Local Search and Optimization

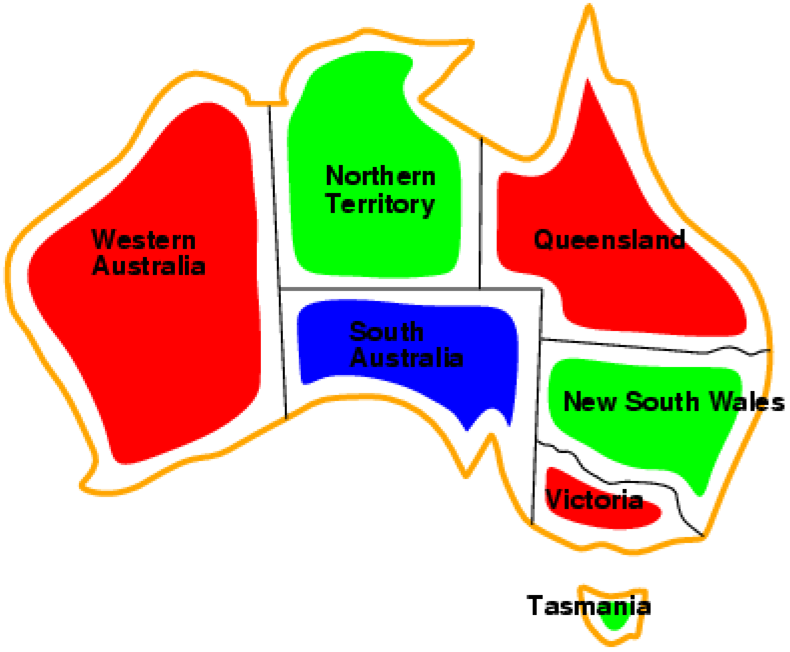
Local Search Algorithms

- The search algorithms we have seen so far keep track of the current state, the "fringe" of the search space, and the *path* to the final state.

- In some problems, one doesn't care about a solution path but only the orientation of the final goal state
  - Example: 8-queen problem

- Local search algorithms operate on a *single* state – current state – and move to one of its neighboring states
  - Solution path needs not be maintained
  - Hence, the search is "local"

Example: Map-Coloring



- Solutions are complete and consistent assignments, e.g., WA = red, NT = green,Q = red,NSW = green,V = red,SA = blue,T = green

Constraint graph

- It can be helpful to visualize a CSP as a **constraint graph**

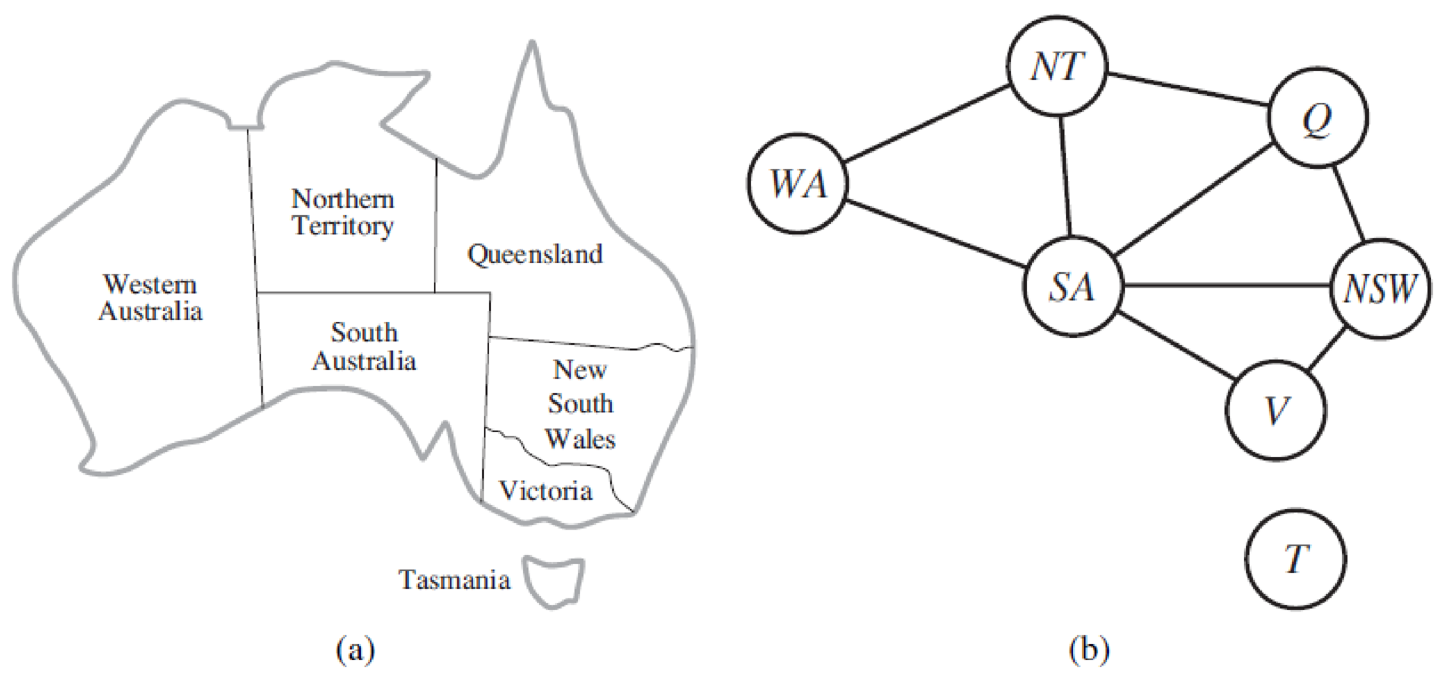
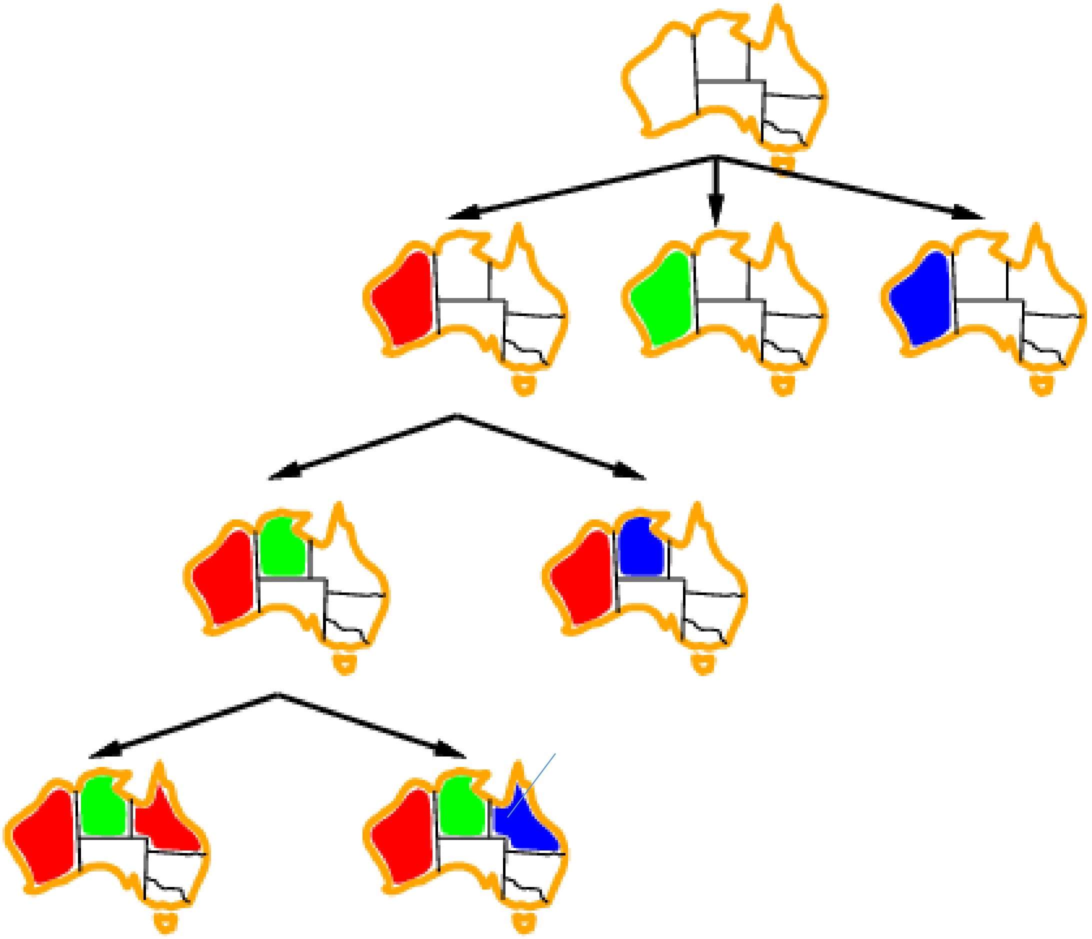


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Example



Local Search Algorithms

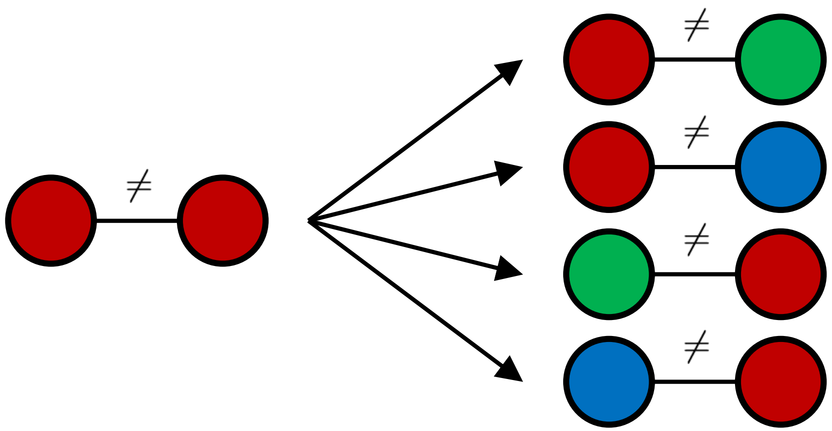
- When there is no need to memorize the path to the goal, just to find the goal.
- Cost of the path and finding optimal path to the goal is not required.
- Start from a single current state and check the neighboring states
- Consumes little memory
- Useful with large state space.

Local Search

- Tree search keeps unexplored alternatives on the fringe (ensures completeness)

- Local search: improve a single option until you can't make it better (no fringe!)

- New successor function: local changes



- Generally much faster and more memory efficient (but incomplete and suboptimal)

Local Search Methods – Applications

- Applicable when seeking Goal State & don't care how to get there. E.g.,
  - N-queens,
  - finding shortest/cheapest round trips
    - **(Travel Salesman Problem, Vehicle Routing Problem)**
  - finding models of propositional formulae (SAT solvers)
  - VLSI layout, planning, scheduling, time-tabling, . . .
  - map coloring,
  - resource allocation
  - protein structure prediction
  - genome sequence assembly

Local Search Algorithms

- In many optimization problems, the **path** to the goal is irrelevant; the goal state itself is the solution
- State space = set of configurations
  - Find a configuration satisfying your constraints, e.g., n-queens
  - Find the best possible state according to a given objective function

- In such cases, we can **use local search algorithms**
  - Keeps a single "current" state, and then shift states, but don't keep track of paths.
  - Use very limited memory
  - Find reasonable solutions in large state spaces.

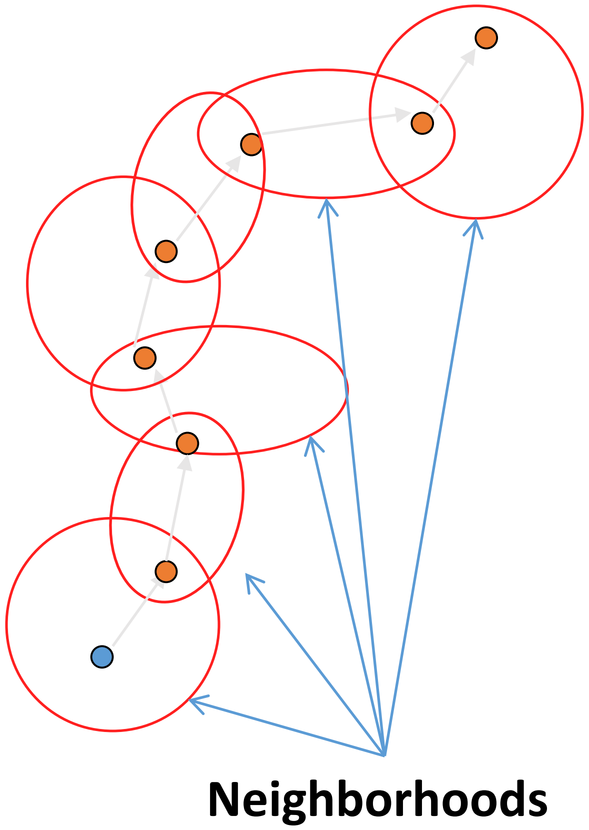
Local Search Algorithm

- Local search algorithms work as follows:
  - Pick a “solution” from the search space and evaluate it. Define this as the current solution.
  - Apply a transformation to the current solution to generate and evaluate a new solution.
  - If the new solution is better than the current solution then exchange it with the current solution; otherwise discard the new solution.
  - Repeat steps 2 and 3 until no transformation in the given set improves the current solution

Local search

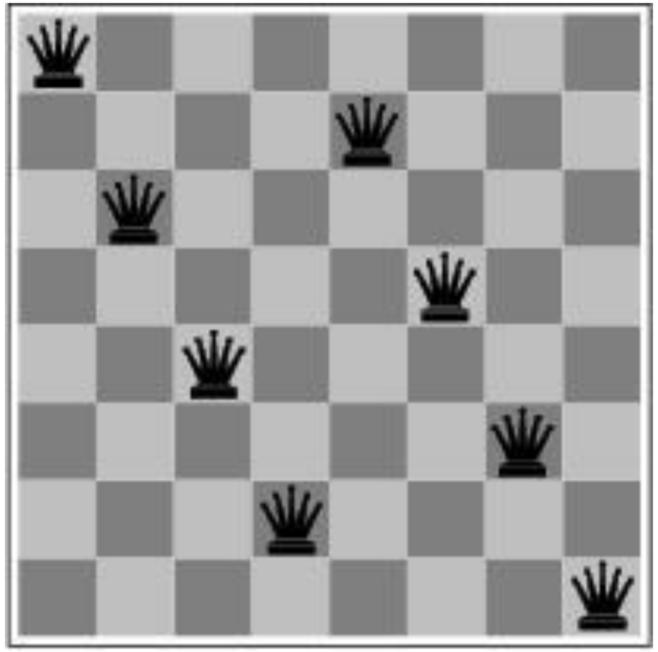
- **Key idea (surprisingly simple):**

1. Select (random) initial state (generate an initial guess)
2. Make **local modification** to **improve** current state
   - Evaluate current state and move to other states
3. Repeat Step 2 until goal state found (or out of time)



