Artificial Intelligence

Agents - Searching

Nguyễn Văn Diêu

HO CHI MINH CITY UNIVERSITY OF TRANSPORT

2025

Kiến thức - Kỹ năng - Sáng tạo - Hội nhập

Sứ mệnh - Tầm nhìn

Triết lý Giáo dục - Giá trị cốt lõi

Outline I

1. AI Overview
   - 1.1 Definition of AI
   - 1.2 Turing Test
2. Agents and Environments
3. Intelligent agents
4. Solving Problems by Searching
   - 4.1 Romania Problem
   - 4.2 Search Problems and Solutions
   - 4.3 Example Problems
5. Search Algorithms
   - 5.1 Search Trees
   - 5.2 Property of Graph Search
   - 5.3 Best-first Search
     - Algorithm

Outline II

Data Structures
Example

5.4 Uninformed Search
Breadth-First Search
Breadth-First Search Example
Uniform-Cost Search
Depth-First Search
Depth-First Search Example
Depth-Limited Search

5.5 Informed Search
Greedy Best-First Search
Example
A* Search
Example
Heuristic Functions

5.6 Complex Environments

Outline III

Local Seach and Optimization
Random Restart Hill-Climbing
Simulated Annealing Search
Local Beam Search
Genetic Algorithms

Definition of AI

- “Intelligence: The ability to learn and solve problems”

  Webster’s Dictionary.

- ”Artificial intelligence (AI) is the intelligence exhibited by machines or software”

  Wikipedia.

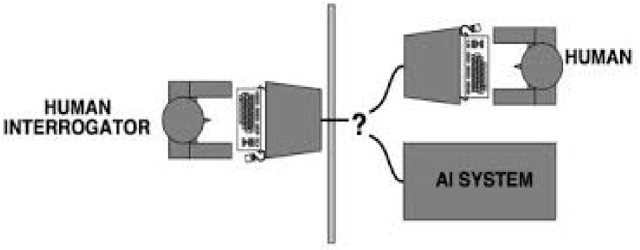
- “The science and engineering of making intelligent machines”

  John McCarthy.

- “The study and design of intelligent agents, where an intelligent agent is a system that perceives its environment and takes actions that maximize its chances of success.”

  Russel and Norvig AI book.

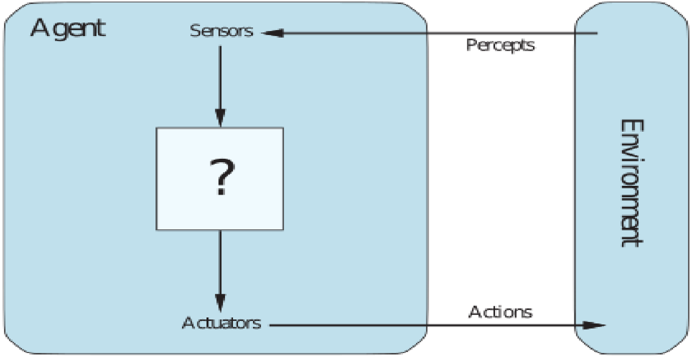
Turing Test



- **Interrogator** posing written questions to **Human** and **Computer**.
- **Human** and **Computer** response.
- **Computer** pass the **Test** if **Interrogator** cannot know responses come from a **Human** or from a **Computer**.

Agents and Environments

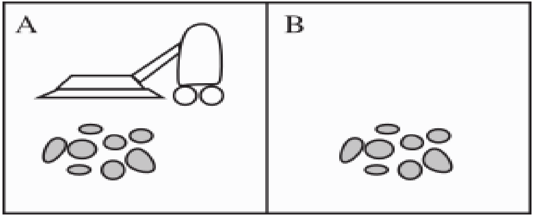
- **Agent:** is anything, can be viewed as:
  - **perceiving** its **environment** through **sensors**
  - **acting** upon that **environment** through **actuators**



Agents and Environments

- **Human Agent:**
  - Sensors: eyes, ears, ...
  - Actuators: hands, legs, mouth, ...
- **Robotic agent:**
  - Sensors: Cameras and infrared.
  - Actuators: Various motors.
- **Agents everywhere!**
  - Cell phone
  - Vacuum cleaner
  - Robot
  - Self-driving car
  - Human
  - ...

Vacuum Cleaner



- Percepts: location and contents e.g., [A, Dirty].
- Actions: Left, Right, Suck, NoOp.
- Agent function: mapping from percepts to actions.

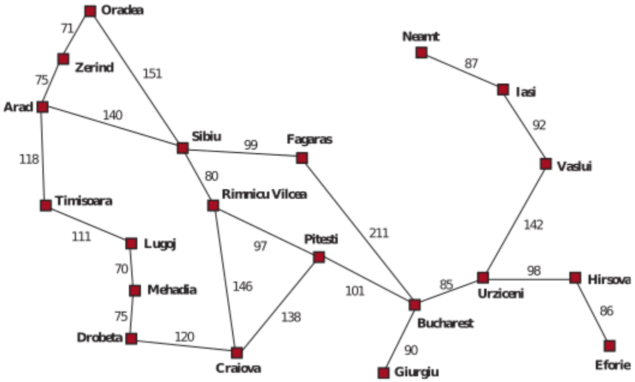
Percept	Action
[A, clean]	Right
[A, dirty]	Suck
[B, clean]	Left
[B, dirty]	Suck

Intelligent agents

- Central in AI.
- The goal of AI is to create intelligent agents that are useful, reactive, autonomous, and even capable of social interaction and proactivity.
- A performance measure evaluates the behavior of the agent.
- An agent that acts to maximize its expected performance measure is called a rational agent.
- Agents can improve their performance through **learning**.

$$\textbf{Agent} = \textbf{Architecture} + \textbf{Program}$$

Agent in **Arad** city and go to **Bucharest** city by road.



A simplified road map of part of Romania.

Search Problems and Solutions

Search problem can be defined formally as follows:

- **States**: An instance of the some aspect of the problem.
- **State space**: A set of all possible states. e.g Cities in Romania problem map.
- **Initial state**: Agent starts in. For example: Arad.
- **Goal states**: One or set of state must reach.

Search Problems and Solutions

Search problem can be defined formally as follows:

- **Actions**: Some thing agent can do. Given a state s, **Action**(s) returns a finite set of actions that can be executed in s.
  e.g. **Action**$(Arad) = \{ToSibiu, ToTimisoara, ToZerind\}$.
- **Transition model**: Describes what each action does.
  Result(s, a) returns the state that results from doing action a in state s.
  e.g. **Result**$(Arad, ToZerind) = Zerind$
- **Action cost function**: **A-Cost**$(s, a, s^{'})$ gives numeric cost of applying action a in state s to reach state $s^{'}$.

The **state space** can be represented as a **graph** in which the **vertices are states** and **edges are actions**

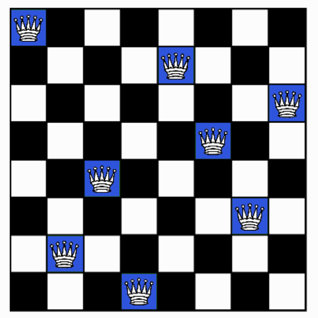
Problem Searching

1. **Define the problem through:**
   - Goal formulation.
   - Problem formulation.
2. **Solving the problem as a 2-stage process:**
   - Search: exploration of several possibilities.
   - Execute the solution found

Problem formulation

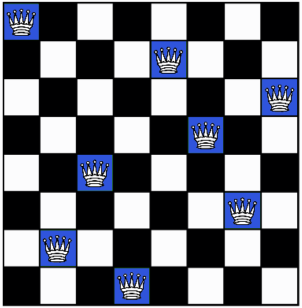
1. **Initial state:** The state in which the agent starts.
2. **States** (**State space**): All states reachable from the initial state by any sequence of actions.
3. **Actions** (**Action space**): Possible actions available to the agent. At a state **s**, **Actions**(**s**) returns the set of actions that can be executed in state **s**.
4. **Transition model:** A description of what each action does **Results**(**s**, **a**).
5. **Goal test:** Determines if a given state is a goal state.
6. **Path cost:** Function that assigns a numeric cost to a path w.r.t. performance measure.

