CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

Metaheuristic Optimization

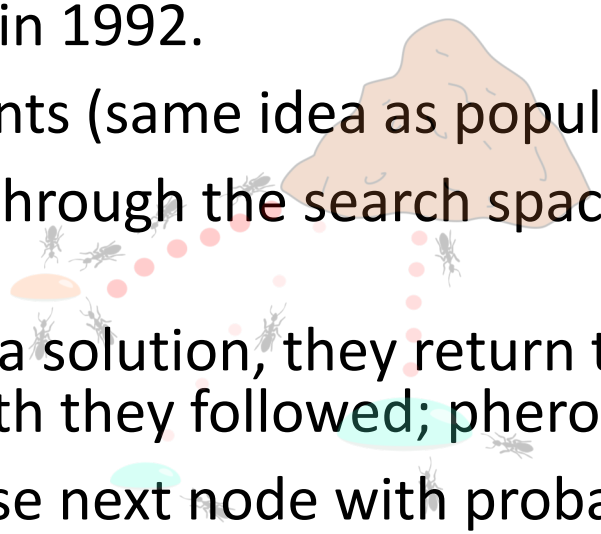
Part 1: ACO and PSO

Part 2: Local Search Introduction

Dr. Diarmuid Grimes

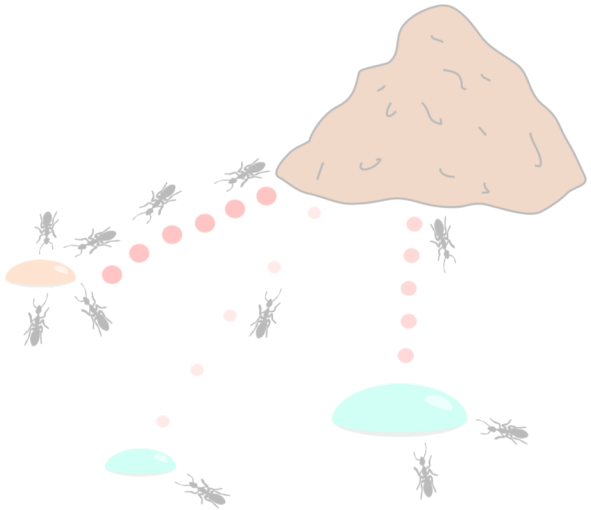
Recap: ACO

- Basic concept
  - Mimic indirect cooperation behavior of ants in finding shortest path to food source from nest.
  - Ants leave pheromone trail on path back to nest, subsequent ants will be attracted to path with most pheromones, pheromones evaporate over time.
- ACO
  - First proposed by Dorigo in 1992.
  - Create a set of artificial ants (same idea as population in GA)
  - Each ant will find a path through the search space such that it generates a candidate solution
  - After all ants have found a solution, they return to the start, and deposit “pheromones” on the path they followed; pheromone function of the solution quality
  - Next iteration, ants choose next node with probability based on pheromone on edge between nodes.



ACO

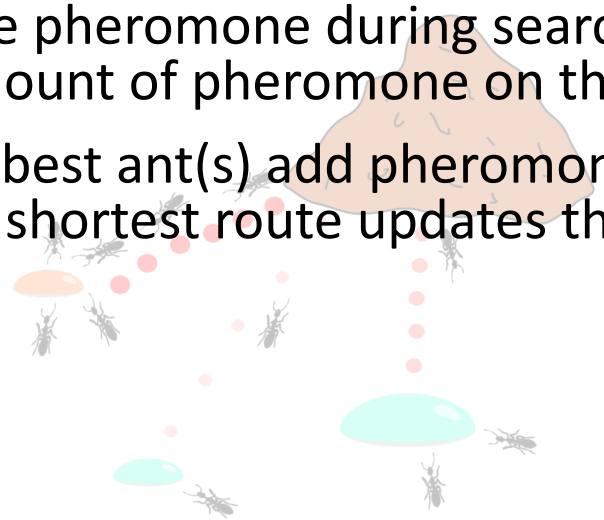
- The three main ideas that this ant colony algorithm has adopted from real ant colonies are:
  - The ants have a probabilistic preference for paths with high pheromone value
  - Shorter paths tend to have a higher rate of growth in pheromone value
  - It uses an indirect communication system through pheromone in edges



ACO

- Ants select the next vertex based on a weighted probability function based on two factors:
  - The number of edges and the associated cost (heuristic information).
  - The trail (pheromone) left behind by other ant agents.

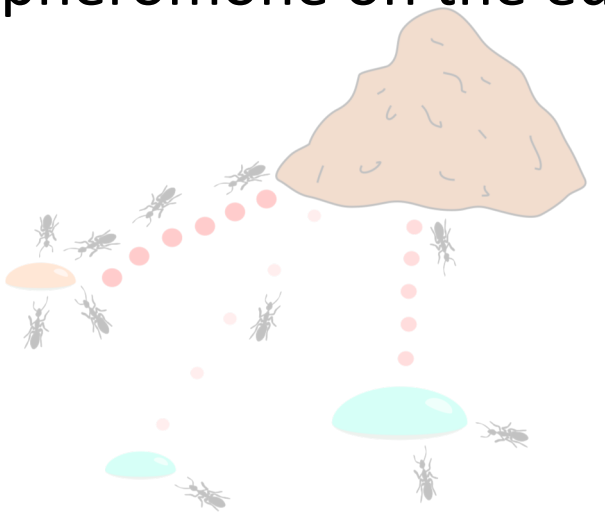
- Variants involve modifying the environment in two different ways :
  - **Local trail updating:** Reduce pheromone during search. E.g. As the ant moves between cities it updates (reduces) the amount of pheromone on the edge
  - **Global trail updating:** Only best ant(s) add pheromone. E.g. When all ants have completed a tour the ant that found the shortest route updates the edges in its path



ACO for TSP

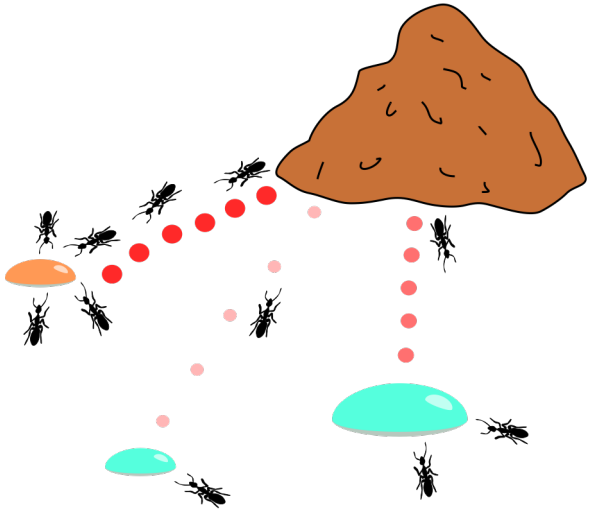
- **Diversification:** Local Updating is used to avoid very strong pheromone edges and hence increase exploration (and hopefully avoid locally optimal solutions).

- **Intensification :**Global Updating gives the shortest path higher reinforcement by increasing the amount of pheromone on the edges of the shortest path.



Common ACO Extensions

- Elitist Ant System
- Rank-Based Ant System (ASrank)
- Max-Min Ant System (MMAS)
- Restart-based ACO
- ACO + Local search
- Continuous Orthogonal Ant Colony (COAC)
- Recursive Ant Colony optimization



Elitist Ant System (EAS)

- First improvement on ACO
- Provide strong additional reinforcement to the arcs belonging to the best tour found since the start of the algorithm

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} + \rho\Delta_{ij}^{best}$$

$1/C^k$ If best ant k travels on edge (i, j)

Rank-Based Ant System (ASrank)

- Each ant deposits an amount of pheromone that decreases with its rank
- In each iteration, only the best (w-1) ranked ants and best-so-far ant are allowed to deposit pheromone

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{w-1} (w-k)\Delta\tau_{ij}^{k} + w\Delta_{ij}^{best}$$

Max-Min Ant System (MMAS)

Stutzle, Hoos 2000 (Good paper, worth reading)

- After all ants construct a solution, pheromone values are updated. Evaporation is the same as described before but only one (best) ant adds pheromones

$$\tau_{ij}(t+1) = \rho \tau_{ij}(t) + \Delta \tau_{ij}^{best}$$

- Best can be either global best found so far, or best found during this iteration.
- ***Lower*** and ***upper*** limits on pheromones limit the probability of selecting a city
- Initial pheromone values are set to the **upper limit**, resulting in initial exploration
- Occasionally pheromones are re-initialized

ACO + local search

- Ants construct solutions
- Local search run on each solution to find nearby improvements
- Pheromone added after local search and process repeats
- Note ACO very suitable for parallelism

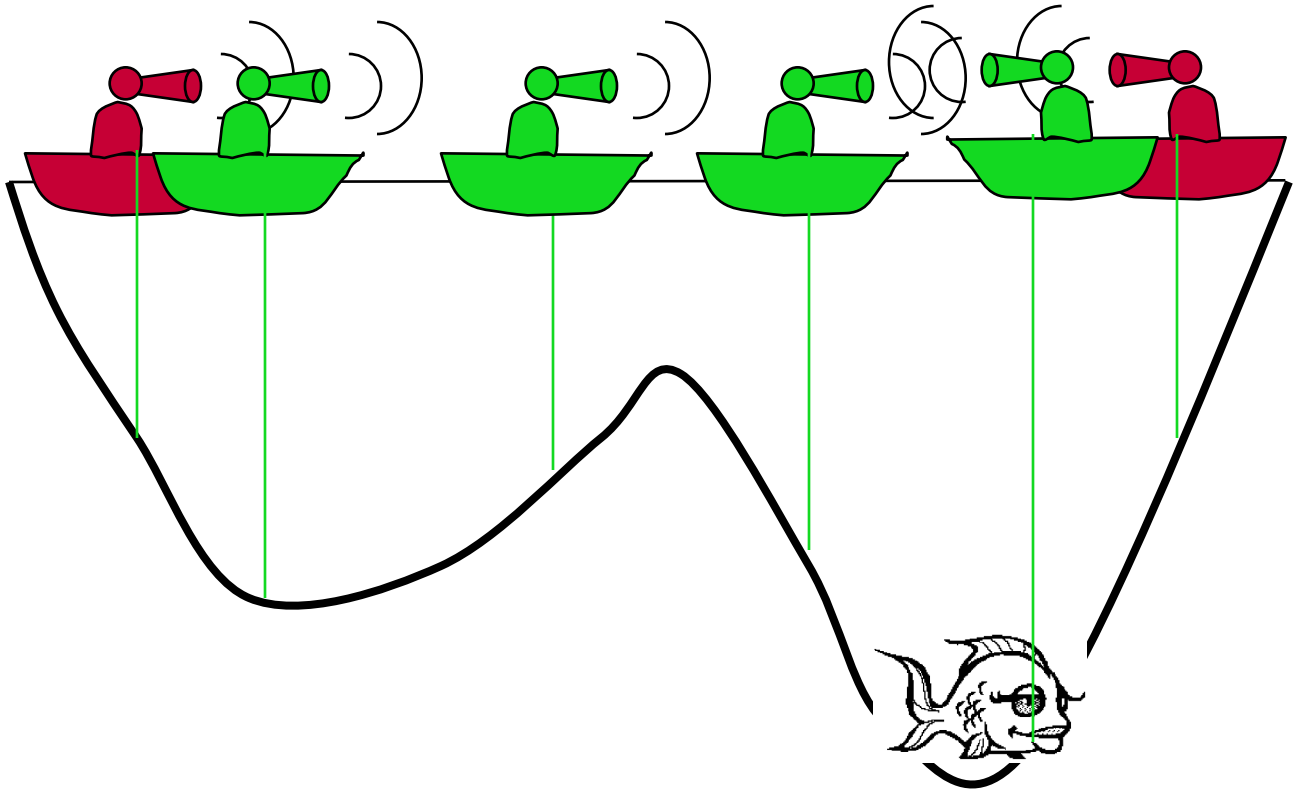
Restart-based ACO

- End of First Run
- Save Best Tour (route and cost)
- All ants die!
- New ants are born

Particle Swarm Optimization

Additional slide material: http://www.cs.armstrong.edu/saad/csci8100/

Cooperation example



PSO: The basic idea

- The PSO algorithm **maintains multiple potential solutions** (or particles) at one time
- During each iteration of the algorithm, each solution (or particle) is evaluated by an objective function to determine its fitness
- Each solution is represented by a particle in the fitness landscape (search space)
- The particles ***fly*** or ***swarm*** through the search space to find the maximum value returned by the objective function

PSO: The basic idea

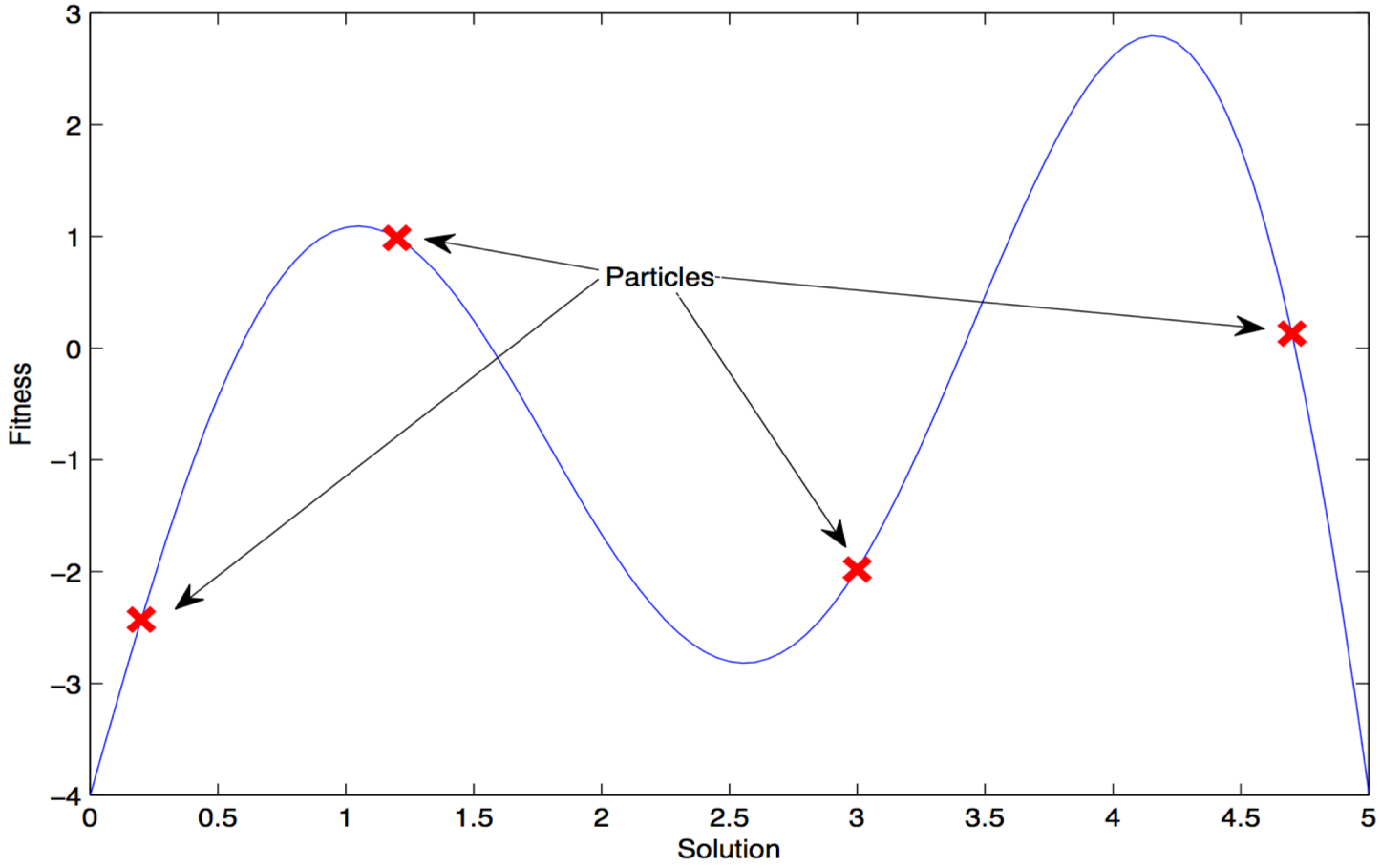
- Each particle is searching for the optimum
- Each particle is moving and hence has a velocity
- Each particle remembers the position it was in where it had its best results so far (personal best)
- But this would not be much good on its own! Particles need to help each other identify best place to search

PSO: The basic idea

- The particles in the swarm **co-operate**. They exchange information about what they have discovered in the places they have visited

- The co-operation is very simple. In the basic PSO it looks like this:
  - A particle has a neighborhood associated with it
  - A particle knows the fitness of those in its neighborhood, and uses the position of the one with the best fitness (global best)
  - This position is simply used to adjust the particle's velocity

Fitness Landscape (2D)

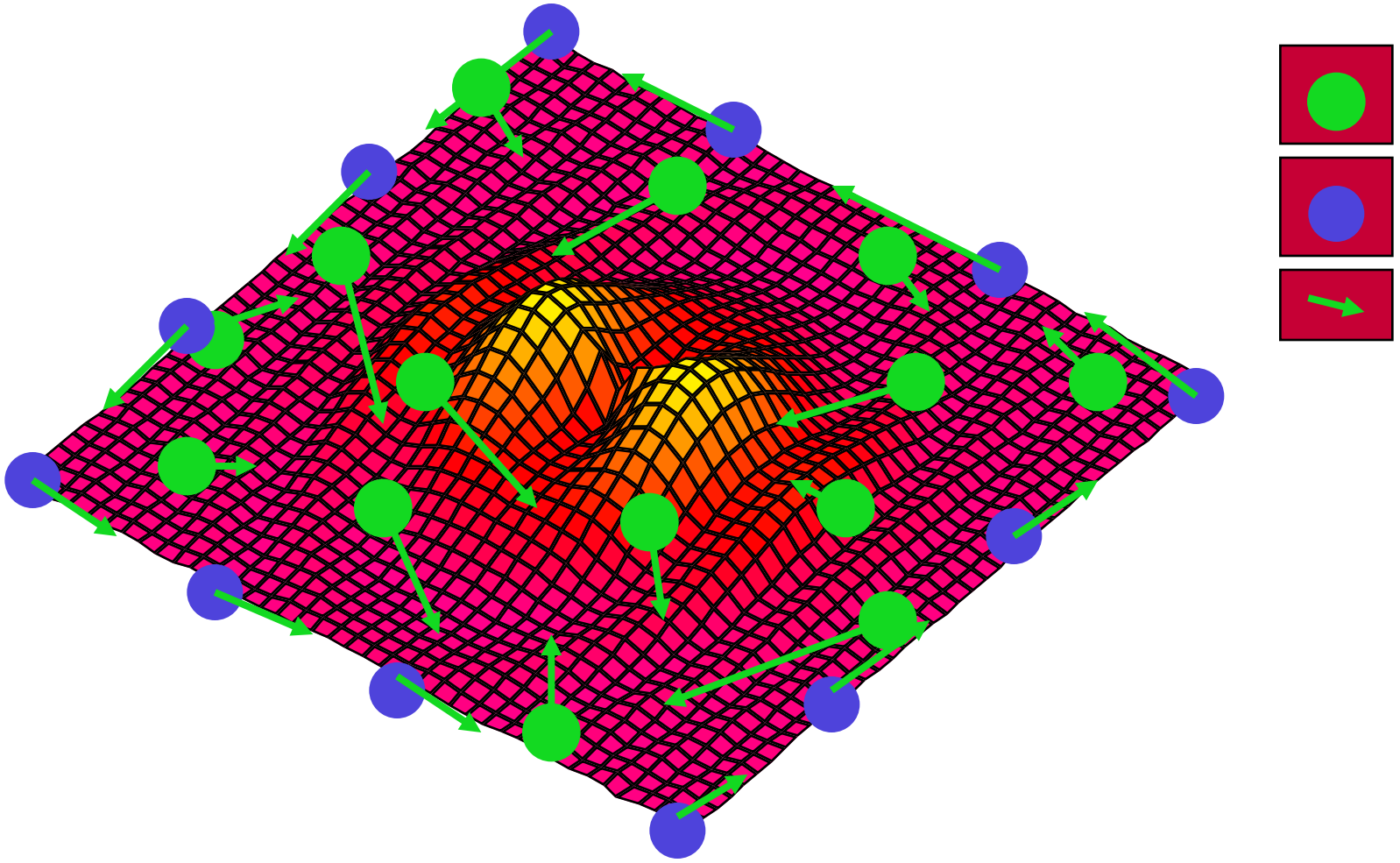


Maintained Information

- Each particle maintains:
  - **Current position** in the search space (solution and fitness)
  - **Velocity**
  - Individual **best position**

- In addition, the system maintains its global best position

Initialization: Positions and velocities



What a particle does

- In each time-step, a particle has to move to a new position. It does this by adjusting its velocity
  - A weighted random portion towards the best found by the particle so far (**local**)
  - A weighted random portion towards the best found across all particles (**global**)

- Having worked out a new velocity, its position is simply the old position + the new velocity

PSO Algorithm

- The PSO algorithm consists of just three steps:
  1. Evaluate fitness of each particle
  2. Update individual and global bests
  3. Update velocity and position of each particle

- These steps are repeated until some stopping condition is met

Velocity Update

- Each particle's velocity is updated using this equation:

$$v_i(t+1) = wv_i(t) + c_1 r_1[\hat{x}_i(t) - x_i(t)] + c_2 r_2[g(t) - x_i(t)]$$

i is the particle index

w is the inertia coefficient

c_1, and c_2 are acceleration coefficients $0 <= c_1, c_2 <= 2$ (learning factors)

r_1, and r_2 are random values ($0 <= r_1, r_2 <= 1$) regenerated every velocity update

Velocity Update

- Each particle's velocity is updated using this equation:

$$v_i(t+1) = wv_i(t) + c_1 r_1[\hat{x}_i(t) - x_i(t)] + c_2 r_2[g(t) - x_i(t)]$$

$v_i(t)$ is the particle's velocity at time t

$x_i(t)$ is the particle's position at time t

$\hat{x}_i$ is the particle's individual best solution as of time t

g(t) is the swarm's best solution as of time t

Velocity Update -- Inertia Component

$$v_i(t+1) = wv_i(t) + c_1 r_1 [\hat{x}_i(t) - x_i(t)] + c_2 r_2 [g(t) - x_i(t)]$$

- Keeps the particle moving in the same direction it was originally heading
- Inertia coefficient ***w*** usually between 0.8 and 1.2
- Lower values speed up convergence, higher values encourage exploring the search space

Velocity Update – Cognitive Component

$$v_i(t+1) = wv_i(t) + c_1 r_1[\hat{x}_i(t) - x_i(t)] + c_2 r_2[g(t) - x_i(t)]$$

- Acts as the particle's memory, causing it to return to its individual best regions of the search space
- Cognitive coefficient c_1 usually close to 2
- Coefficient limits the size of the step particle takes towards its individual best $\hat{x}_i(t)$