Local Search Algorithms for optimization Problems

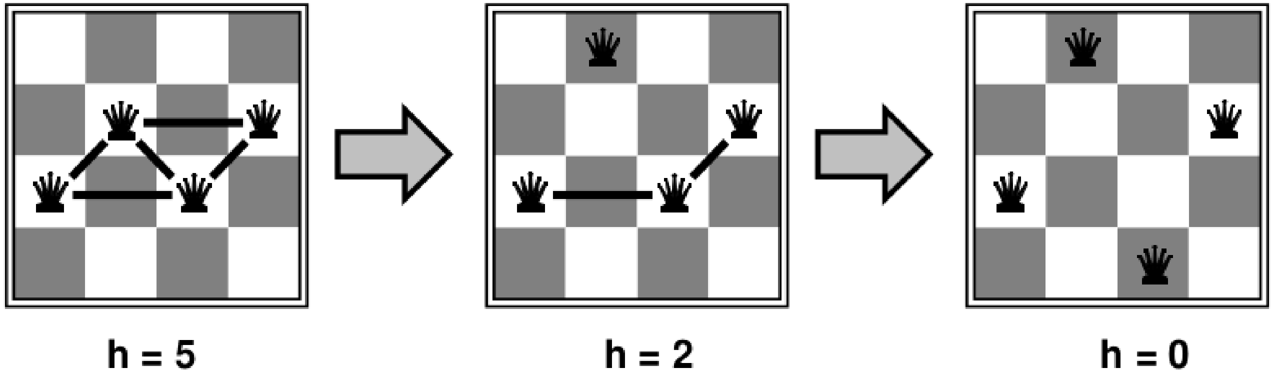
- Local search algorithms are very useful for optimization problems
- systematic search doesn't work
- however, can start with a suboptimal solution and improve it
- Goal: find a state such that the objective function is optimized



Minimize the number of attacks

Example: n-Queen

- Put N Queens on an $n \times n$ board with no two queens on the same row, column, or diagonal
- Move a queen to reduce number of conflicts

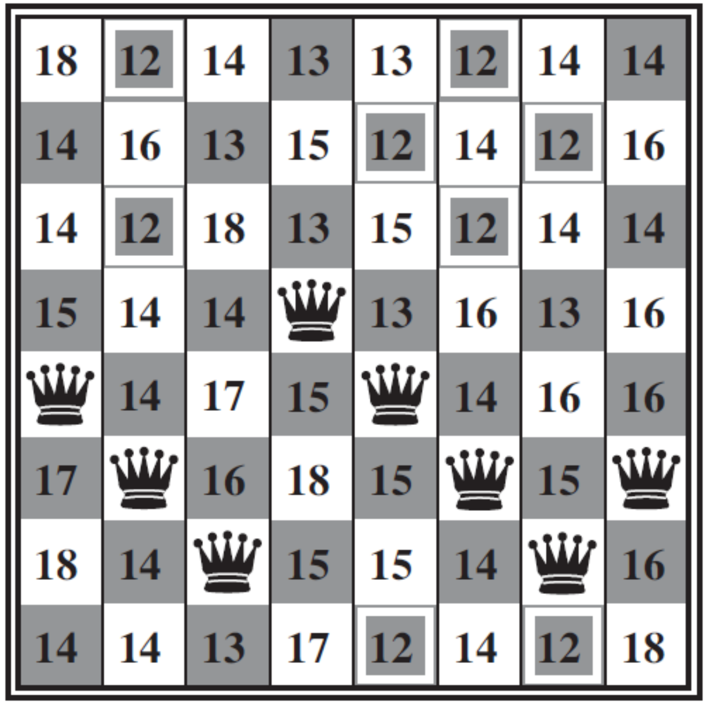


Initial state … Improve it … using local transformations

Almost always solves n-queens problems instantaneously for very large n, e.g., n = 1 million

Example: n-Queen

- Put N Queens on an $n \times n$ board with no two queens on the same row, column, or diagonal



Example: n-Queen

- Put N Queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

<u>HEURISTIC</u>:

h = # of pairs of queens that are attacking each other, either **directly or indirectly**

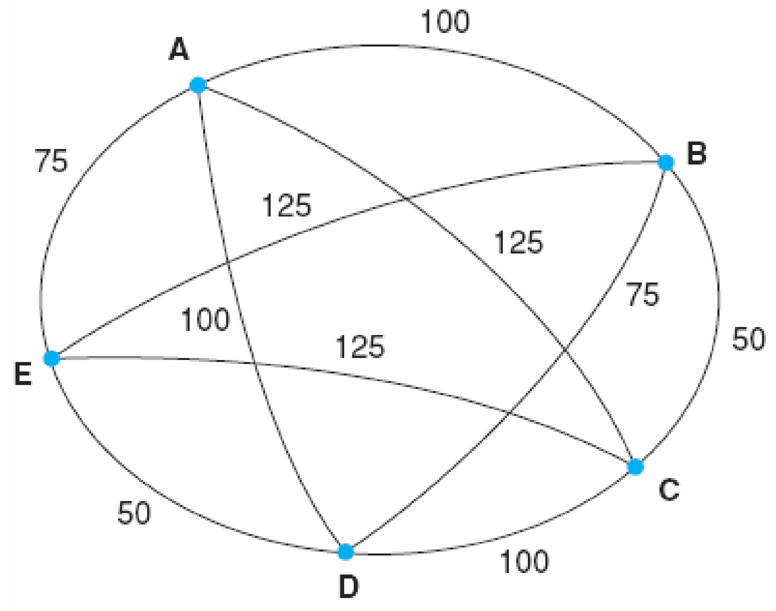
18	$\boxed{12}$	14	13	13	$\boxed{12}$	14	14
14	16	13	15	$\boxed{12}$	14	$\boxed{12}$	16
14	$\boxed{12}$	18	13	15	$\boxed{12}$	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	$\boxed{12}$	14	$\boxed{12}$	18

Each number indicates *h* if we move a queen in its column to that square

12 (boxed) = best *h* among all neighors; <u>select one randomly</u>

Example: Travelling Salesman Problem

- **Travelling Salesperson Problem (TSP)**
  - Suppose a salesman has five cities to visit and ten must return home.
  - The goal of the problem is to find the shortest path for salesman to travel, visiting each city, and then returning to the starting city.



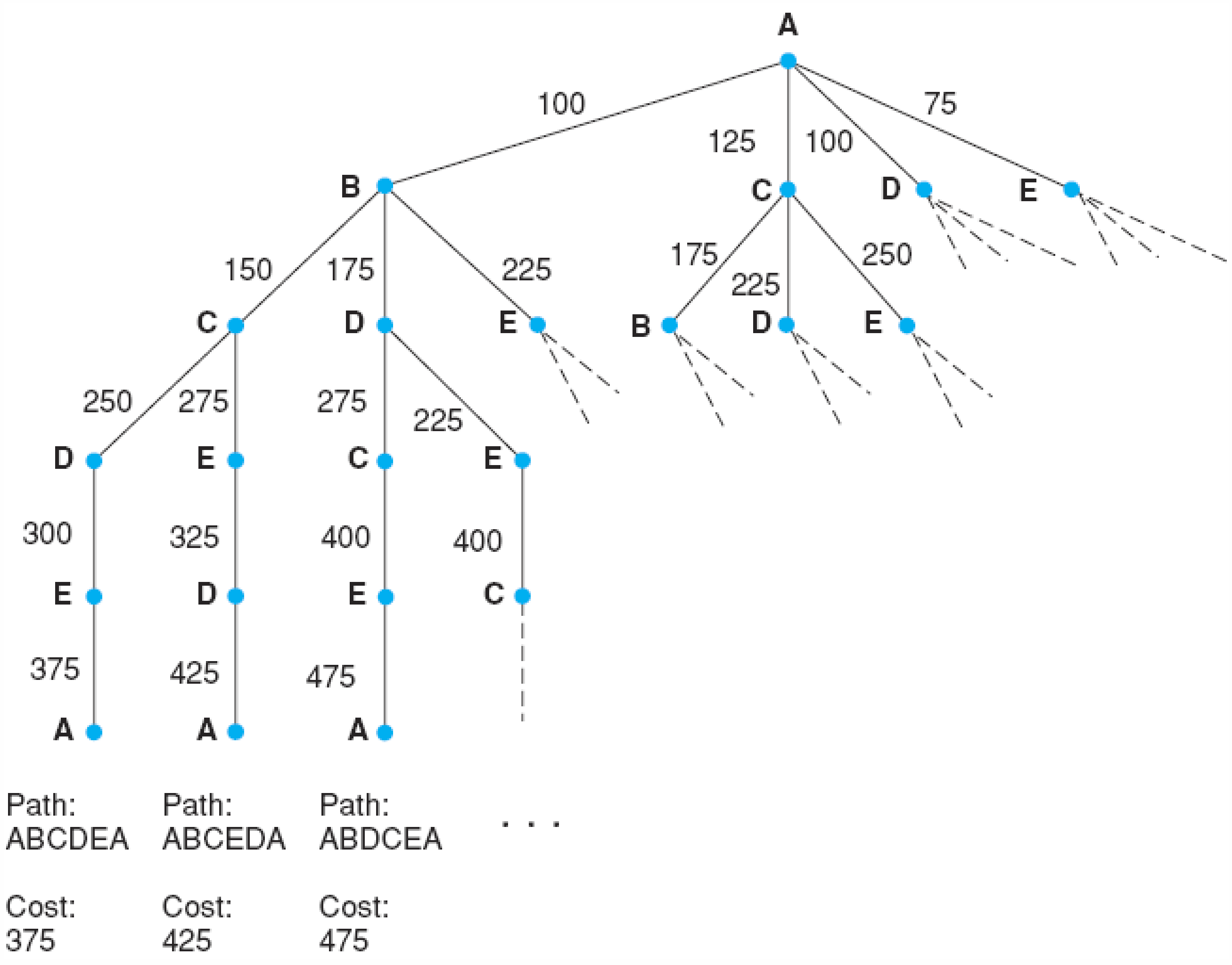
Example: Travelling Salesman Problem

- A Solution: Exhaustive Search
  - (Generate and Test) !!
- The number of all tours is about (n-1)!/2

- If n = 36 the number is about:
- 566573983193072464833325668761600000000

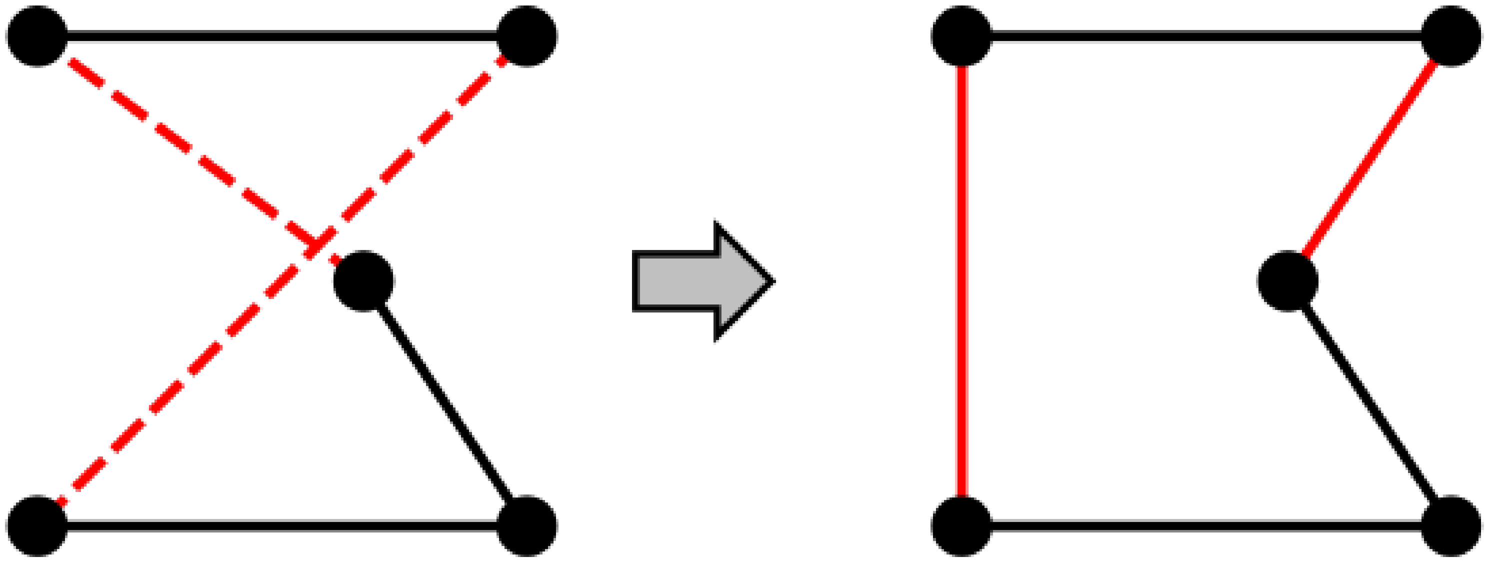
- Not Viable Approach !!

Search for the travelling salesperson problem. Each arc is marked with the total weight of all paths from the start node (A) to its endpoint.