8-Queen Problem



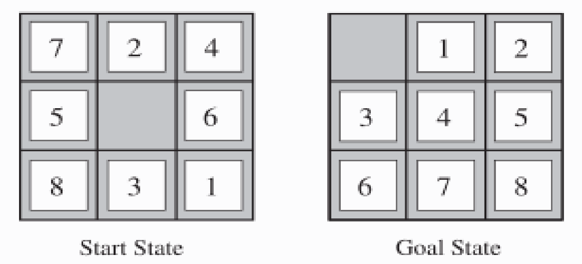
- Place 8 queens so that no queen is attacking any other horizontally, vertically or diagonally.
- Number of possible sequences to investigate:
  $64 * 63 * 62 * ... * 57 = 1.8 * 10^{14}$

8-Queens Problem



1. **Initial state:** Any arrangement of 0 to 8 queens on the board is a state.
2. **States**: No queen on the board.
3. **Actions**: Add a queen to any empty square.
4. **Transition model:** Returns the board with a queen added to the specified square.
5. **Goal test:** 8 queens on the board with none attacked.

8-puzzle Problem

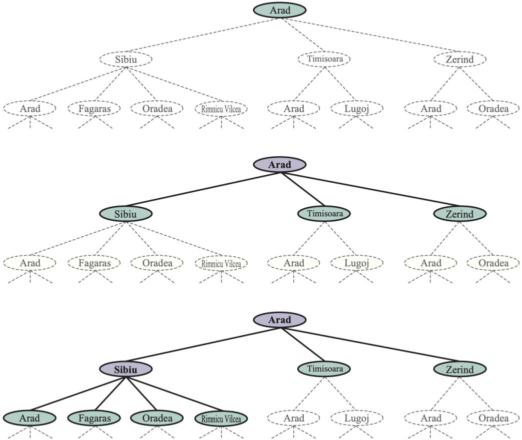


1. **States:** Location of each of the 8 tiles in the 3x3 grid.
2. **Initial state:** Any state.
3. **Actions**: Move Left, Right, Up or Down.
4. **Transition model:** Given a state and an action, returns resulting state.
5. **Goal test:** State matches the goal configuration.
6. **Path cost:** Each step costs 1. Path cost is the number of steps in the path.

Search Algorithms

- **States space**: Graph is formed by various paths from the initial state, trying to find a path that reaches a goal state.
- **Search tree**: Describes paths between these states, reaching towards the goal.
- **Node**: Corresponds to a state in the states space.

Search Trees

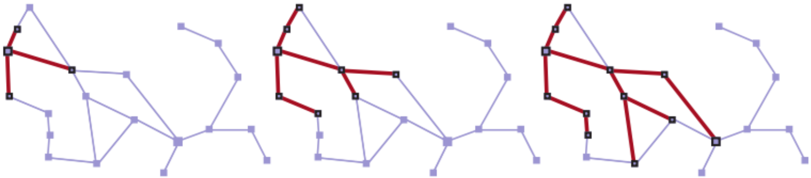


3 partial search trees for finding a route from **Arad** to **Bucharest**.

Search Trees

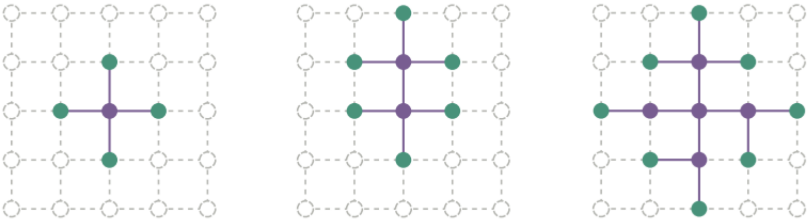
- Expanded: set of Lavender Nodes.
- Frontier: Generated node but not yet expanded. (Green nodes).
- Reached nodes = Expanded nodes + Frontier nodes.

Sequence of Search Trees



Sequence of search trees generated by a graph search on the Romania problem.

Property of graph search



(a) (b) (c)

- Frontier: Set of nodes (and corresponding states) that have been reached but not yet expanded.
- Interior: Set of nodes (and corresponding states) that have been expanded.
- Exterior: Set of states that have not been reached.

Best-first Search

```
func Best-First-Search(problem, f) return Solution node or Failure
    node ← Node(State = problem.Initial)
    frontier ← a priority Queue by f, node as an element
    reached ← lookup table, with key problem.Initial and value node
    while not Empty (frontier) do
        node ← Pop (frontier)
        if problem.Goal(node.State) then
            return node
        foreach child ∈ Expand (problem, node) do
            s ← child.State
            if problem.Goal(s) then
                return child
            if s ∉ reached or child.P-Cost < reached[s].P-Cost then
                reached ← child
                add child to frontier
    return Failure
```

function Expand

func Expand$(problem, node)$ **yields** *nodes*

$s \leftarrow node.\textbf{State}$

foreach $action \in problem.\textbf{Action}(s)$ **do**

$s' \leftarrow problem.\textbf{Result}(s, action)$

$cost \leftarrow node.\textbf{P-Cost} + problem.\textbf{A-Cost}(s, action, s')$

yields $\textbf{Node}(\textbf{State} = s', \textbf{Parent} = node, \textbf{Action} = action,$
$\textbf{P-Cost} = cost)$

Data Structures

Data structure help to keep track of the search tree.

A node with four components:

- *node*.**State**: the state to which the node corresponds.
- *node*.**Parent**: the node in the tree that generated this node.
- *node*.**Action**: the action that was applied to the parent's state to generate this node;
- *node*.**P-Cost**: the total cost of the path from the initial state to this node.

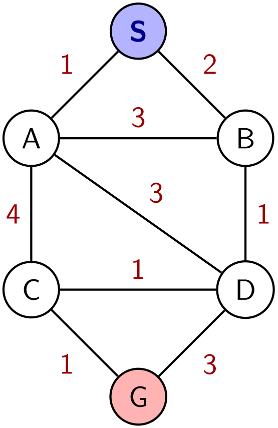
Data structure of frontier:

- **Empty**(*frontier*)
- **Pop**(*frontier*)
- **Top**(*frontier*)
- **Add**(*node*, *frontier*)

Three type of queue:

- **Priority queue**
- **FIFO queue**
- **LIFO queue**

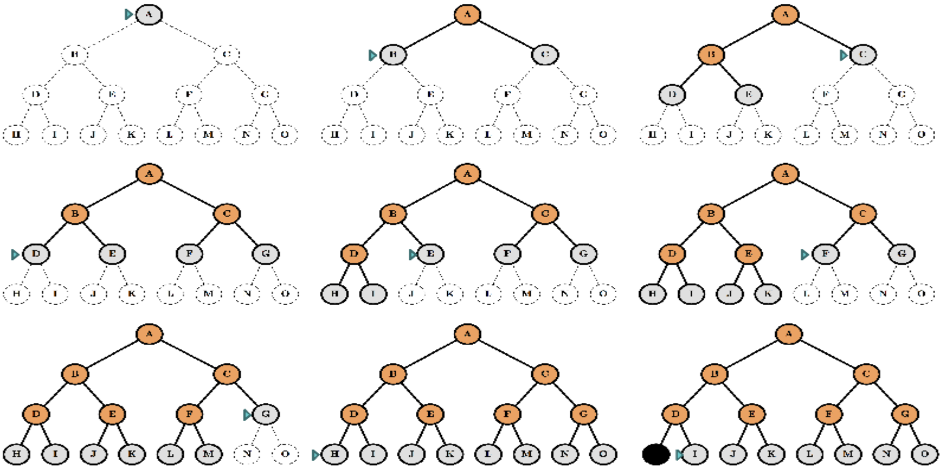
Best-first Search using Priority Queue.