Velocity Update – Social Component

$$v_i(t+1) = wv_i(t) + c_1 r_1[\hat{x}_i(t) - x_i(t)] + c_2 r_2[g(t) - x_i(t)]$$

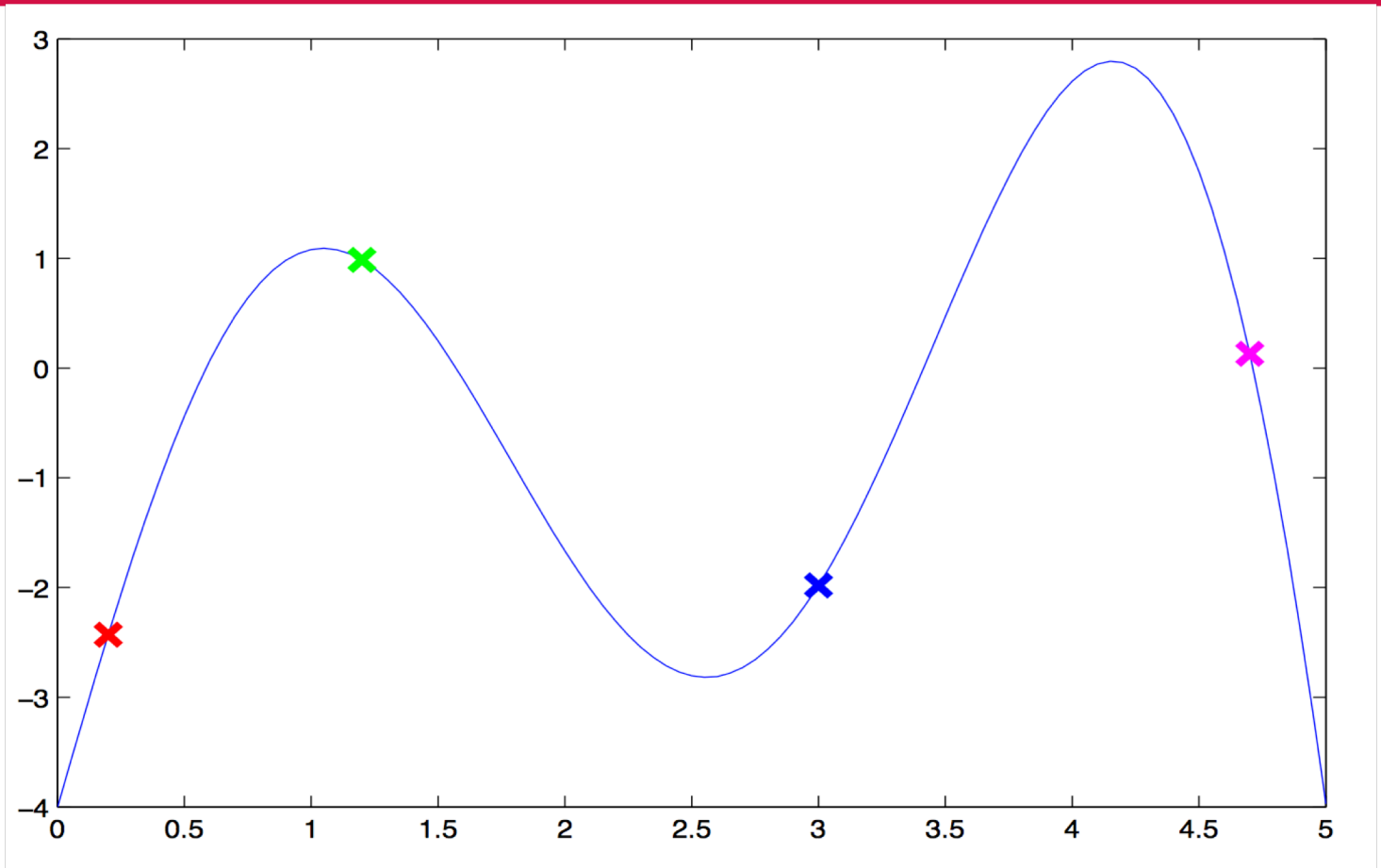
- Causes the particle to move to the best regions the swarm has found so far
- Social coefficient c_2 usually close to 2
- Coefficient limits the size of the step the particle takes towards global best g

Position update

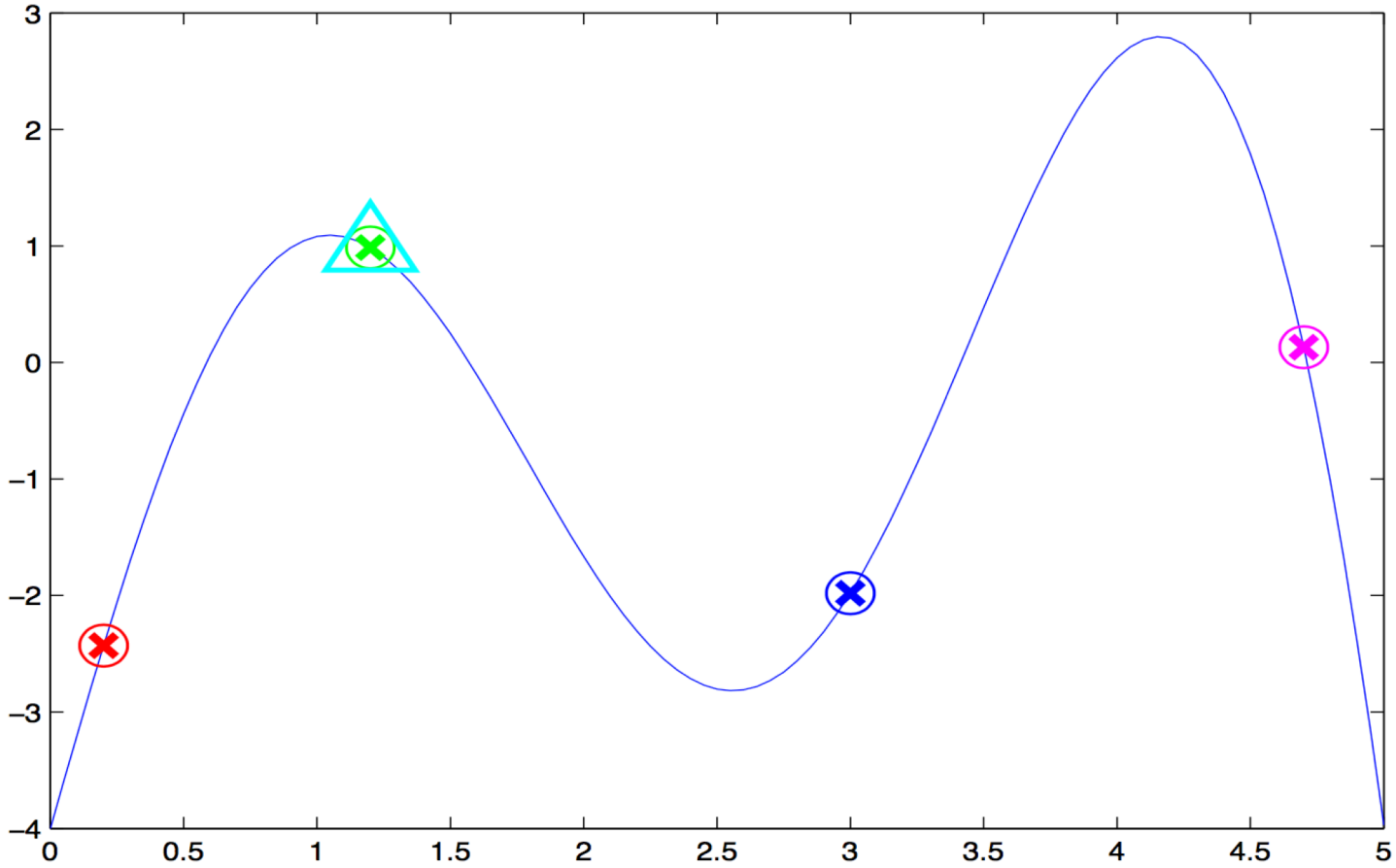
- Each particle's position is updated using this equation:

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

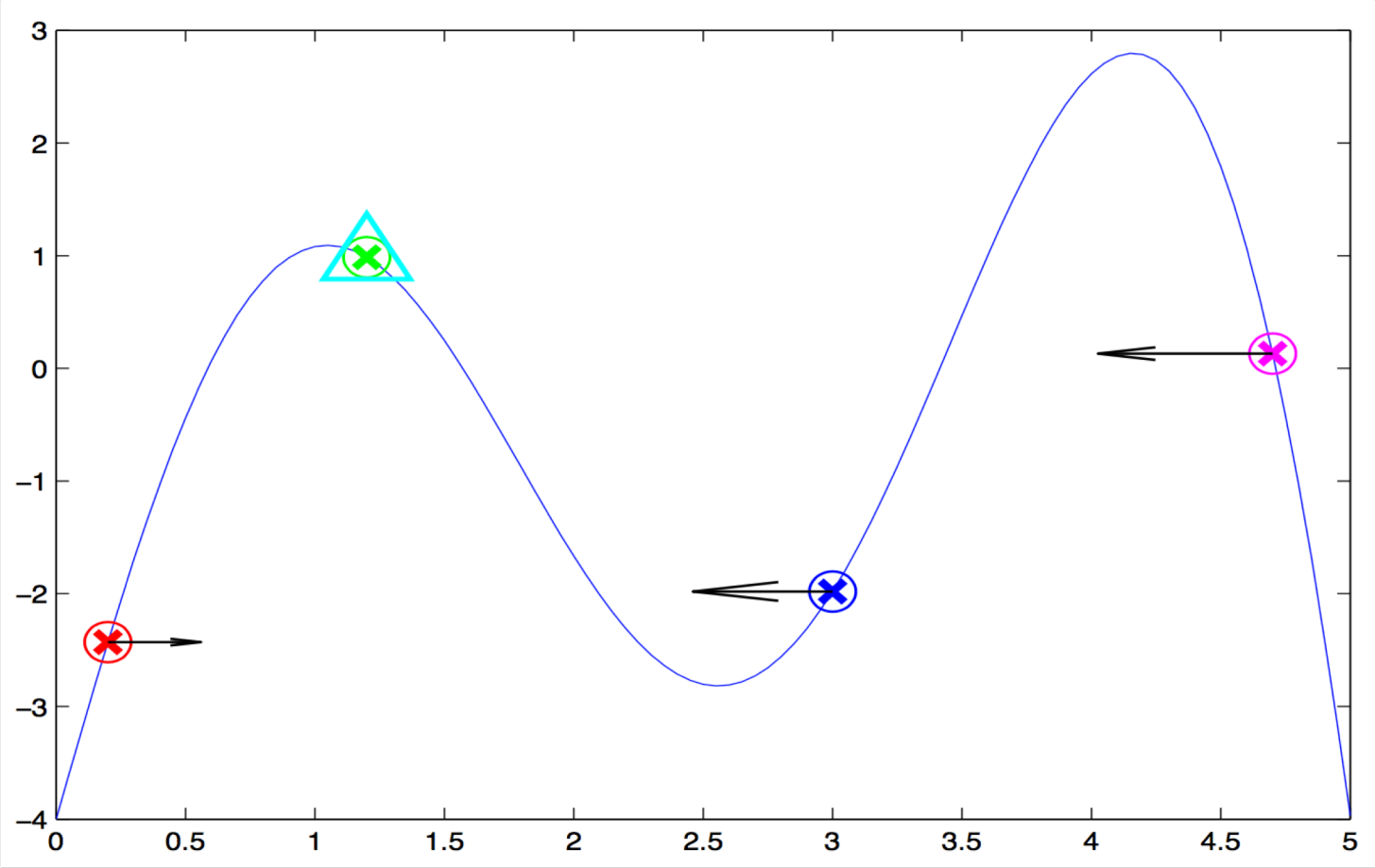
Example -- Fitness Evaluation (t=1)



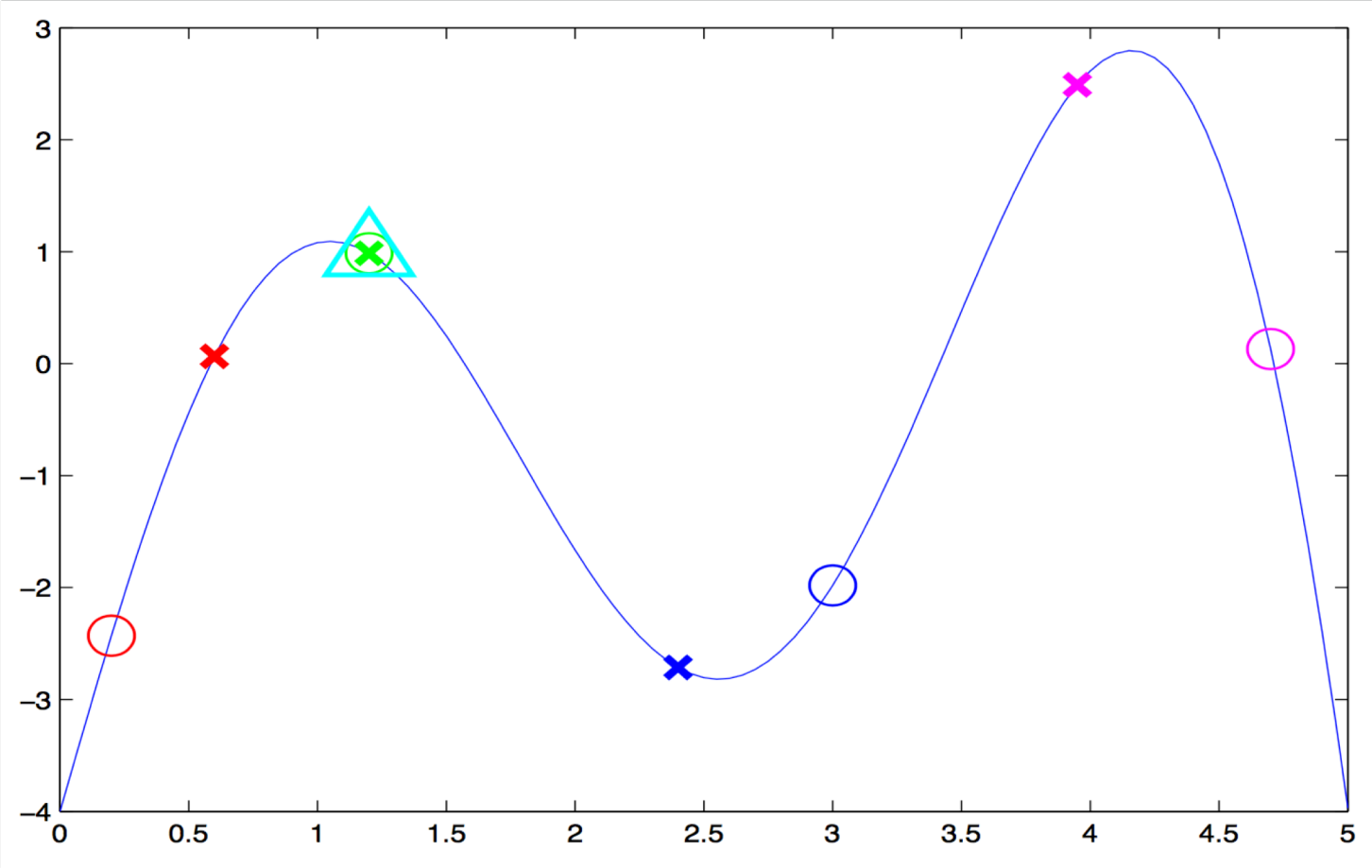
Update Individual/Global Best (t=1)



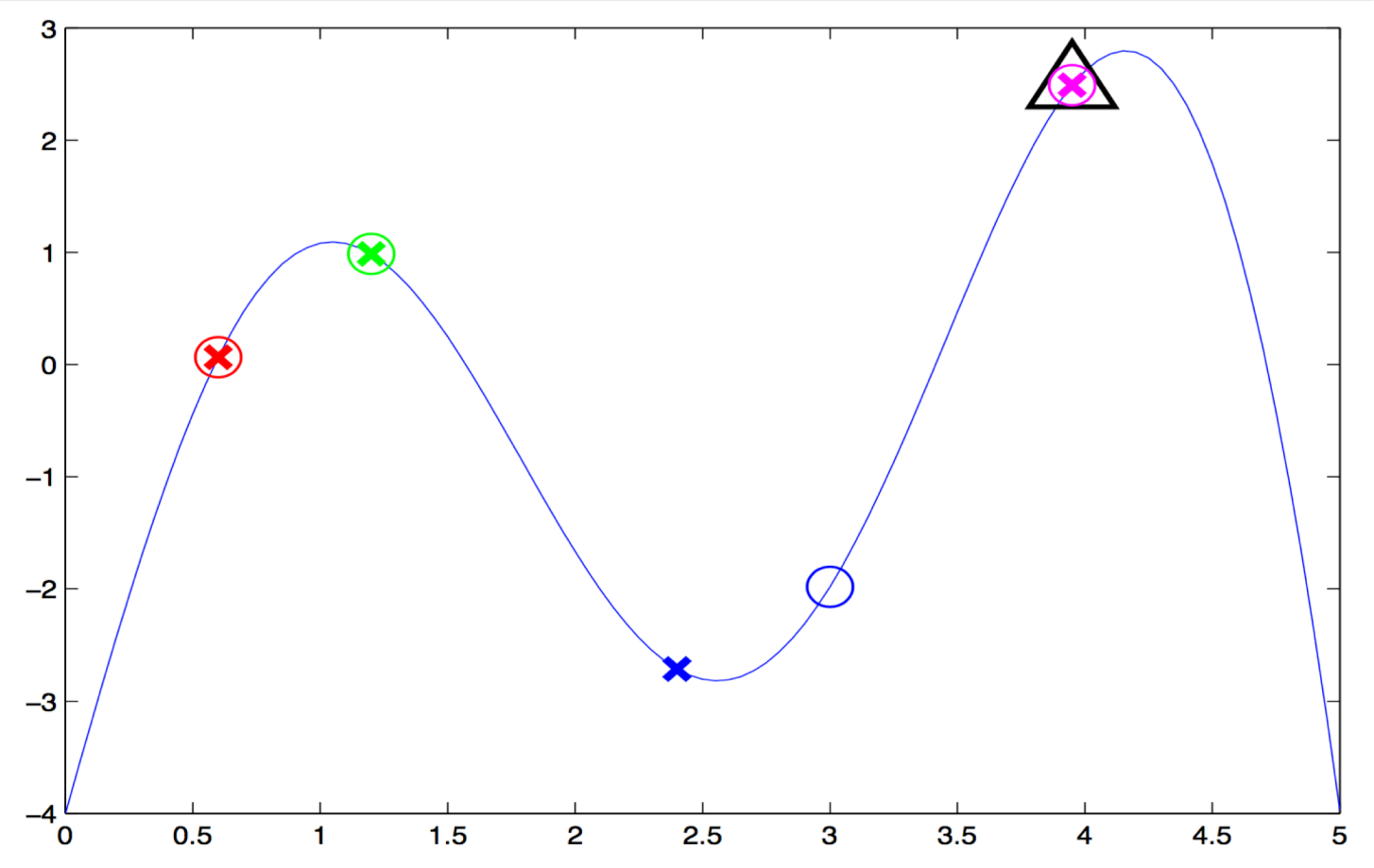
Update Velocity and Position (t=1)



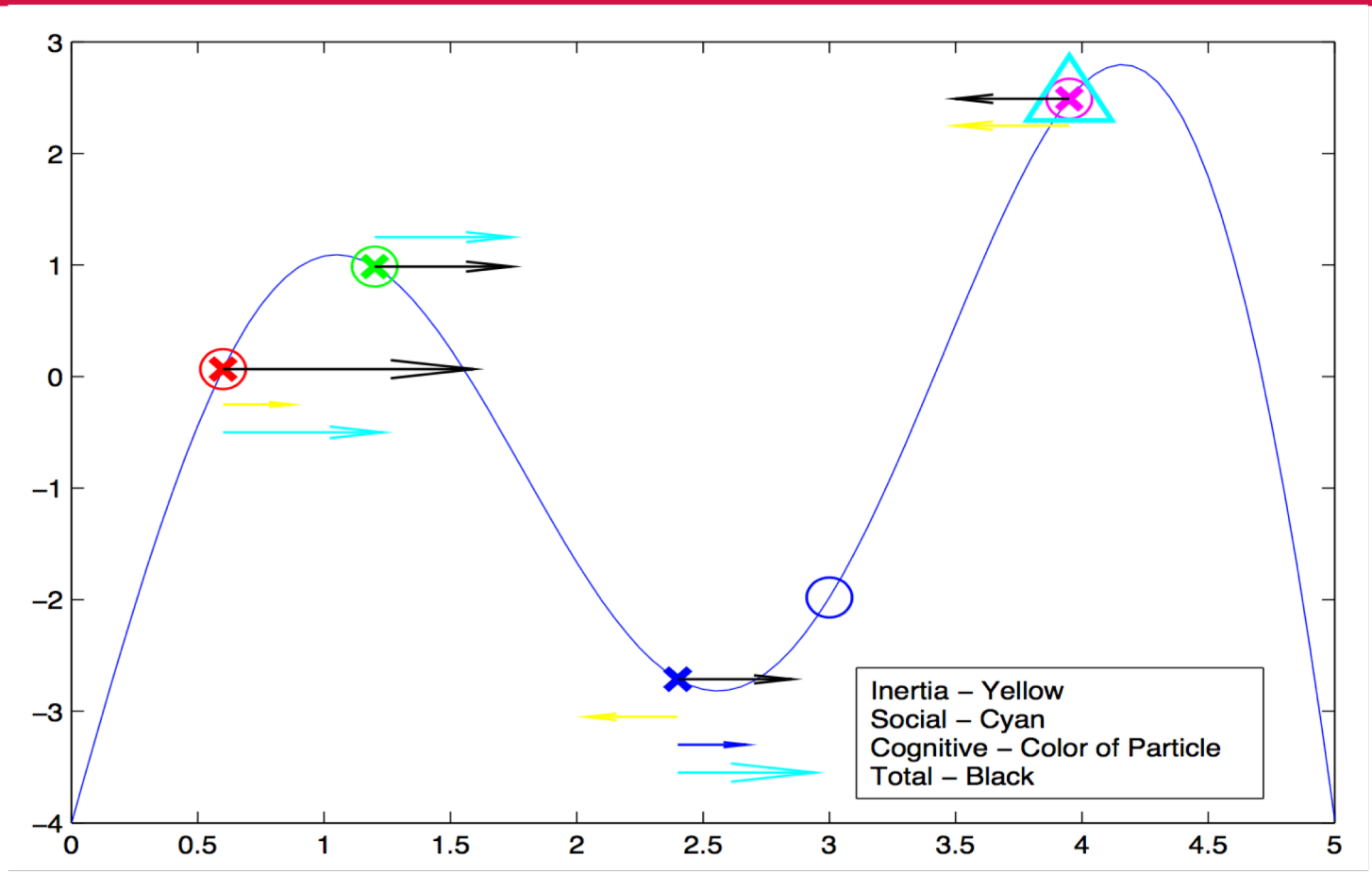
Fitness Evaluation (t=2)



Update Individual/Global Best (t=2)