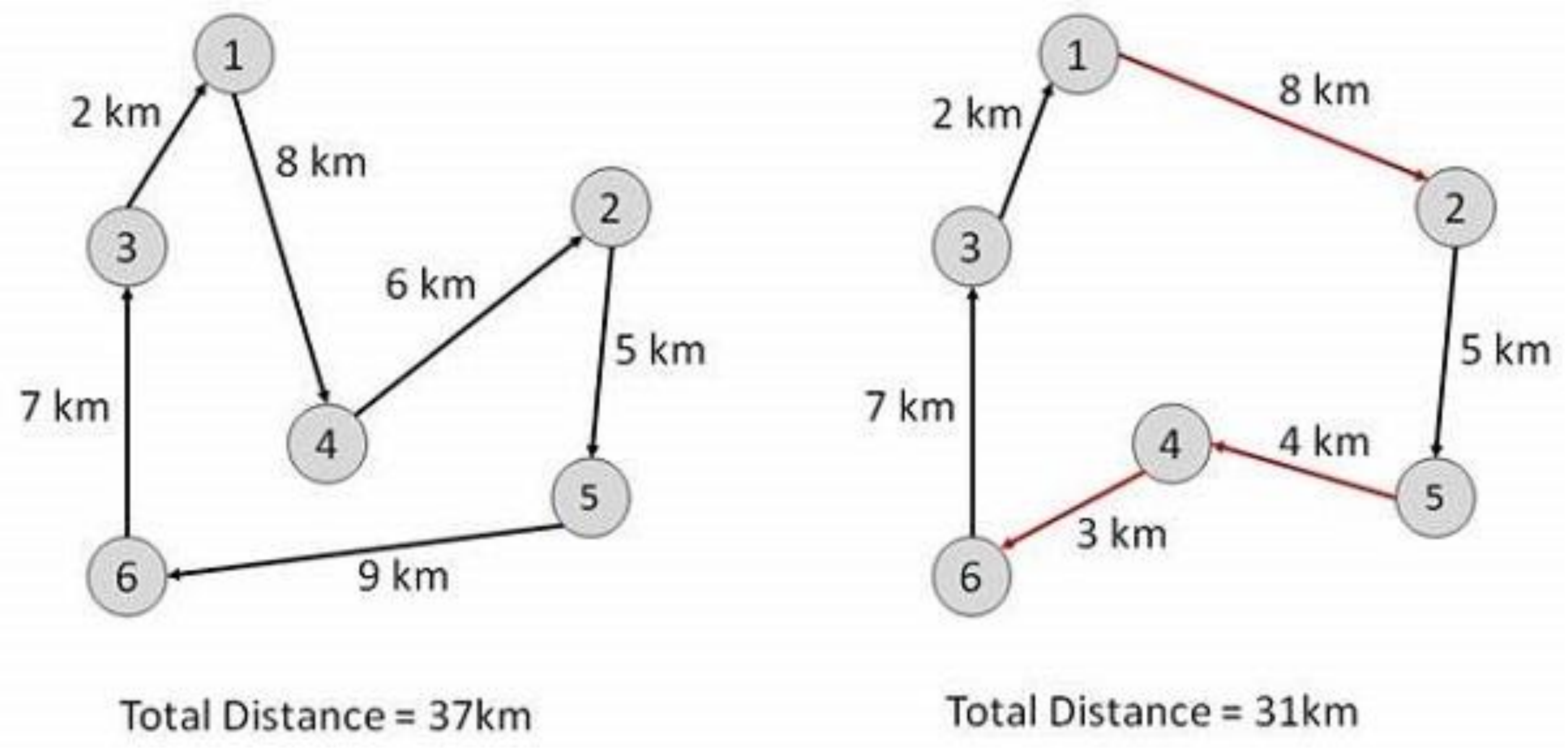


Example: Travelling Salesman Problem

Start with any complete tour, perform pairwise exchanges



Variants of this approach get within 1% of optimal very quickly with thousands of cities


1
2 km
8 km
3
2
6 km
5 km
7 km
4
5
6
9 km
Total Distance = 37km
1
8 km
2 km
2
3
5 km
7 km
4
4 km
5
3 km
6
Total Distance = 31km

Hill Climbing Algorithm

Gradient Ascent/Descent Algorithm

Hill Climbing

- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit
- What's bad about this approach?
  - Complete?
  - Optimal?

"Like climbing Everest in thick fog with amnesia"

- Hill climbing algorithm (a.k.a. greedy local search) uses a loop that continually moves in the direction of increasing values (that is uphill).
- It terminates when it reaches a peak where no neighbor has a higher value.

Hill-Climbing Algorithm

1. Pick a random point in the search space
2. Consider all the neighbors of the current state
3. Choose the neighbor with the best quality and move to that state
4. Repeat 2 to 4 until all the neighboring states are of lower quality
5. Return the current state as the solution state.

Hill Climbing Search

- Steepest Ascent Search, Greedy Local Search
- Iteratively maximize value of current state by replacing its successors state that has highest value , as long as possible.

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

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

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

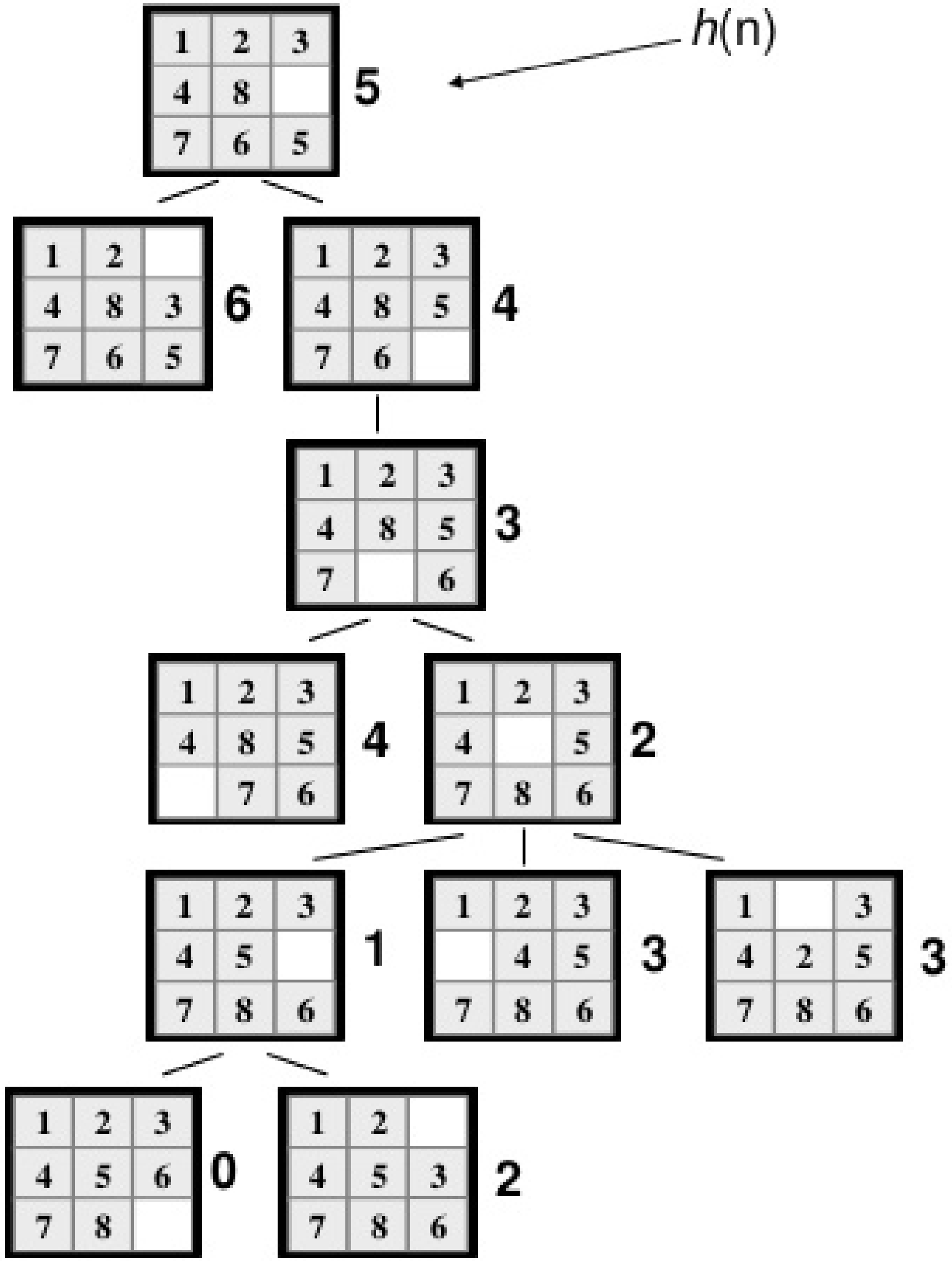
Hill Climbing Search

- Hill-climbing strategies expand the current state of the search and evaluate its children.
- The “best” child is selected for further expansion; neither its siblings nor its parent are retained.
- Because it keeps no history, the algorithm cannot recover from failures of its strategy.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.

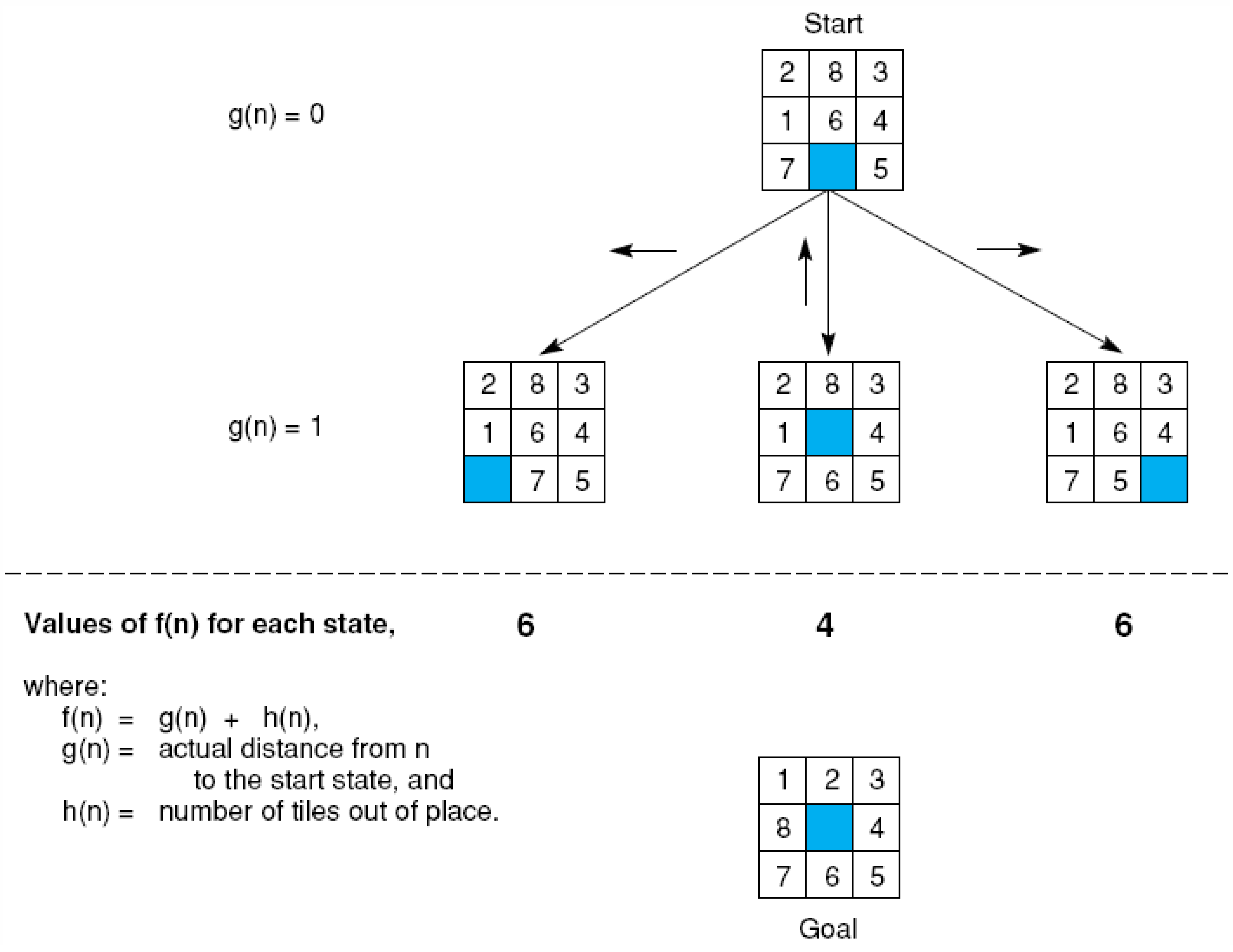
- This is "hill climbing"
- We can use heuristics to guide "hill climbing" search.

- In this example, the Manhattan Distance heuristic helps us quickly find a solution to the 8-puzzle.

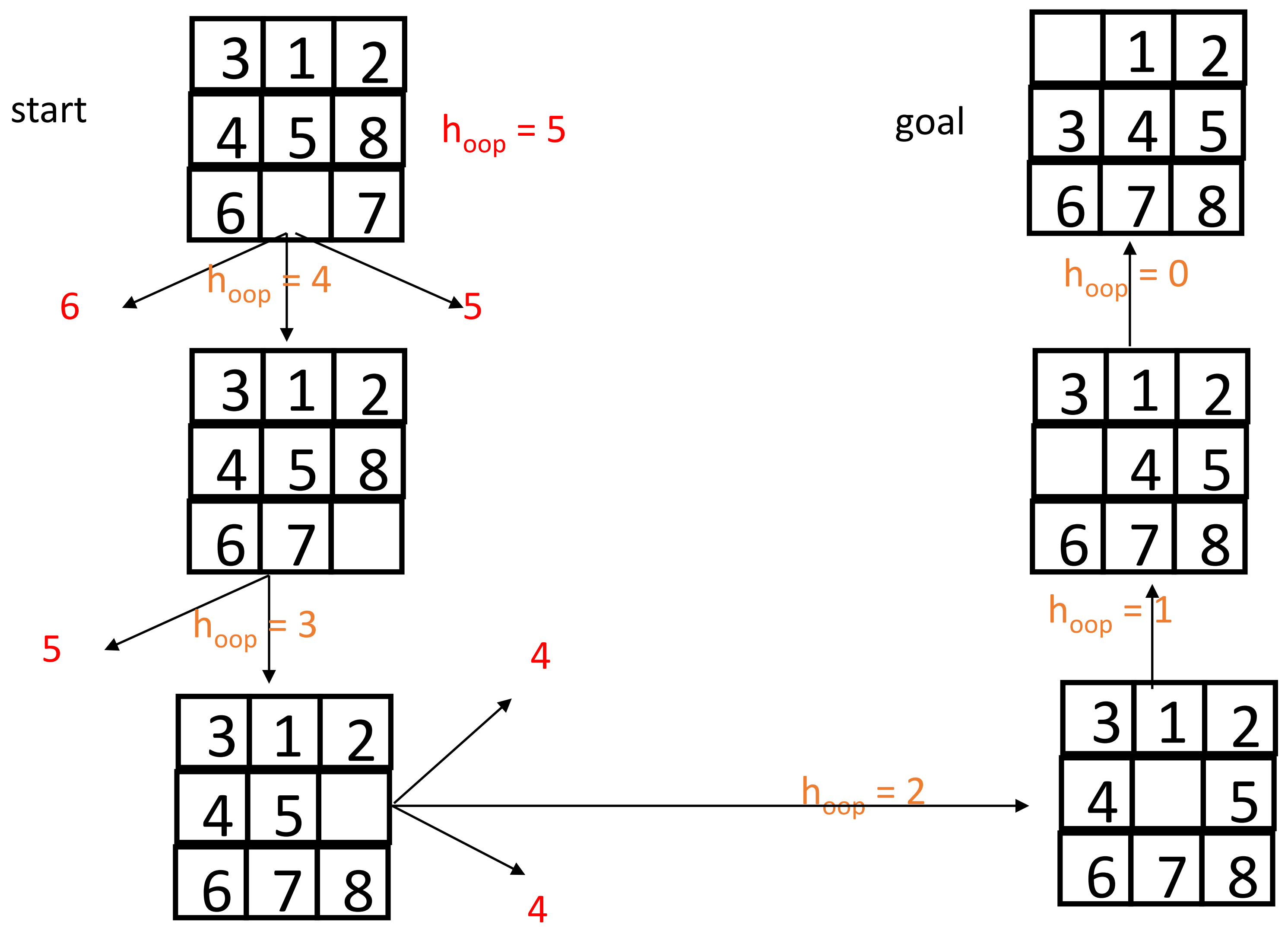
But "hill climbing has a problem..."

