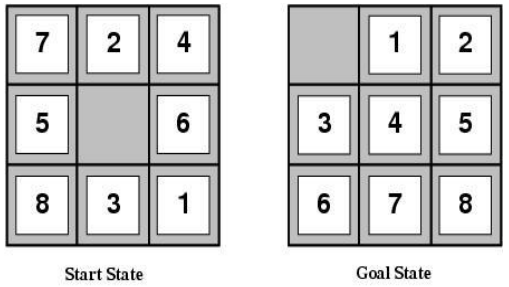
3		**Solving Problems by Searching**	12
	3.1	Definition, State space representation, Problem as a state space search, Problem formulation, Well-defined problems	
	3.2	Solving Problems by Searching, Performance evaluation of search strategies, Time Complexity, Space Complexity, Completeness, Optimality	
	3.3	Uninformed Search: Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Search, Uniform Cost Search, Bidirectional Search	
	3.4	Informed Search: Heuristic Function, Admissible Heuristic, Informed Search Technique, Greedy Best First Search, A* Search, Local Search: Hill Climbing Search, Simulated Annealing Search, Optimization: Genetic Algorithm	
	3.5	Game Playing, Adversarial Search Techniques, Mini-max Search, Alpha-Beta Pruning	

Informed search algorithms

Limitations of uninformed search

- Search Space Size makes search tedious
  - **Combinatorial Explosion**
- For example, 8-puzzle
  - Avg. solution cost is about 22 steps
  - branching factor ~ 3
  - Exhaustive search to depth 22:
    - 3.1×10^{10} states
  - E.g., d=12, IDS expands 3.6 million states on average



[24 puzzle has 10^{24} states (much worse)]

informed search algorithm

The informed search algorithm is also called heuristic search or directed search. In contrast to uninformed search algorithms, informed search algorithms require details such as distance to reach the goal, steps to reach the goal, cost of the paths which makes this algorithm more efficient.

Here, the goal state can be achieved by using the heuristic function.

The heuristic function is used to achieve the goal state with the lowest cost possible. This function estimates how close a state is to the goal.

heuristic function

In an informed search, a heuristic is a function that estimates how close a state is to the goal state. For example – Manhattan distance, Euclidean distance, etc. (Lesser the distance, closer the goal.)

The heuristic function is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close the agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in a reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

$h(n) <= h^*(n)$

Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

Admissible heuristic

- In artificial intelligence (AI), an Admissible heuristic is a guiding principle that helps estimate the distance or cost between a current state and a goal state in problem-solving tasks.
- Admissible heuristics are a type of search algorithm that are used in algorithms related to pathfinding.
- They are called admissible because they always find the shortest path to the goal state.

Admissible heuristic

- Estimation — The heuristic function, denoted as h(n), provides an estimated cost from the current node n to the goal state.

- Admissibility — For h(n) to be admissible, it must always be less than or equal to the true cost to reach the goal from n, denoted as h*(n).
- This means h(n) <= h*(n) for all nodes n.

What are the benefits of using admissible heuristics?

Admissible heuristics offer several advantages in search algorithms:

