on the path has $f(n)<C^*$.
  - We say these surely expanded nodes
  - Potential problems: large memory requirements
    - All nodes with $f(n)<C^*$

- A* never expands nodes with $f(n)> C^*$
  - Pruning: an important concept in many areas of AI

Pruning example



- T and Z are pruned
- In UCS, T and Z are expanded.

Memory bounded search

- Beam search
  - Frontier only maintains the nodes with top-k best f-scores
  - UCS and A* spreading out everywhere in concentric contours
    - Cost contours based on g(n) or g(n)+h(n)
  - Beam search explores only a focused portion of those contours
    - The portion has the k best f-scores
- Iterative-deepening A*
  - Iterative-deepening DFS, each time the depth +1
  - Iterative-deepening A*, each time the f-cost, i.e., g+h value, +1

Outline

- Informed Search
  - Greedy best-first search
  - A* search
  - Admissible heuristics
  - Weighted A* search
  - Design heuristics functions
- Search in complex environment
  - Local search and optimization problem

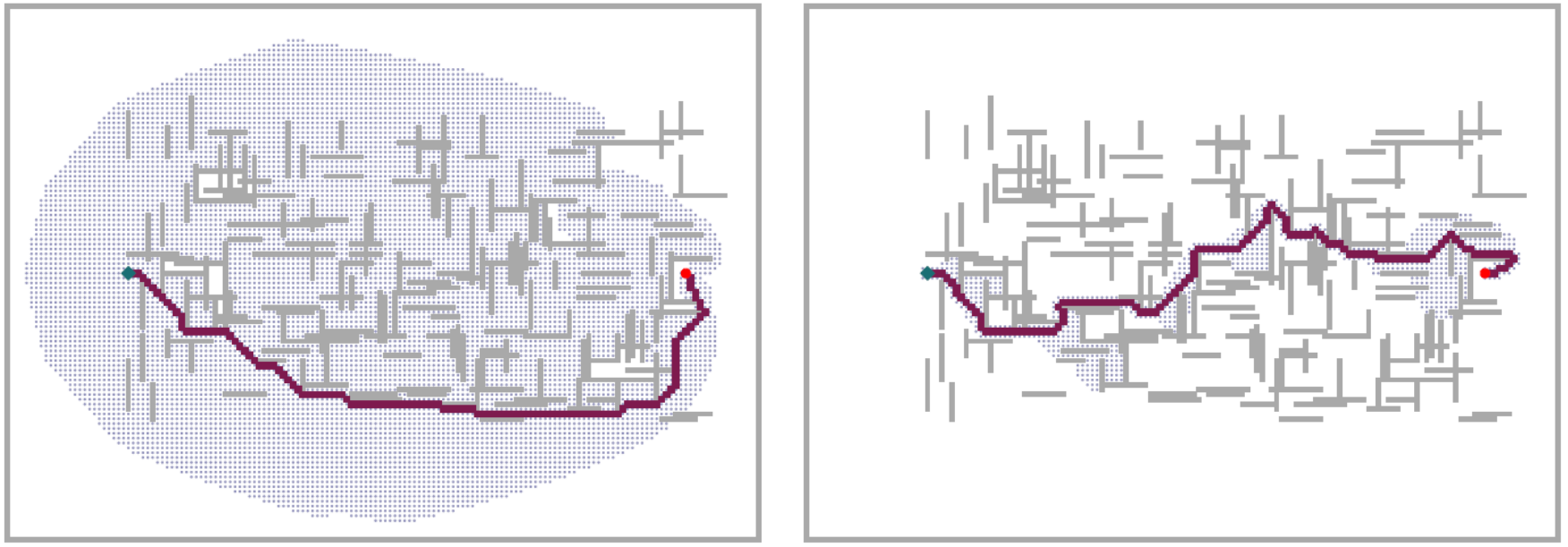
Satisficing search

- Take less time and space, if we are willing to take **good enough** solutions, but not always looking for “optimal solutions.”
  - Use inadmissible heuristics
- Detour index
  - A multiplier to the straight-line distance due to curvature of roads
  - E.g., 1.2 to 1.6

- Weighted A* with a weight W on heuristic function
  - f(n)=g(n)+W*h(n)

Example

f(n)=g(n)+W*h(n)



(a) (b)

Figure 3.21 Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight $W = 2$. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly.