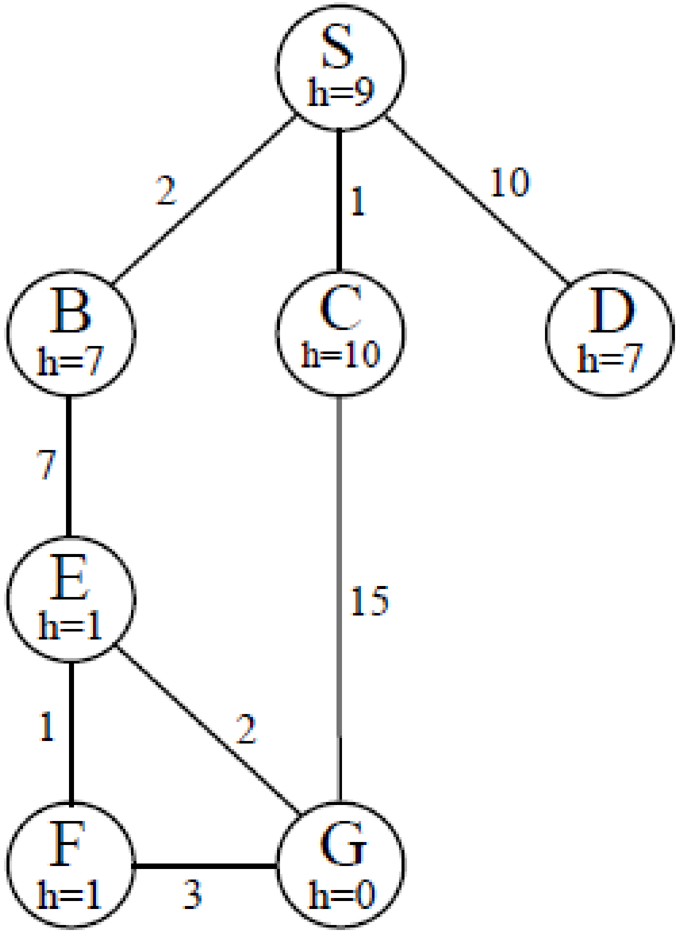


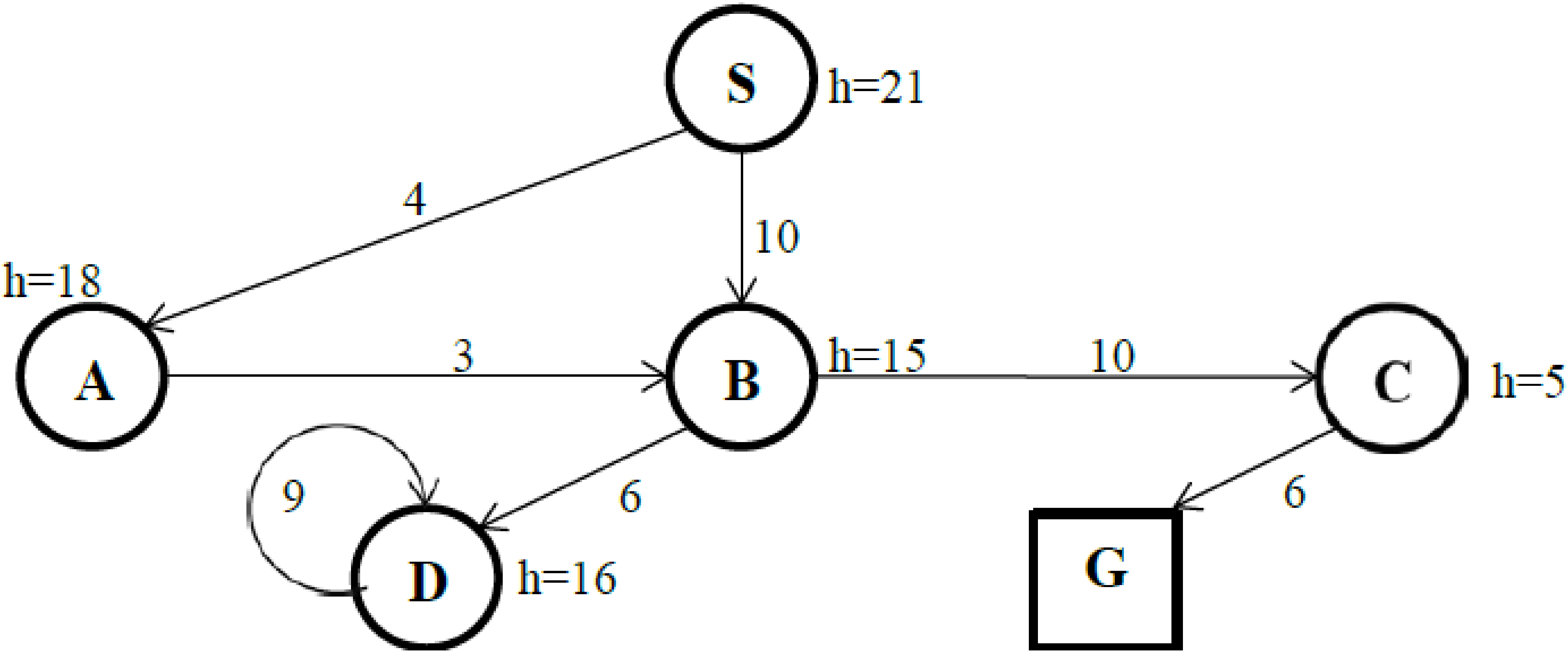
The heuristic f applied to states in the 8-puzzle.

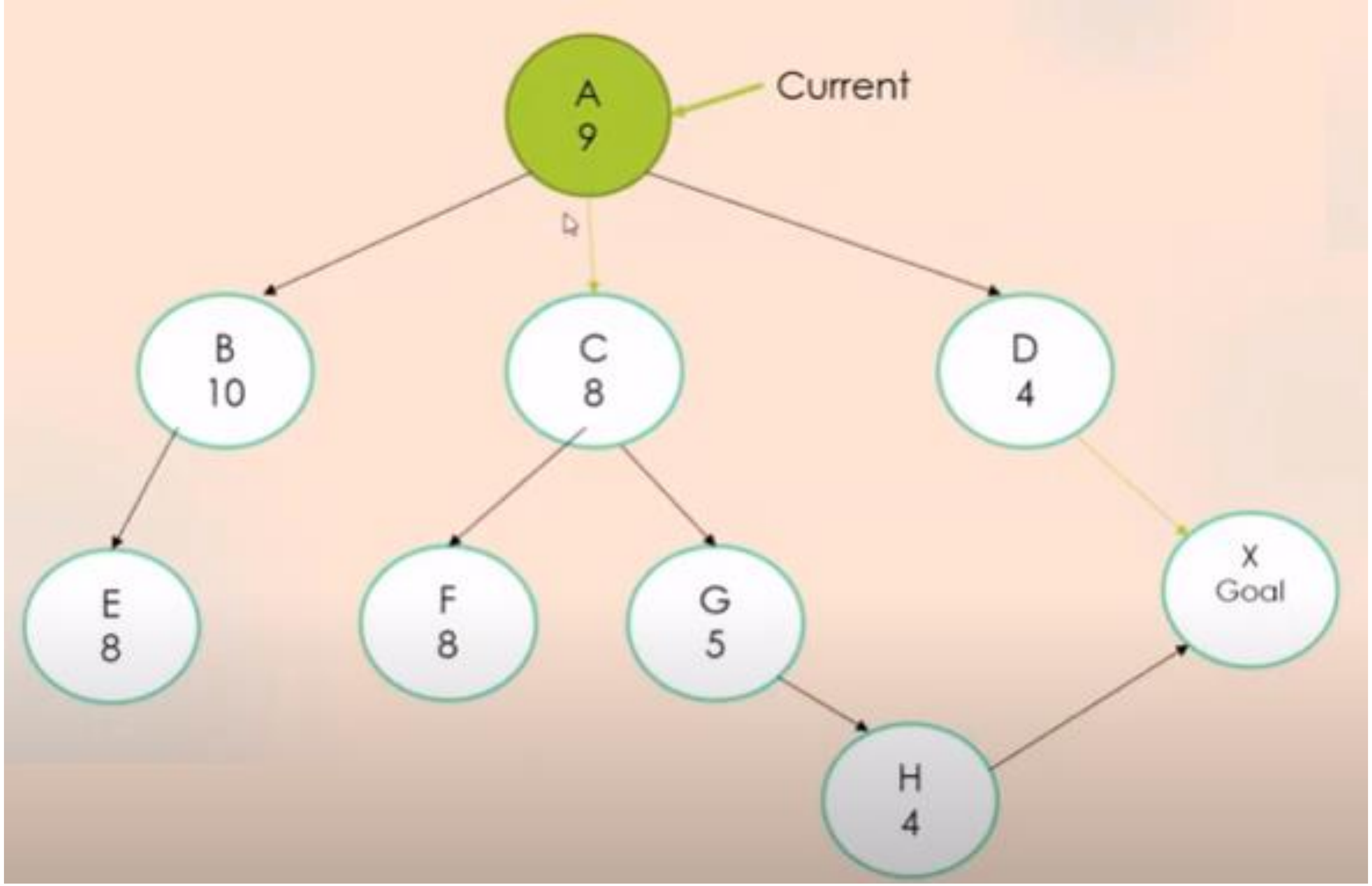


Hill climbing example (*minimizing* h)



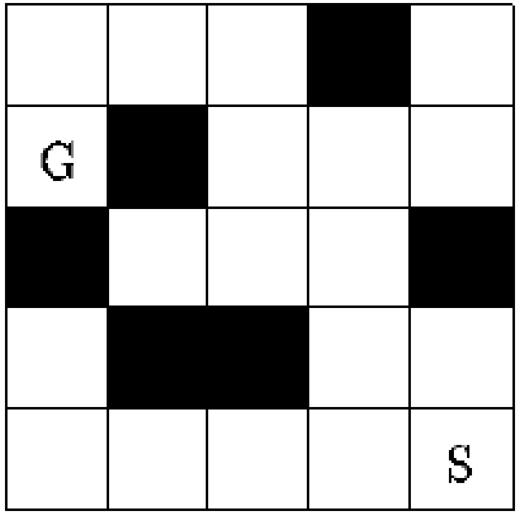

S
h=9
2
1
10
B
h=7
C
h=10
D
h=7
7
15
E
h=1
1
2
F
h=1
3
G
h=0

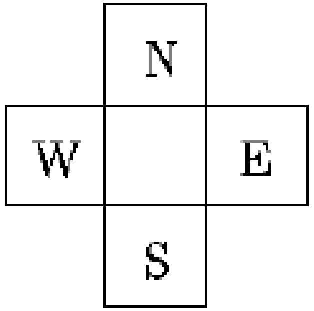

S
h=21
4
10
h=18
A
3
B
h=15
10
C
h=5
9
6
6
D
h=16
G

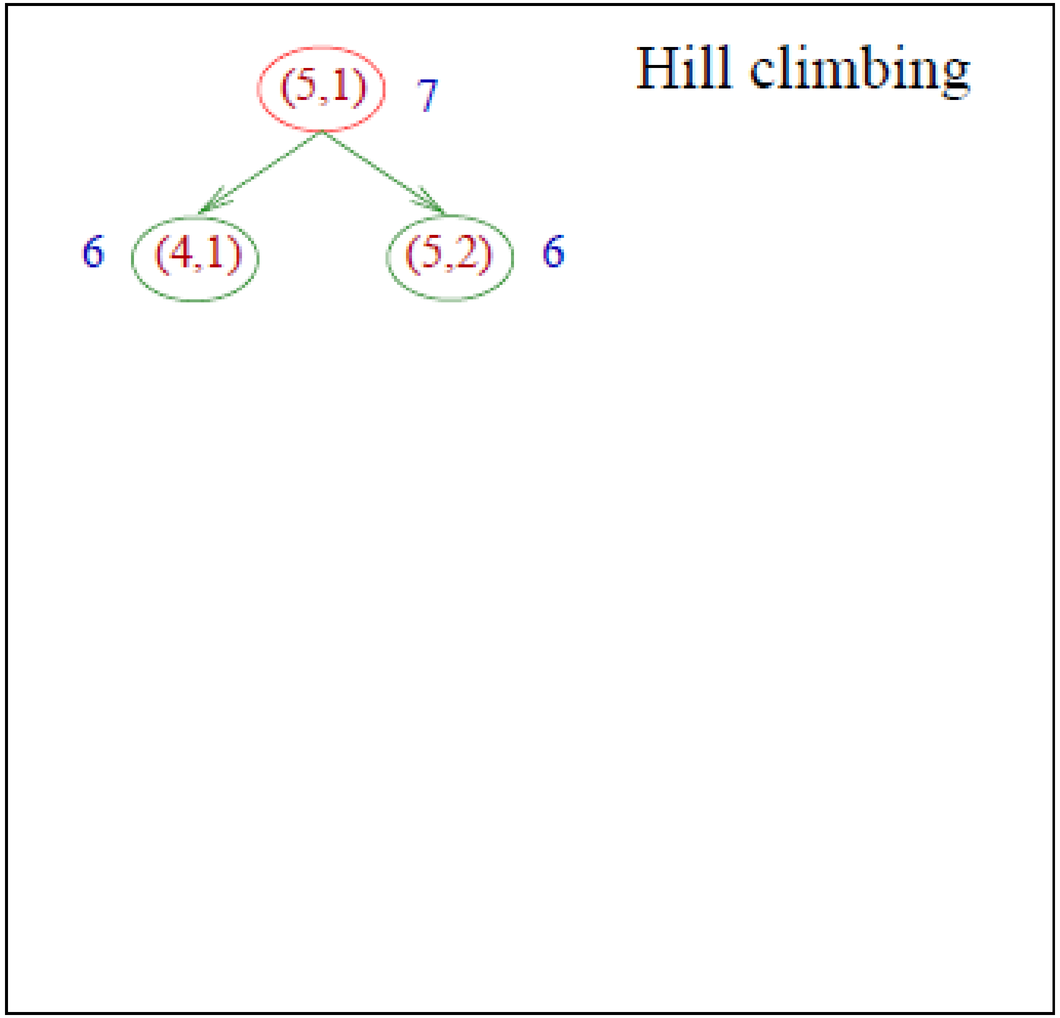

A
9
Current
B
10
C
8
D
4
E
8
F
8
G
5
X
Goal
H
4

Example of hill-climbing for the maze problem

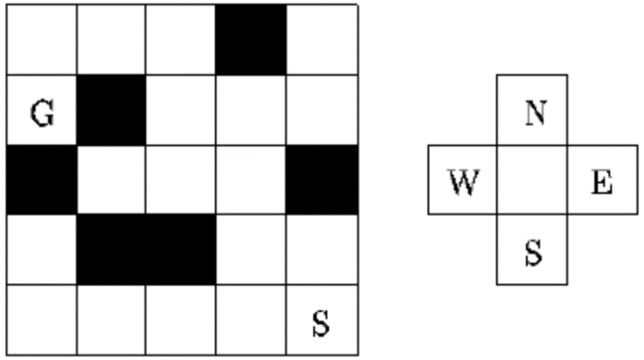
- Consider the following maze:
- The problem is to get from the start node S to the goal node G, by moving horizontally and vertically and avoiding the black obstacles in the above maze.