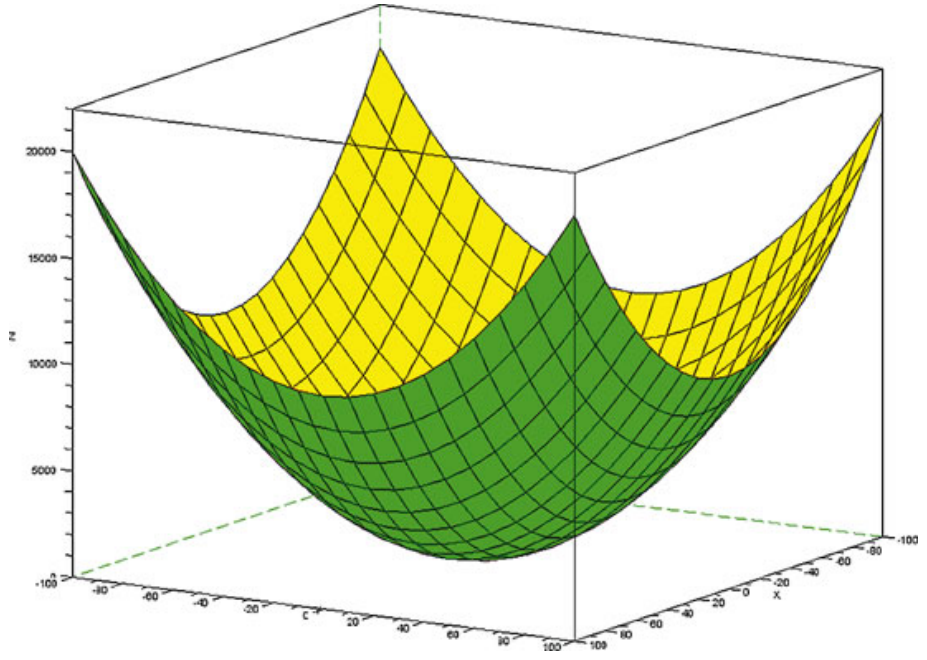
Update Velocity and Position (t=2)



PSO - Example

- Suppose we want to find the values for x_0 and x_1 that minimise the following function:
  - $f = 3 + x_0^2 + x_1$

- The solution should be $x_0 = 0$ and $x_1 = 0$



PSO Example

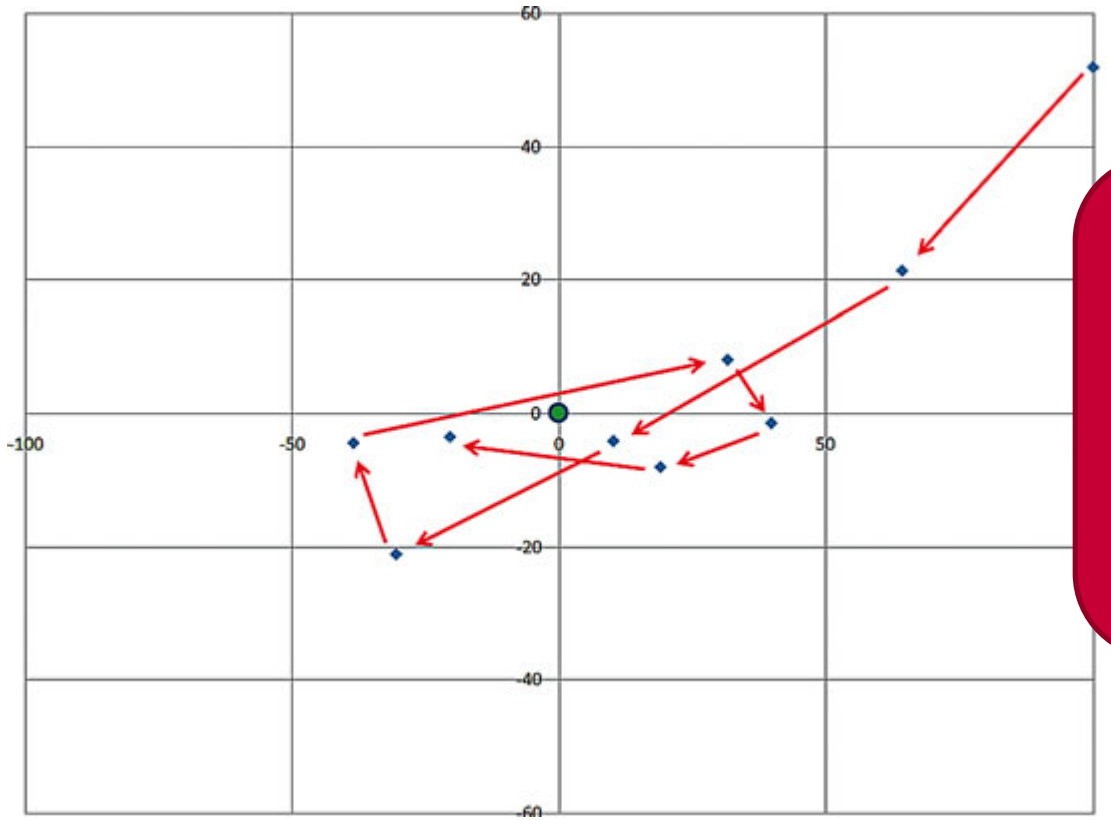
- A particle's current position, x(t), is [x0, x1] = [3.0, 4.0]
- The particle's current velocity, v(t), is [-1.0, -1.5].
- Constant w = 0.7 (inertia weight), constant c1 = 1.4 (local weight), constant c2 = 1.4 (social weight), and that random numbers r1 and r2 are 0.5 and 0.6 respectively.
- Particle's best known position is p(t) = [2.5, 3.6]
- Global best known position by any particle in the swarm is g(t) = [2.3, 3.4].

$$v(t+1) = (0.7 * \{-1.0,-1.5\}) + (1.4 * 0.5 * \{2.5, 3.6\} - \{3.0, 4.0\}) + (1.4 * 0.6 * \{2.3, 3.4\} - \{3.0, 4.0\})$$
$$= \{-0.70, -1.05\} + \{-0.35, -0.28\} + \{-0.59, -0.50\}$$
$$= \{-1.64, -1.83\}$$

$$x(t+1) = \{3.0, 4.0\} + \{-1.64, -1.83\}$$
$$= \{1.36, 2.17\}$$

Recall that the optimal solution is {x0, x1} = (0.0, 0.0}. Observe that the update process has improved the old position/solution from (3.0, 4.0} to {1.36, 2.17}.

PSO



The movement of one of the particles during the first eight iterations of a PSO run.

The particle starts at x0 = 100.0, x1 = 80.4 and tends to move toward the optimal solution of x0 = 0, x1 = 0.

PSO vs. GAs -- Commonalities

- PSO and GA are both population based stochastic optimization
- Both algorithms start with a group of a randomly generated population
- Both have fitness values to evaluate the population
- Both update the population and search for the optimum with random values
- Neither system can guarantee success

PSO vs. GAs -- Differences

- PSO does not have genetic operators like crossover and mutation. Particles update themselves with the internal velocity.
- They also have memory, which is important to the algorithm
- Particles do not die
- The information sharing mechanism in PSO is significantly different
  - Info from best to others, GA population moves together

PSO vs. GAs

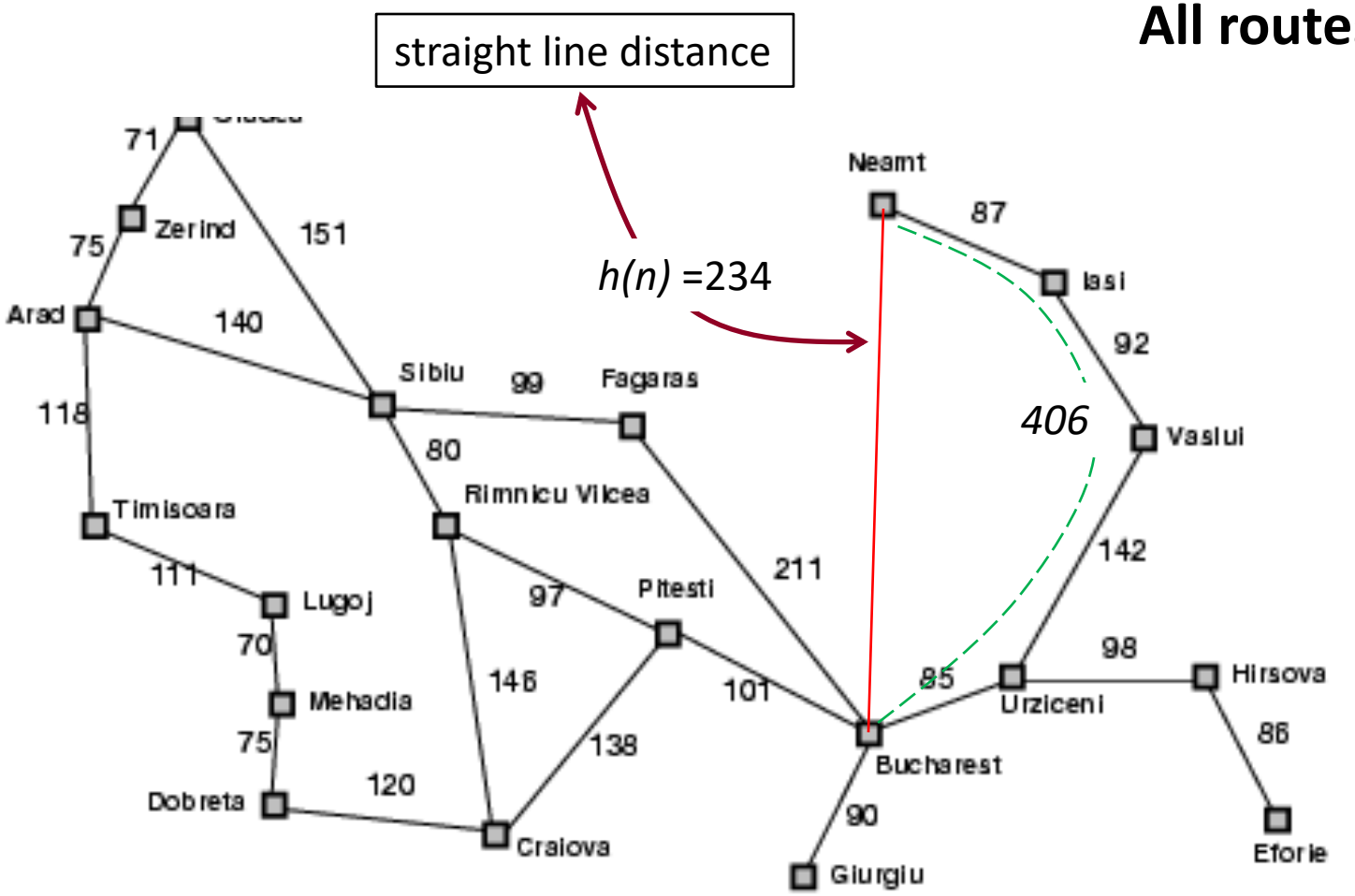
- PSO has a **memory**: not ***what*** the best solution was, but ***where*** that best solution was
- **Quality**: population responds to quality factors pBest *x* and gBest *g*
- **Diverse response**: responses allocated between pBest and gBest
- **Stability**: population changes state only when gBest changes
- **Adaptability**: population does change state when gBest changes

Local Search

Local Search

- GAs looked at combining good solutions
  - But don’t look at why/where a solution is good or bad.
- Local search methods try and make a change not at random, but one that will result in an improvement in the objective value or fitness of the solution.
- As with GA, need to have a method to generate initial solution.

Greedy best-first search

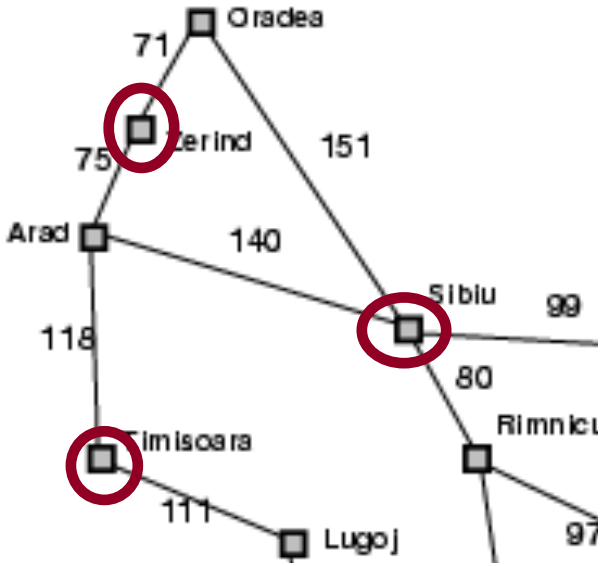


All routes lead to ~~Rome~~ Bucharest

Euclidean Distance

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

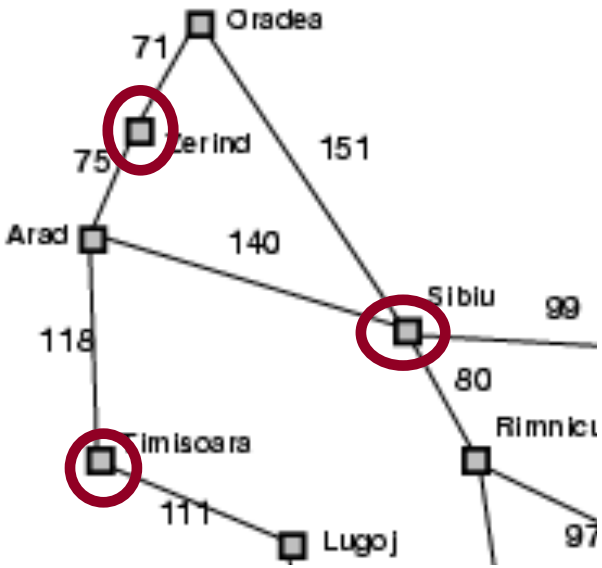
Greedy best-first search example

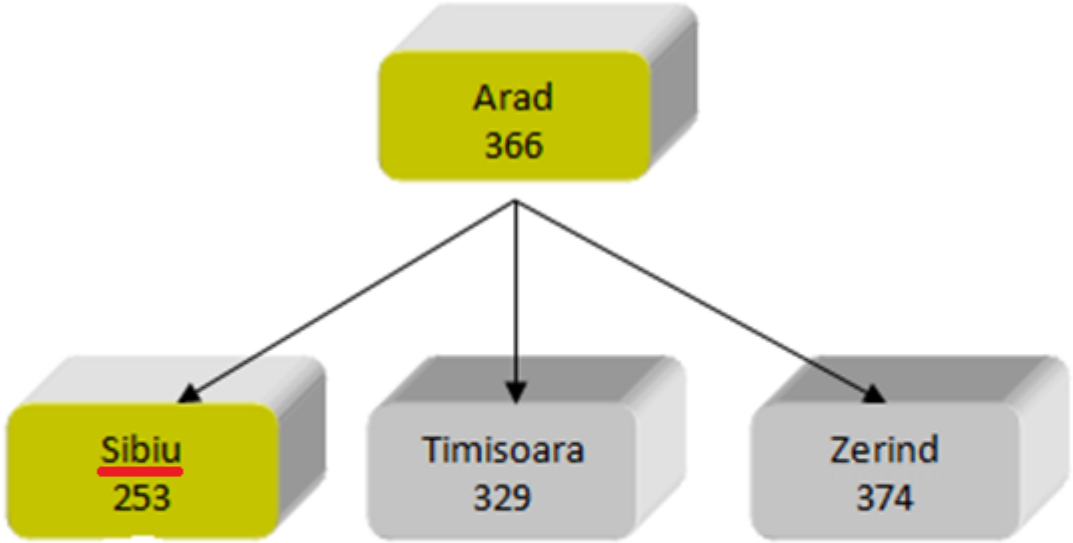


Euclidean Distance

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

Greedy best-first search example

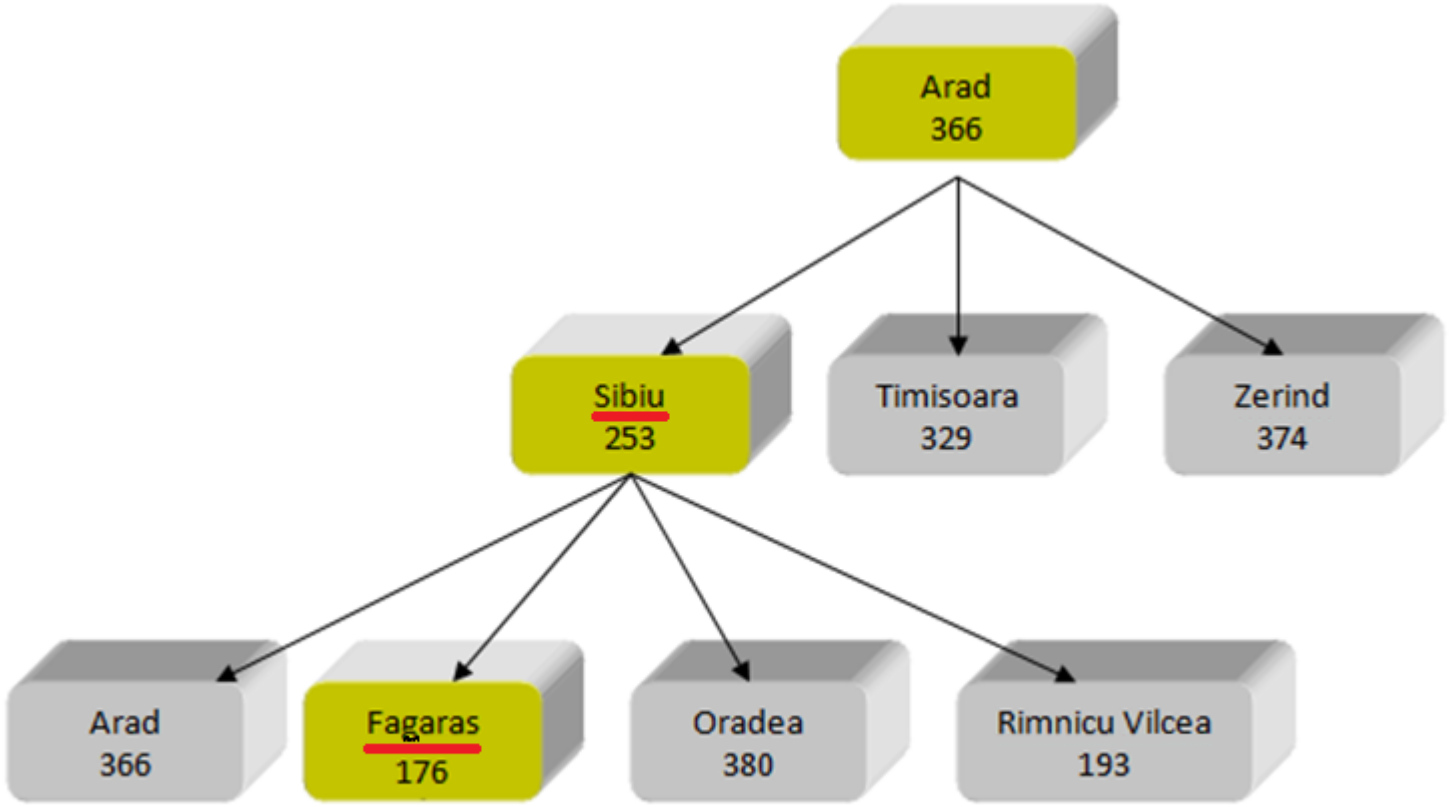




Euclidean Distance

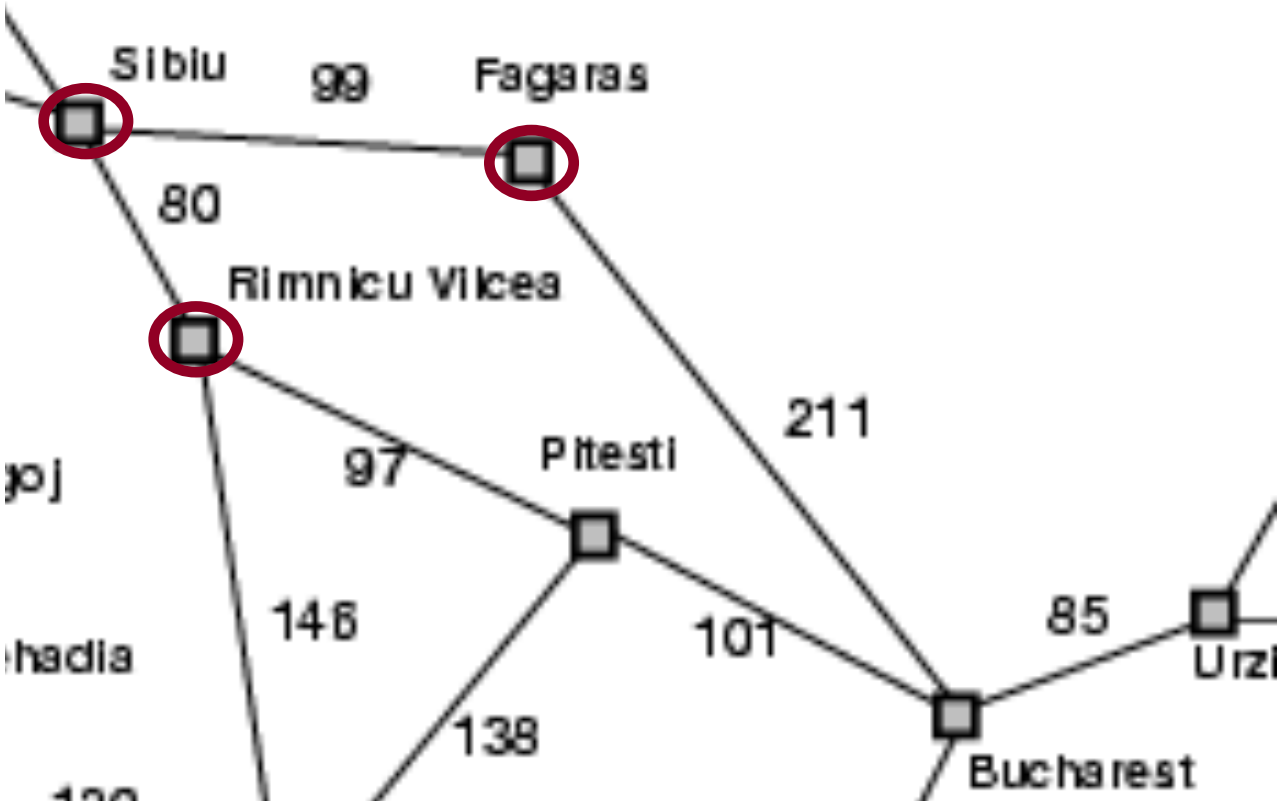
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
→ Sibiu	253
→ Timisoara	329
Urziceni	80
Vaslui	199
→ Zerind	374