Weighted A* often runs much faster, but may not find an optimal solution.

Weighted A*

$$\begin{aligned}
\text{A* search:} &\quad g(n) + h(n) &\quad (W = 1) \\
\text{Uniform-cost search:} &\quad g(n) &\quad (W = 0) \\
\text{Greedy best-first search:} &\quad h(n) &\quad (W = \infty) \\
\text{Weighted A* search:} &\quad g(n) + W \times h(n) &\quad (1 < W < \infty)
\end{aligned}$$

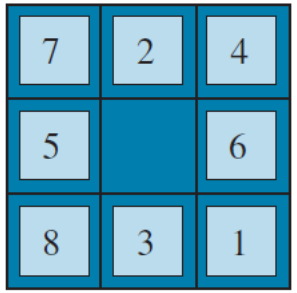
Outline

- Informed Search
  - Greedy best-first search
  - A* search
  - Admissible heuristics
  - Weighted A* search
  - Design heuristics functions
- Search in complex environment
  - Local search and optimization problem

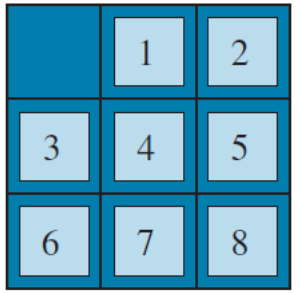
Heuristic functions

- How heuristic functions can be invented?
  - Generating heuristics from relaxed problems
  - Generating heuristics from subproblems, pattern databases
  - Generating heuristics from with landmarks
  - Learning heuristics from experience

8-puzzle problem



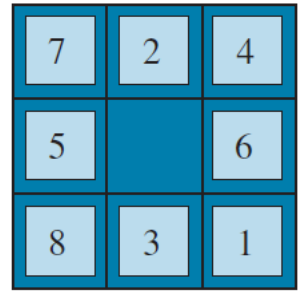
Start State



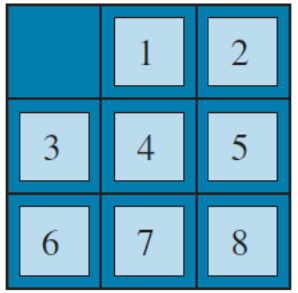
Goal State

- A tile can move from square X to square Y if
  - X is adjacent to Y and Y is blank
- State space
  - A set of possible states that the environment can be in. A state description specifies the location of each of the tiles.
- Initial state
  - Where the agent starts. Any state.
- Goal states
  - Any state. Often a state with the numbers in order.
- Actions
  - Blank moving *Left, right, up, down.*
- Transition model
  - From one state to a new state. Start State, Left: 5 and the blank switched.
- Action cost function
  - Each action costs 1.

From relaxed problems



Start State

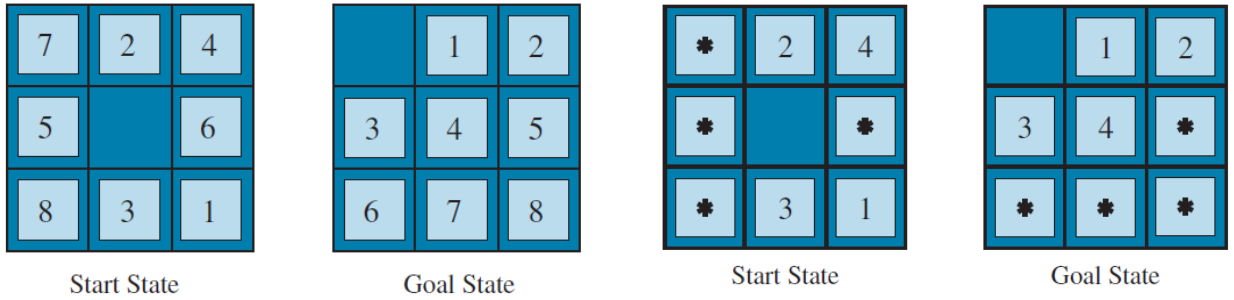


Goal State

N!