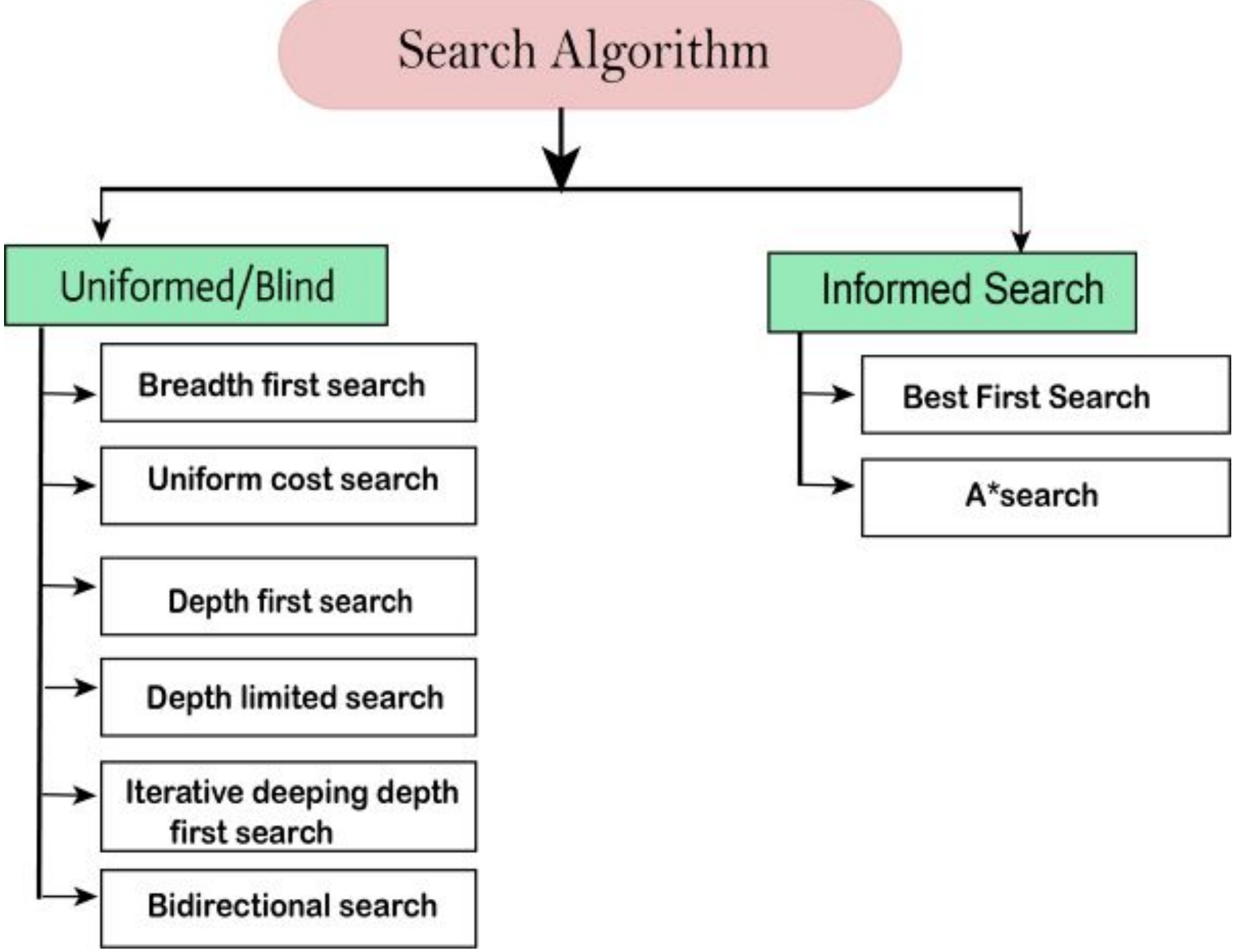
1. **Guaranteed Optimal Path** — They ensure that the shortest path to the goal state is found, provided that a path exists.

2. **Efficiency** — Admissible heuristics can be more efficient than other search strategies like breadth-first search because they only need to explore a part of the search space.

3. **Problem Simplification** — They simplify complex problems by providing a feasible path to the solution, which is particularly useful in AI, computer games, logistics, and navigation systems.

4. **Reduced Search Space** — They can prune large portions of the search space, reducing the number of nodes expanded during the search.

5. **Resource Management** — They help to minimize the system's load by facilitating real-time monitoring of events using fewer resources.


Search Algorithm
Uniformed/Blind
Informed Search
Breadth first search
Uniform cost search
Depth first search
Depth limited search
Iterative deeping depth first search
Bidirectional search
Best First Search
A*search

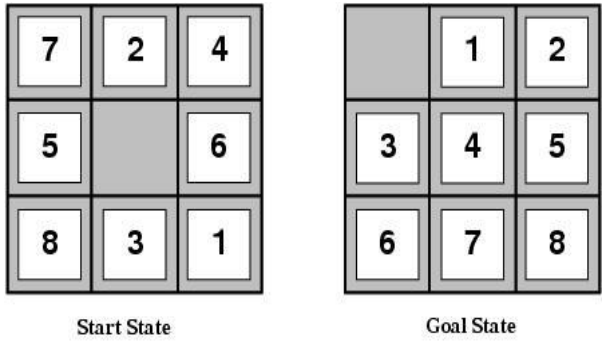
1. Greedy best-first search algorithm

Greedy best-first search uses the properties of both depth-first search and breadth-first search. Greedy best-first search traverses the node by selecting the path which appears best at the moment. The closest path is selected by using the heuristic function.

Consider the below graph with the heuristic values.