Best-First Search

No.	Reached	
	Expanded	Frontier [Priority Queue]
0		S(0)
1	**S(0)**	A(1) B(2)
2	A(1)	B(2) C(5) D(4)
3	B(2)	C(5) ~~D(4)~~ D(3)
4	D(3)	~~C(5)~~ C(4) G(6)
5	C(4)	~~G(6)~~ G(5)
6	**G(5)**	
$S \rightarrow B(2) \rightarrow D(3) \rightarrow C(4) \rightarrow G(5)$		

BFS: Expand **shallowest** first.



BFS Search algo.

```
func Breadth-First-Search(problem) return Solution node or Failure
    node ← Node(problem.Initial)
    if problem.Goal (node.State) then
        return Solution (node)
    frontier ← a FIFO queue with node as an element
    reached ← problem.Initial
    while not Empty (frontier) do
        node ← Pop (frontier)
        foreach child ∈ Expand (problem, node) do
            s ← child.State
            if problem.Goal(s) then
                return child
            if s ∉ reached then
                add s to reached
                add child to frontier
    return Failure
```

Expand Function

func Expand($problem$, *node*) **yields** *nodes*

$s \leftarrow$ *node*.**State**

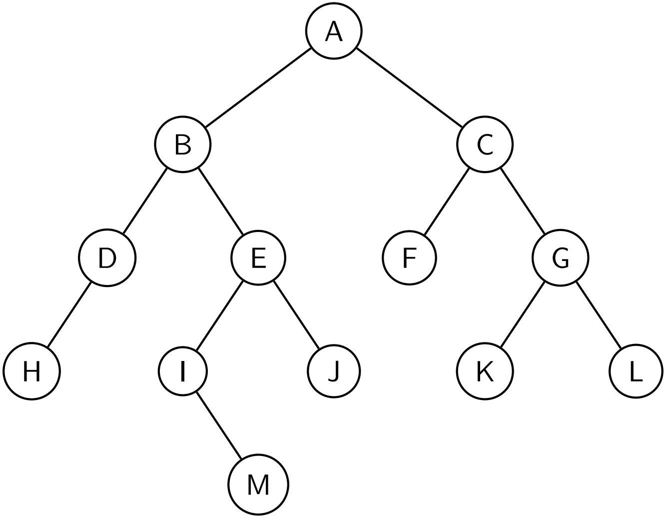
foreach $action \in problem.$**Action**(s) **do**

$s' \leftarrow problem.$**Result**$(s, action)$

$cost \leftarrow$ *node*.**P-Cost** $+$ $problem.$**A-Cost**$(s, action, s')$

yields Node(**State** $= s'$, **Parent** $=$ *node*, **Action** $= action$,
P-Cost $= cost$)

e.g. BFS 01

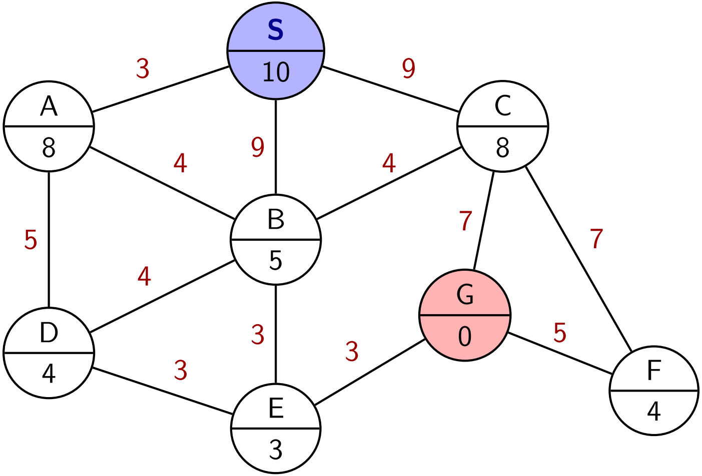


e.g. BFS 01

Breadth-First Search

No.	Reached	
	Expanded	Frontier [Queue (Head-Tail)]
0		A
1	A	B C
2	B	C D E
3	C	D E F G
4	D	E F G H
5	E	F G H I J
6	F	G H I J
7	G	H I J K L
8	H	I J K L
9	I	J K L M
10	J	K L M
11	K	L M
12	L	M
13	M	

e.g.02 BFS



e.g.02 BFS

Breadth-First Search

No.	Reached	
	Expanded	Frontier [Queue (Head-Tail)]
0		S
1	S	<u>A B C</u>
2	A	B C <u>D</u>
3	B	C D <u>E</u>
4	C	D E <u>G</u>
5	D	E G
6	E	G
7	G	
$S \rightarrow C \rightarrow G$		

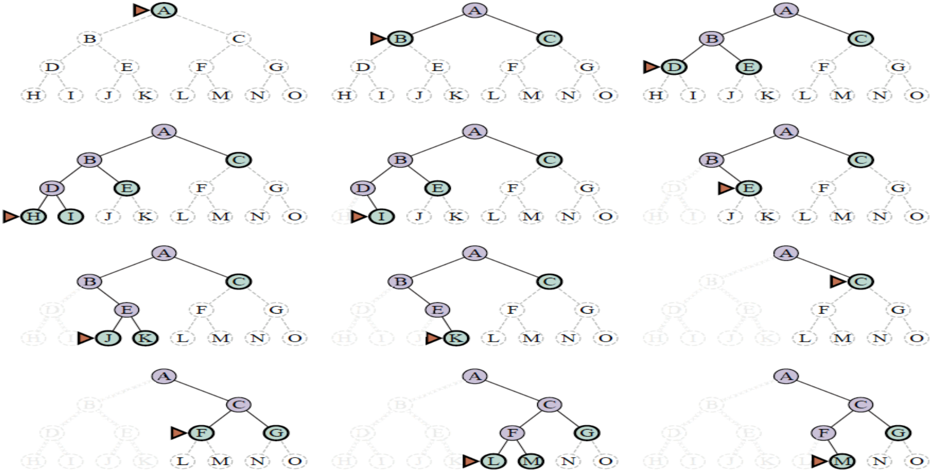
Uniform-Cost Search

Uniform Cost search is the Best First search.

func `Uniform-Cost-Search`(*problem*) **return Solution** or **Failure**
  Best-First-Search(*problem*, PATH-COST)

Depth-first Search (DFS)

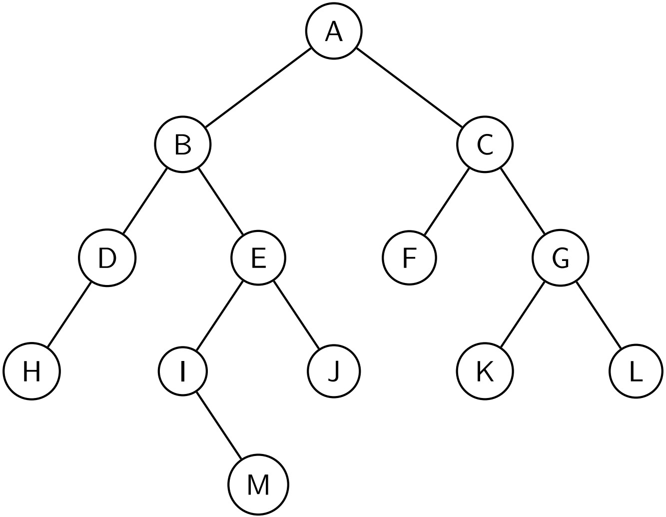
DFS: Expand **deepest** first.