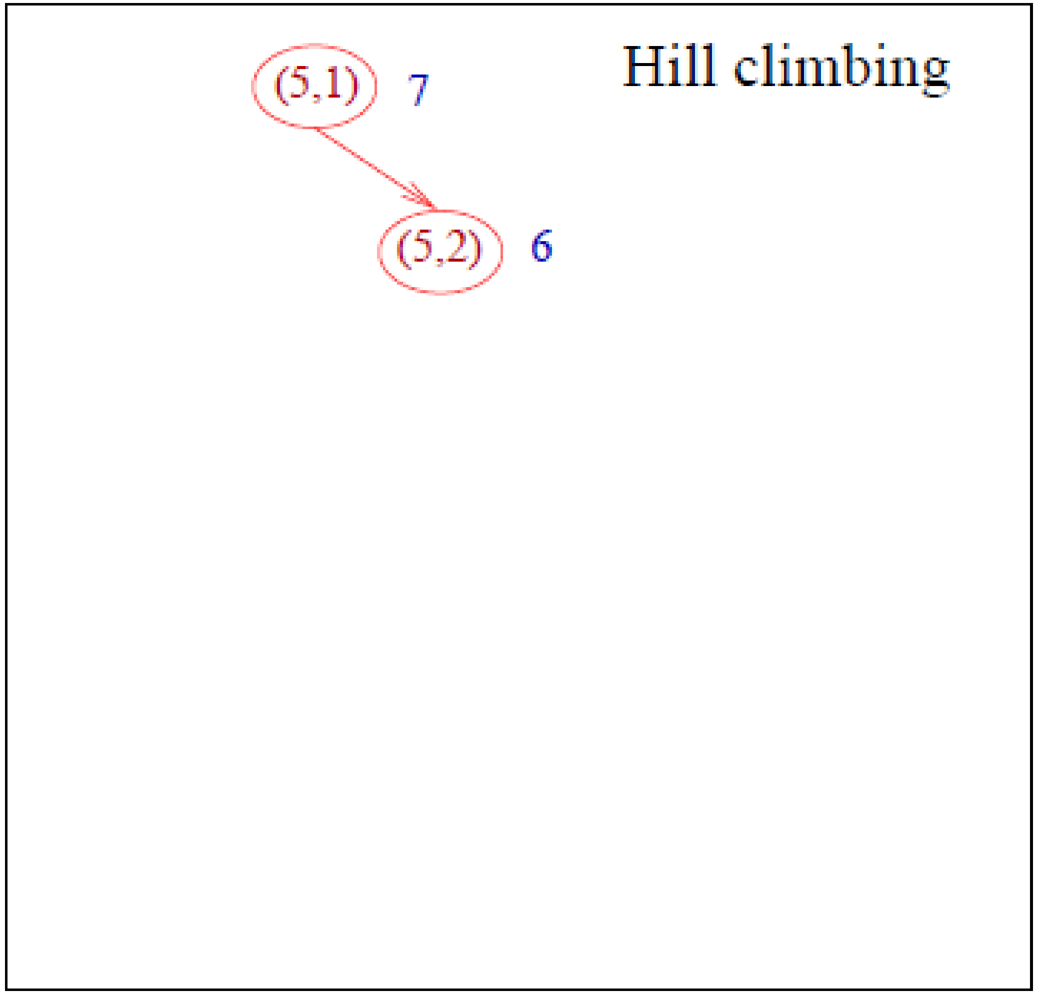




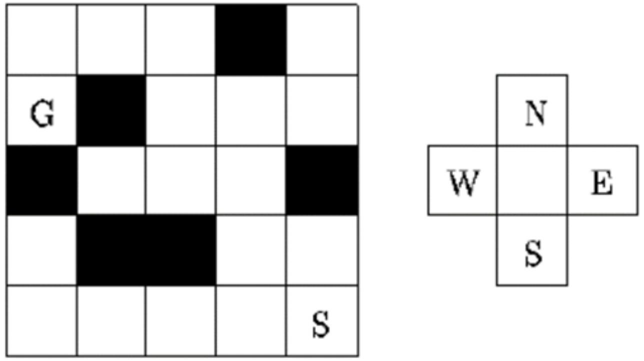


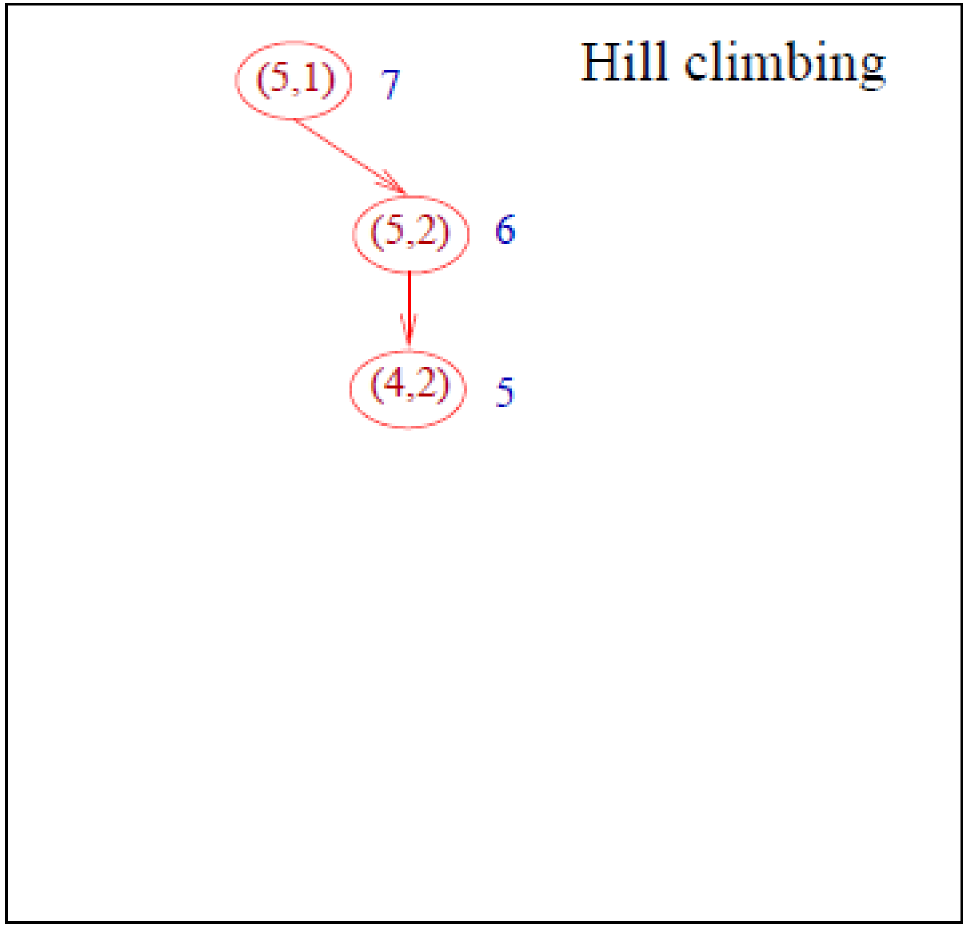
- In the figures describing the steps of the algorithm
- each position will be denoted by a pair of numbers which are the horizontal and vertical coordinates (that is, for (x, y), x denotes the horizontal coordinate and y denotes the vertical coordinate).
- The Manhattan distance between the points (x, y) and (z, t) is given by d((x, y), (z, t)) = |x − z| + |y − t|.



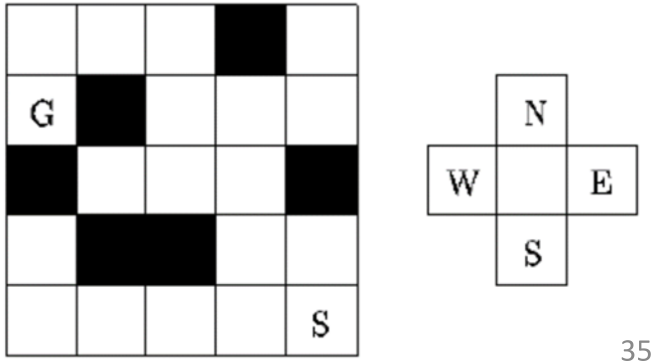


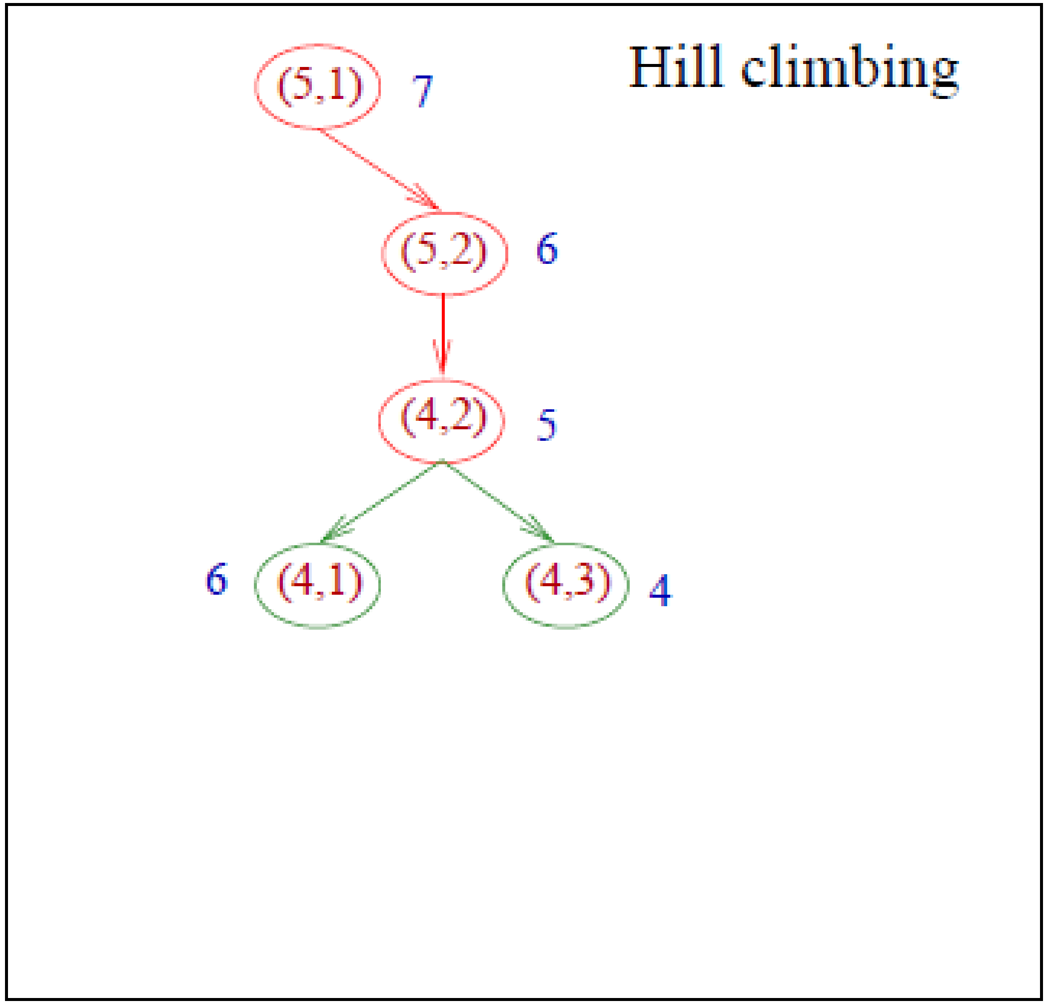
- In the figures describing the steps of the algorithm
- each position will be denoted by a pair of numbers which are the horizontal and vertical coordinates (that is, for (x, y), x denotes the horizontal coordinate and y denotes the vertical coordinate).
- The Manhattan distance between the points (x, y) and (z, t) is given by d((x, y), (z, t)) = |x − z| + |y − t|.



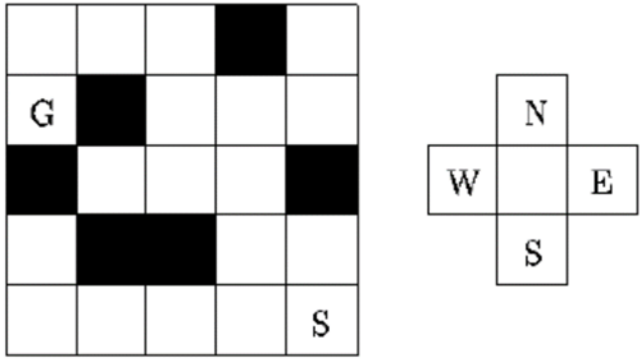


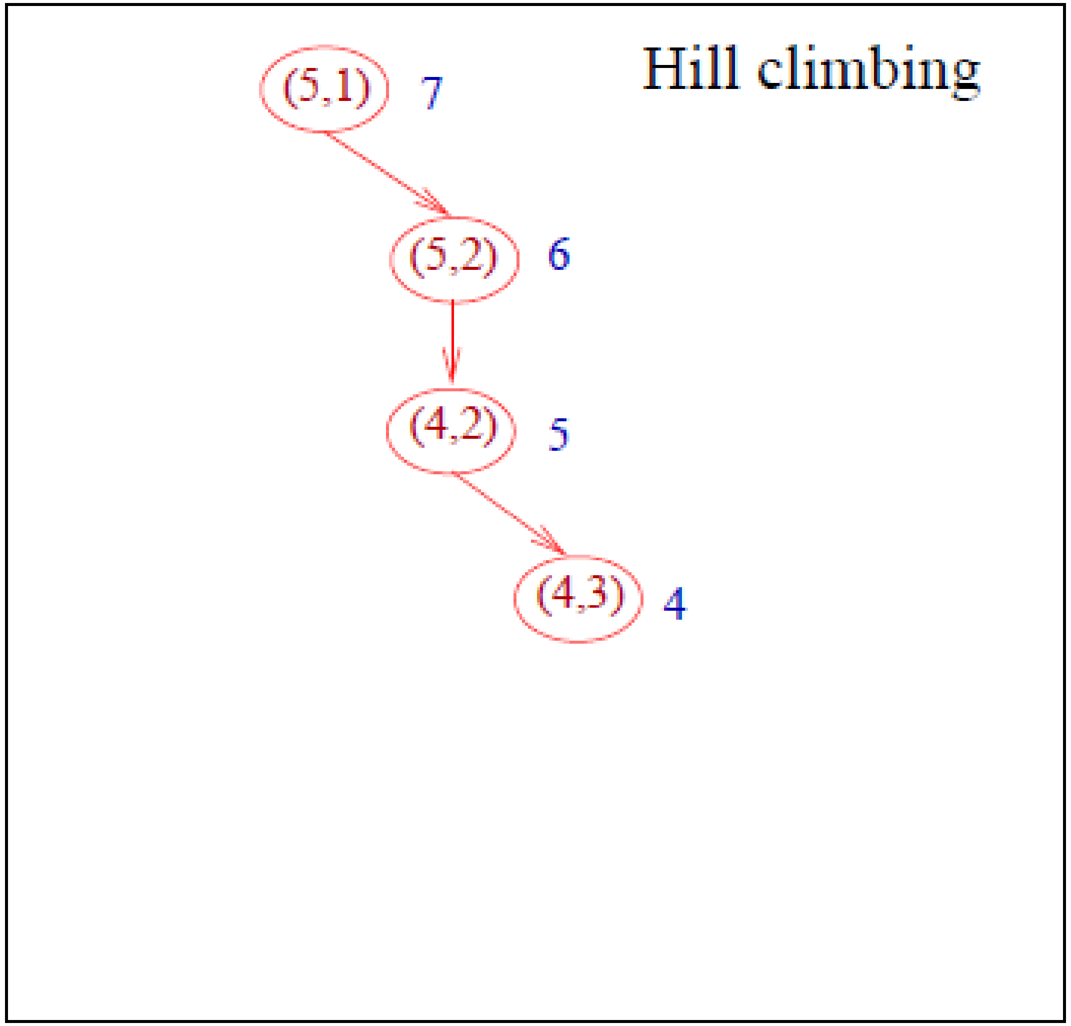
- In the figures describing the steps of the algorithm
- each position will be denoted by a pair of numbers which are the horizontal and vertical coordinates (that is, for (x, y), x denotes the horizontal coordinate and y denotes the vertical coordinate).
- The Manhattan distance between the points (x, y) and (z, t) is given by d((x, y), (z, t)) = |x − z| + |y − t|.