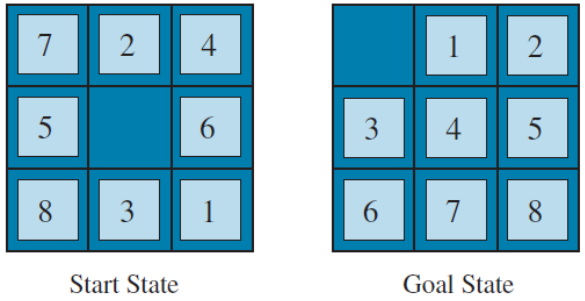
- 8-puzzle problem
- A tile can move from square X to square Y if
  - X is adjacent to Y and Y is blank
- Relaxed problems
  - Removing one or both of the conditions
- RP1: A tile can move from square X to square Y
  - $h_1()$ returns the number of misplaced tiles, 8
  - Admissible, each misplaced tile requires at least one move to the right place
- RP2: A tile can move from square X to square Y if X is adjacent to Y
  - $h_2()$ returns the sum of the Manhattan distance of the tiles from their goal locations
    - 3+1+2+2+2+3+3+2=18
  - Admissible, move one tile one step closer to the goal
- RP3: A tile can move from square X to square Y if Y is blank
- When a collection of admissible heuristics is available, choose the most accurate one
  - $h(n)=\max\{h_1(), h_2(), \cdots, h_k()\}$

From subproblems



- Subproblem
  - Same problem, smaller in size
  - Getting tiles 1, 2, 3, 4 and blank to the correct positions, ignoring other tiles
- Cost of the optimal solution of a subproblem is a lower bound on the cost of the complete problem
- Store exact solution costs for every possible subproblem instance into a database
- Different subproblems, multiple databases
  - 1-2-3-4, 2-4-5-6, 2-5-8, ⋯
  - $h(n)=\max\{h_1(), h_2(), \cdots, h_k()\}$

Learning heuristics from experience



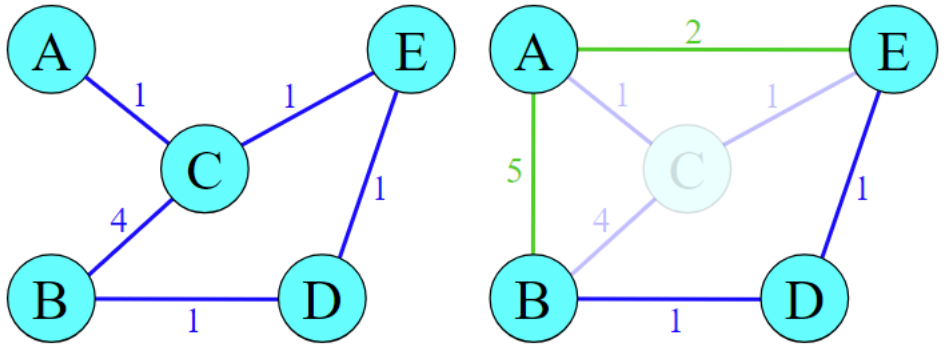
- Experience
  - Solving lost of 8-puzzle problems
  - Many optimal solutions, each corresponds to a path
  - Learn a function to approximate the true path cost for other states
- State feature based learning
  - Some machine learning methods work better with “features” of a state rather than the raw state description
  - $x_1(n)$: number of misplaced tiles
    - When x1(n)=5, the average cost is around 14.
  - $X_2(n)$: number of pairs of adjacent tiles that are not adjacent in the goal state

$$h(n) = c_1 x_1(n) + c_2 x_2(n).$$

From landmarks

- Route planning
- Use landmarks
  - For each landmark L and each other vertex v in the graph
  - Compute and store the exact cost of the optimal path from v to L, denoted as C*(v, L), and from L to v, denoted as C*(L, v)
  - Inadmissible though.
- Add shortcuts
  - Contraction hierarchies

$$h_L(n) = \min_{L \in Landmarks} C^*(n, L) + C^*(L, goal)$$



Simon Aagaard Pedersen, Bin Yang, Christian S. Jensen: Fast stochastic routing under time-varying uncertainty. VLDB J. 29(4): 819-839 (2020)
Chenjuan Guo, Bin Yang, Jilin Hu, Christian S. Jensen, Lu Chen: Context-aware, preference-based vehicle routing. VLDB J. 29(5): 1149-1170 (2020)