DFS Search

```
func Depth_First_Search(initialState, goalTest)
    frontier = Stack.new(initialState)
    explored = Set.new()
    while not frontier.isEmpty() do
        state = frontier.pop()
        explored.add(state)
        if goalTest(state) then
            return Solution (state)
    for neighbor ∈ state.neighbors() do
        if neighbor ∉ (frontier ∪ explored) then
            frontier.push(neighbor)
    return Failure
```

e.g.01 DFS

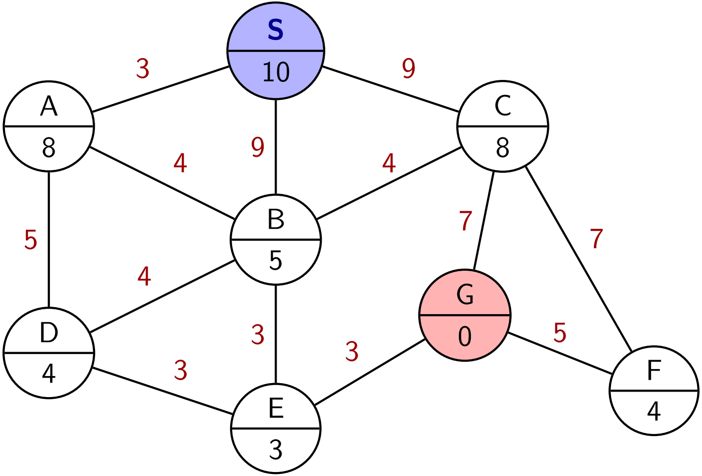


e.g.01 DFS

Depth-First Search

No.	Reached	
	Expanded	Frontier [Stack (Bottom-Top)]
0		A
1	A	C B
2	B	C E D
3	D	C E H
4	H	C E
5	E	C J I
6	I	C J M
7	M	C J
8	J	C
9	C	G F
10	F	G
11	G	L K
12	K	L
13	L	

e.g.02 DFS



e.g.02 DFS

Depth-First Search

No.	Reached	
	Expanded	Frontier [Stack (Bottom-Top)]
0		S
1	S	C B A
2	A	C B D
3	D	C B E
4	E	C B G
5	G	C B
$S \rightarrow A \rightarrow D \rightarrow E \rightarrow G$		

Depth-Limited Search

```
func Depth_Limited_Search(problem, l)
    return node or failure or cutoff
    frontier ← a LIFO queue (stack) with Node(problem.Initial) as an element
    result ← failure
    while not Empty(frontier) do
        node ← Pop(frontier)
        if problem.Goal(node.State) then
            return node
        if Depth(node) > l then
            result ← cutoff
        else
            if not Cycle(node) then
                foreach child ∈ Expand(problem, node) do
                    add child to frontier
    return result
```

Informed (Heuristic) Search

Informed search strategy:

1. Problem $\rightarrow$ Problem-specific knowledge.
2. Find solutions *more efficiently* than *uninformed strategy*.

General approach is called **best-first search**:

– Instance of **Tree-Search** or **Graph-Search** algorithm.

– Node *n* is selected based on evaluation of function, $f(n)$.

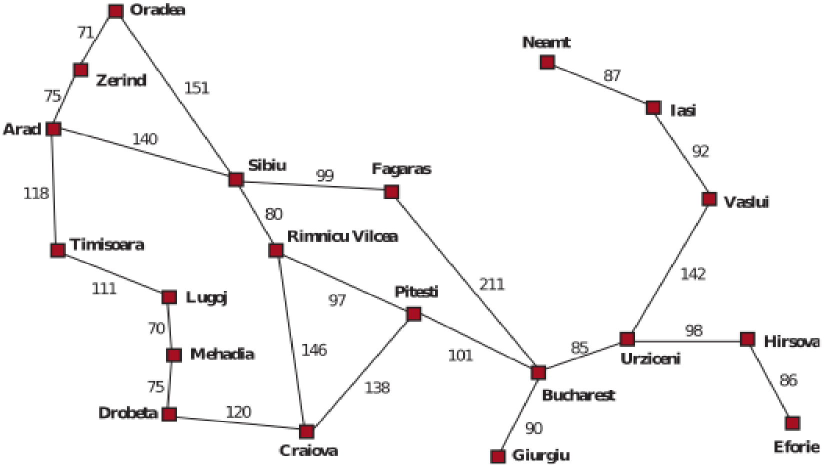
– **Cost estimate** $f(n) \rightarrow$ node n, **lowest evaluation** is expanded first.

– A component of $f(n)$, **heuristic** function $h(n)$

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

Greedy Best-First Search

Example: Finding a route from **Arad** to **Bucharest**.



Greedy Best-First Search

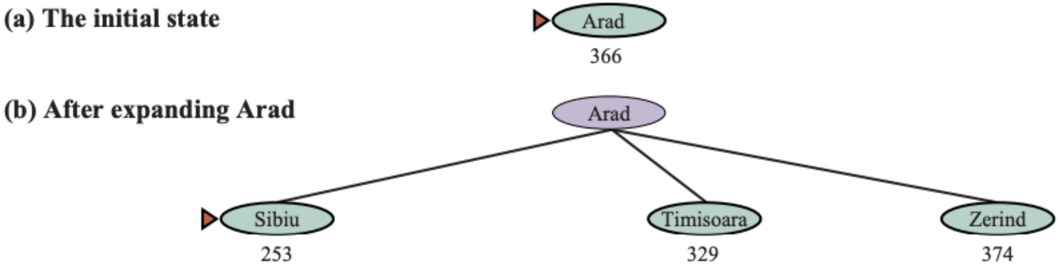
Straight-line distance heuristic: h_{SLD}.

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

Straight-line distance from **City** to **Bucharest**.

Greedy Best-First Search

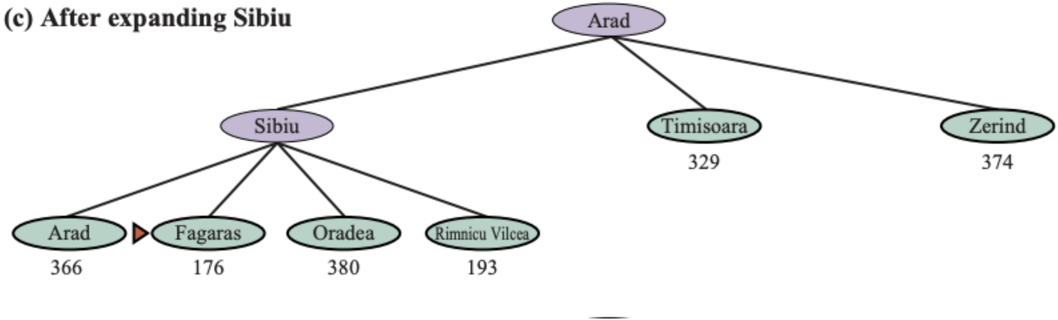
$h_{SLD}(Arad) = 366;$ $h_{SLD}(Sibiu) = 253$
$h_{SLD}(Timisoara) = 329;$ $h_{SLD}(Zerind) = 366$