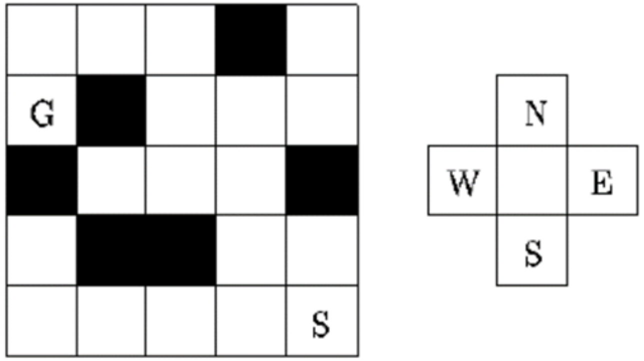


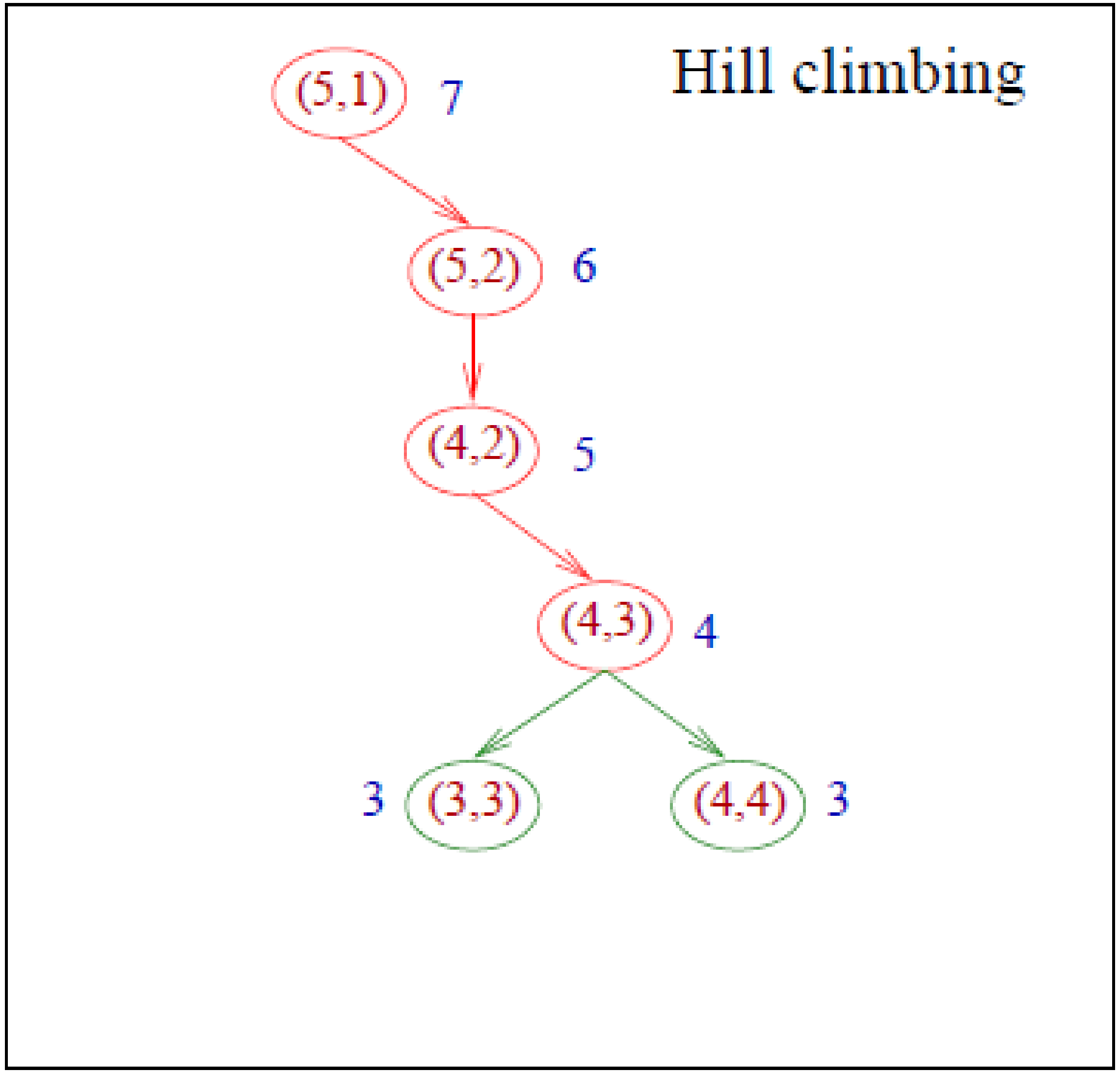
- In the figures describing the steps of the algorithm
- each position will be denoted by a pair of numbers which are the horizontal and vertical coordinates (that is, for (x, y), x denotes the horizontal coordinate and y denotes the vertical coordinate).
- The Manhattan distance between the points (x, y) and (z, t) is given by d((x, y), (z, t)) = |x − z| + |y − t|.



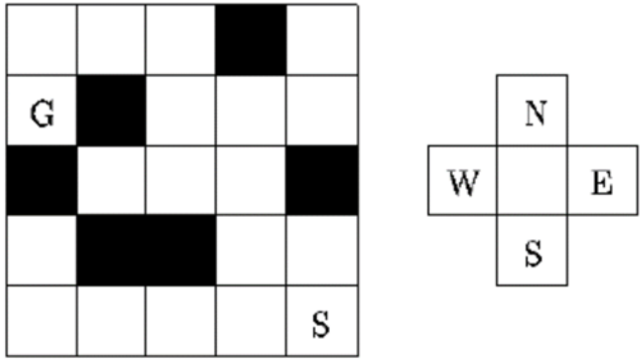


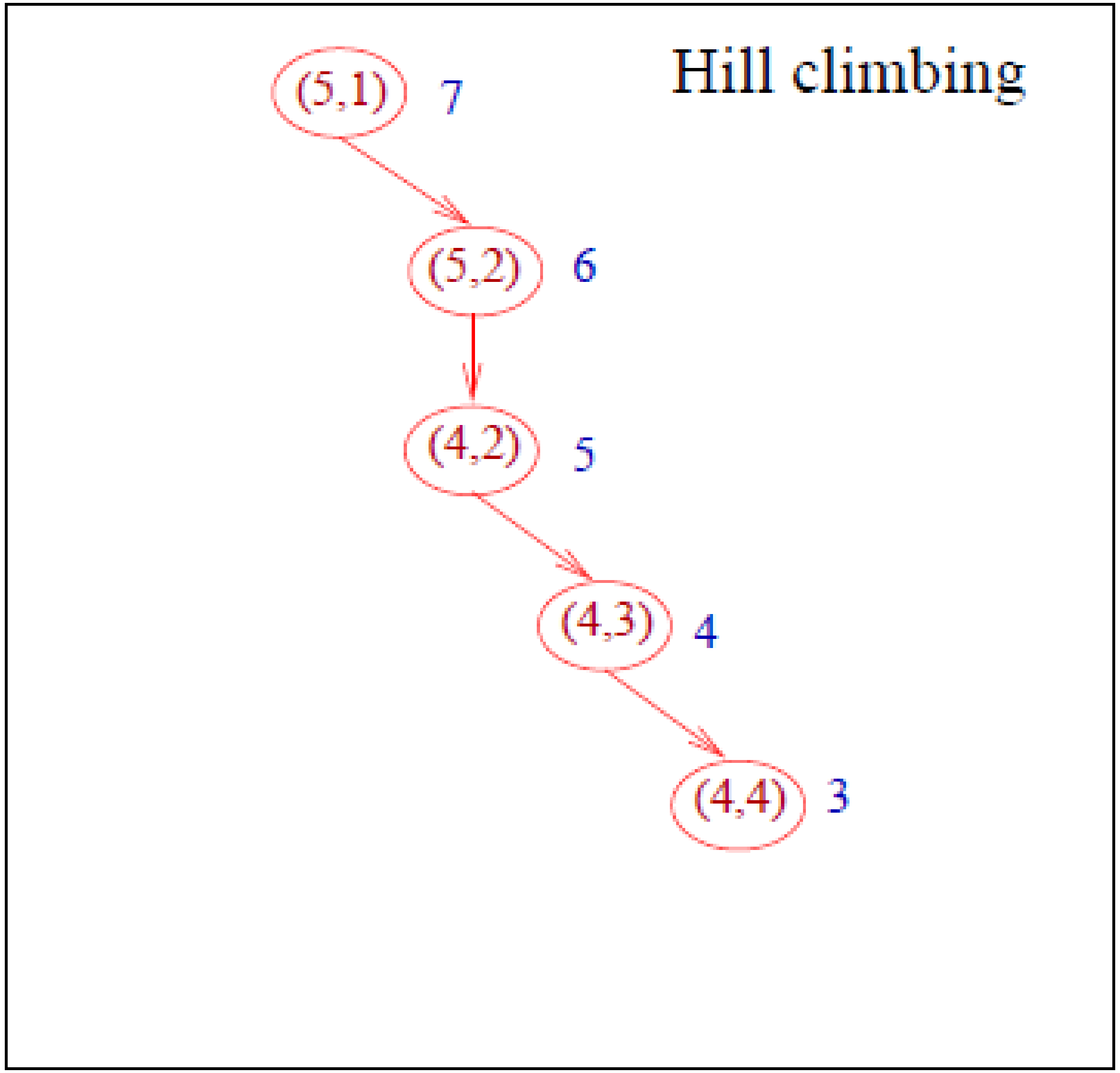
- In the figures describing the steps of the algorithm
- each position will be denoted by a pair of numbers which are the horizontal and vertical coordinates (that is, for (x, y), x denotes the horizontal coordinate and y denotes the vertical coordinate).
- The Manhattan distance between the points (x, y) and (z, t) is given by d((x, y), (z, t)) = |x − z| + |y − t|.



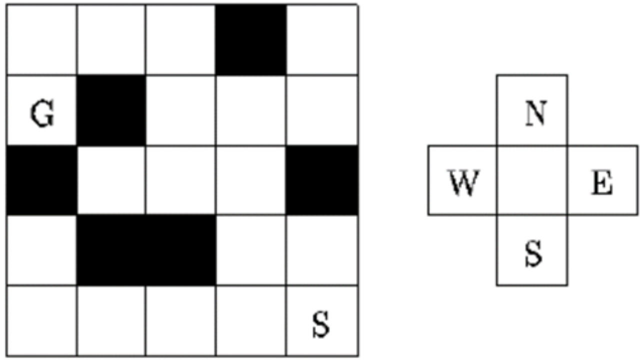


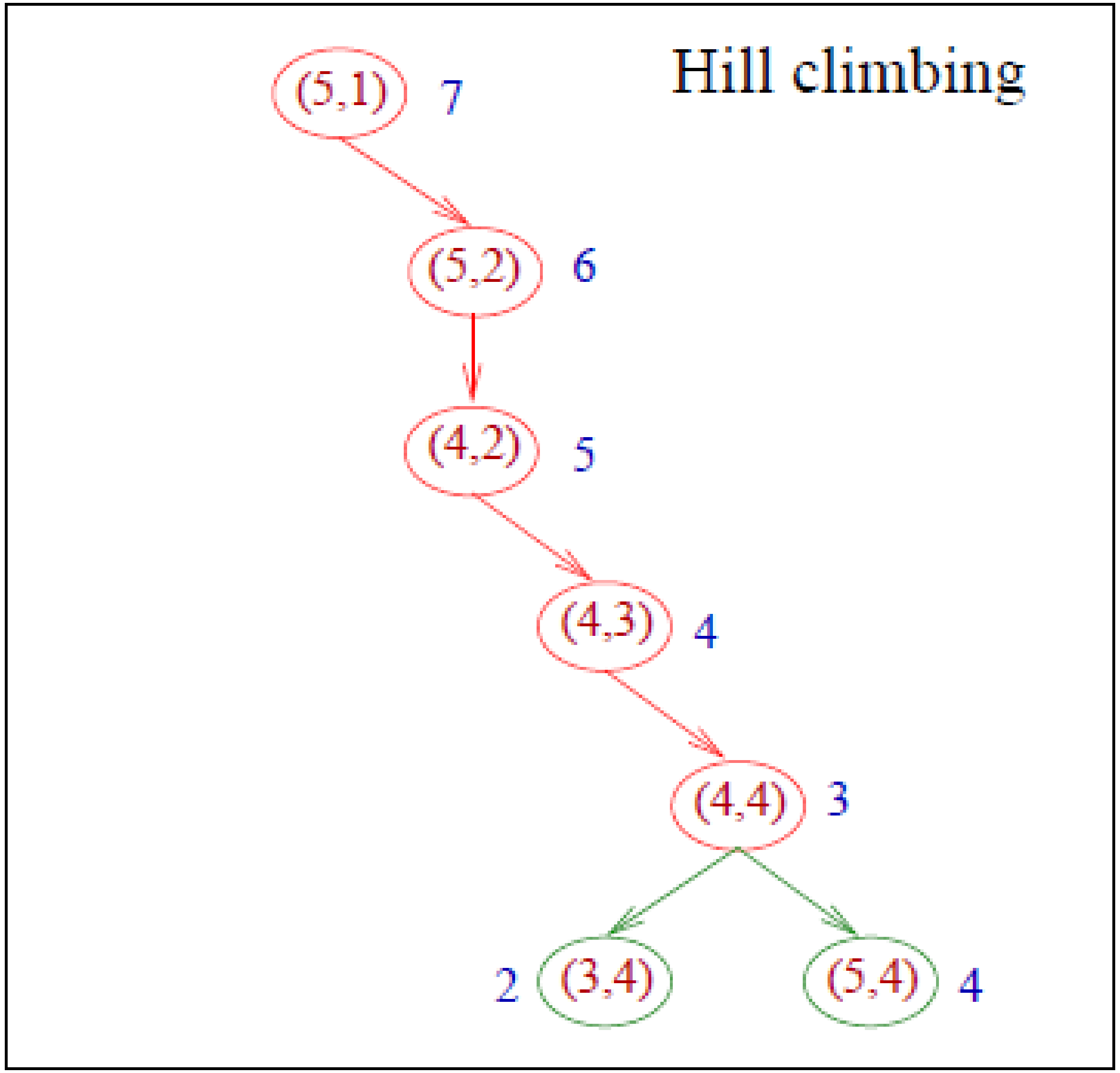
- In the figures describing the steps of the algorithm
- each position will be denoted by a pair of numbers which are the horizontal and vertical coordinates (that is, for (x, y), x denotes the horizontal coordinate and y denotes the vertical coordinate).
- The Manhattan distance between the points (x, y) and (z, t) is given by d((x, y), (z, t)) = |x – z| + |y – t|.