Heuristic functions for 8-puzzle

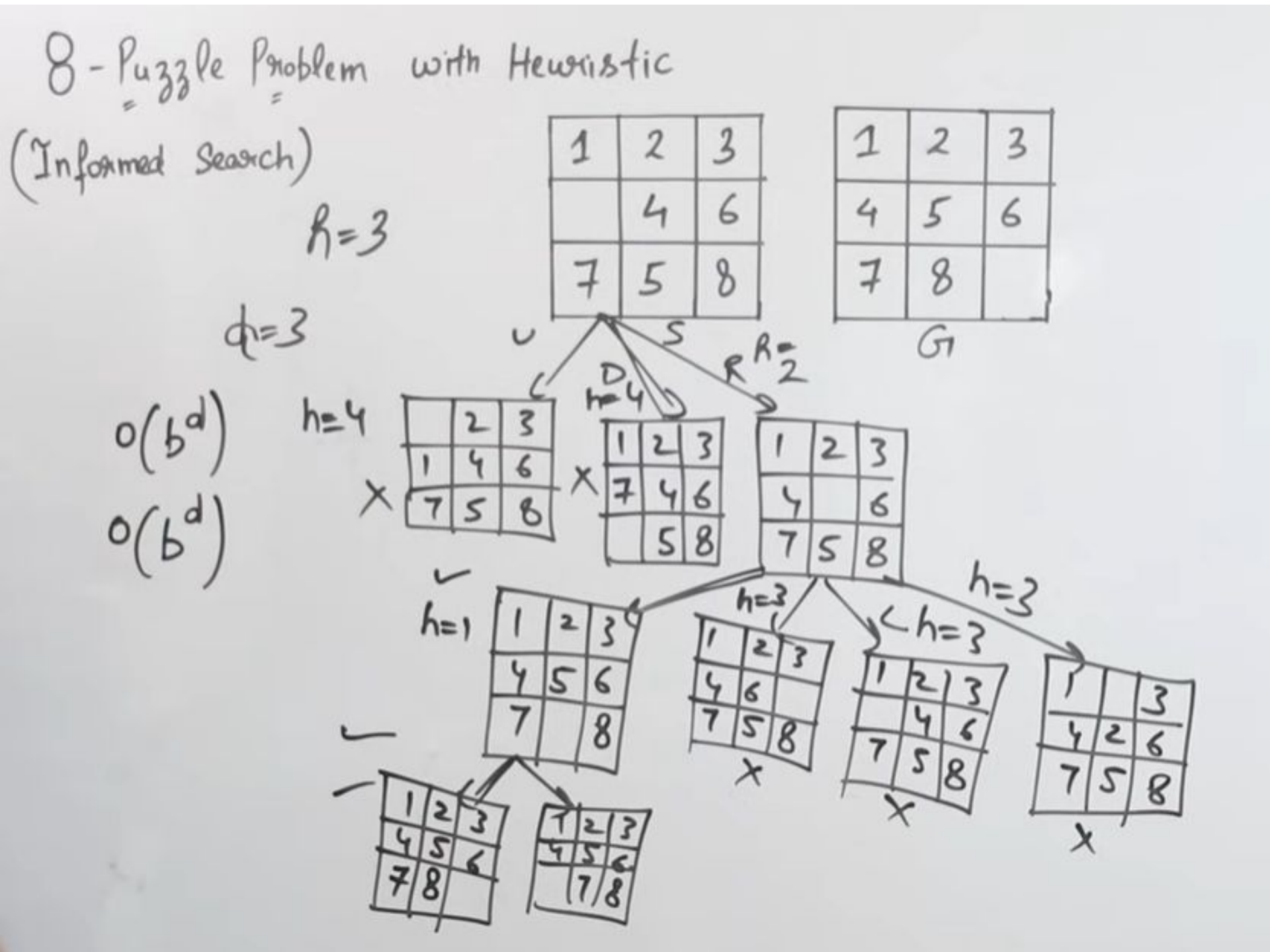
- 8-puzzle
  - Avg. solution cost is about 22 steps
  - branching factor ~ 3
  - Exhaustive search to depth 22:
    - 3.1 x 10^{10} states.
  - A good heuristic function can reduce the search process.



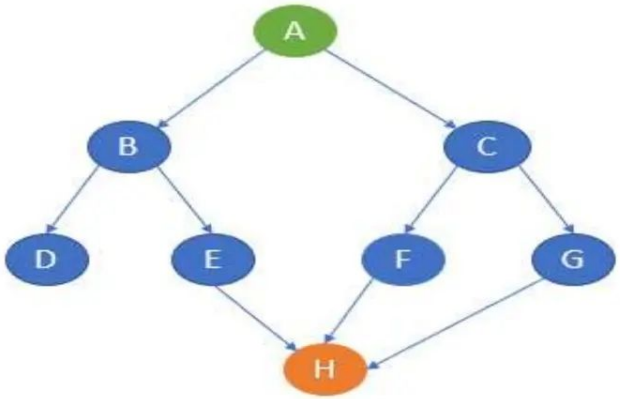
- Two commonly used heuristics
  - h_1 = the number of misplaced tiles
    - $h_1(s)=8$
  - h_2 = the sum of the distances of the tiles from their goal positions (Manhattan distance).
    - $h_2(s)=3+1+2+2+2+3+3+2=18$

8-Puzzle Problem with Heuristic

(Informed Search)