Greedy Best-First Search

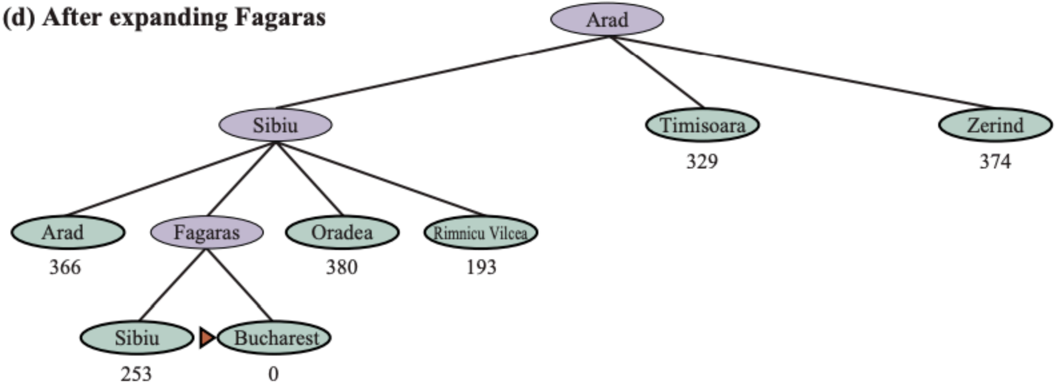
$h_{SLD}(Arad) = 366;$ $h_{SLD}(Fagaras) = 176$

$h_{SLD}(Oradea) = 380;$ $h_{SLD}(RimnicuVilcea) = 193$



Greedy Best-First Search

$h_{SLD}(Sibiu) = 253; \qquad h_{SLD}(Bucharest) = 0$



Greedy Best-First Search

- **Greedy Best-First** search Using **Best-First-Search** algo. with $f(n) = h(n)$

Greedy-Best-First Search

```
func Greedy-Best-First-Search(problem, f) return Solution node or Failure
    node ← Node(State = problem.Initial)
    frontier ← a priority Queue by f, node as an element
    reached ← lookup table, with key problem.Initial and value node
    while not Empty (frontier) do
        node ← Pop (frontier)
        if problem.Goal(node.State) then
            return node
        foreach child ∈ Expand (problem, node) do
            s ← child.State
            if problem.Goal(s) then
                return child
            if s ∉ reached or child.h() < reached[s].h() then
                reached ← child
                add child to frontier
    return Failure
```

update function Expand

func $\texttt{Expand}(\mathit{problem}, \mathit{node})$ **yields** *nodes*

$s \leftarrow \mathit{node}.\textbf{State}$

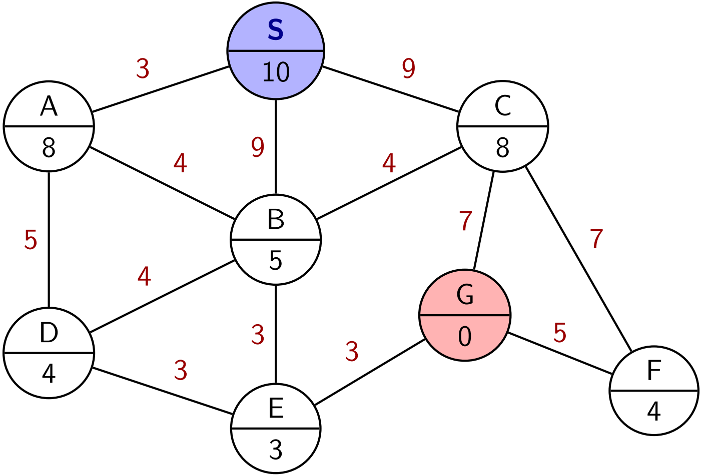
foreach $\mathit{action} \in \mathit{problem}.\textbf{Action}(s)$ **do**

$s' \leftarrow \mathit{problem}.\textbf{Result}(s, \mathit{action})$

$\mathit{cost} \leftarrow \mathit{problem}.\textbf{h()}(s, \mathit{action}, s')$

yields $\textbf{Node}(\textbf{State} = s', \textbf{Parent} = \mathit{node}, \textbf{Action} = \mathit{action}, \textbf{h()} = \mathit{cost})$

e.g. Greedy BFS



e.g. Gready BFS

Greedy Best-First Search

No.	Reached	
	Expanded	Frontier [Priority Queue]
0		S(10)
1	**S(10)**	A(8) B(5) C(8)
2	B(5)	A(8) C(8) D(4) E(3)
3	E(3)	A(8) C(8) D(4) G(0)
4	**G(0)**	A(8) C(8) D(4)
$S \rightarrow B \rightarrow E \rightarrow G$		

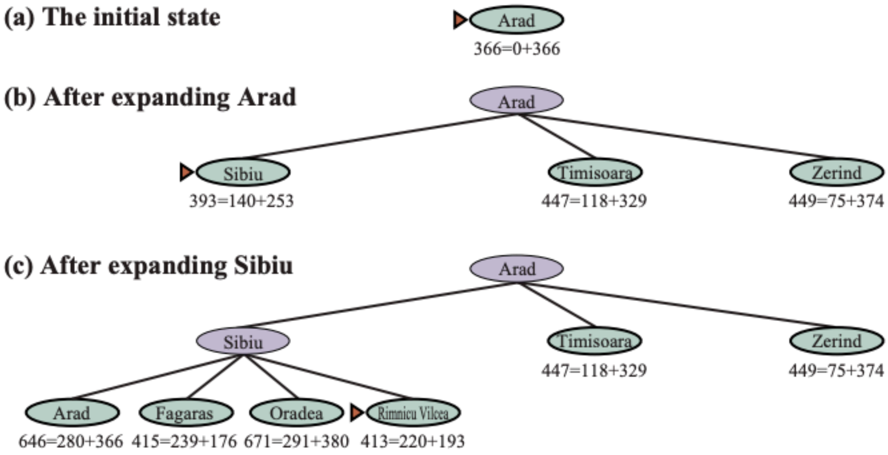
A* Search

- $g(n)$: Cost to reach the node *n*.
- $h(n)$: Cost from the node *n* to the goal:

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

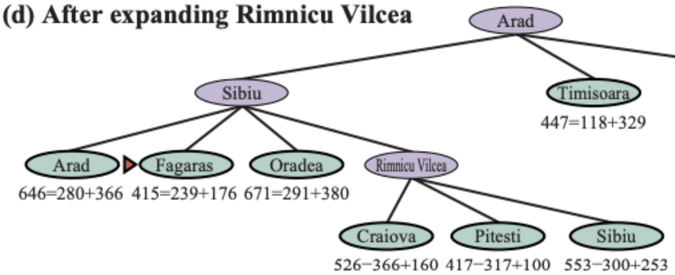
$f(n) =$ estimated cost of the cheapest solution through *n*.