Greedy best-first search example



Greedy best-first search

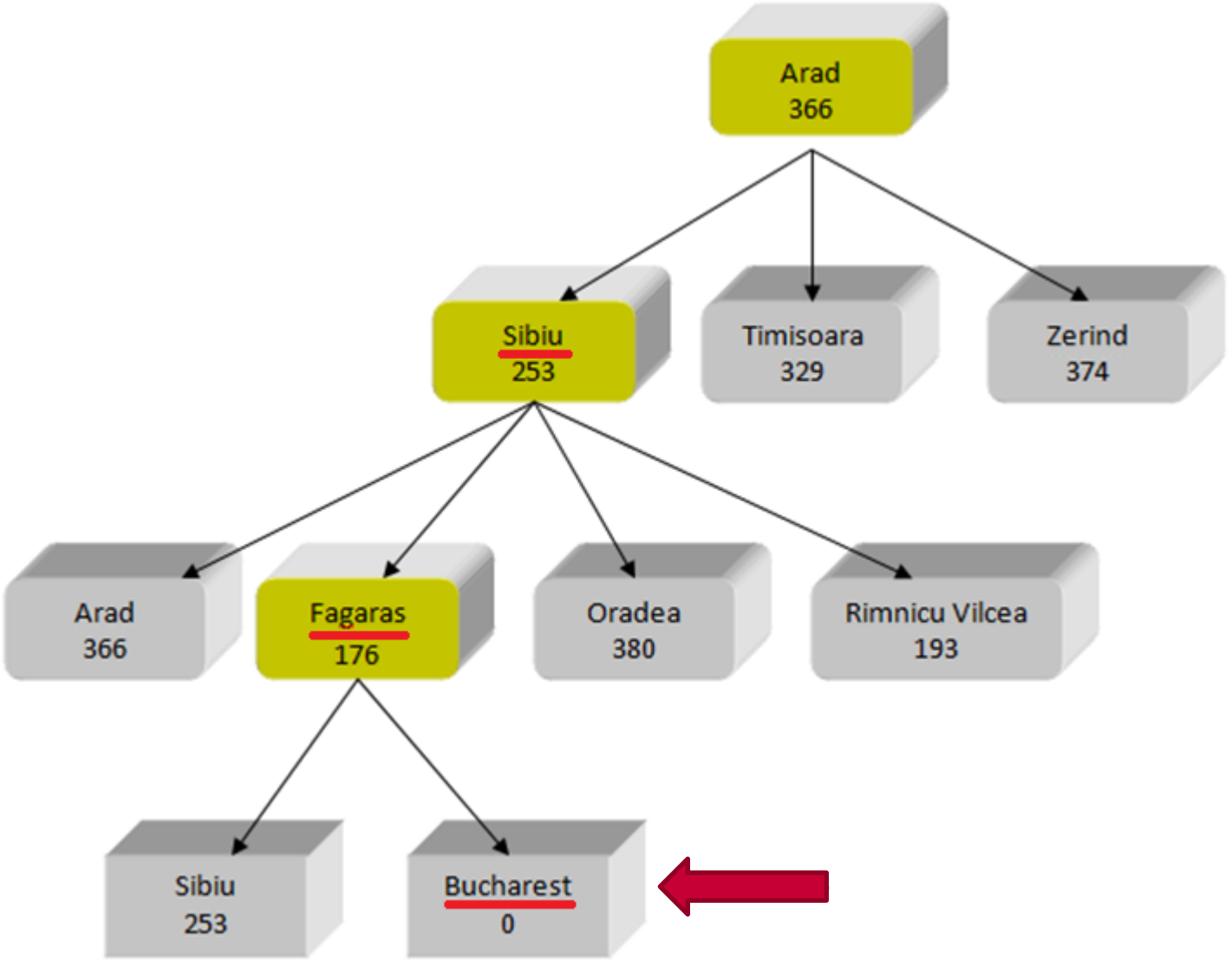
All routes lead to ~~Rome~~ Bucharest



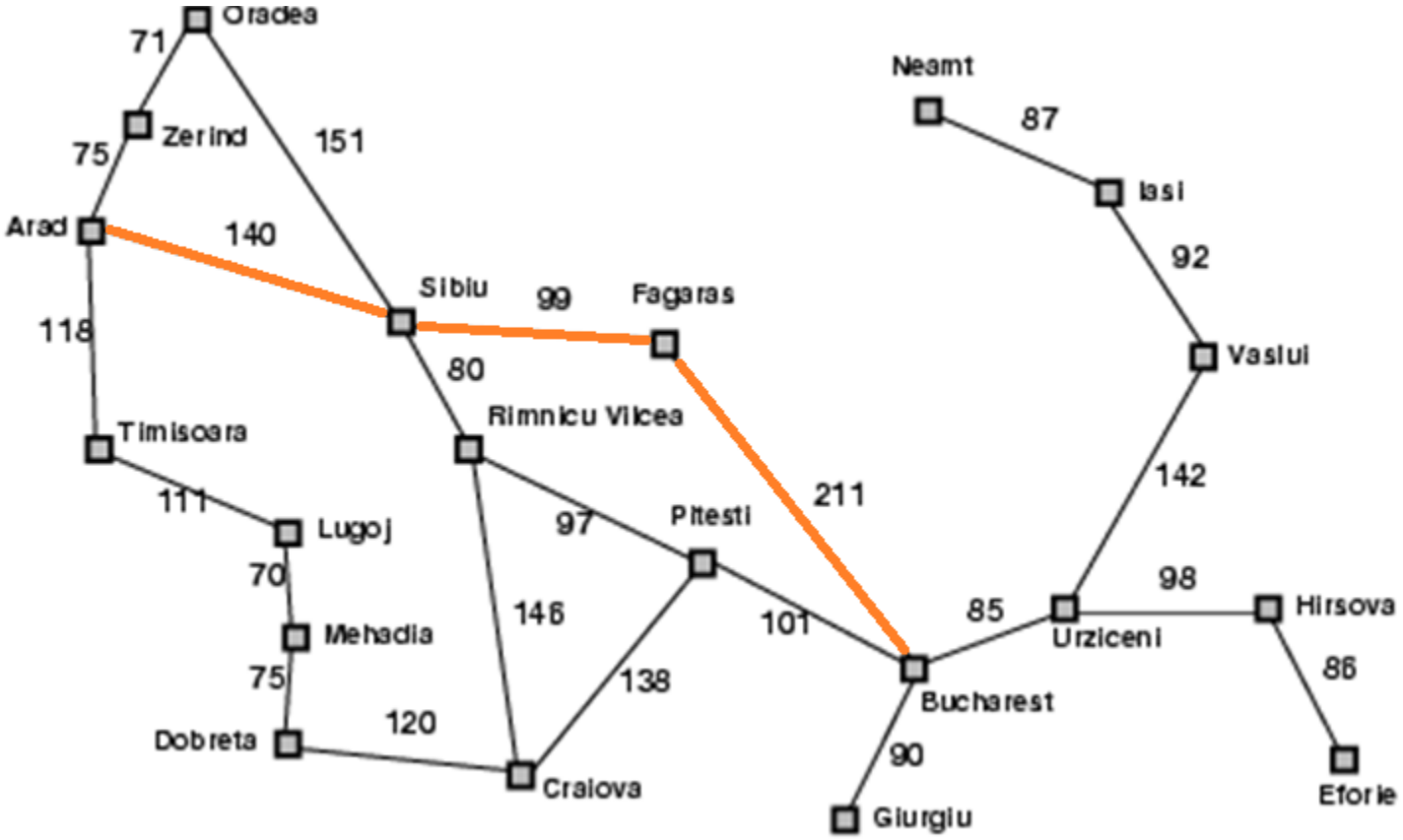
Euclidean Distance

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

Greedy best-first search example



Greedy best-first search example



Arad ➔ Sibiu ➔ Fagaras ➔ Bucharest

Euclidean Distance

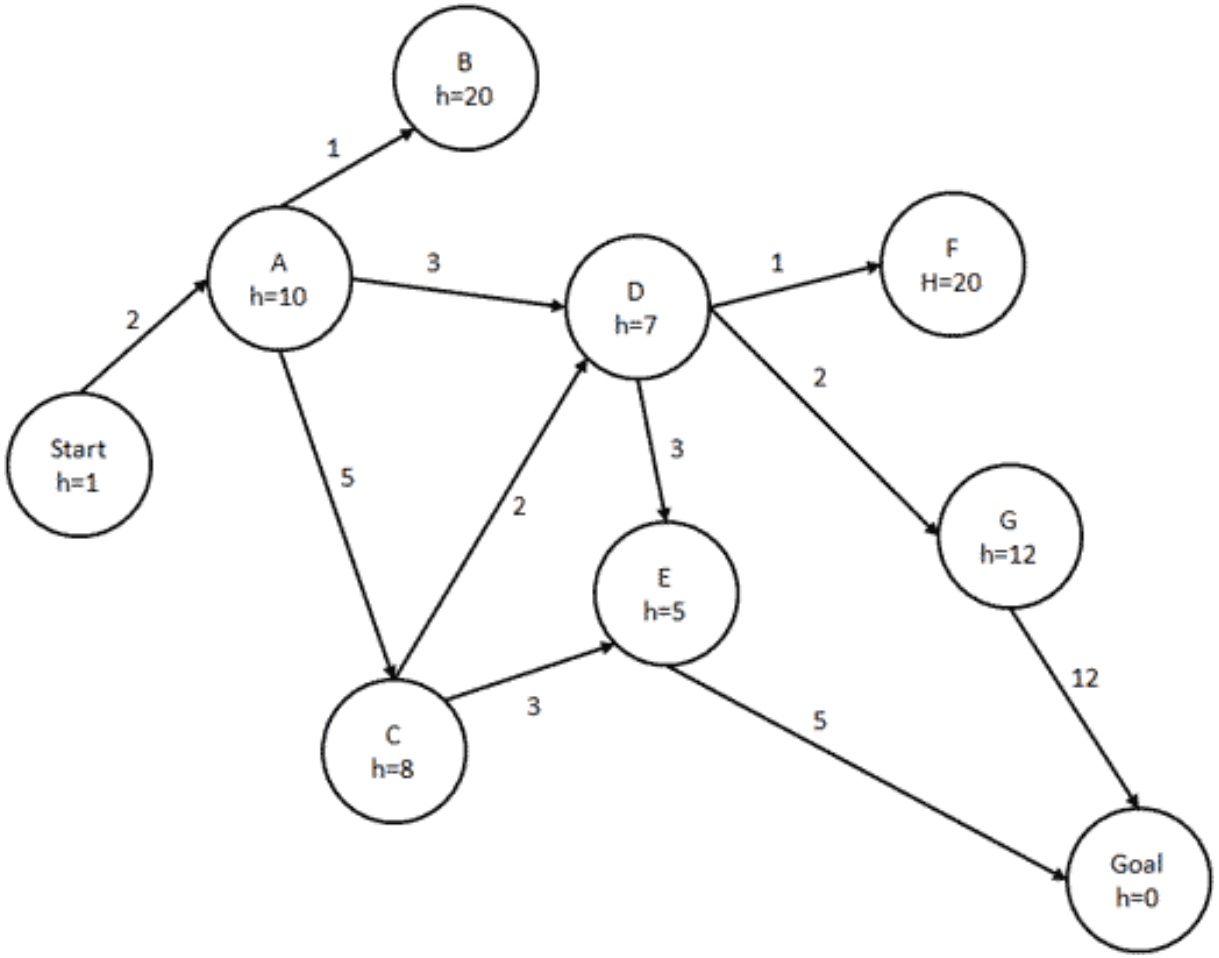
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Properties of greedy best-first search

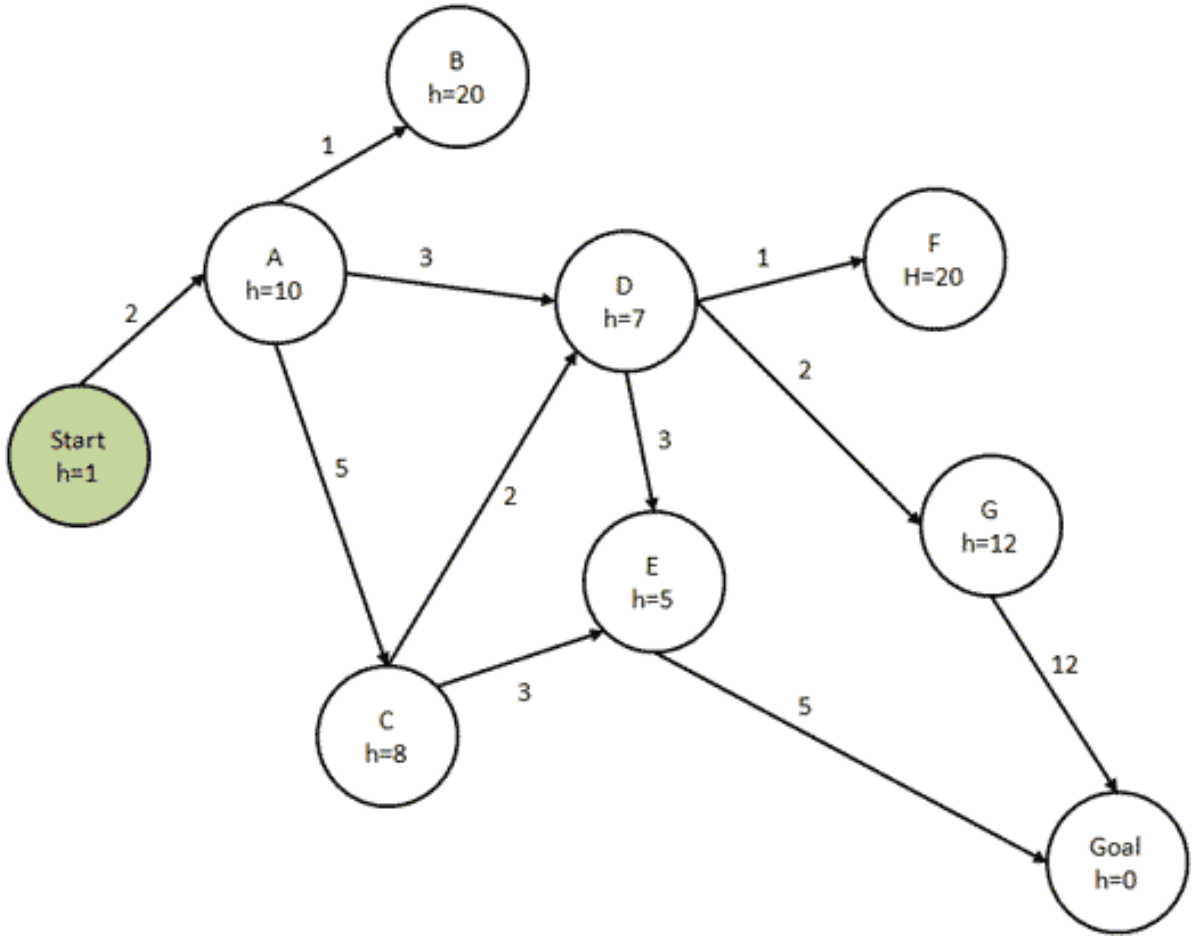
- **Complete?**
  - No – can get stuck in loops!
  - E.g. best move from city A is city D, but best move from city D is city A

- But, complete in finite space with repeated state elimination,
  - i.e. can’t return to previous state.