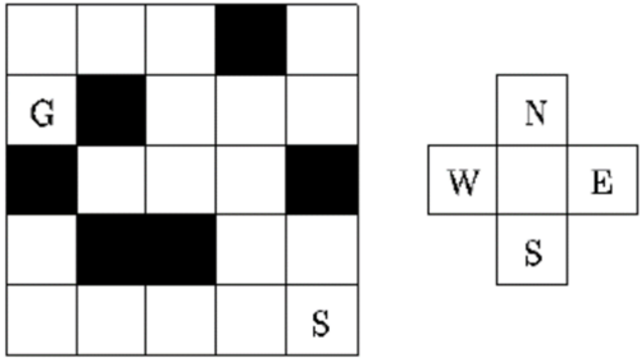


- In the figures describing the steps of the algorithm
- each position will be denoted by a pair of numbers which are the horizontal and vertical coordinates (that is, for (x, y), x denotes the horizontal coordinate and y denotes the vertical coordinate).
- The Manhattan distance between the points (x, y) and (z, t) is given by $d((x, y), (z, t)) = |x - z| + |y - t|$.



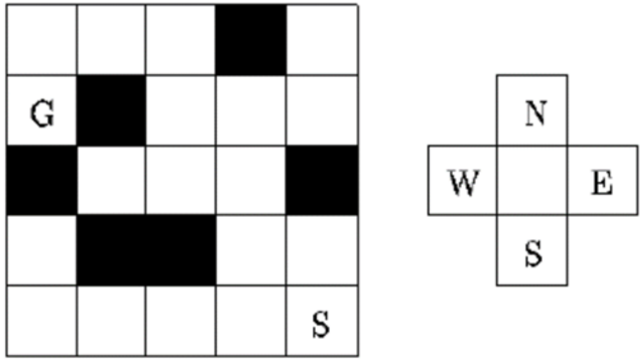


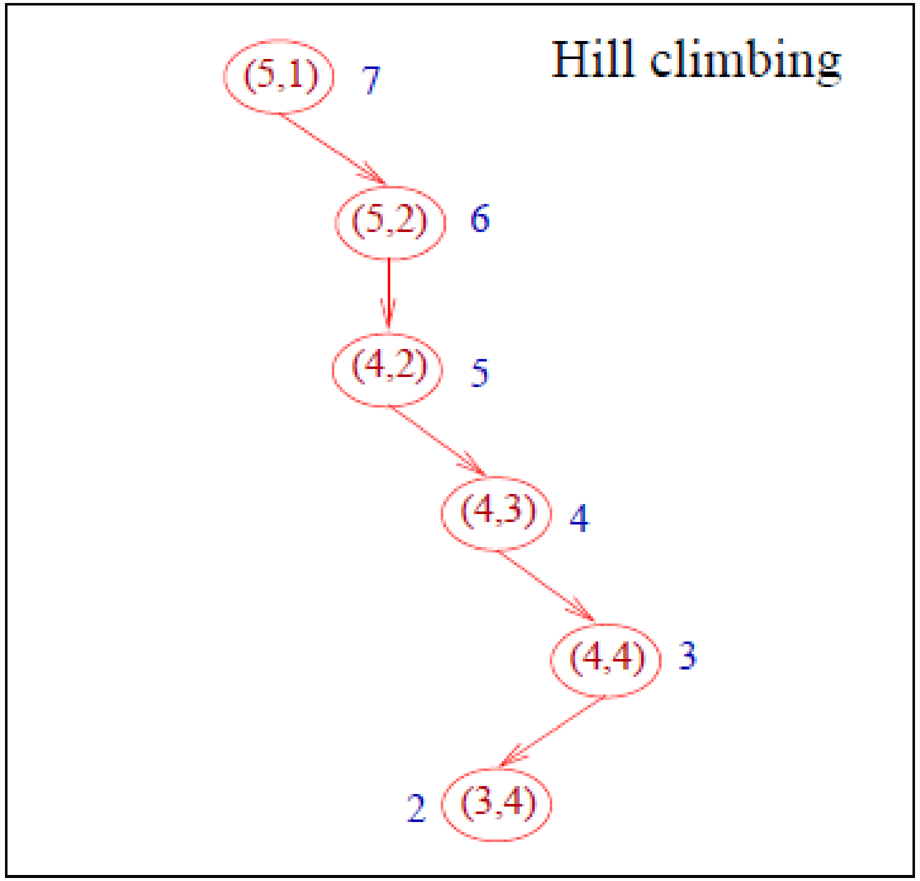
- In the figures describing the steps of the algorithm
- each position will be denoted by a pair of numbers which are the horizontal and vertical coordinates (that is, for (x, y), x denotes the horizontal coordinate and y denotes the vertical coordinate).
- The Manhattan distance between the points (x, y) and (z, t) is given by d((x, y), (z, t)) = |x − z| + |y − t|.



Example of hill-climbing for the maze problem

- The algorithm is stuck in (3,4) because all neigbours of (3,4) are worse than (3,4) (i.e., they have a higher Manhattan distance).





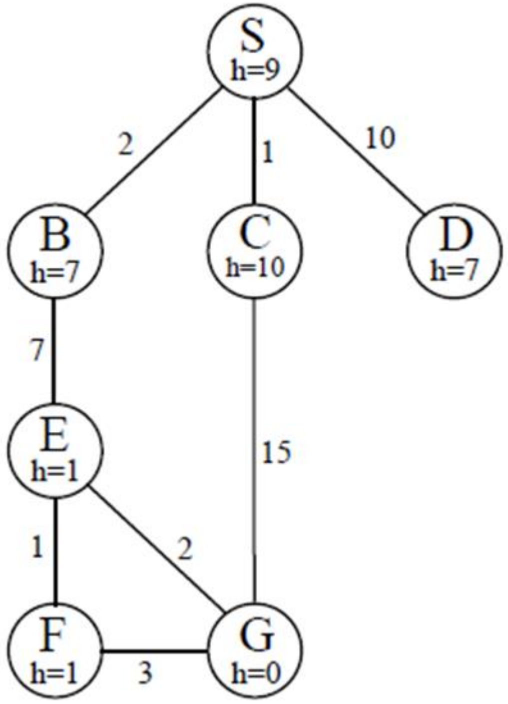
Example of hill-climbing for finding the minimum of a quadratic function

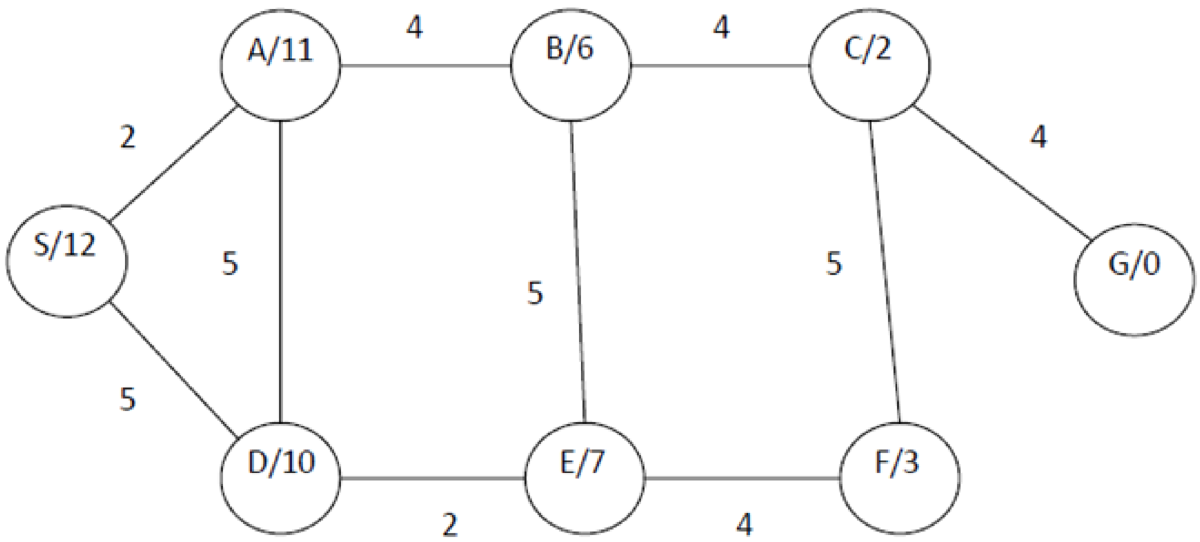
- Consider the function $f(x) = x^2 + 3x + 5$ defined on integer numbers in the interval $[-20,20]$.
- Use the hill-climbing algorithm to find the function's minimum value on this interval.
- First define the search space, then the neighborhood. Try to find the minimum using starting point $x = 0$. Then try using starting point $x = -6$.

Example of hill-climbing for finding the minimum of a quadratic function

- The search space is the set {−20,−19, ...,−1, 0, 1, 2, ..., 19, 20}.
- The neighborhood for $x = 0$ is $\{-1,0,1\}$, for $x = -1$, $\{-2,-1,0\}$ etc. In general for an x, the neighborhood is $\{x-1, x, x+1\}$.
  - f(0) = 5, f(−1) = 3, f(1) = 9, as we are looking for the minimum, hill climbing will select x = −1
  - f(−1) = 3, f(−2) = 3, f(0) = 5, so the algorithm will stop and give the answer x = −1, with the minimum value f(−1) = 3
- Starting from x = −6,
  - f(−6) = 23, f(−7) = 33, f(−5) = 15, so select x = −5.
  - f(−5) = 15, f(−6) = 23, f(−4) = 9, so x = −4 is selected.
  - f(−4) = 9, f(−5) = 15, f(−3) = 5, so x = −3 is selected.
  - f(−3) = 5, f(−4) = 9, f(−2) = 3, so x = −2 is selected.
  - f(−2) = 3, f(−3) = 5, f(−1) = 3, so x = −2 is returned as solution.

	Frontier (Hill Climbing(	Expand	Explored
1	(S,9)	S	
2	(S-B,7)(S-C,10)(S-D,7)	B	
3	(S-B-E,1)	E	
4	(S-B-E-F,1) (S-B-E-G,0)	G	
5	(S-B-E-G,0)		



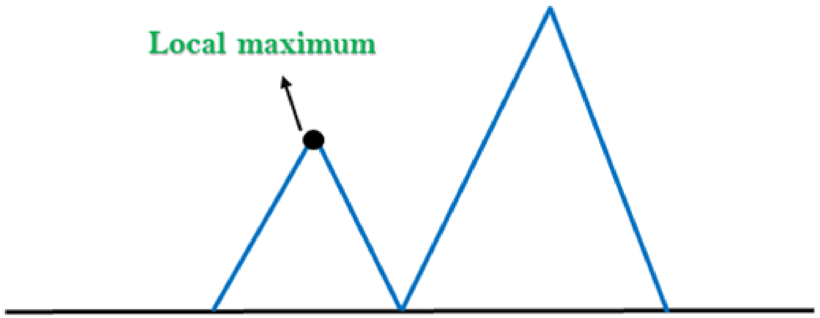


	Frontier (Hill Climbing(	Expand	Explored
1	(S,12)	S	
2	(S-A,11)(S-D,10)	D	S
3	(S-D-S,12) (S-D-A,11) (S-D-E,7)	E	
4	(S-D-E-D,10) (S-D-E-B,6) (S-D-E-F,3)	F	
5	(S-D-E-F-C,3)		
6			

Problems with Hill Climbing in AI

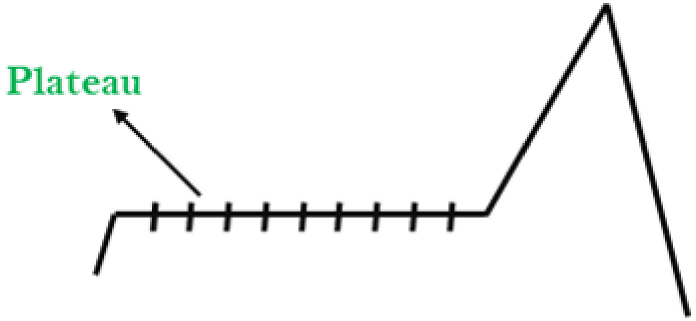
Local Maximum

- A local maximum is a **peak state in the landscape** which is better **than each of its neighboring states**, but there **is another state also present which is higher than the local maximum.**



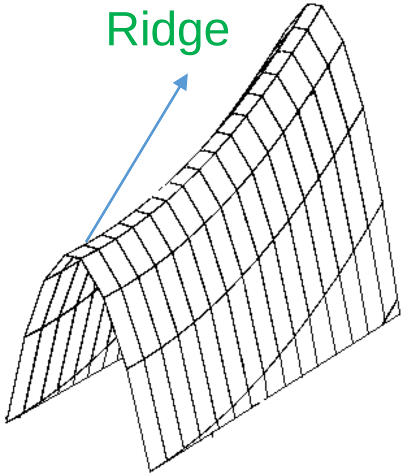
Plateau

- A plateau is the **flat area** of the search space in which all the **neighbor states of the current state contains the same value**, because of this algorithm does not **find any best direction to move**. A hill-climbing search might be **lost** in the plateau area.