Outline

- Informed Search
  - Greedy best-first search
  - A* search
  - Admissible heuristics
  - Weighted A* search
  - Design heuristics functions

- Search in complex environment
  - Local search and optimization problem

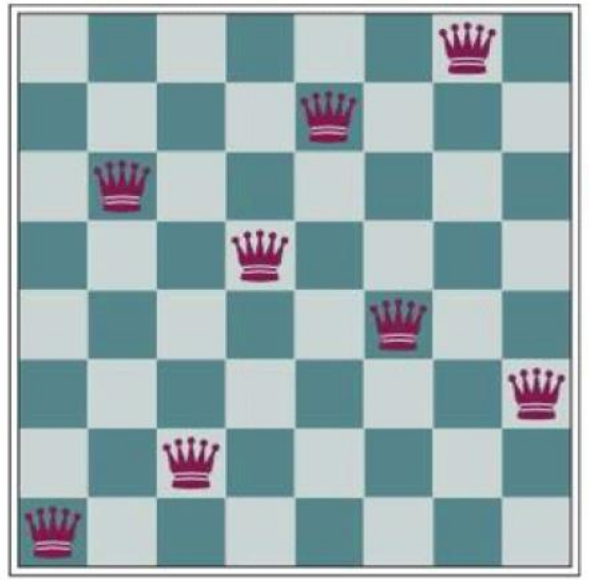
Search in complex environment

- So far, the simplest environment:
  - Episodic, single agent, fully observable, deterministic, static, discrete, and known.
  - A solution is a sequence of actions.
- We relax the simplifying assumptions, to get closer to the real world.
  - Finding a good state without worrying about the path to get there, covering both discrete and continuous states.
  - Relaxing determinism to nondeterministic states.
  - Partial observability.
  - Unknown environment.

Local search and optimization problem

- Sometimes we care only about the final state, but not the path to get there.
  - because if you know the final state, it is trivial to reconstruct the steps that created it.

- The 8-queens problem
  - Place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.)