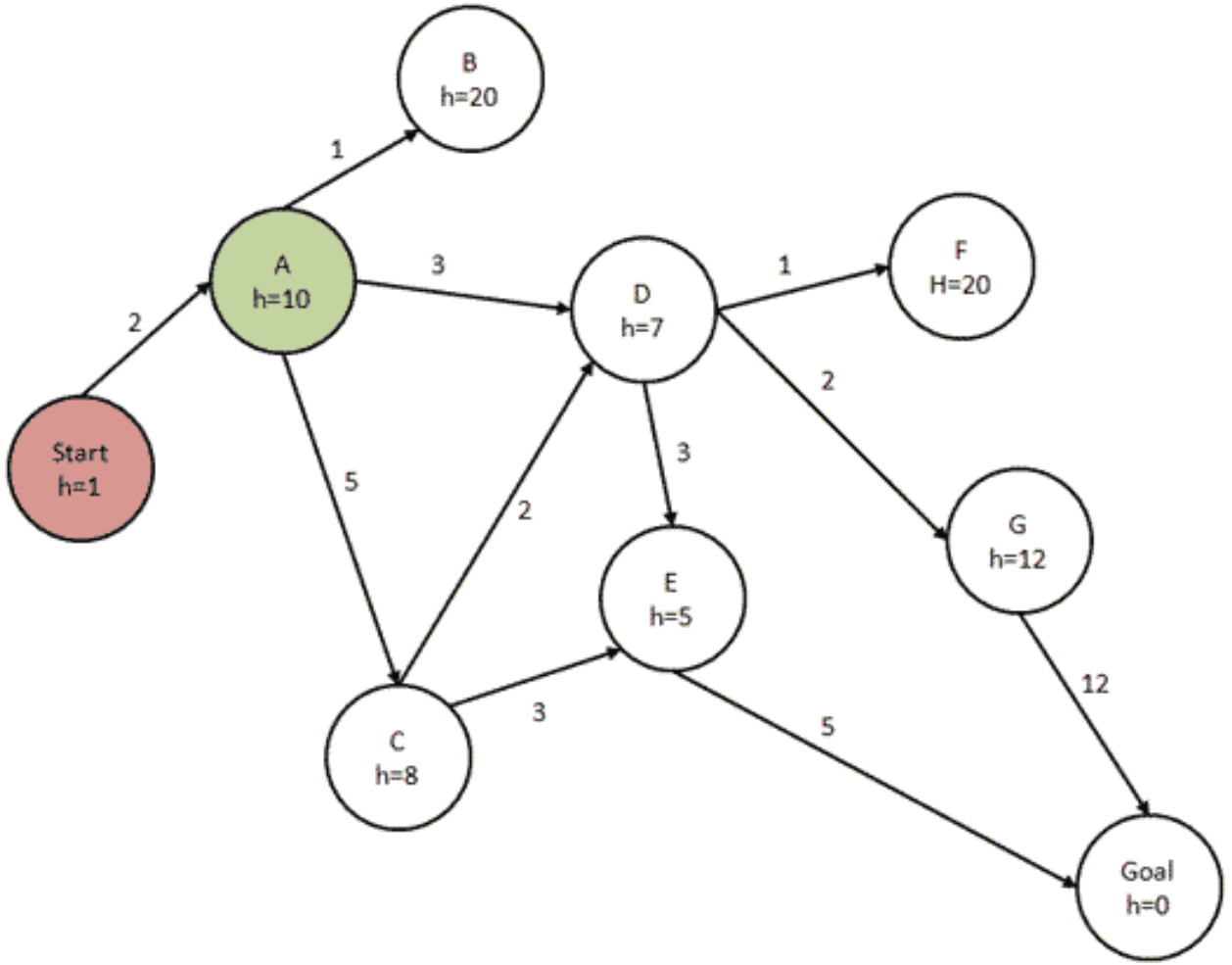
Another example



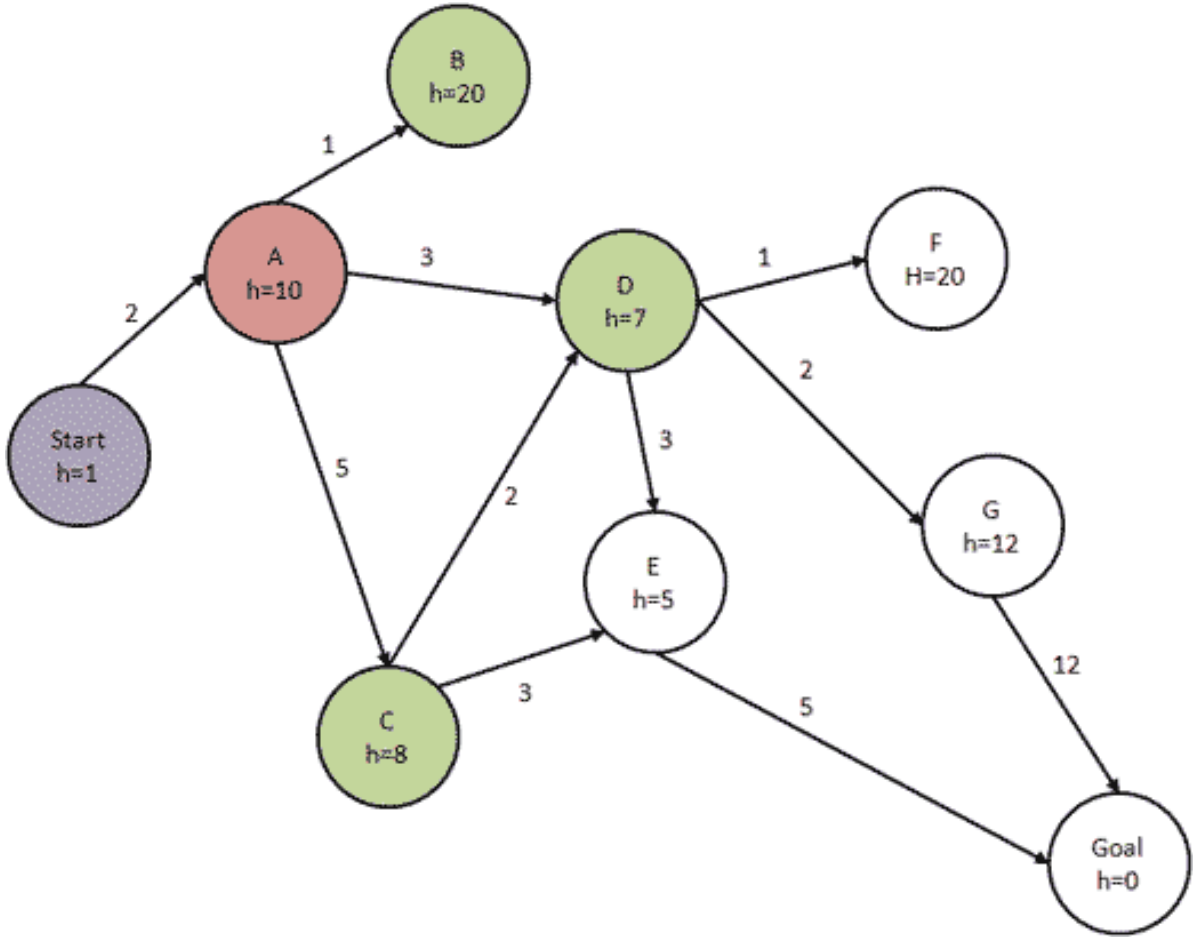
Another example



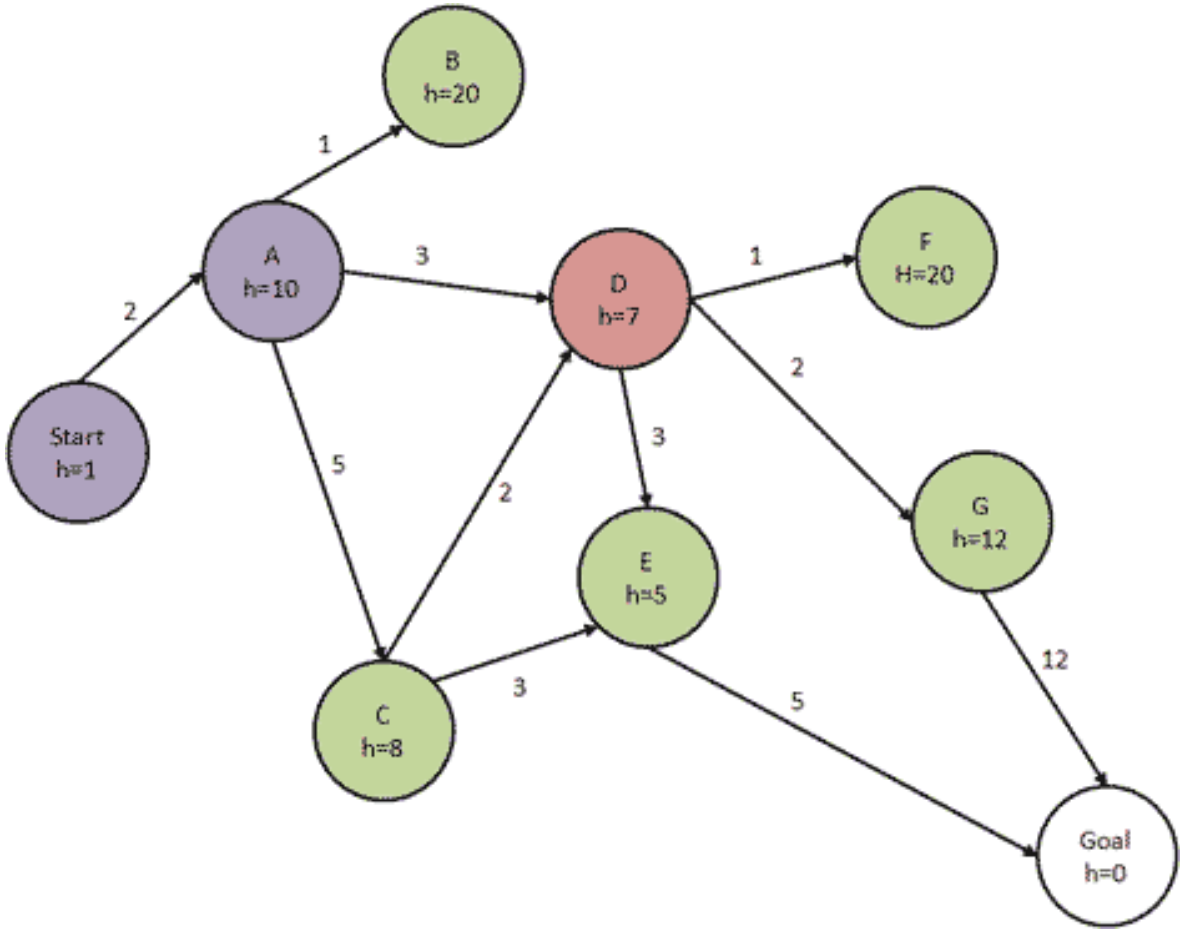
Another example



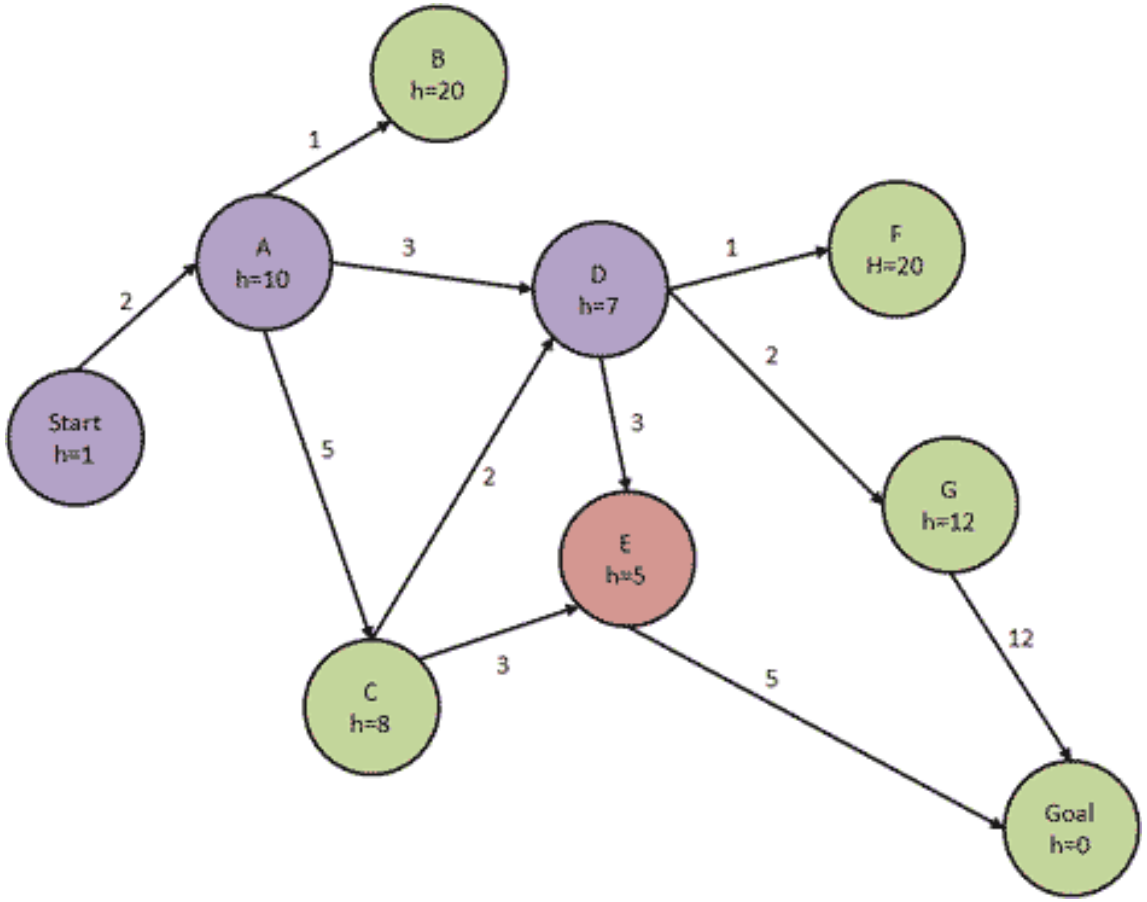
Another example



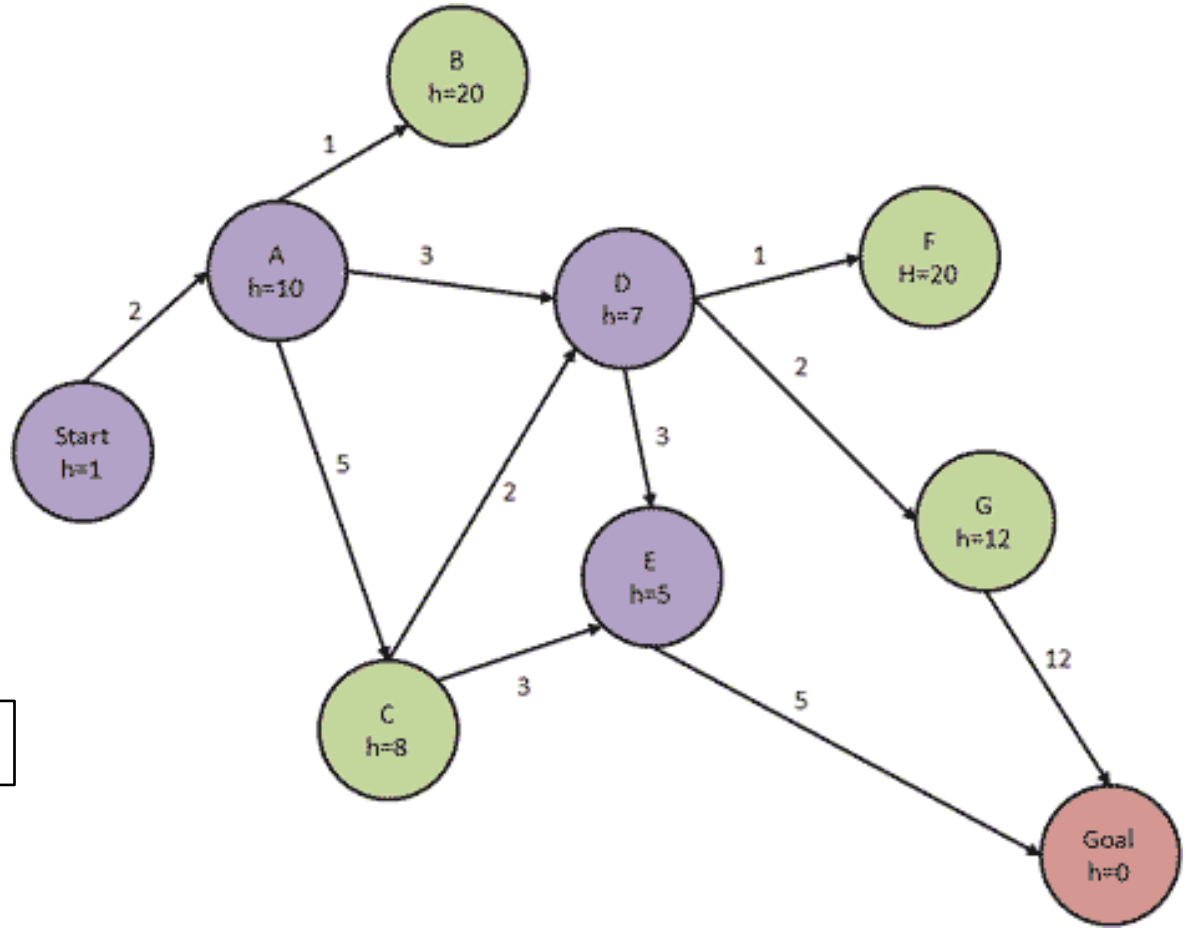
Another example



Another example

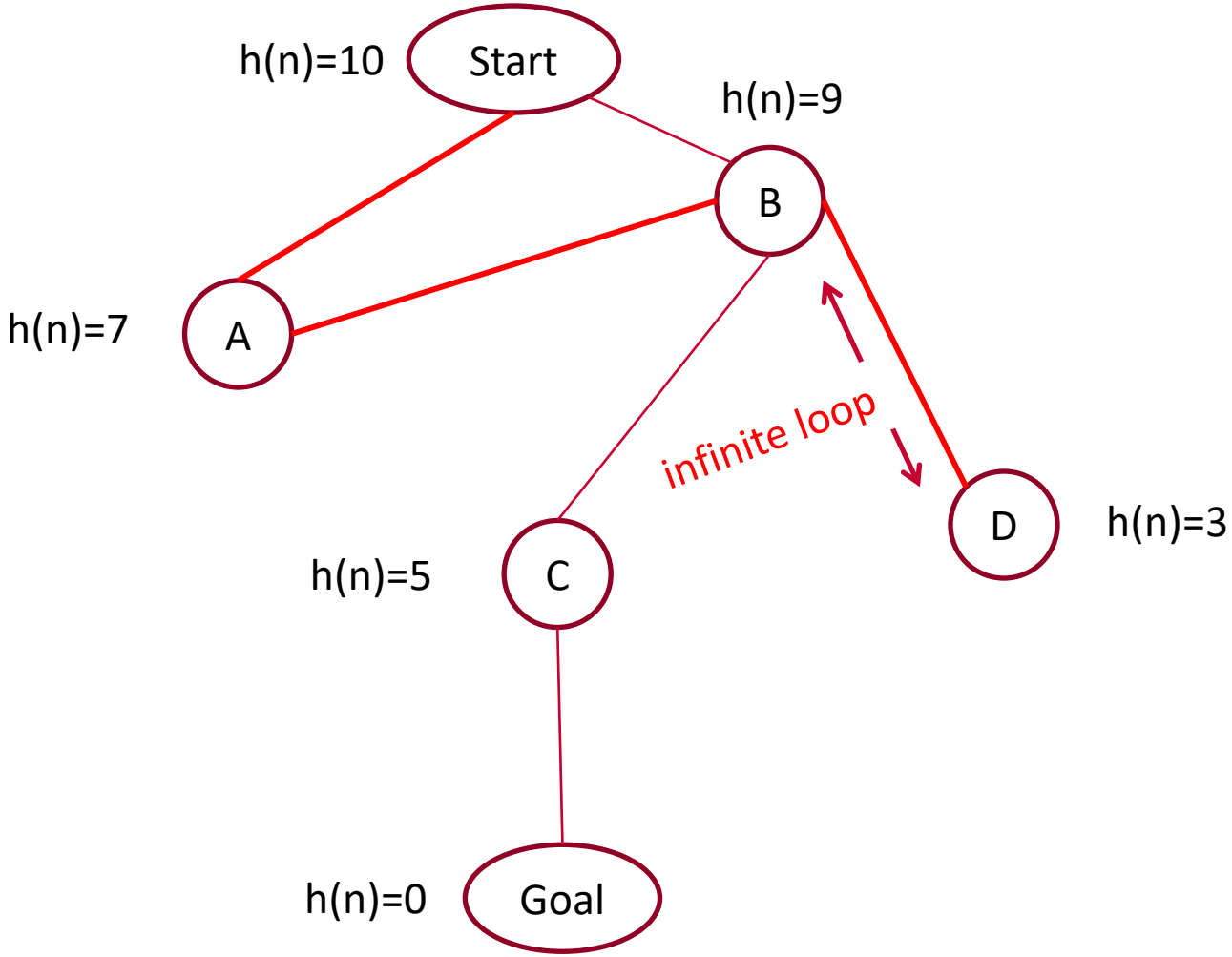


Another example



Start ➔ A ➔ D ➔ E ➔ Goal

Greedy best-first infinite loop



Local search and optimization

- Local search
  - Keep track of single current state
  - Move only to neighboring states
  - Ignore paths

- Advantages:
  - Uses very little memory
  - Can often find reasonable solutions in large or infinite (continuous) state spaces.

- “Pure Optimization” problems
  - All states have an objective function
  - Goal is to find state with max (or min) objective value
  - Local search can do quite well on these problems.

Representation

	1	2	3	4	5	6	7	8
1						♛		
2			♛					♛
3	♛							
4				♛				
5		♛						
6								
7							♛	
8					♛			

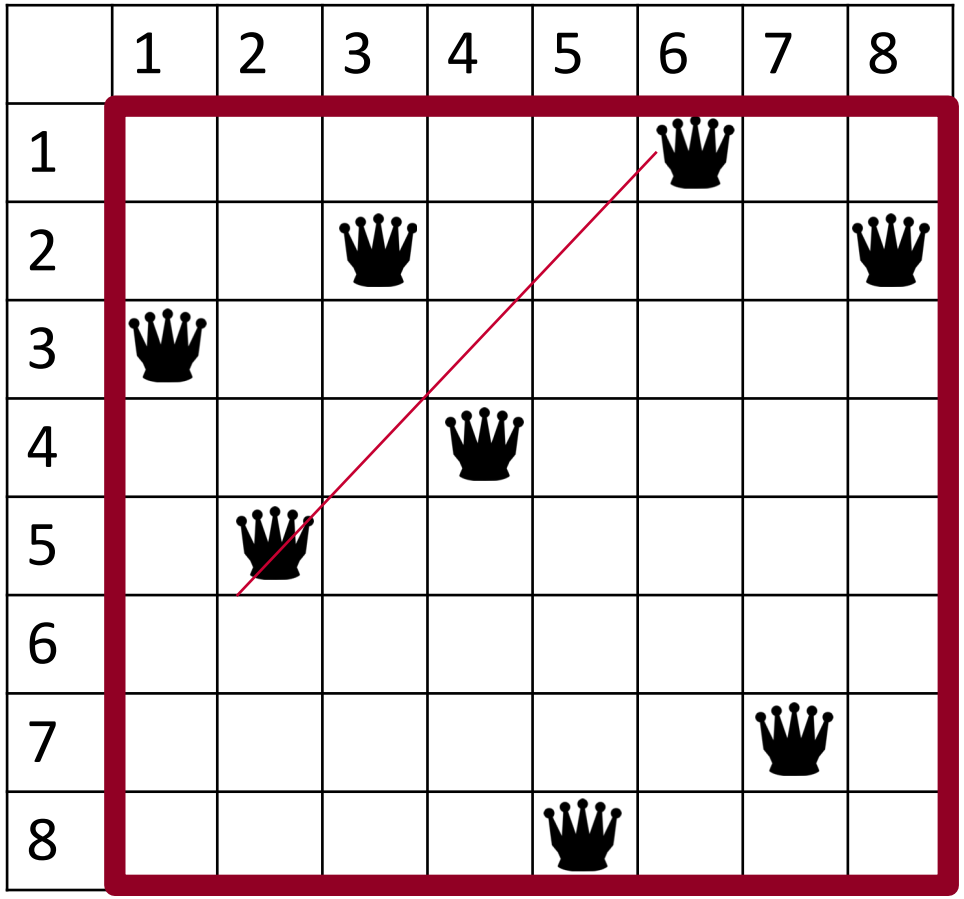
n variables

- Domain:
  - n^2?
  - n?

What's the difference?

- d^n
- For 8-Queens there are 16,777,216 configurations.
- Or 281,474,976,710,656 configurations with n^2

Representation

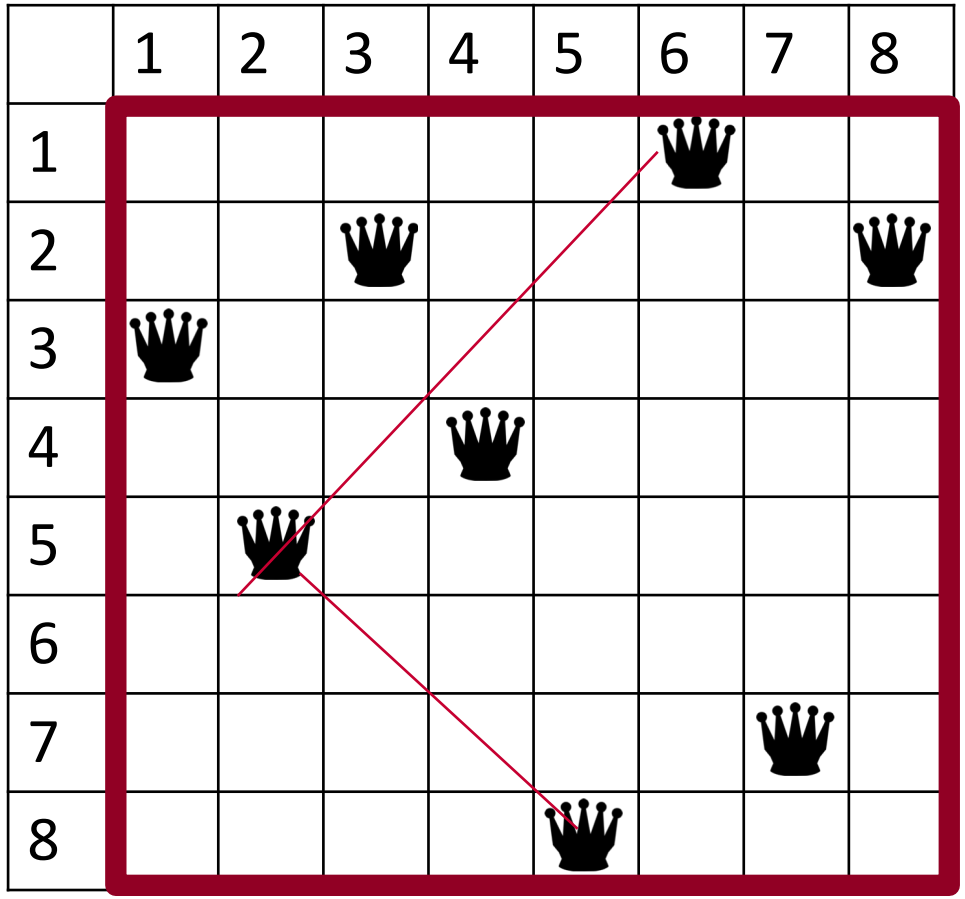


3, 5, 2, 4, 8, 1, 7, 2

Pair of Attacking queens:

?

Representation

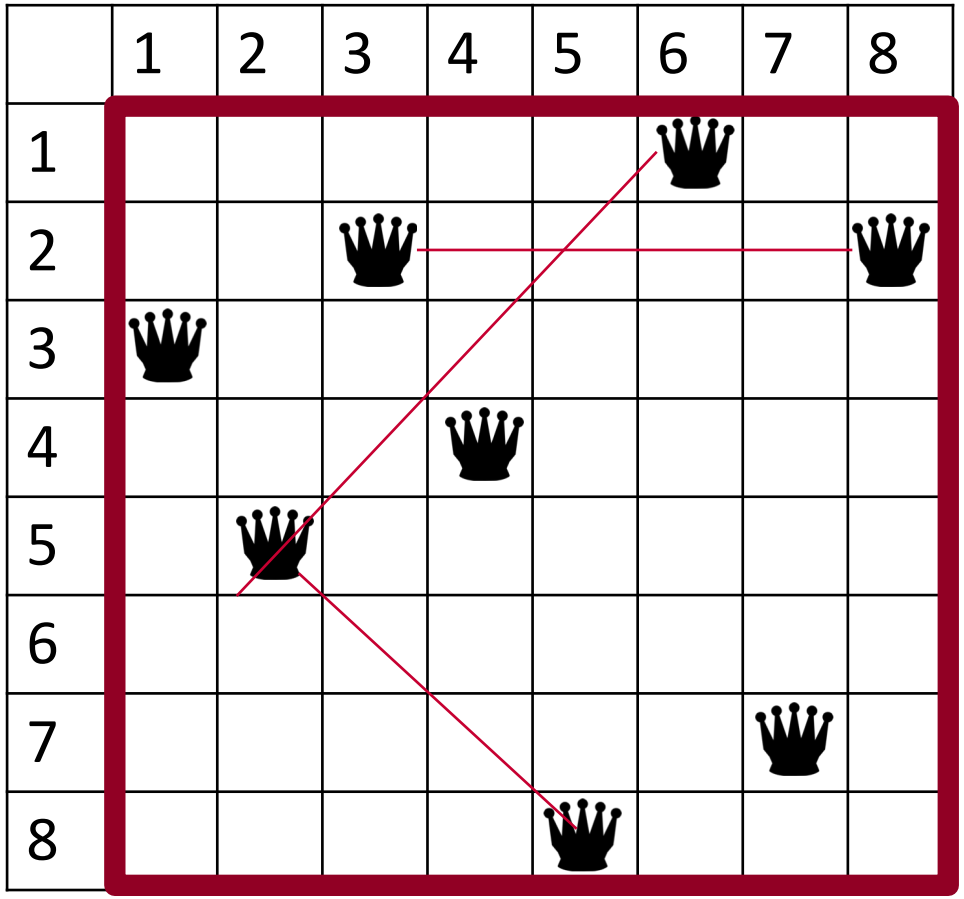


3, 5, 2, 4, 8, 1, 7, 2

Pair of Attacking queens:

?