A* Search



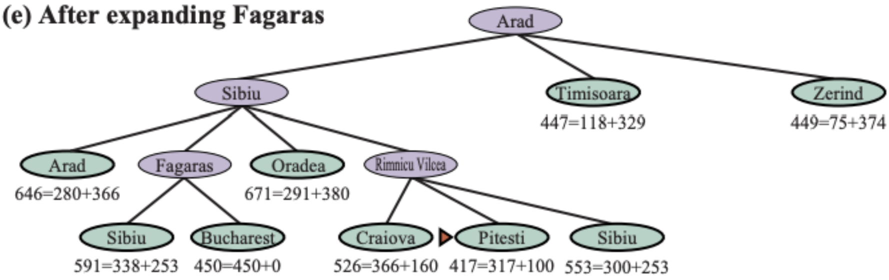
(a) The initial state
Arad
366=0+366
(b) After expanding Arad
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
(c) After expanding Sibiu
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

A* Search

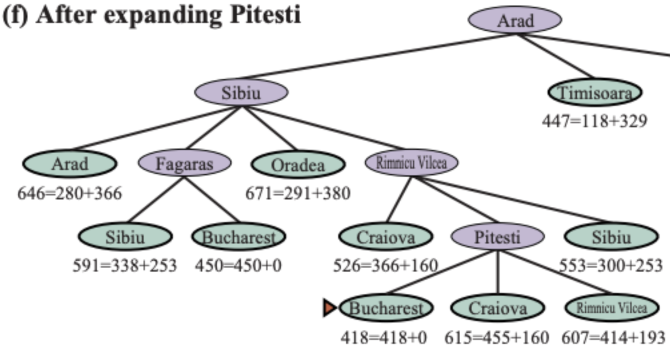


(d) After expanding Rimnicu Vilcea
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* Search


(e) After expanding Fagaras
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* Search


(f) After expanding Pitesti
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

A* Search

- **A* search** algo. using **Best-First-Search** algo. with $f(n) = g(n) + h(n)$

A* Search

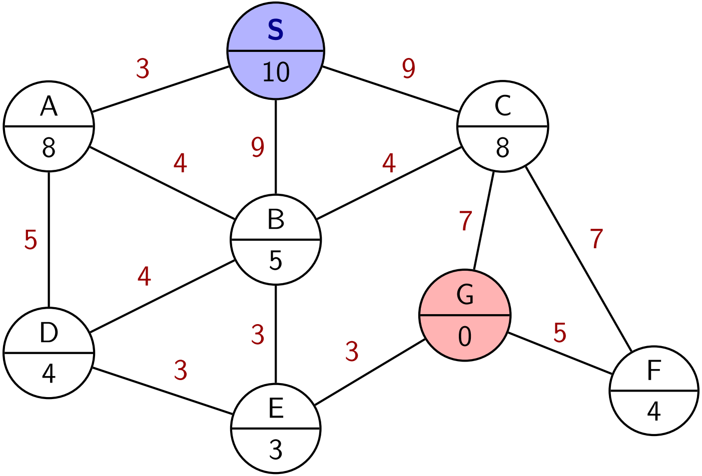
```
func A-Star-Search(problem, f) return Solution node or Failure
    node ← Node(State = problem.Initial)
    frontier ← a priority Queue by f, node as an element
    reached ← lookup table, with key problem.Initial and value node
    while not Empty (frontier) do
        node ← Pop (frontier)
        if problem.Goal(node.State) then
            return node
        foreach child ∈ Expand (problem, node) do
            s ← child.State
            if problem.Goal(s) then
                return child
            if s ∉ reached or child.P-Cost < reached[s].P-Cost then
                reached ← child
                add child to frontier
    return Failure
```

update function Expand

func Expand$(problem, node)$ **yields** *nodes*

- $s \leftarrow node.\textbf{State}$
- **foreach** $action \in problem.\textbf{Action}(s)$ **do**
  - $s' \leftarrow problem.\textbf{Result}(s, action)$
  - $cost \leftarrow node.\textbf{h()} + problem.\textbf{A-Cost}(s, action, s')$
  - **yields** $\textbf{Node}(\textbf{State} = s', \textbf{Parent} = node, \textbf{Action} = action, \textbf{P-Cost} = cost)$

e.g. A^*



e.g. A*

A* Search

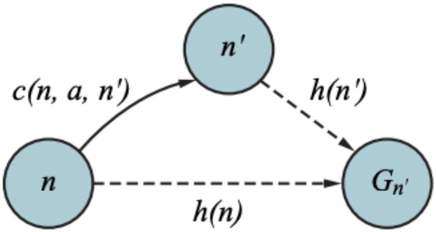
No.	Reached	
	Expanded	Frontier [Priority Queue]
0		S(0+10=10)
1	**S(0+10=10)**	A(3+8=11) B(9+5=14) C(9+8=17)
2	A(3+8=11)	~~B(9+5=14)~~ C(9+8=17) D(8+4=12) B(7+5=12)
3	B(7+5=12)	C(9+8=17) D(8+4=12) E(10+3=13)
4	D(8+4=12)	C(9+8=17) E(10+3=13)
5	E(10+3=13)	C(9+8=17) G(13+0=13)
6	**G(13+0=13)**	C(9+8=17)
$S \rightarrow A(3) \rightarrow B(7) \rightarrow E(10) \rightarrow G(13)$		

A* Consistency

- Node n : $h(n)$
- Node n' is successor of n : $h(n')$
- $n \rightarrow n'$: $c(n, a, n')$

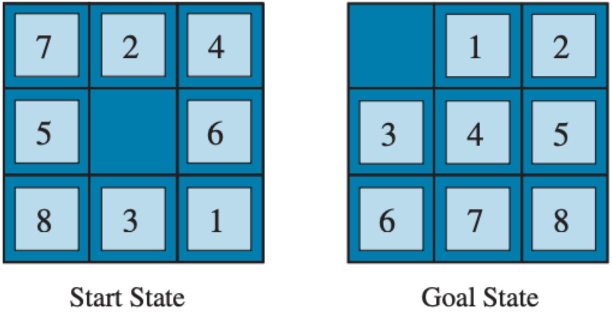
Triangle inequality:

$$h(n) \leqslant c(n, a, n') + h(n')$$



Heuristic Functions

e.g.



Heuristic Functions

There are two common heuristic function:

- h_1 = the number of misplaced tiles (blank not included). For figure above, all eight tiles are out of position, so the start state has $h_1 = 8$.
- h_2 = the sum of the distances of the tiles from their goal positions. Manhattan distance.

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

Local Search

From current state:

- Searching to neighbor,
- Without keep track of the paths, nor the set of states that have been reached.

Advantages:

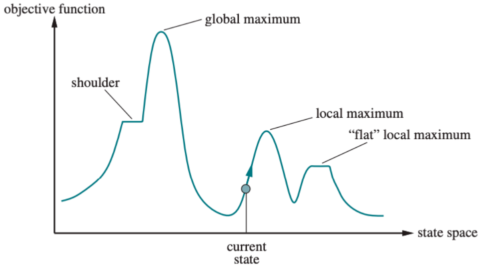
- Using very little memory;
- Can be often find reasonable solutions in large or infinite state spaces.

Hill-Climbing Search

"Like Climbing Everest in thick fog with amnesia"

Sometimes called greedy local search

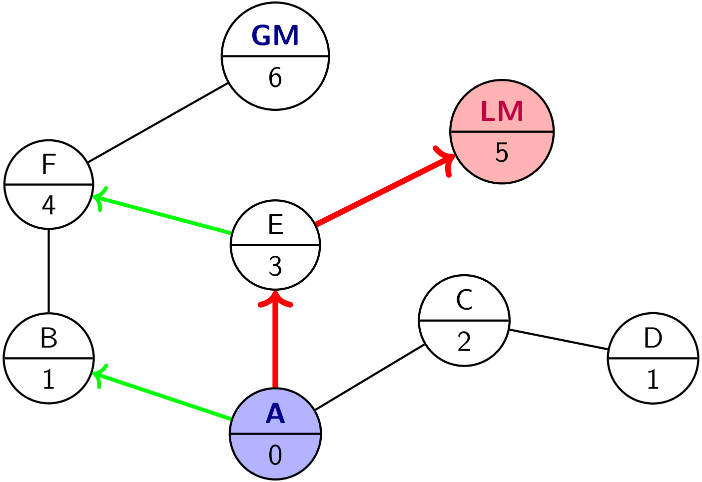
- From current state continuously moves in the direction of increasing value to find the peak of mountain or best solution to the problem.
- Keep track of current state and moves to the neighboring state with highest value.