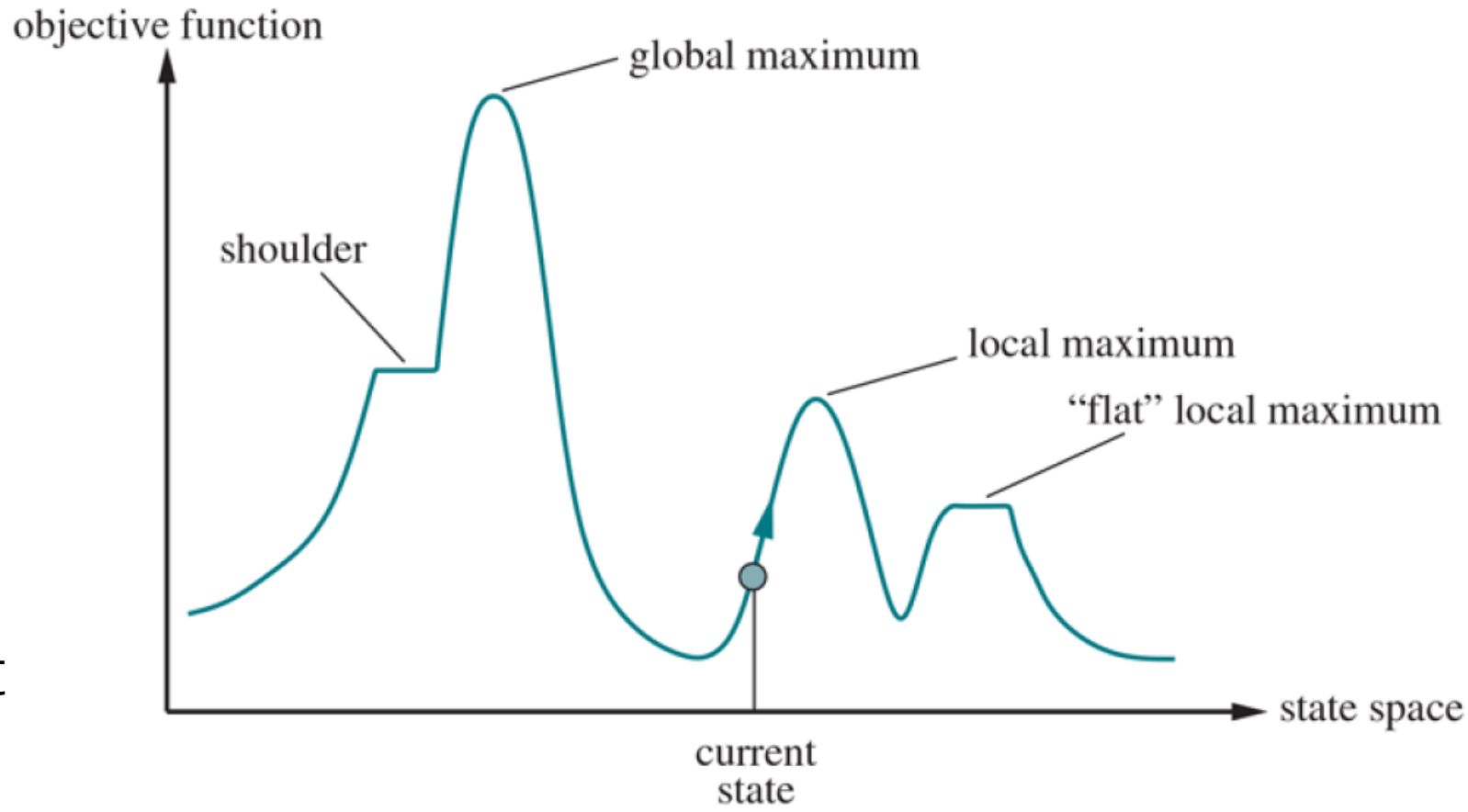


Local search

- Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached.
  - Use very little memory
  - Find reasonable solutions in large or infinite state spaces
- Local search can also solve optimization problems, in which the aim is to find the best state according to an objective function.

Local search

- Each state has an elevation, defined by the value of the objective function.
- The aim is to find the state with the highest elevation.
- Hill climbing