Representation

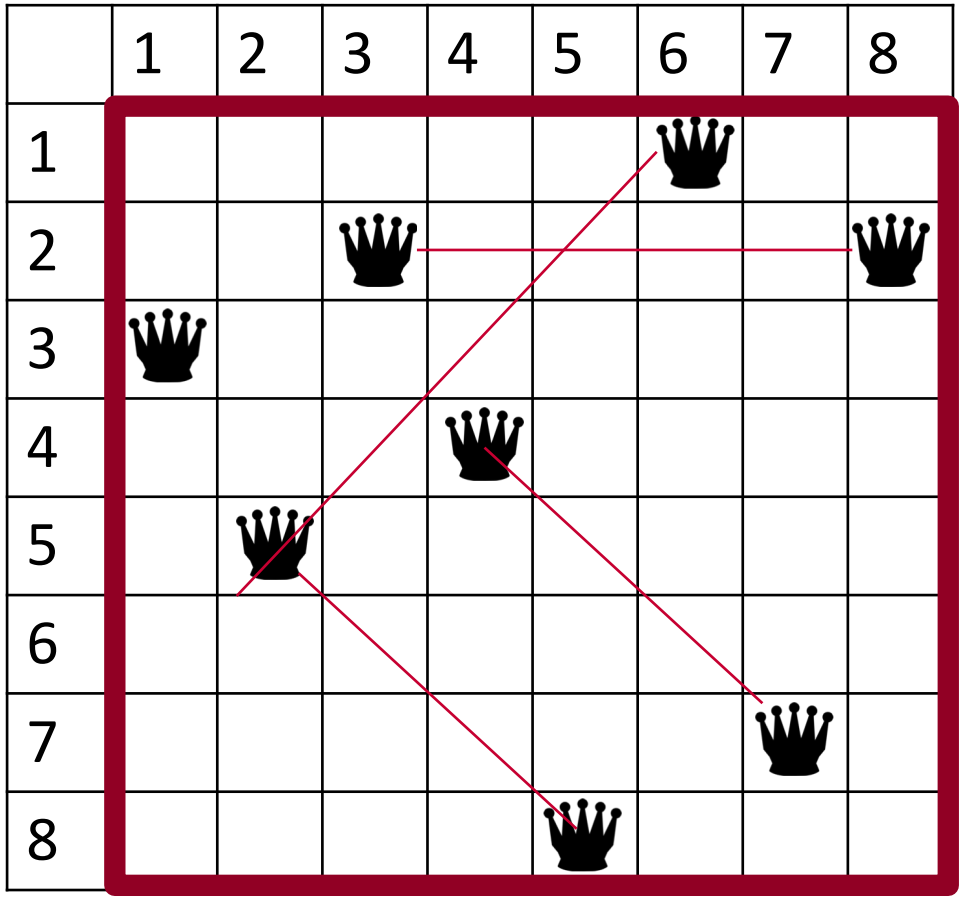


3, 5, 2, 4, 8, 1, 7, 2

Pair of Attacking queens:

?

Representation

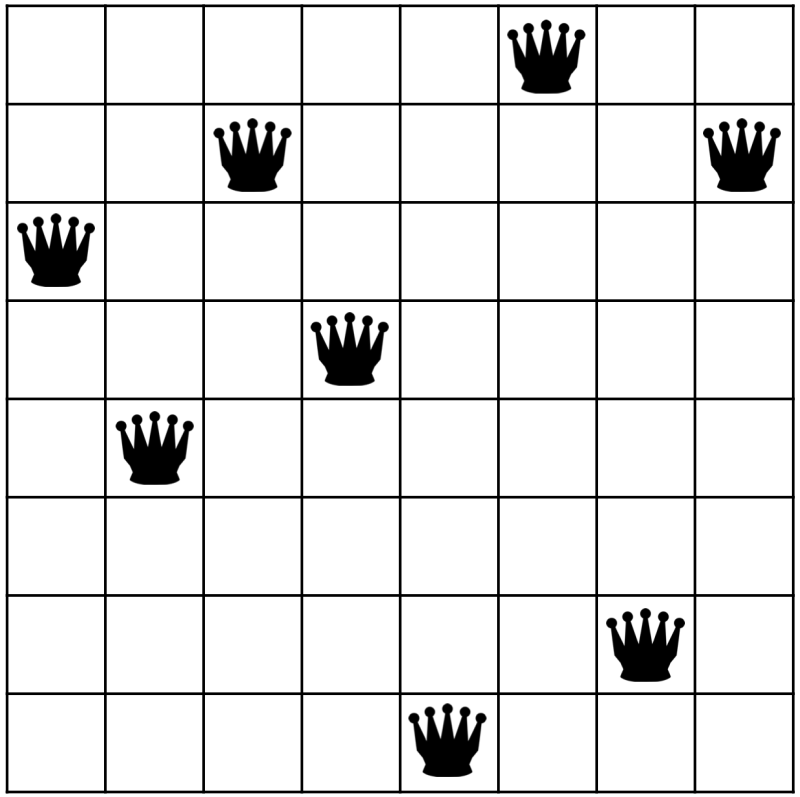


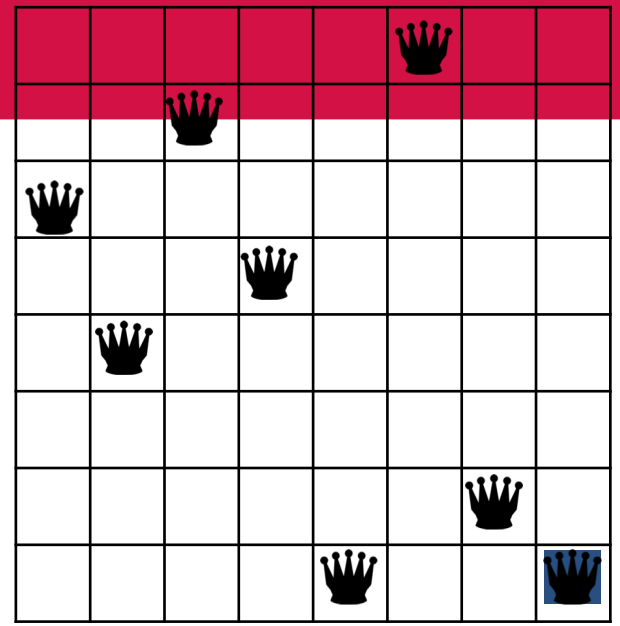
3, 5, 2, 4, 8, 1, 7, 2

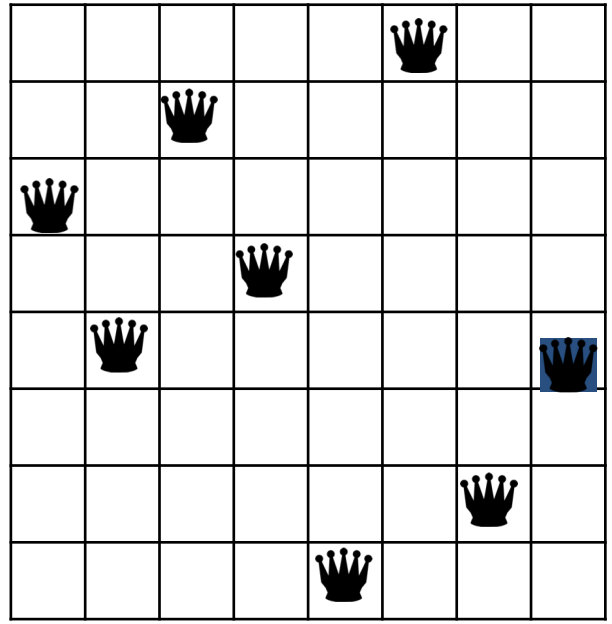
Pair of Attacking queens:

4

Neighbors: N-queens





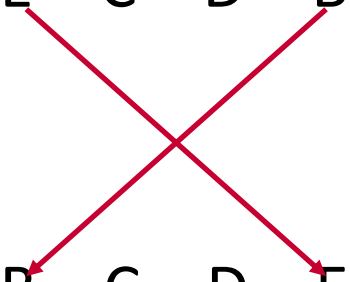


Neighbors: TSP

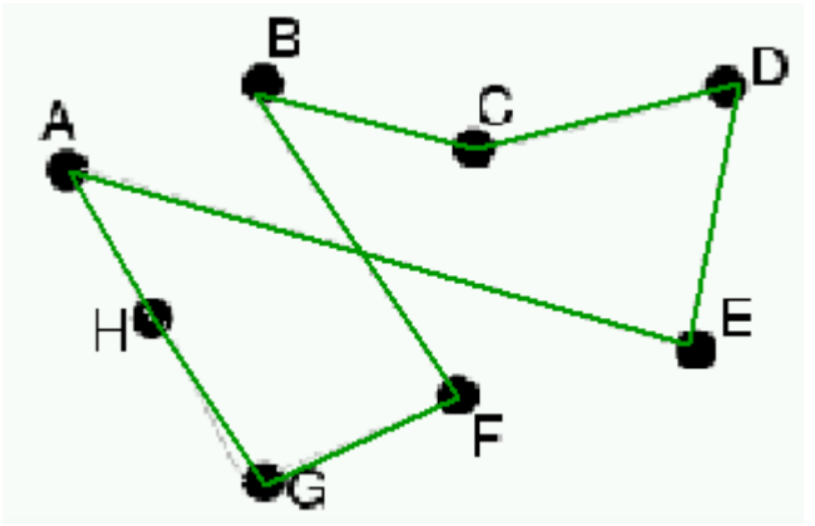
State: A – E – C – D – B – F – G – H – A

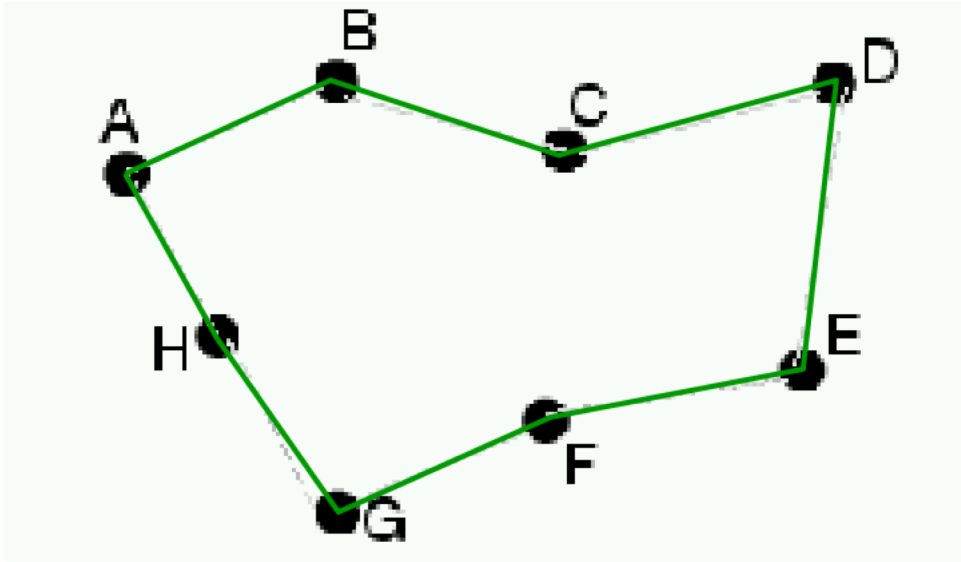
One possibility: 2-change

A – E – C – D – B – F – G – H – A



A – B – C – D – E – F – G – H – A





Neighbors: SAT

State: (A=T, B = F, C=T, D=T, E=T)

Neighbor: Flip the assignment of one variable

(**A=F**, B = F, C=T, D=T, E=T)

(A=T, **B = T**, C=T, D=T, E=T)

(A=T, B = F, **C=F**, D=T, E=T)

(A=T, B = F, C=T, **D=F**, E=T)

(A=T, B = F, C=T, D=T, **E=F**)

$$A \vee \neg B \vee C$$
$$\neg A \vee C \vee D$$
$$B \vee D \vee \neg E$$
$$\neg C \vee \neg D \vee \neg E$$
$$\neg A \vee \neg C \vee E$$

Hill Climbing

- What's a neighbor?
  - The neighborhood must be small enough for efficiency
  - Designing the neighborhood is critical.

- Pick which neighbor?
- What if no neighbor is better than the current state?

Hill Climbing

- What's a neighbor?
  - The neighborhood must be small enough for efficiency
  - Designing the neighborhood is critical.
- Pick which neighbor? → **The Best one (Greedy)**
- What if no neighbor is better than the current state? → **Stop**

Aside:

- **Constrained optimization problem to unconstrained optimization problem**
- **Objective function includes penalty for each constraint violated**
- **Cost of a single constraint violation should be greater than the cost of the worst solution to the constrained optimization problem**
- **Best neighbor will never increase the number of violated constraints**

Hill-climbing (Greedy LS) Maximisation Version

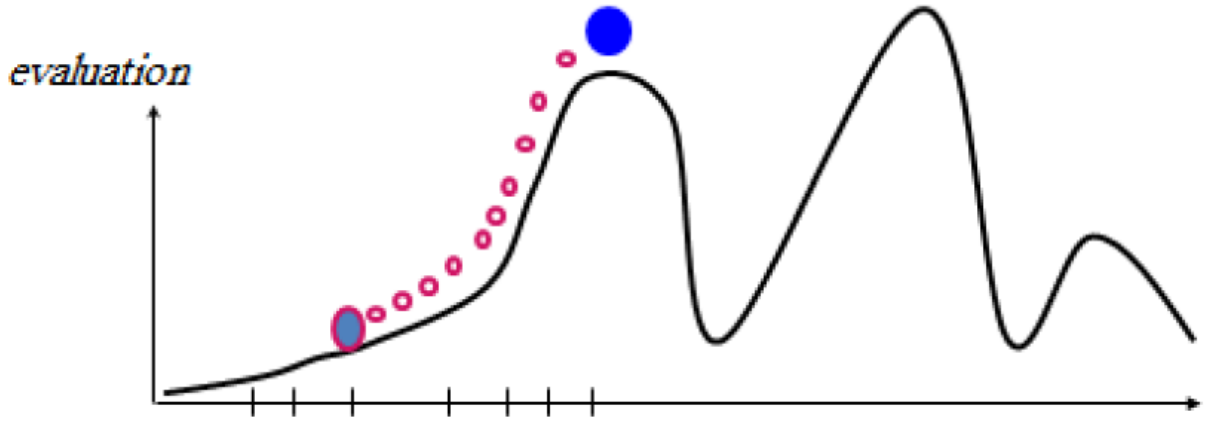
function HILL-CLIMBING(*problem*) **return** a state that is a local maximum
input: *problem*, a problem
local variables: *current*, **a node.**
neighbor, **a node.**

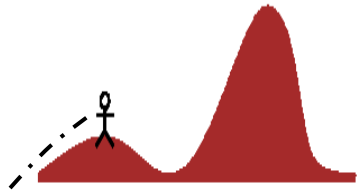
current ← MAKE-NODE(INITIAL-STATE[*problem*])
loop do
neighbor ← a highest valued successor of *current*
if VALUE [*neighbor*] ≤ VALUE[*current*] **then return** STATE[*current*]
current ← *neighbor*

Minimisation version will reverse inequalities and look for lowest valued successor

Hill-climbing

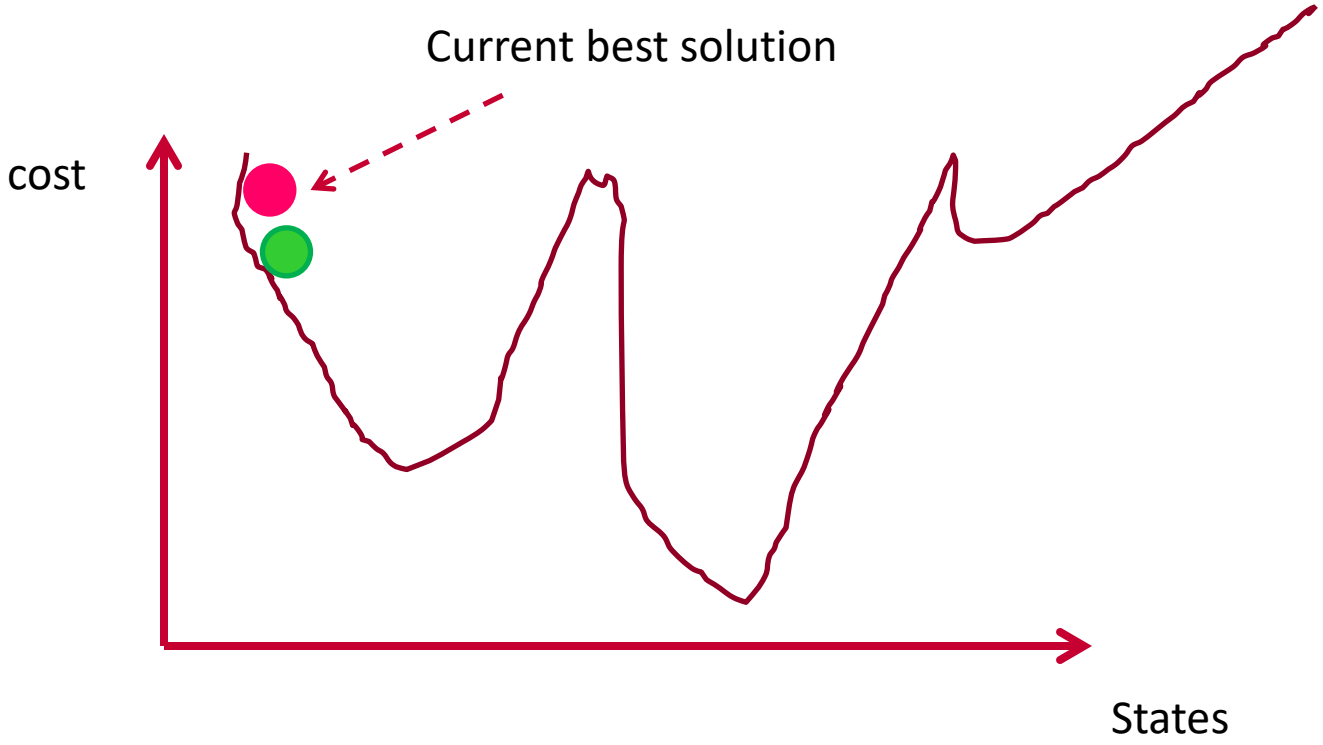
- **Main Idea**: Keep a single current node and move to a neighboring state to improve it.
- Uses a loop that continuously moves in the direction of increasing value (**uphill**):
- Choose the best successor, choose randomly if there is more than one.
- Terminate when a peak reached where no neighbor has a higher value.
- It also called **greedy local search**, steepest ascent/descent.



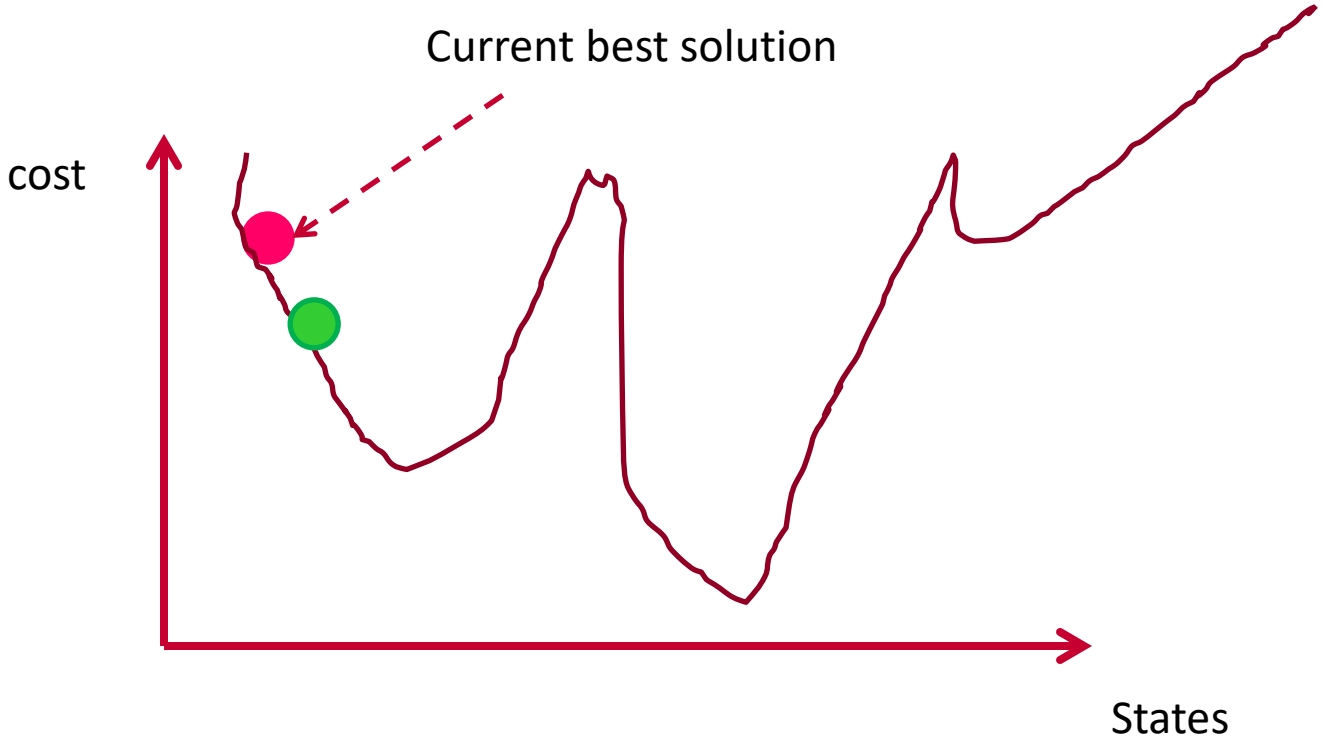


Hill-Climbing in Action …

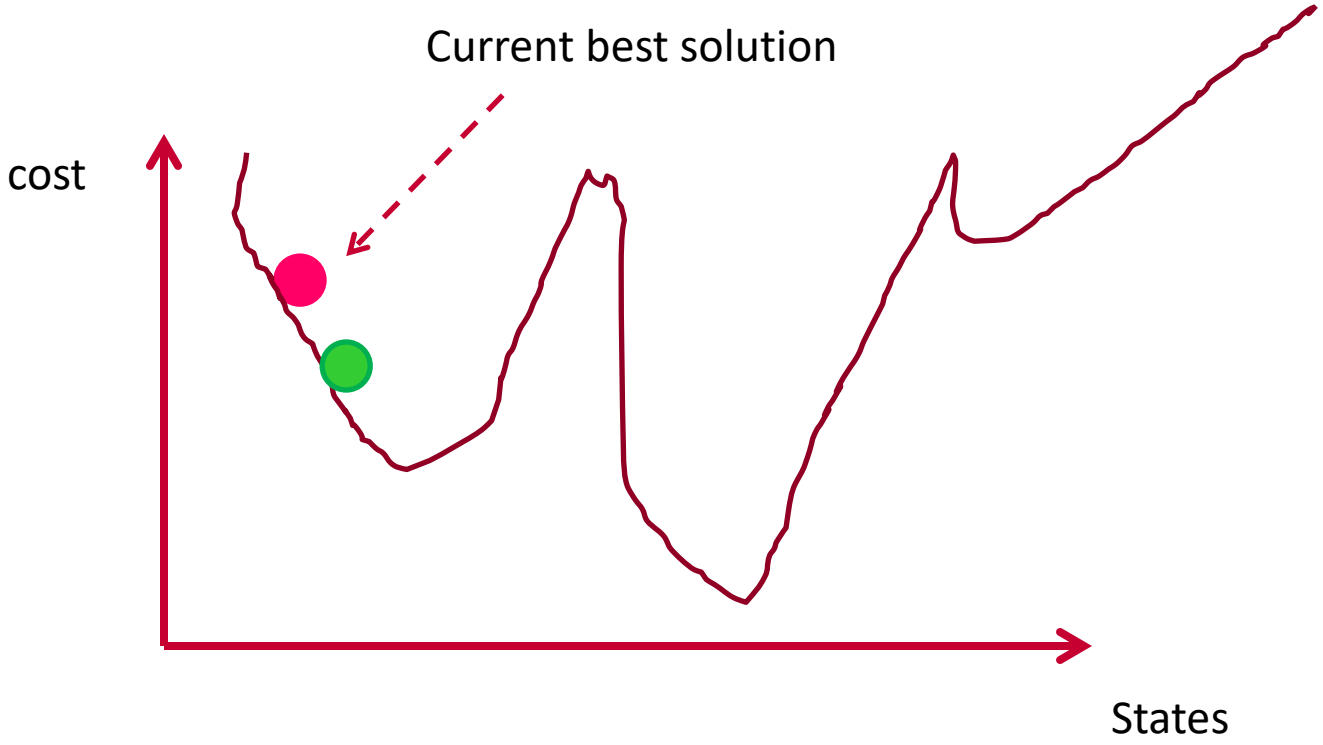
Minimisation Problem!