Greedy best-first search algorithm



NODES	HEURISTICS
A	13
B	12
C	4
D	7
E	3
F	8
G	2
H	0

Greedy best-first search algorithm

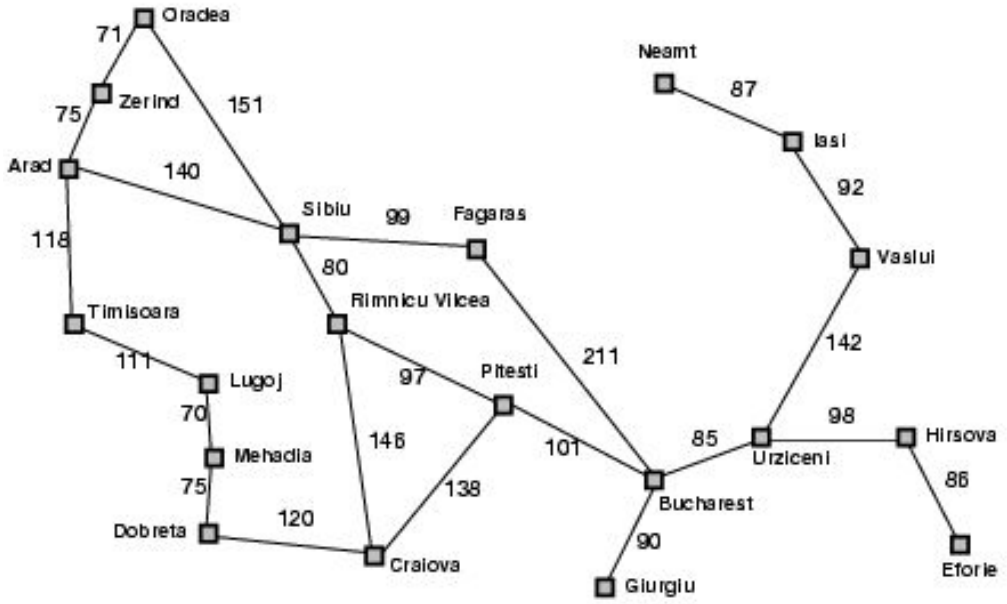
Here, A is the start node and H is the goal node.
Greedy best-first search first starts with A and then examines the next neighbour B and C. Here, the heuristics of B is 12 and C is 4. The best path at the moment is C and hence it goes to C. From C, it explores the neighbours F and G. the heuristics of F is 8 and G is 2. Hence it goes to G. From G, it goes to H whose heuristic is 0 which is also our goal state.

Greedy best-first search
(often called just "best-first")

- *h(n)* = estimate of cost from *n* to *goal*
  - e.g., *h(n)* = straight-line distance from *n* to Bucharest

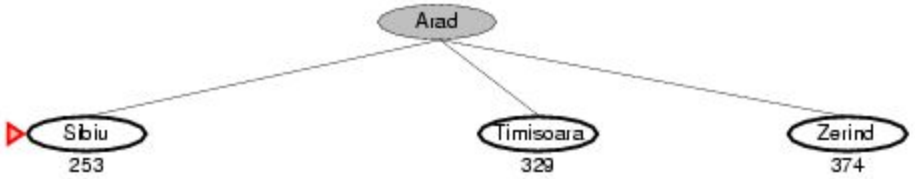
- Greedy best-first search expands the node that appears to be closest to goal.
  - *Priority queue sort function = h(n)*

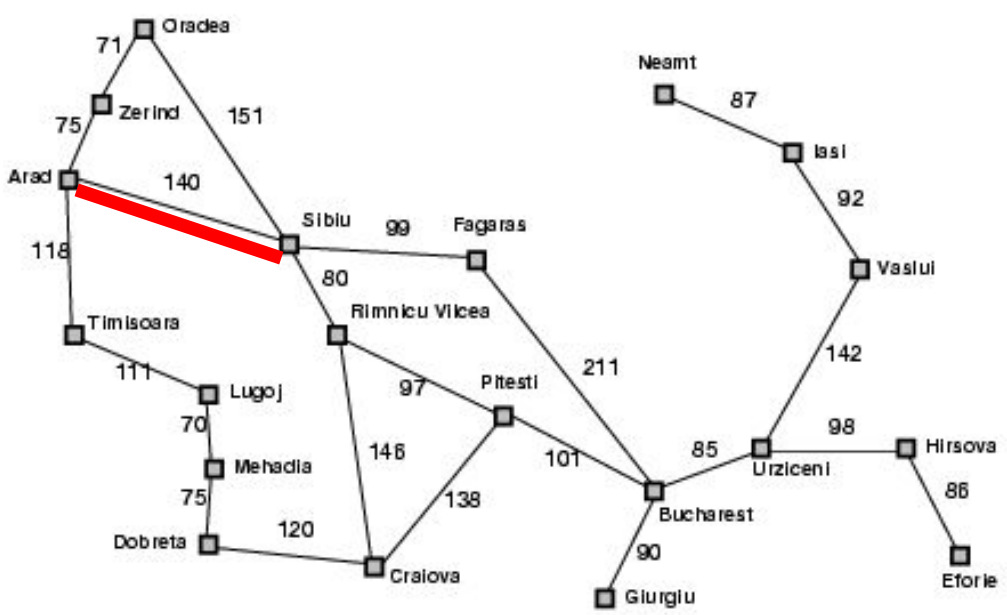
Greedy best-first search example



Straight–line distance to Bucharest	
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