A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

Hill Climbing HC

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← problem.INITIAL
  while true do
    neighbor ← a highest-valued successor state of current
    if VALUE(neighbor) ≤ VALUE(current) then return current
    current ← neighbor
```

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.

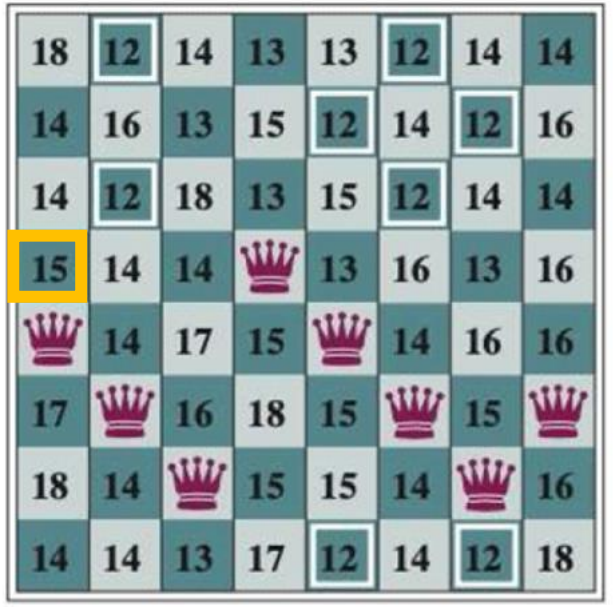
- From current state, move to the neighboring state with the highest value
  - Heads to the direction with the highest ascent.
- It terminates when it reaches a peak where no neighbors has a higher value.
- Use the negative of the heuristic function as the objective function.

Example of HC

- 8*7=56 successors
- Heuristic cost function h is the number of queens that are attacking each other.
- 15
  - (1, 4)
  - (2, 6), (2, 8) , (2, 3) , (2, 4)
  - (3, 7), (3, 5)
  - (4, 5), (4, 6), (4, 7),
  - (5, 6), (5, 7)
  - (6, 7), (6, 8)
  - (7, 8)
  - 1+4+2+3+2+2+1=15

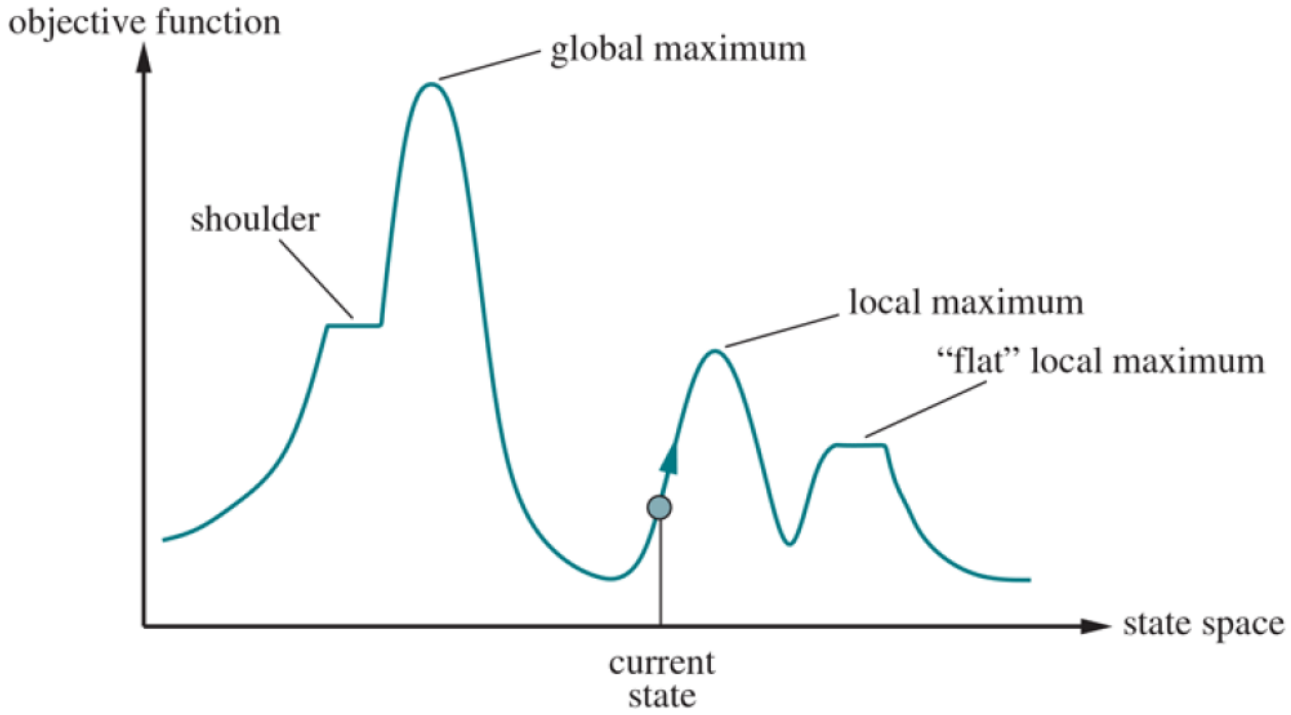
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  current ← *problem*.INITIAL
  while *true* **do**
    neighbor ← a highest-valued successor state of *current*
    if VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    current ← *neighbor*

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.

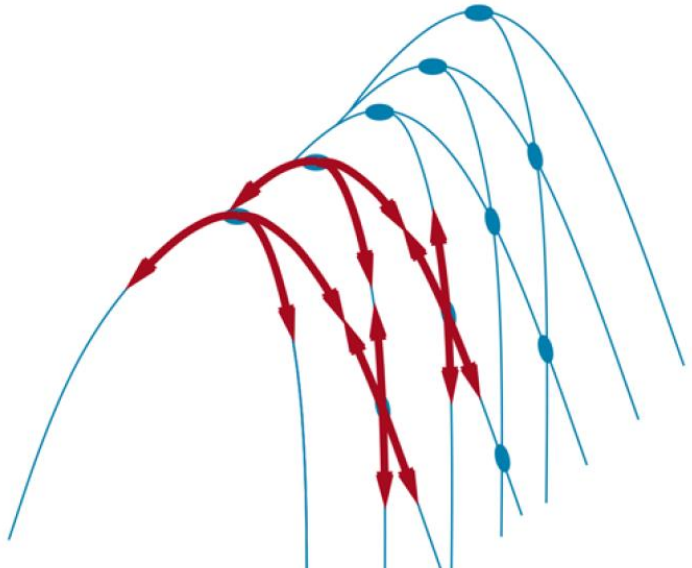


Problems of HC

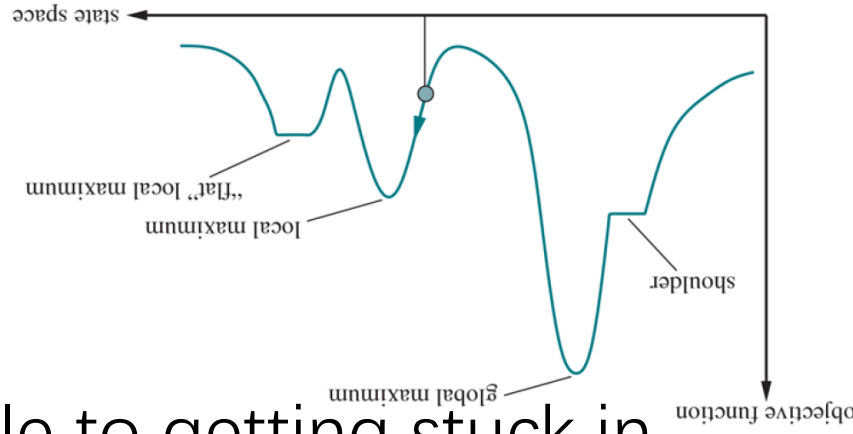
- Local maxima
  - A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum
- Plateaus
  - A plateau is a flat area of the state-space landscape