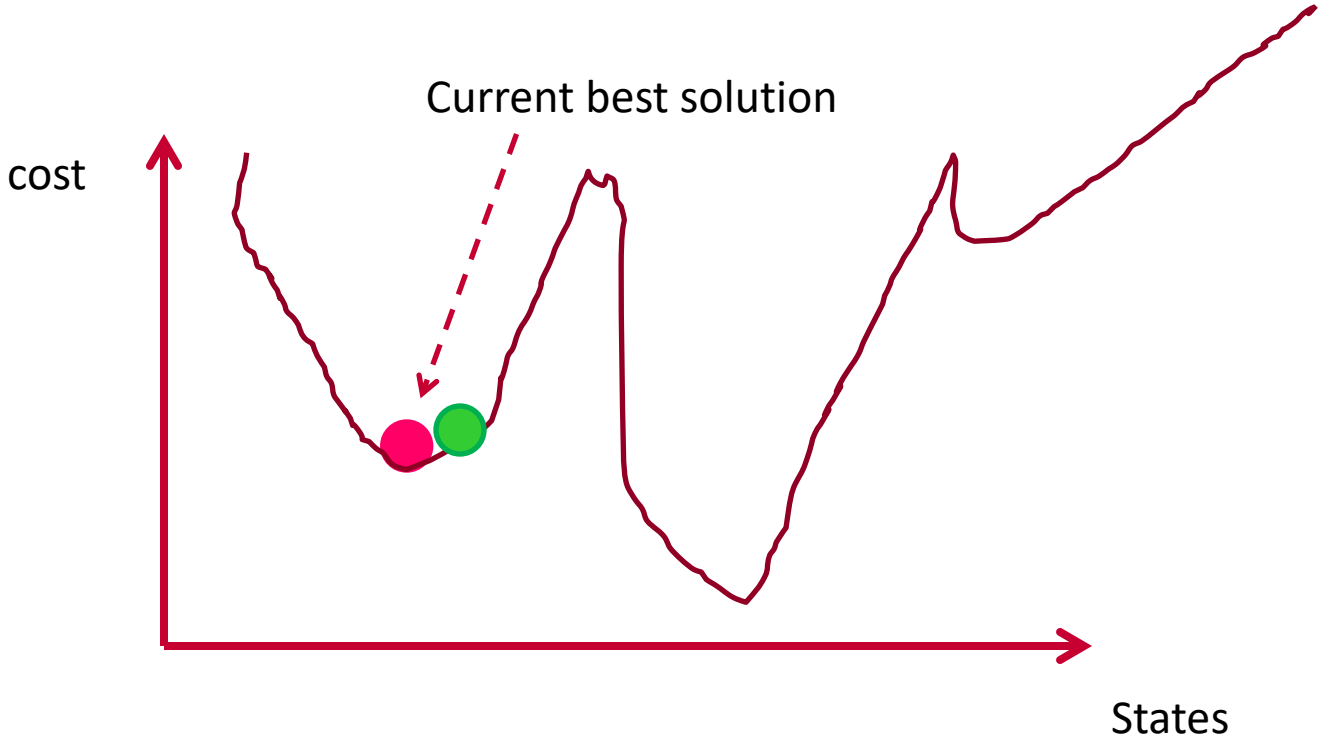
Hill-Climbing in Action …



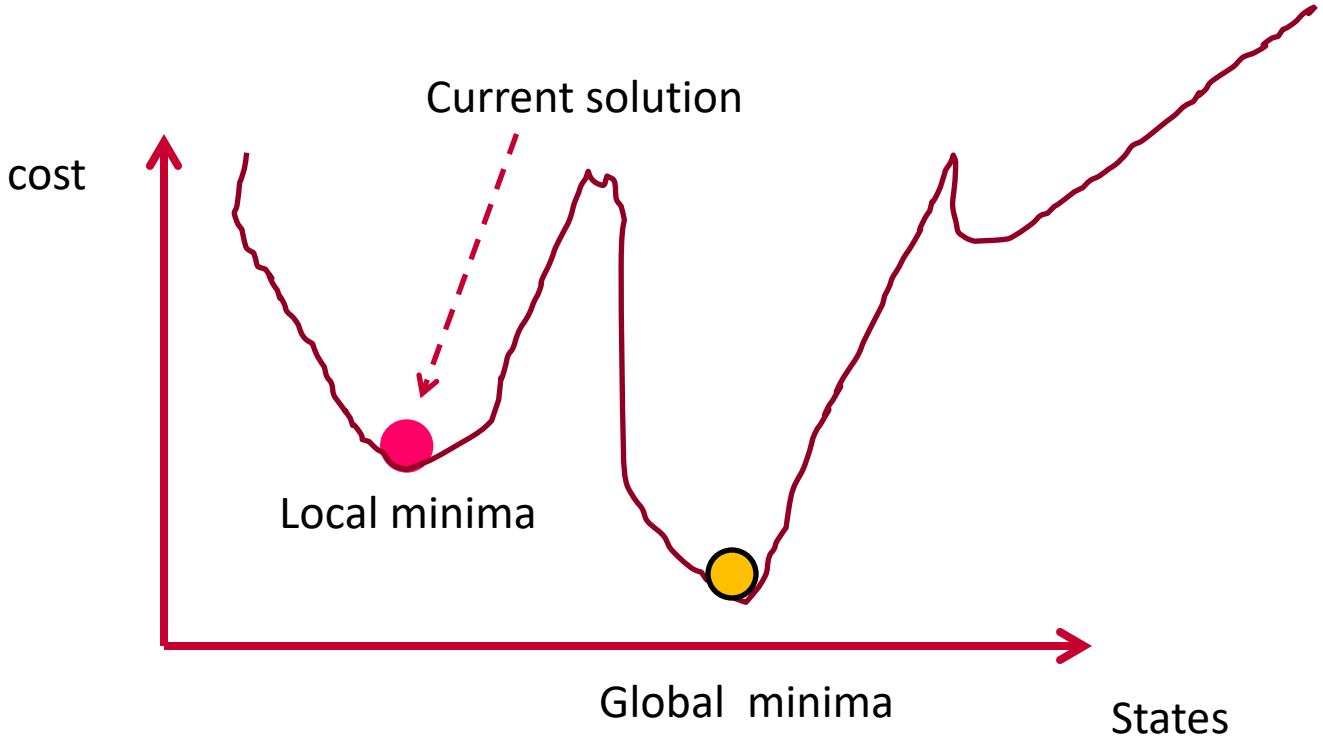
Hill-Climbing in Action …



Hill-Climbing in Action …

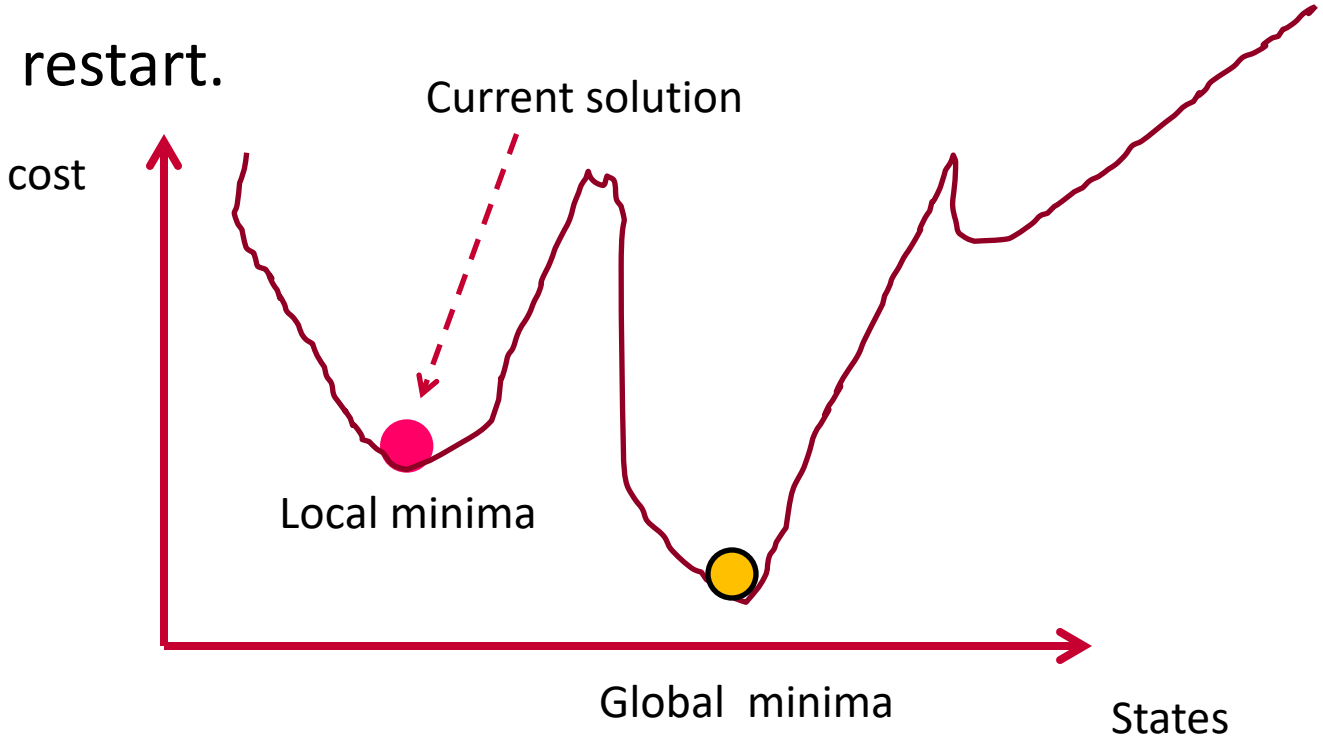


Hill-Climbing in Action …



Hill-Climbing in Action …

- Drawback: Depending on initial state, it can get stuck in **local maxima/minimum** or flat local maximum and not find the solution.
- Cure: Random restart.



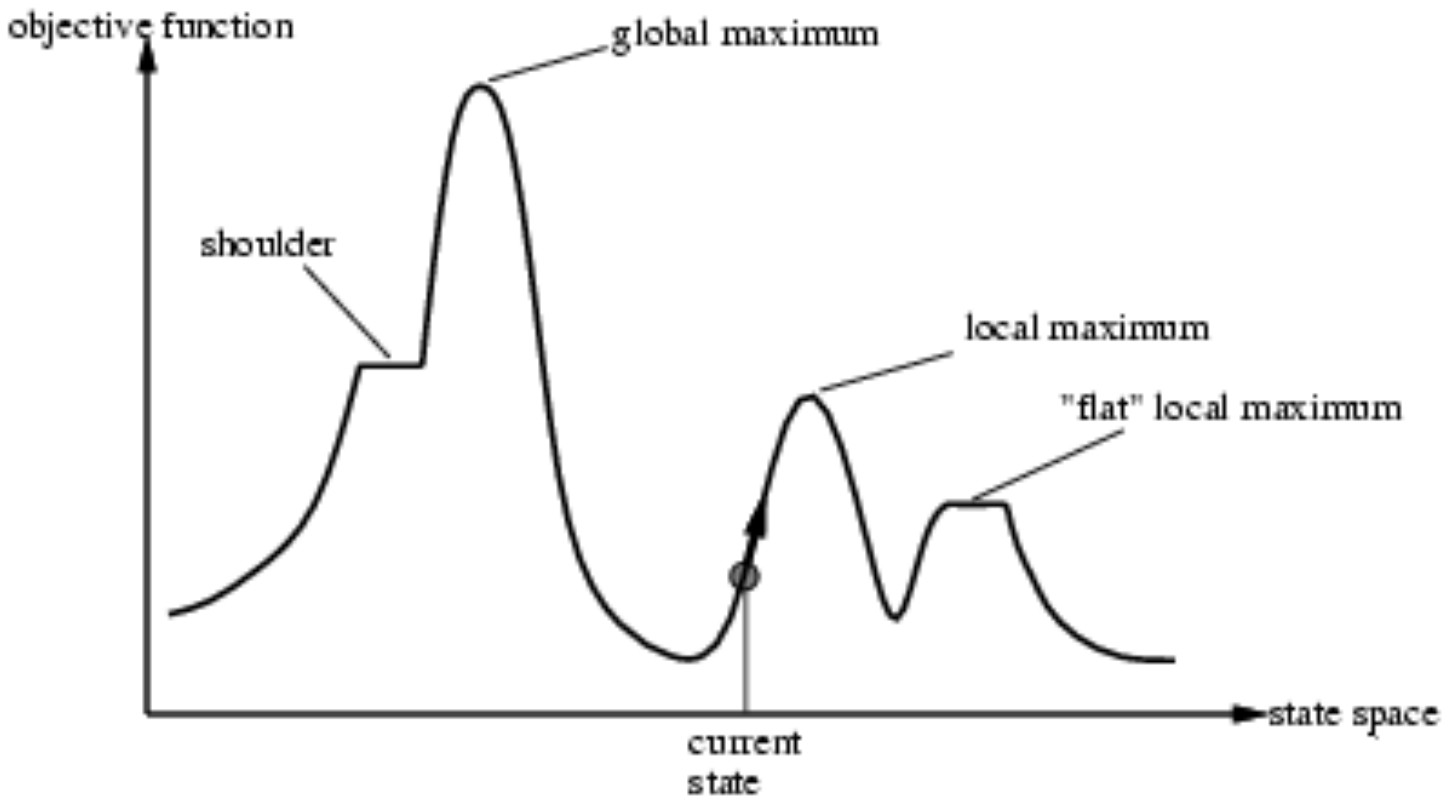
Hill-climbing search

For maximization problem (opposite direction for minimization):

- "a loop that continuously moves towards increasing value"
  - terminates when a peak is reached
  - Aka greedy local search
- Value can be either
  - Objective function value
  - Heuristic function value
- Hill climbing does not look ahead of the immediate neighbors
- Can randomly choose among the set of best successors
  - Random tie-breaking if multiple have the best value

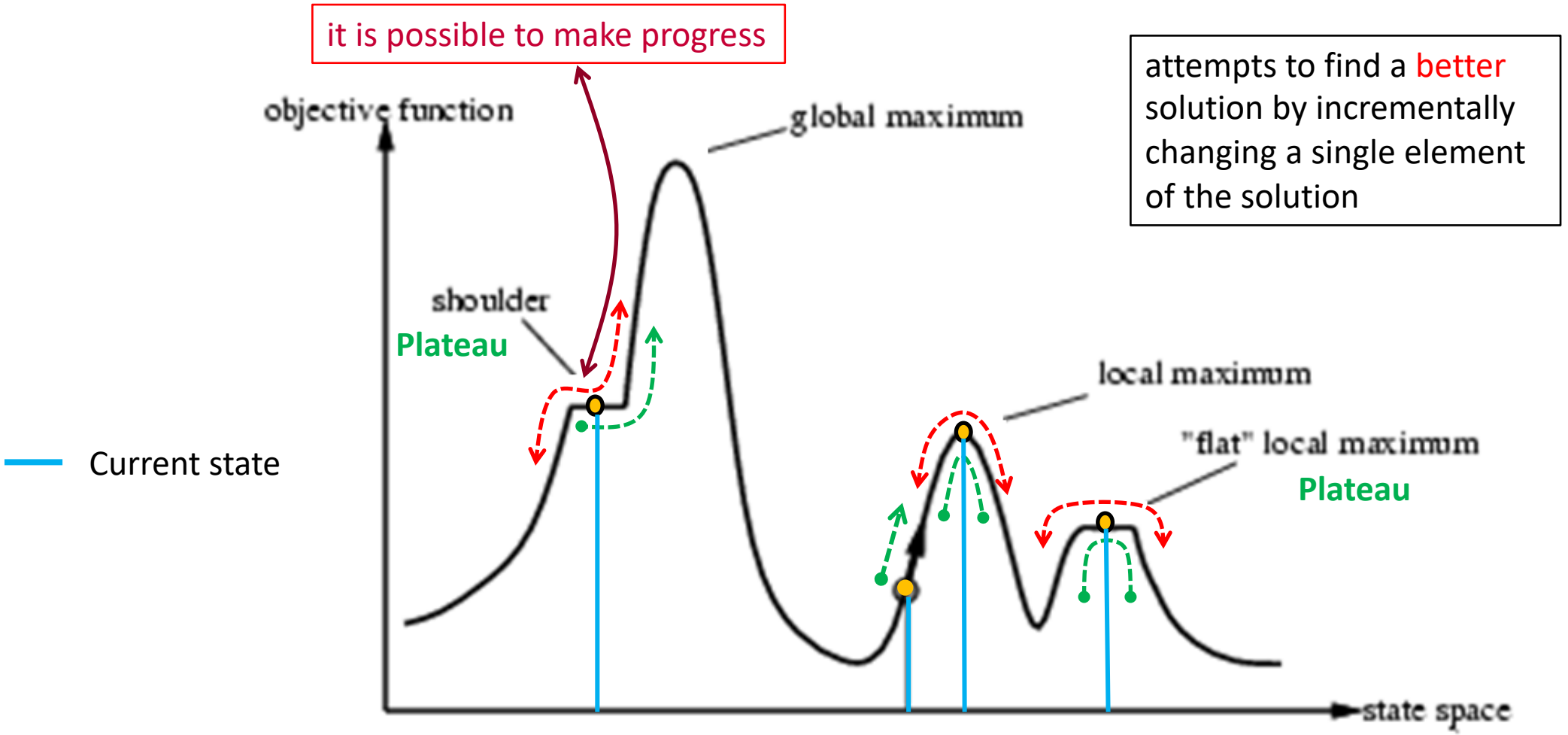
"Climbing Mount Everest in a thick fog with amnesia"

Landscape of Search



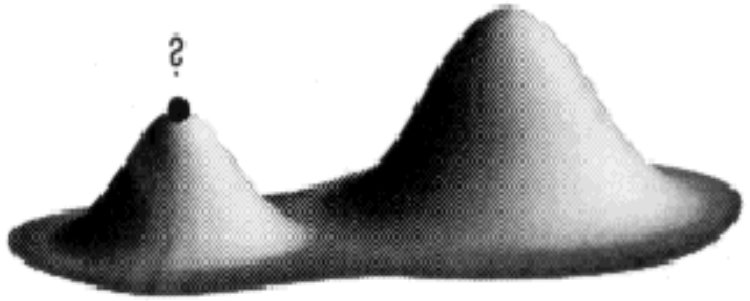
Hill Climbing gets stuck in local minima

Hill-climbing search

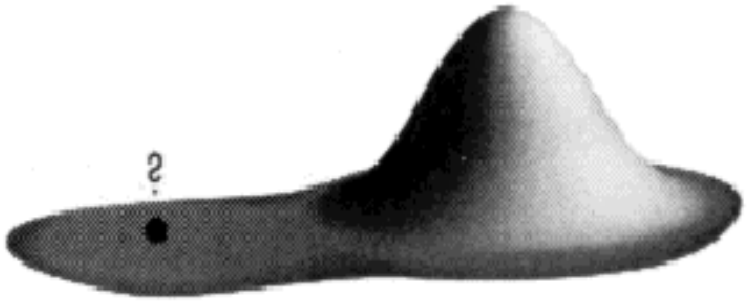


Hill-climbing search

- Local maxima: a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go.



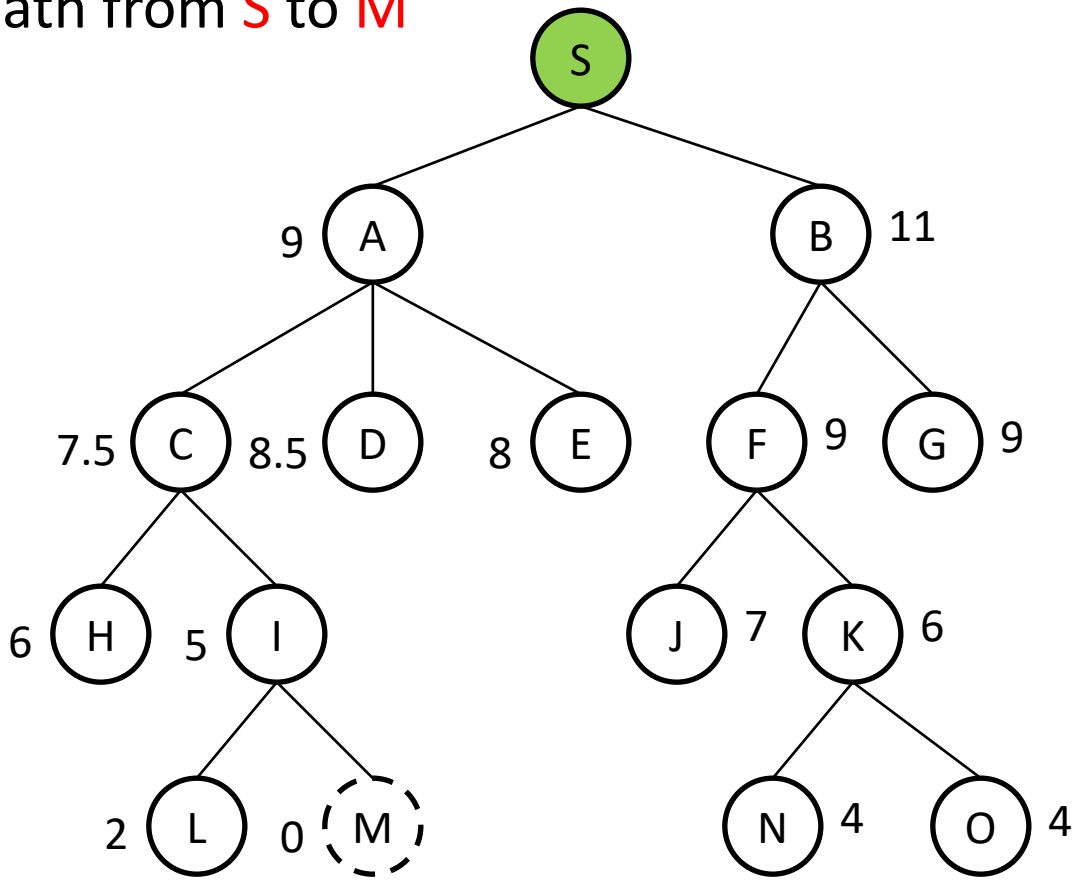
- Plateau: a plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which it is possible to make progress.