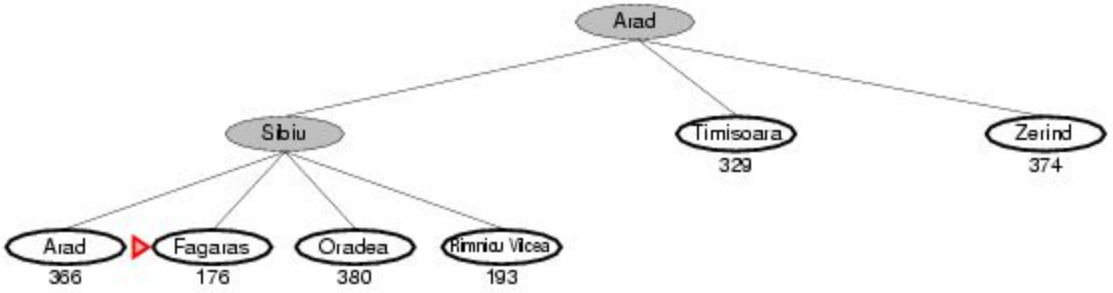
Greedy best-first search example

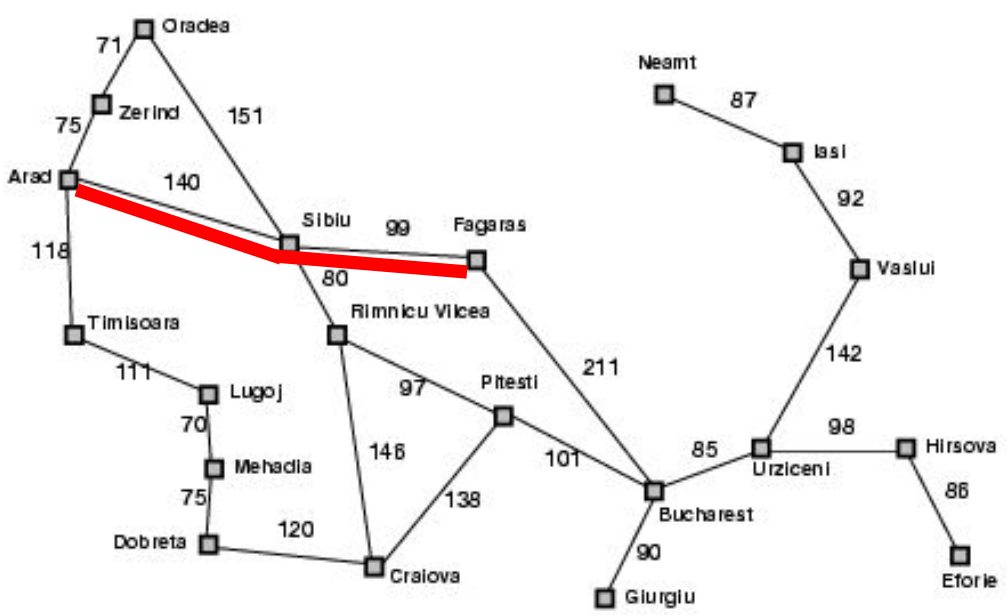




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

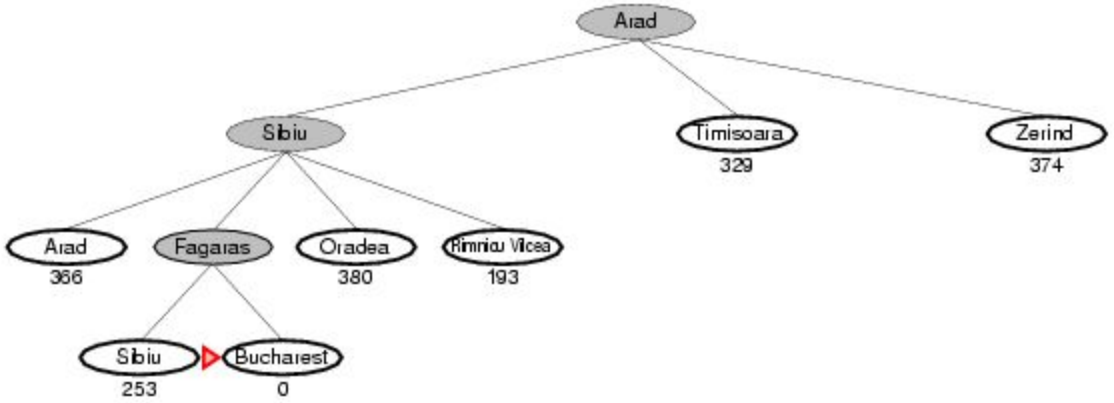


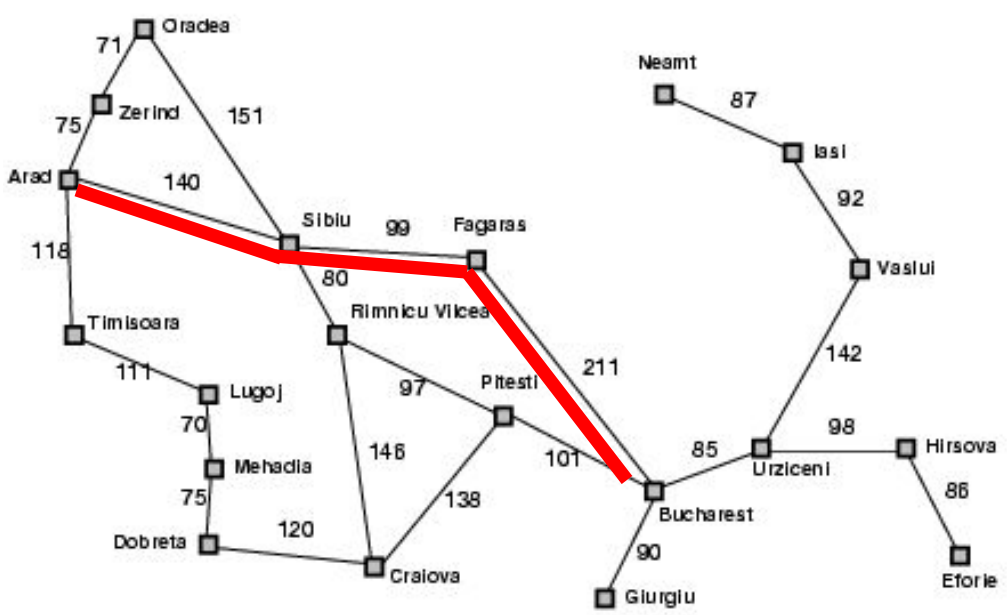


Straight–line distance to Bucharest

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

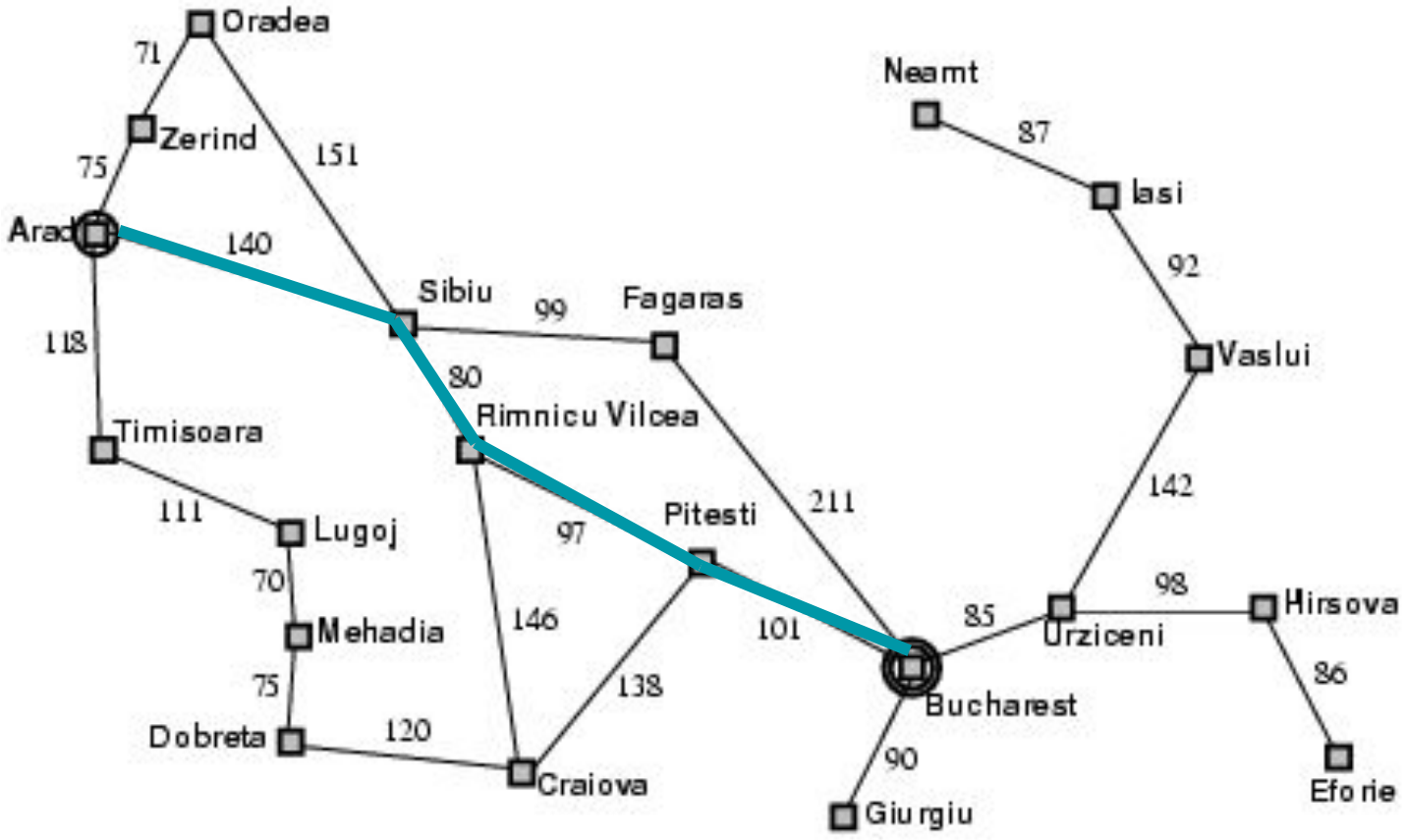




Straight–line distance to Bucharest

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

Optimal Path



Properties of greedy best-first search

- Complete?
  - Tree version can get stuck in loops.
  - Graph version is complete in finite spaces.
- Time? $O(b^m)$
  - A good heuristic can give **dramatic** improvement
- Space? $O(b^m)$
  - Keeps all nodes in memory
- Optimal? No

  e.g., Arad ☐ Sibiu ☐ Rimnicu Vilcea ☐ Pitesti ☐ Bucharest is shorter!

3		**Solving Problems by Searching**	12