  - Shoulder, or flat local maximum
- Ridges



A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.



Simulated Annealing (SA)



- A HC neve makes downhill moves, is vulnerable to getting stuck in a local maximum.
- A purely random walk will eventually stumble upon the global maximum, but very inefficient
- Combining these two
  - Instead of picking the best move, SA picks a random move.
  - If the move improves the situation, it is always accepted.
  - Otherwise, SA accepts the move with some probability less than 1.
    - Badness of the move. Worse the move, smaller the probability.
    - Probability goes down as temperature goes down.

Local beam search

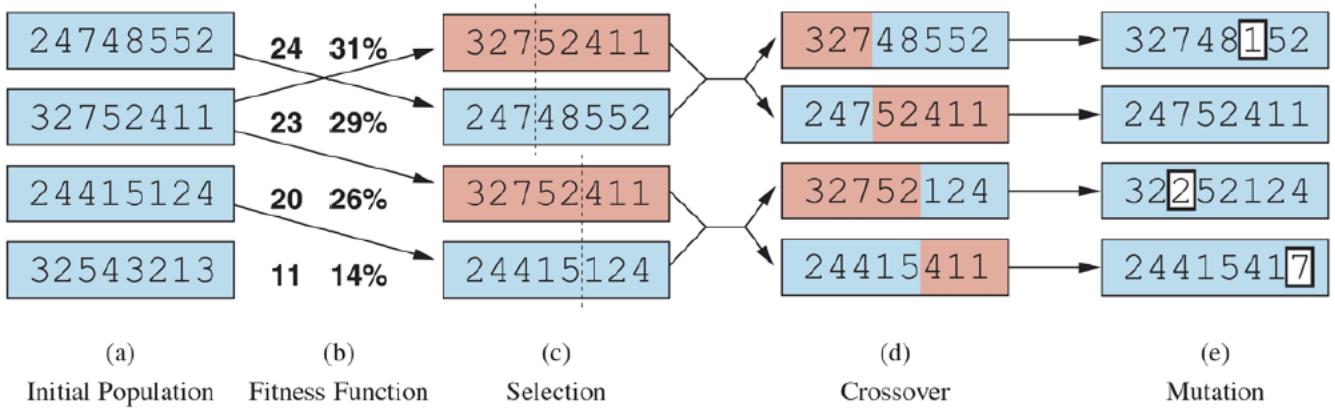
- Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations.
- The local beam search algorithm keeps track of k states rather than just one.
  - It begins with k randomly generated states. At each step, all the successors of all states are generated.
  - If any one is a goal, the algorithm halts.
  - Otherwise, it selects the best k successors from the complete list and repeats.

Evolutionary algorithms

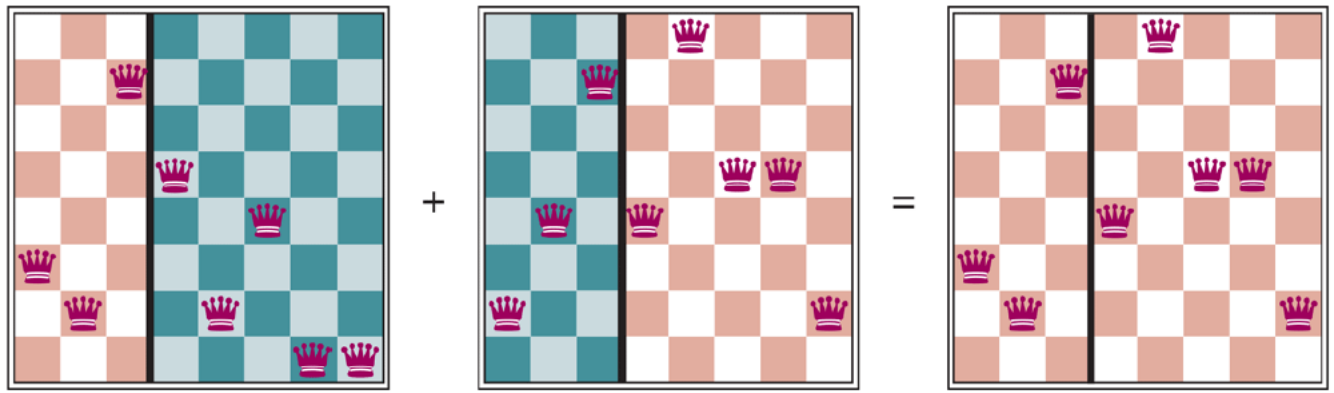
- Motivated by the natural selection in biology
- There is a population of individuals (states)
- The fittest (highest value) individuals produce offspring (successor states) that populate the next generation

- **The mixing number**, ρ, which is the number of parents that come together to form offspring.
- **Selection**: selecting the individuals who will become the parents of the next generation
- **Recombination**: randomly select a crossover point to split each of the parent strings and recombine the parts to form children.
- **Mutation rate:** how often offspring have random mutations.

8-queens Example



A genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by a fitness function in (b) resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).



The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The green columns are lost in the crossover step and the red columns are retained. (To interpret the numbers in Figure 4.6: row 1 is the bottom row, and 8 is the top row.)

- The c-th digit represents the row number of the queen in column c.
- **The mixing number**, ρ=2
- **Selection:** Fitness values: non-attacking pairs. Solution has 8*7/2=26 non-attacking pairs.
  - Transform the fitness values to probabilities
  - Two pairs of parents are selected w.r.t. the probabilities
- **Recombination:** Each pair has a randomly chosen crossover point.
  - Left from parent 1 + right from parent 2
  - Right from parent 1 + left from parent 2
- **Mutation:** Each location is subject to random mutation with a small independent probability.

Lecture 3 ILOs

- Informed Search Strategies
  - Heuristic function
  - Greedy best-first search
  - A* search, cost optimality, admissibility
  - Memory bounded search, weighted A* search
  - Design heuristics functions
- Search in complex environments
  - Local search
    - Hill climbing
    - Simulated annealing
    - Local beam search
    - Evolutionary search