- Allow sideways moves … but
  - What would we need in this case?
  - Memory or stopping condition in non-improving moves

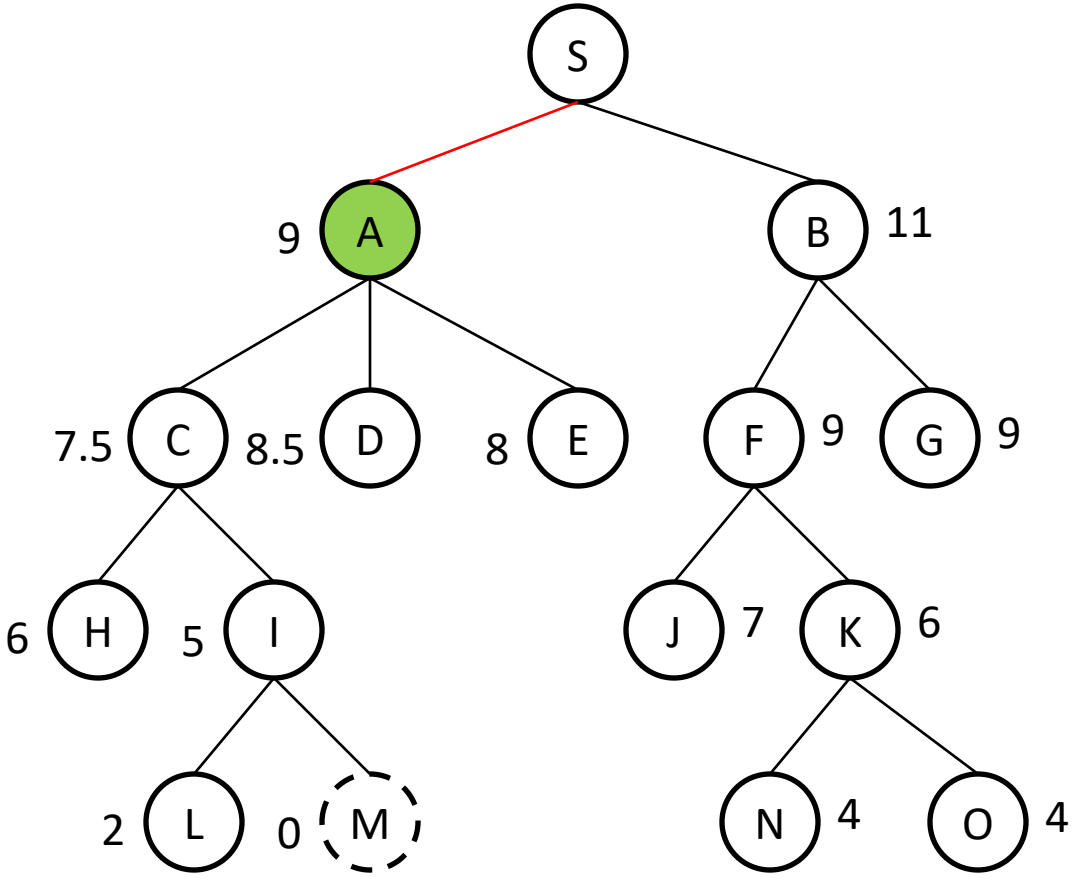
Hill-climbing search example

Our aim is to find a path from S to M

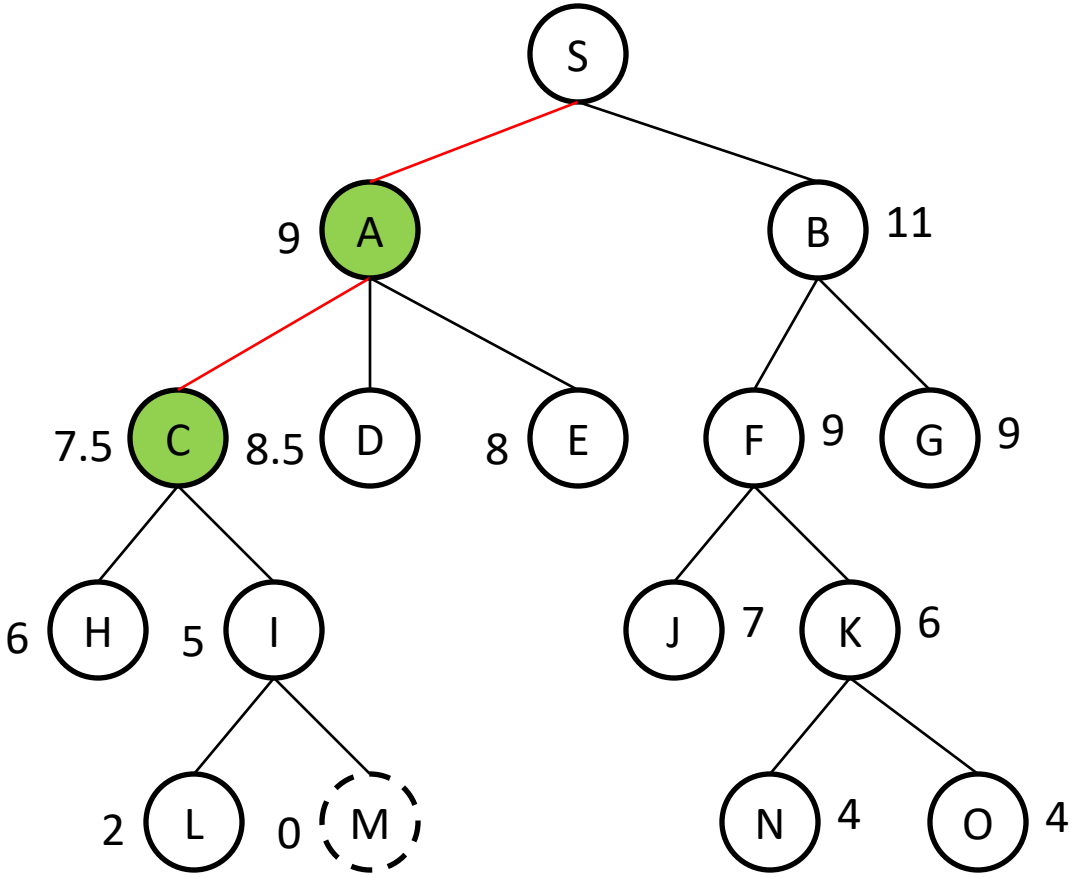


Associate heuristics with every node, For instance:
Number of unsatisfied clauses (SAT)

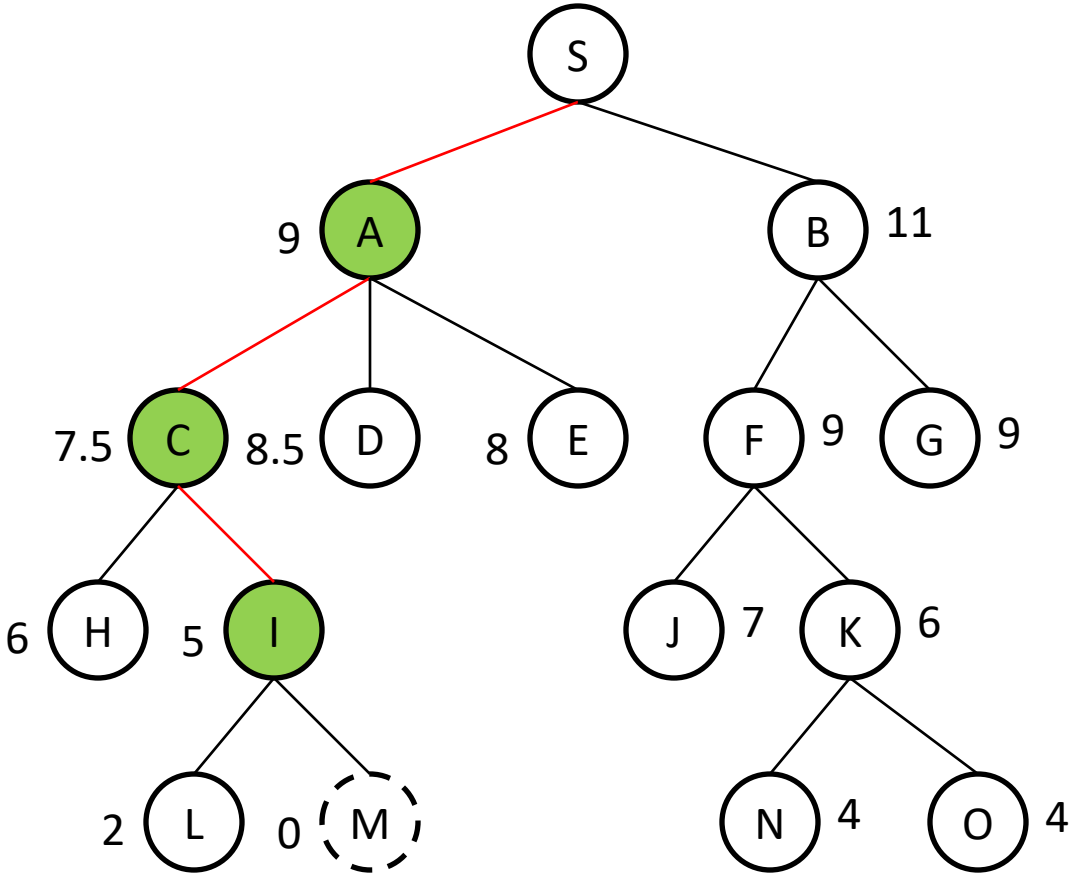
Hill-climbing search example



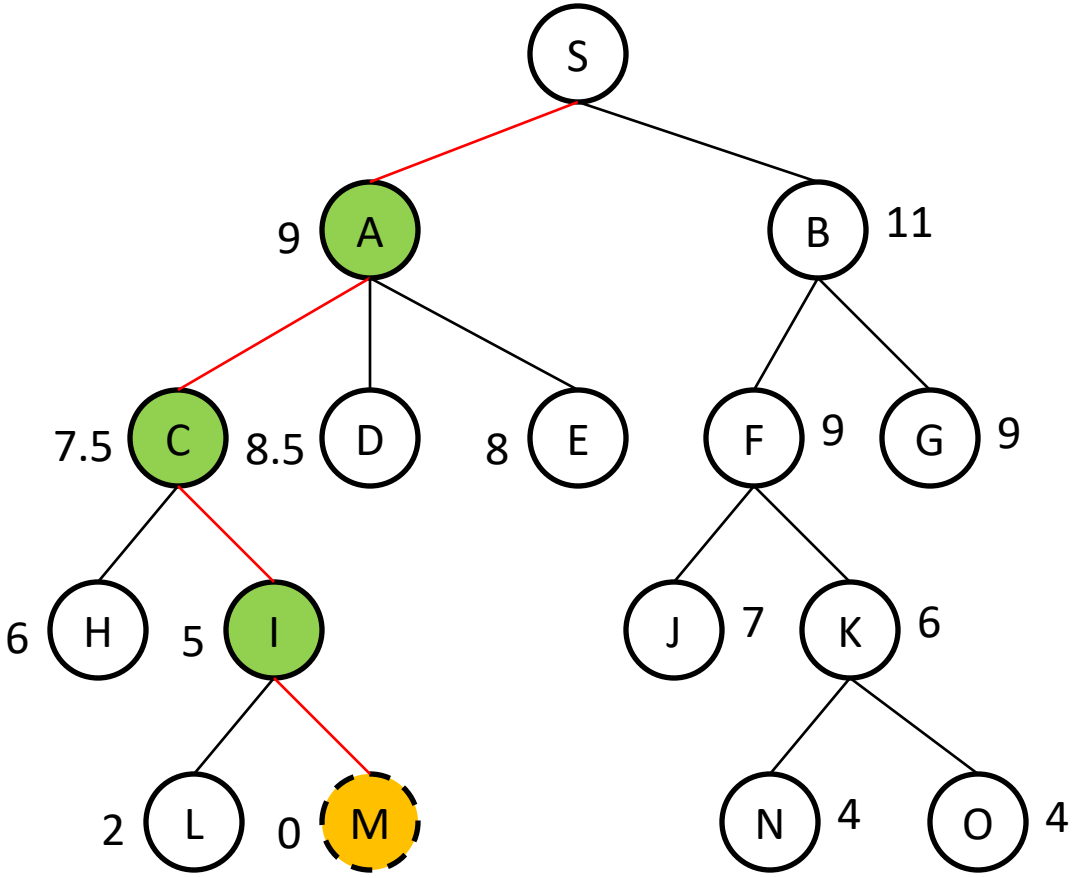
Hill-climbing search example



Hill-climbing search example

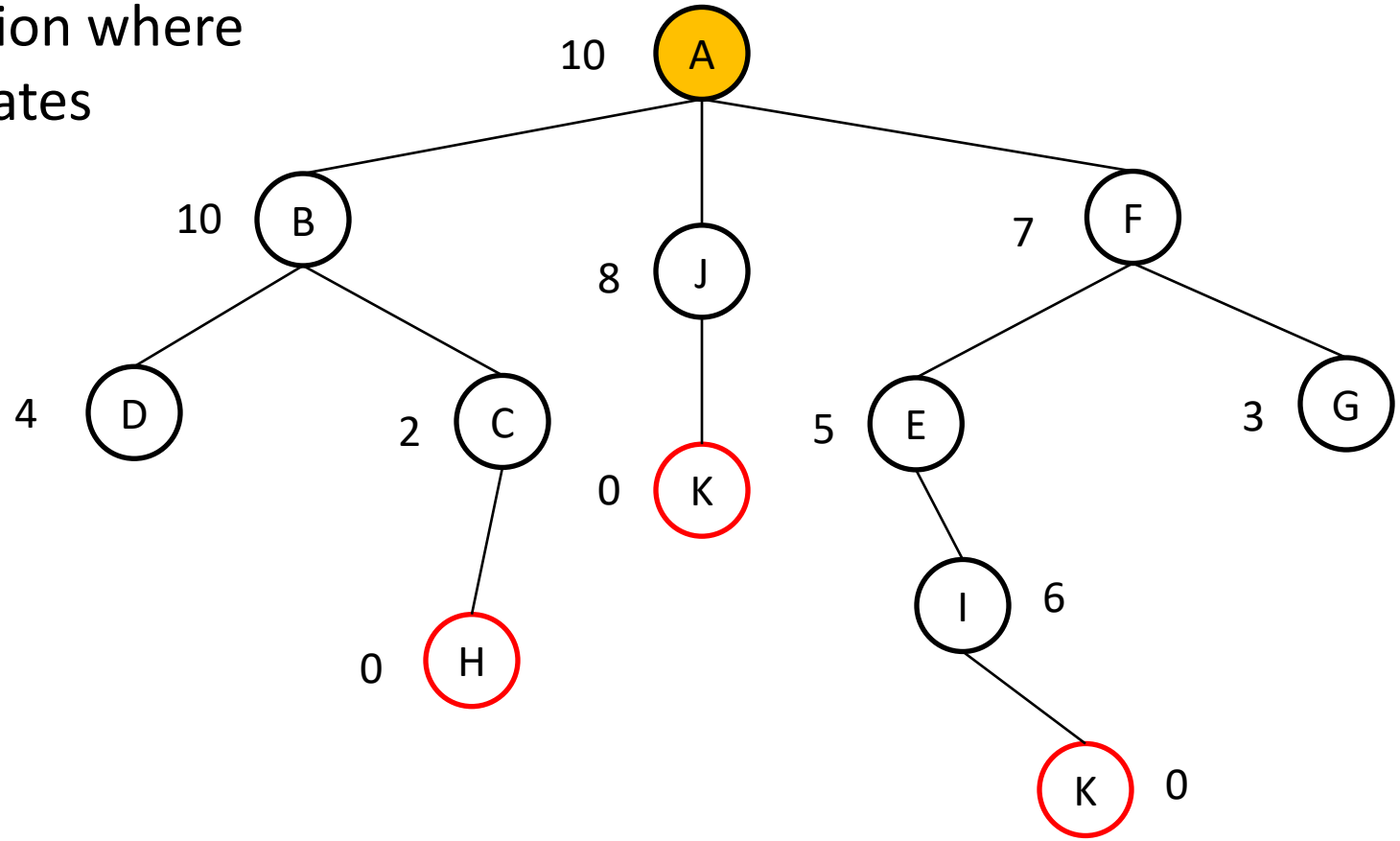


Hill-climbing search example

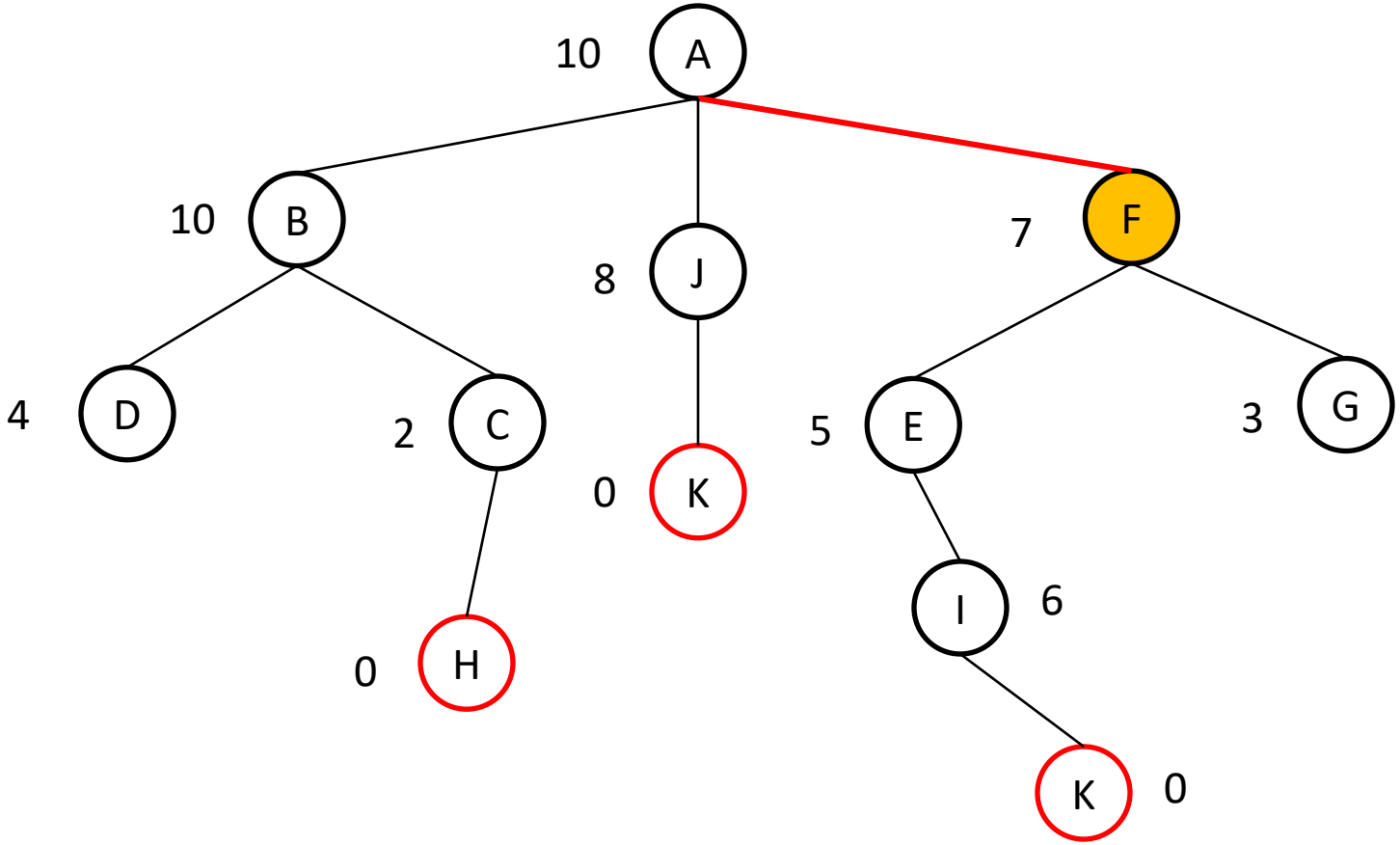


Hill-climbing search example (Local Maximum)

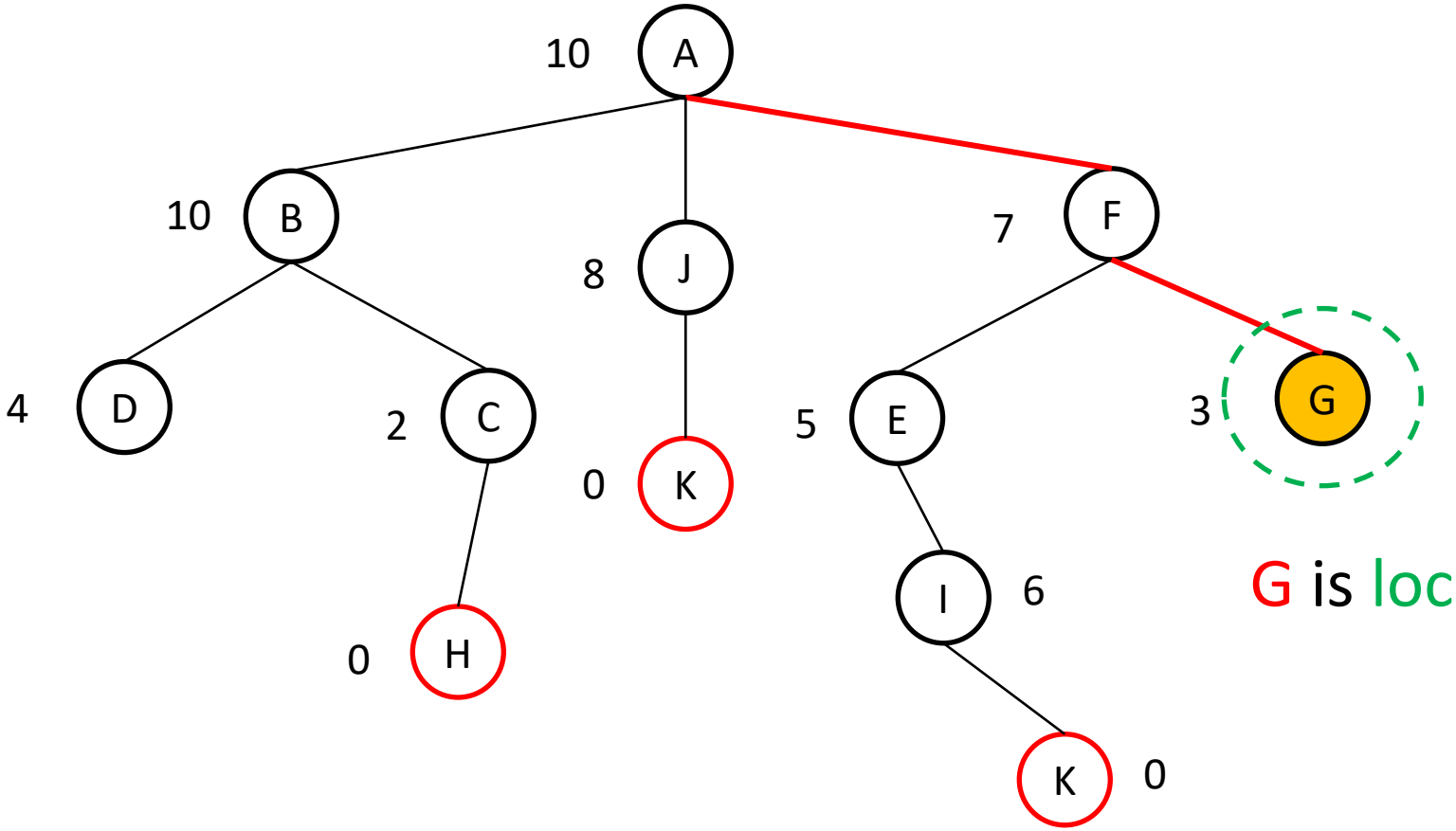
From A find a solution where
H and K are final states



Hill-climbing search example (Local Maximum)



Hill-climbing search example (Local Minimum)



Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next.

Hill-climbing search example
Local Maximum -- Local Minimum