Hill-Climbing Algorithm

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*

e.g. Hill-Climbing Problems - Local Maximum



e.g. Hill-Climbing Problems - Local Maximum

Hill-Climbing $h =$ **Elevation**

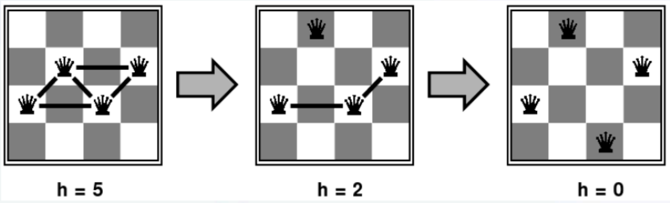
No.	Reached	
	Expanded	Frontier [Priority Queue]
0		**A(0)**
1	**A(0)**	B(1) C(2) E(3)
2	E(3)	S(4) LM(5)
3	LM(5)	
$A \to E \to LM$		

e.g. 4-Queens

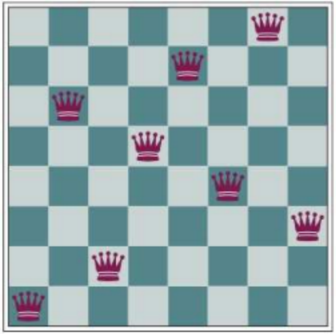
1	4	3	2
2	4	♕	3
♕	4	5	♕
3	♕	4	2

- Heuristic cost function *h*: the number of pairs of queens that are attacking each other.
- $h = 3$ for this board.
- Queen move within its column and update *h*.
- $h = 1$ is the best.
- The Hill-climbing algorithm will pick one of these.

e.g. 4-Queens



e.g. 8-Queens



- $h = 1$ for this board.

e.g. 8-Queens

- $h = 17$
- After update, $h = 12$ is the best.

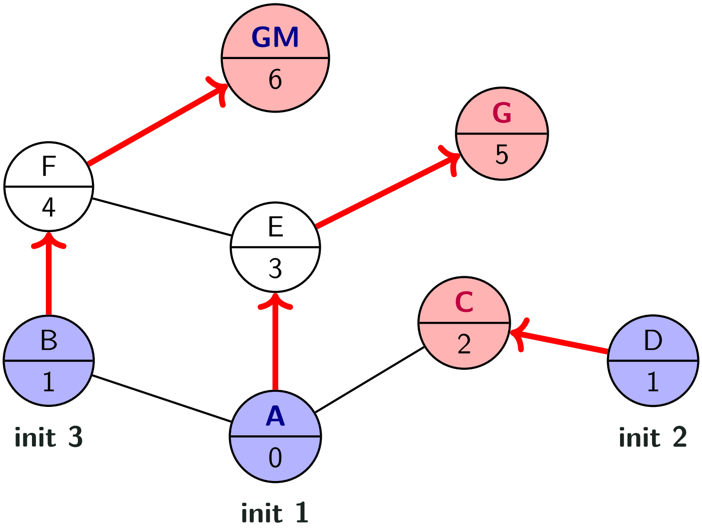
Hill-Climbing Problems

- Can get stuck in local maximum
- Can be stuck by ridges (a series of local maxima that occur close together)
- Can be stuck by plateaux

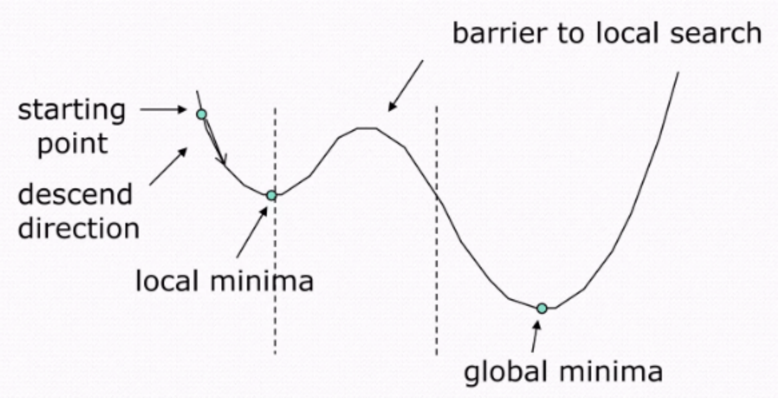
Resolutions

- Random restart Hill-Climbing
- Simulated Annealing
- Local Beam Search

e.g. Random Restart Hill-Climbing



Hill-Climbing Problems



Physical Annealing

In metallurgy, annealing is a process where a material (like metal) is heated to a high temperature and then cooled down gradually. The steps of this process are:

- The material is heated, allowing the atoms to move freely and escape their current positions.
- As the material cools down slowly, the atoms settle into a more stable, low-energy configuration, resulting in a stronger, more stable structure.

Simulated Annealing (SA)

- The Simulated Annealing (SA) algorithm simulates this annealing process to solve optimization problems.
- In SA, the goal is to find the best solution (global optimum) by starting from an initial solution and improving it over time.
- Importantly, the algorithm allows for occasional acceptance of worse solutions to avoid getting stuck in local optima.

Simulated Annealing (cont.)

Key steps of the algorithm:

1. Initialization: Start with a random solution.
2. Initial Temperature: Set a high initial temperature.
3. Make a change: Make a random change to the current solution to generate a new one.
4. Evaluate Energy (Cost): Compare the new solution to the current one based on a cost function (representing the energy or objective function of the system).
   - If the new solution is better, accept it.
   - If the new solution is worse, don't reject it outright but accept it with a certain probability.

Simulated Annealing (cont.)

Probability of accepting a worse solution:

The probability of accepting a worse solution is determined by an exponential function:

$$P = e^{-\Delta E/T}$$

where:

- ΔE is the difference in energy (cost) between the new solution and the current solution.
- T is the current temperature (which decreases over time).

If $\Delta E > 0$ (meaning the new solution is worse), there is still a probability P of accepting it. When the temperature T is high, this probability is higher, allowing the algorithm to explore and escape local minima.

Simulated Annealing (cont.)

Why use this probability distribution?

Using this exponential distribution to accept worse solutions is essential for preventing the algorithm from getting trapped in **local minima**:

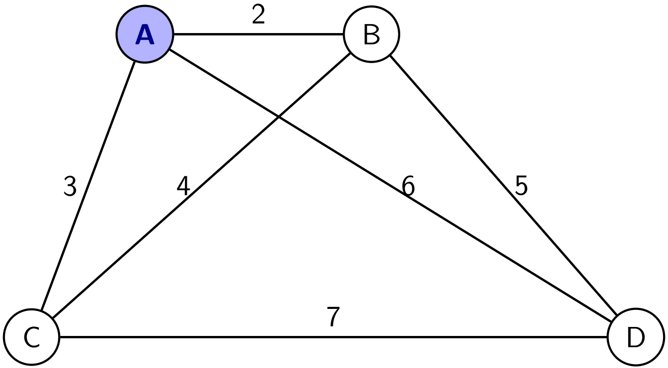
- At high temperatures (early in the process), accepting worse solutions allows the algorithm to explore more broadly and potentially escape local minima.
- As the temperature decreases, the probability of accepting worse solutions also decreases, making the algorithm converge to a more optimal solution.

Thus, by mimicking the physical annealing process, the algorithm can effectively search the solution space and find the global optimum, rather than settling for suboptimal local minima.

Simulated Annealing (cont.)