	3.1	Definition, State space representation, Problem as a state space search, Problem formulation, Well-defined problems	
	3.2	Solving Problems by Searching, Performance evaluation of search strategies, Time Complexity, Space Complexity, Completeness, Optimality	
	3.3	Uninformed Search: Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Search, Uniform Cost Search, Bidirectional Search	
	3.4	Informed Search: Heuristic Function, Admissible Heuristic, Informed Search Technique, Greedy Best First Search, A* Search, Local Search: Hill Climbing Search, Simulated Annealing Search, Optimization: Genetic Algorithm	
	3.5	Game Playing, Adversarial Search Techniques, Mini-max Search, Alpha-Beta Pruning	

A* search

- Idea: avoid paths that are already expensive
  - Generally the preferred simple heuristic search
  - Optimal if heuristic is: admissible(tree)/consistent(graph)
- Evaluation function *f(n) = g(n) + h(n)*
  - g(n) = known path cost so far to node n.
  - h(n) = estimate of (optimal) cost to goal from node n.
  - f(n) = g(n)+h(n)
    = estimate of total cost to goal through node n.
- *Priority queue sort function = f(n)*

Admissible heuristics

- A heuristic $h(n)$ is admissible if for every node n,
  $h(n) \leq h^*(n)$,
  where $h^*(n)$ is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic (or at least, never pessimistic)
  - Example: $h_{SLD}(n)$ (never overestimates actual road distance)
- Theorem:
  If $h(n)$ is admissible, A* using `TREE-SEARCH` is optimal

A* Search Algorithm

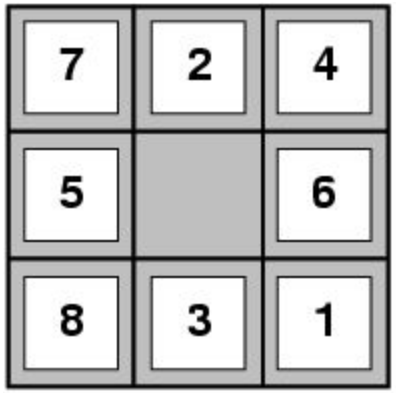
- A* combines the advantages of Dijkstra's algorithm and Greedy Best-First Search.
- Like Dijkstra's algorithm A* ensures that the path found is as **short as possible** but does so more **efficiently** by directing its search through a heuristic similar to Greedy Best-First Search.
- A heuristic function, denoted h(n), estimates the cost of getting from any given node n to the destination node.