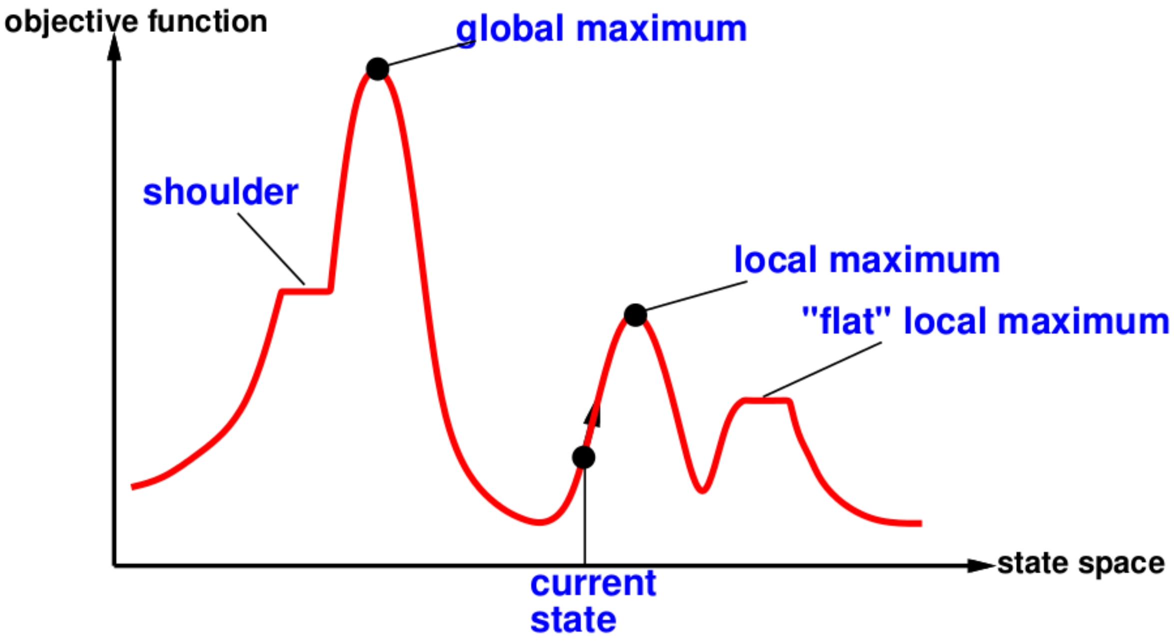
Ridge

- A ridge is a **special form of the local maximum**. It has an area which is **higher than its surrounding areas**, but itself has a **slope**, and **cannot be reached in a single move.**

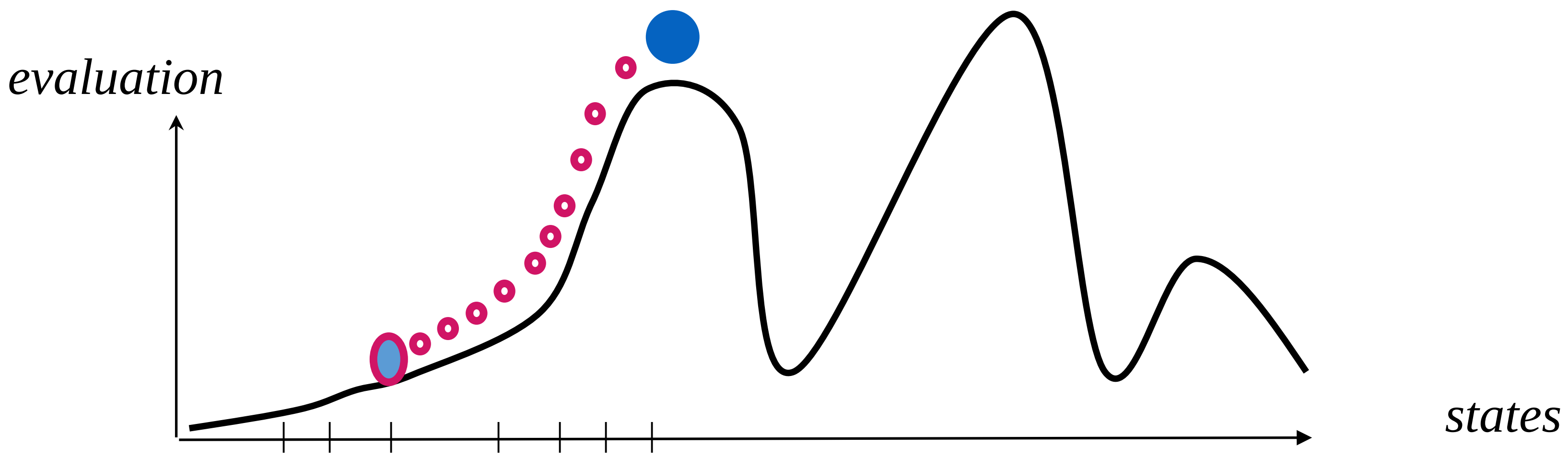


State Space Landscape

- A state space landscape is a graph of states associated with their costs
- A landscape has both “location” (defined by the state) and “elevation” (defined by the value of the heuristic cost function or objective function).



Hill Climbing



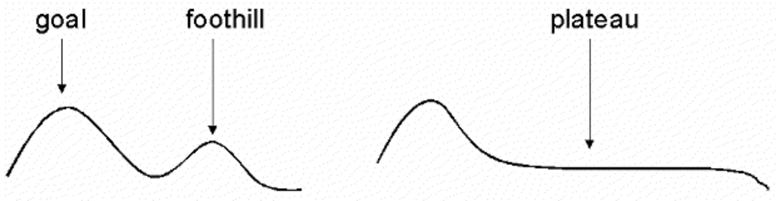
Hill-Climbing – Two Versions

- **Greedy Local Search**
  - Systematically search the Neighbourhood.
  - Accept the {first or best} cost decreasing move found.

- **Stochastic Local Search**
  - Choose randomly from amongst the cost-decreasing moves found.

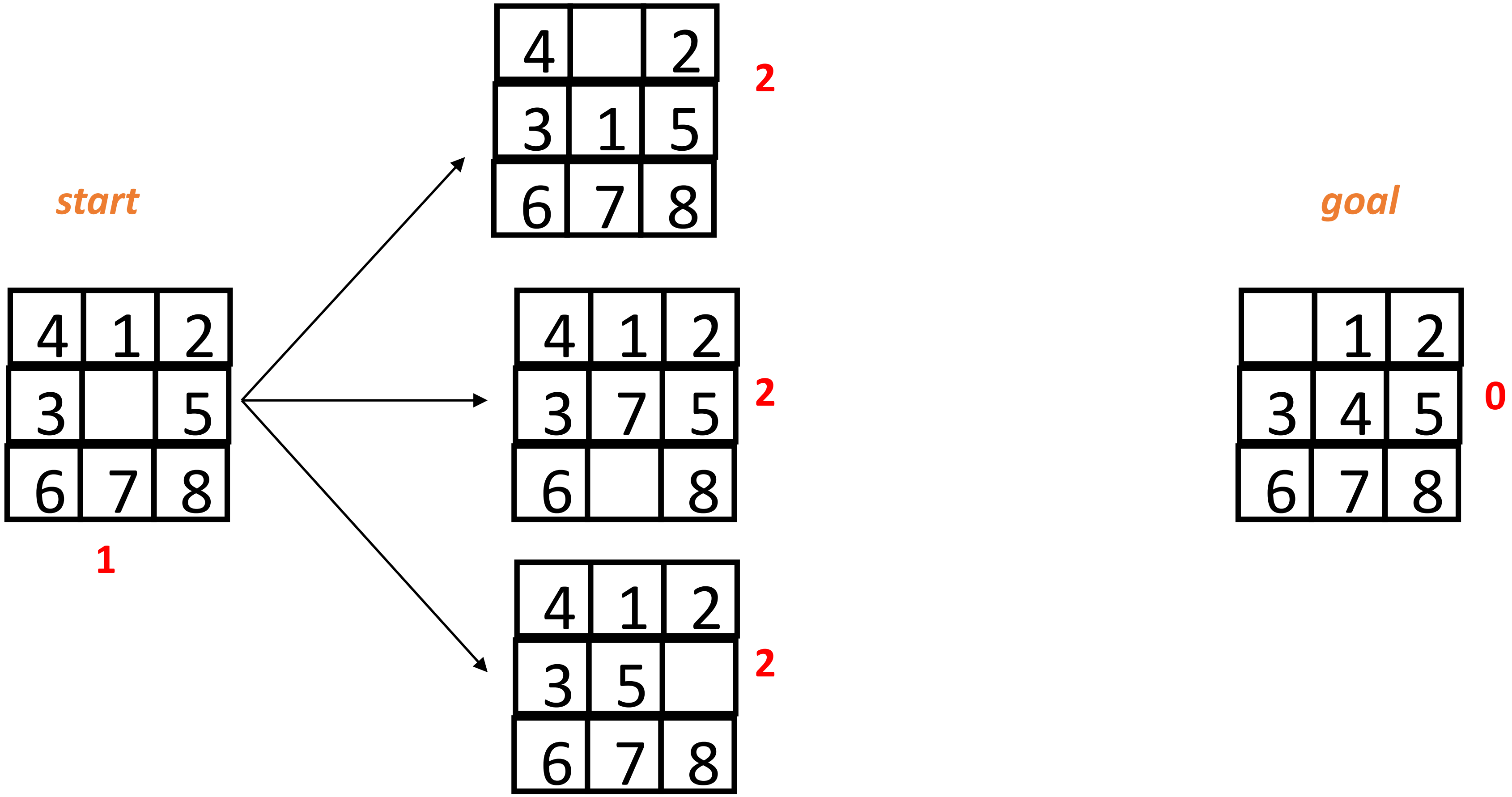
Hill-Climbing Problems

- Greedy Local Search: grabs a good neighbour state without thinking about where to go next
  - However, greedy algos do make good progress generally towards the solution



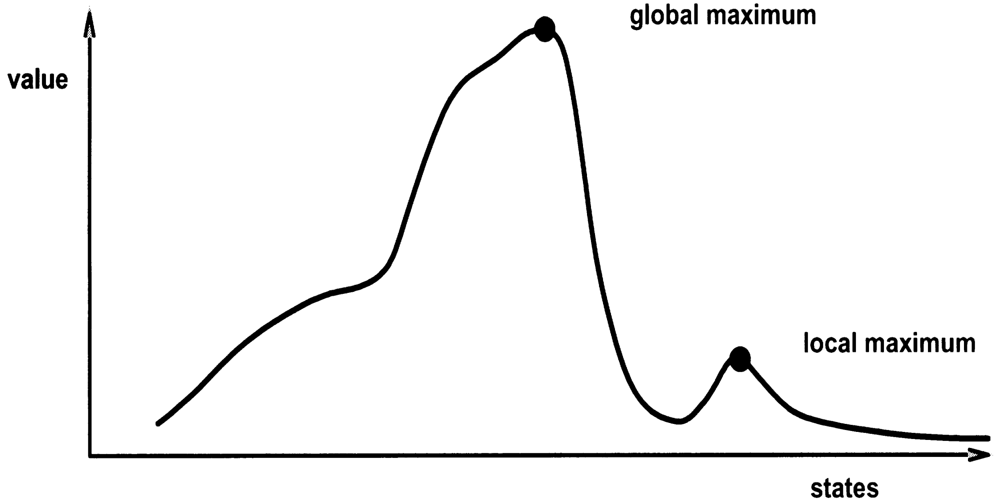
- Unfortunately, hill-climbing
  - Can get stuck in **local maxima**
  - Can be stuck by **ridges** (a series of local maxima that occur close together)
  - Can be stuck by **plateau** (a flat area in the state space landscape)
    - Shoulder: if the flat area rises uphill later on
    - Flat local maximum: no uphill rise exists.

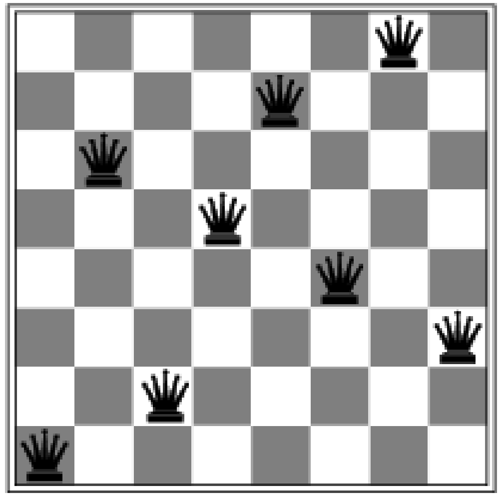
Example of a local "maximum"



Hill-Climbing Problems

- Local maxima/minima : local search can get stuck on a local maximum/minimum and not find the optimal solution





Local minimum

•Cure

+ Random restart
+ Good for Only few local maxima

Improvements

- **Random-restart Hill Climbing**: Selects a series of initial nodes randomly until the solution is found.

- Some problem spaces are great for hill climbing and others are terrible.

Variants of Hill Climbing

- Stochastic hill climbing:
  - chooses at random from among uphill moves
  - converges more slowly, but finds better solutions in some landscapes.
- First-choice hill climbing:
  - generate successors randomly until one is better than the current
  - good when a state has many successors
- Random-restart hill climbing:
  - conducts a series of hill climbing searches from randomly generated initial states, stops when a goal is found
  - It’s complete with probability approaching 1

Variants

- In **simple hill climbing**, the first closer node is chosen, whereas in **steepest ascent hill climbing** all successors are compared and the closest to the solution is chosen. Steepest ascent hill climbing is similar to best-first search, which tries all possible extensions of the current path instead of only one.
- **Stochastic hill climbing** does not examine all neighbors before deciding how to move. Rather, it selects a neighbor at random, and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.
- **Random-restart hill climbing** iteratively does hill-climbing, each time with a random initial condition. The best is kept: if a new run of hill climbing produces a better than the stored state, it replaces the stored state.

Stochastic Hill Climbing

- Stochastic hill climbing chooses at random from among the uphill moves;
- The probability of selection can vary with the steepness of the uphill move.
- Stochastic hill climbing usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.
- Stochastic hill climbing is NOT complete, but it may be less likely to get stuck

First Choice Hill Climbing

- First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
- This is a good strategy when a state has many of successors.
- First-choice hill climbing is also NOT complete,

Random Restart Hill Climbing

- Random-Restart Hill Climbing conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.
- Random-Restart Hill Climbing is complete if infinite (or sufficiently many tries) are allowed.
- If each hill-climbing search has a probability p of success, then the expected number of restarts required is 1/p.
- The success of hill climbing depends very much on the shape of the state-space landscape:
- If there are few local maxima and plateau, random-restart hill climbing will find a good solution very quickly.
- On the other hand, many real problems have many local maxima to get stuck on.

Nature-Inspired Heuristic Algorithms

- **Genetic Algorithm** (evolution)
- Neural Network (artificial neural network)
- Simulated Annealing (metal annealing)
- Tabu Search (animal's brain)
- Ant Colony Optimization
- Bee Colony Optimization