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  current ← *problem*.INITIAL
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow schedule(t)$
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE(*current*) – VALUE(*next*)
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

e.g. Traveling Salesperson Problem (TSP)



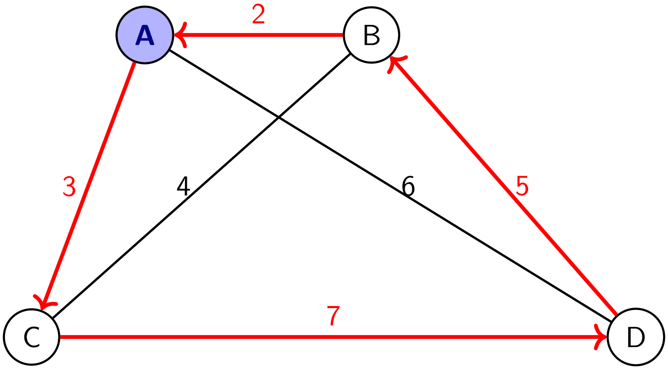
e.g. Traveling Salesperson Problem (TSP)

Loop $t = 100$; Series of random: 0.1, 0.9, 0.5, 0.8, 0.7, 0.3, 0.6, 0.5, ...
$T_0 = 100$; T rate $= 0.95$; $P = e^{\Delta/T}$
if $\Delta \leqslant 0$ and $P \leqslant Random$: Current $\leftarrow$ Next

Simulated Annealing: TSP

No.	Current	Next	Δ	T	P	Random	Solution
1	ABCDA(19)	ABDCA(17)	2	100	-	0.1	Next
2	ABDCA(17)	ACBDA(18)	-1	95	0.98	0.9	Next
3	ACBDA(18)	ACDBA(17)	1	90	-	0.5	Next
4	ACDBA(17)	ADBCA(19)	-2	85	0.97	0.8	Current
5	ACDBA(17)	ADCBA(19)	-2	81	0.97	0.7	Current
6	ACDBA(17)	-	-	-	-	-	-
-	-	-	-	-	-	-	-

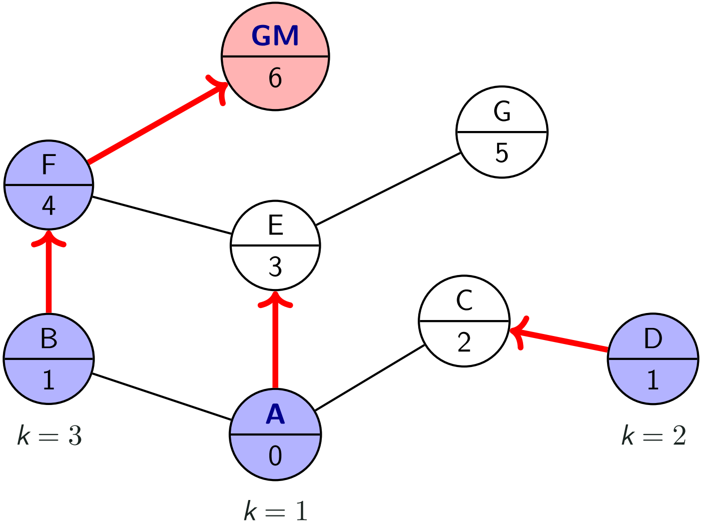
e.g. Traveling Salesperson Problem (TSP)



Local Beam Search

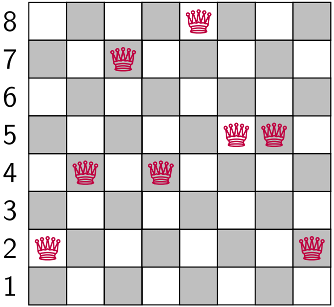
- Beginning with k randomly generated states.
- At each step, all the successors of all k states are generated.
- If any one is a goal, the algorithm halts.
- Otherwise, it selects the k best successors from the complete list and repeats.

e.g. Local Beam Search



Definition:

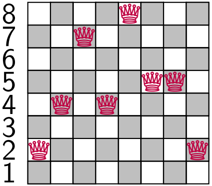
- **Individual**: Each individual (**gene**) is encoded by a string (characters or numbers) e.g. 8-Queens



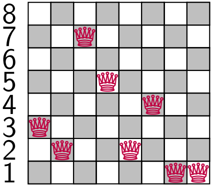
Gene = Encoded: **(2 4 7 4 8 5 5 2)**

- **Population:** Subset of $\sum(individuals - Chromosomes)$

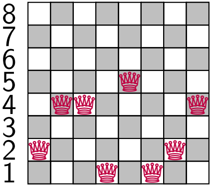
  e.g. 8-Queens



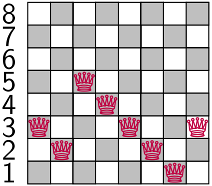
(2 4 7 4 8 5 5 2)



(3 2 7 5 2 4 1 1)



(2 4 4 1 5 1 2 4)

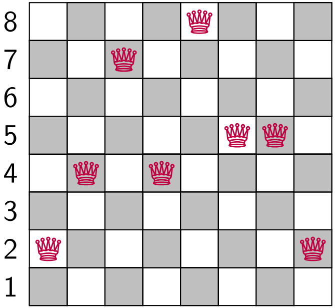


(3 2 5 4 3 2 1 3)

Genetic Algorithms (GA)

- **Fitness:** Score function of individual F_i

e.g. 8-Queens:
Fitness = number of non attacking pairs of queens. **Final solution:** $(8 * 7)/2 = 28$.



(3 2 5 4 3 2 1 3)

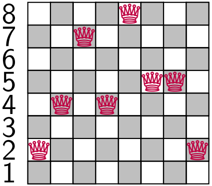
- Queen 1: 6
- Queen 2: 5
- Queen 3: 4
- Queen 4: 4
- Queen 5: 3
- Queen 6: 1
- Queen 7: 1
- Queen 8: 0

Fitness score $F_1 = 24$

Genetic Algorithms (GA)

- **Normalized fitness to probabilities:** $P_i = F_i / \sum_1^n F_i$

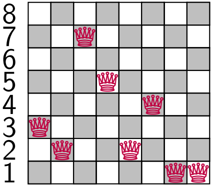
e.g. 8-Queens:



(2 4 7 4 8 5 5 2)

$F_1 = 24$

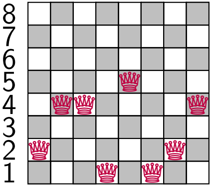
$P_1 = 31\%$



(3 2 7 5 2 4 1 1)

$F_2 = 23$

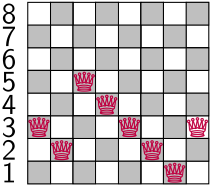
$P_2 = 29\%$



(2 4 4 1 5 1 2 4)

$F_3 = 20$

$P_3 = 26\%$



(3 2 5 4 3 2 1 3)

$F_4 = 11$

$P_4 = 14\%$

Genetic Algorithms (GA)

- **Mixing number ρ :** Number of parents that come together to form offspring. The most common case is $\rho = 2$: two parents combine their "genes" (parts of their representation) to form offspring.
- **Selection:** Selecting the individuals who will become the parents of the next generation:
- **Crossover:** Randomly select a Crossover point to split each of the parent strings, and recombine the parts to form two children.
- **Mutation rate:** Determine the frequency of offspring with random mutations.

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
    weights ← WEIGHTED-BY(population, fitness)
    population2 ← empty list
    for i = 1 to SIZE(population) do
      parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
      child ← REPRODUCE(parent1, parent2)
      if (small random probability) then child ← MUTATE(child)
      add child to population2
    population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

Genetic Algorithms (GA)

- population is an ordered list of individuals.
- fitness is a function to compute these values.
- weights is a list of corresponding fitness values for each individual.

Genetic Algorithms (GA)