Admissible heuristics

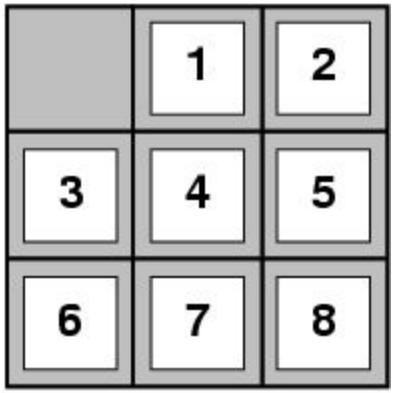
E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



Start State



Goal State

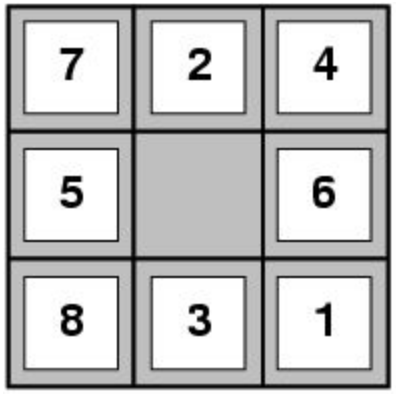
- $h_1(S) = ?$
- $h_2(S) = ?$

Admissible heuristics

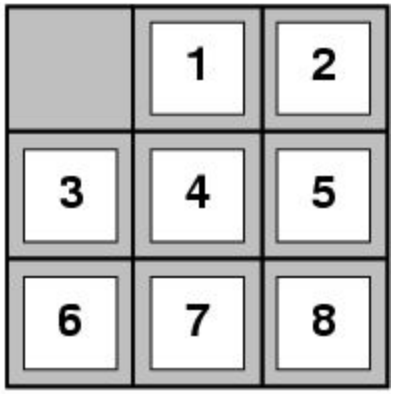
E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



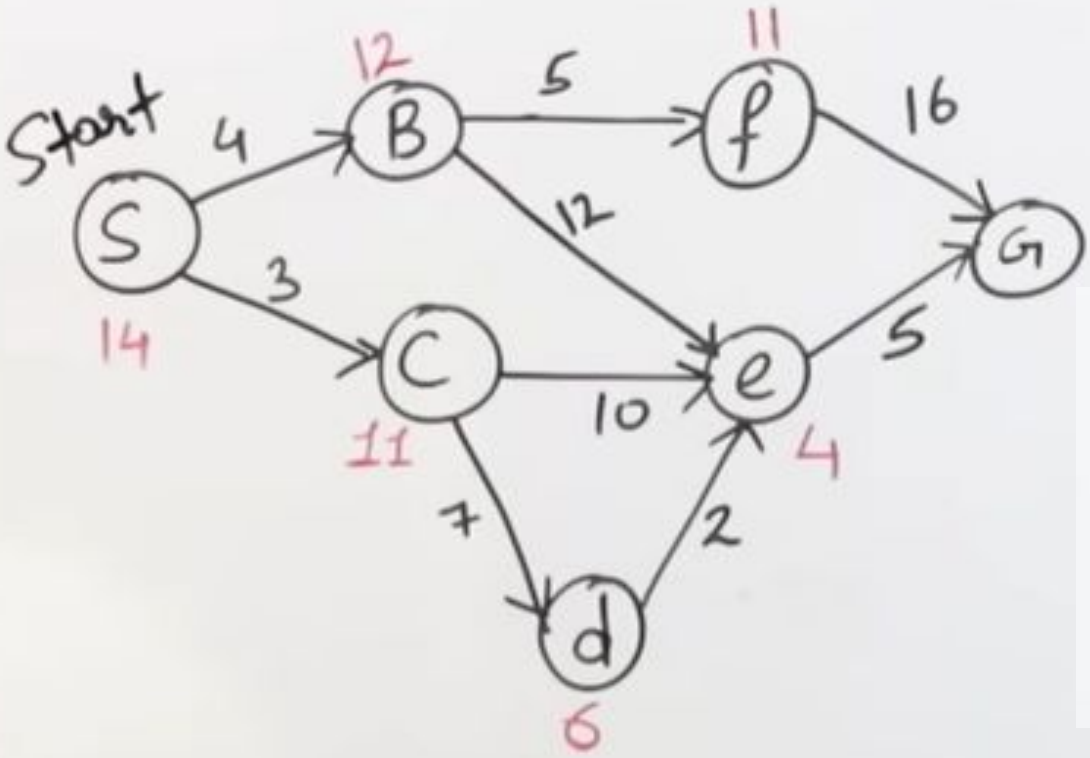
Start State



Goal State

- $h_1(S) = ?$ 8
- $h_2(S) = ?$ 3+1+2+2+2+3+3+2 = 18

A* algorithm → Informed Searching



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

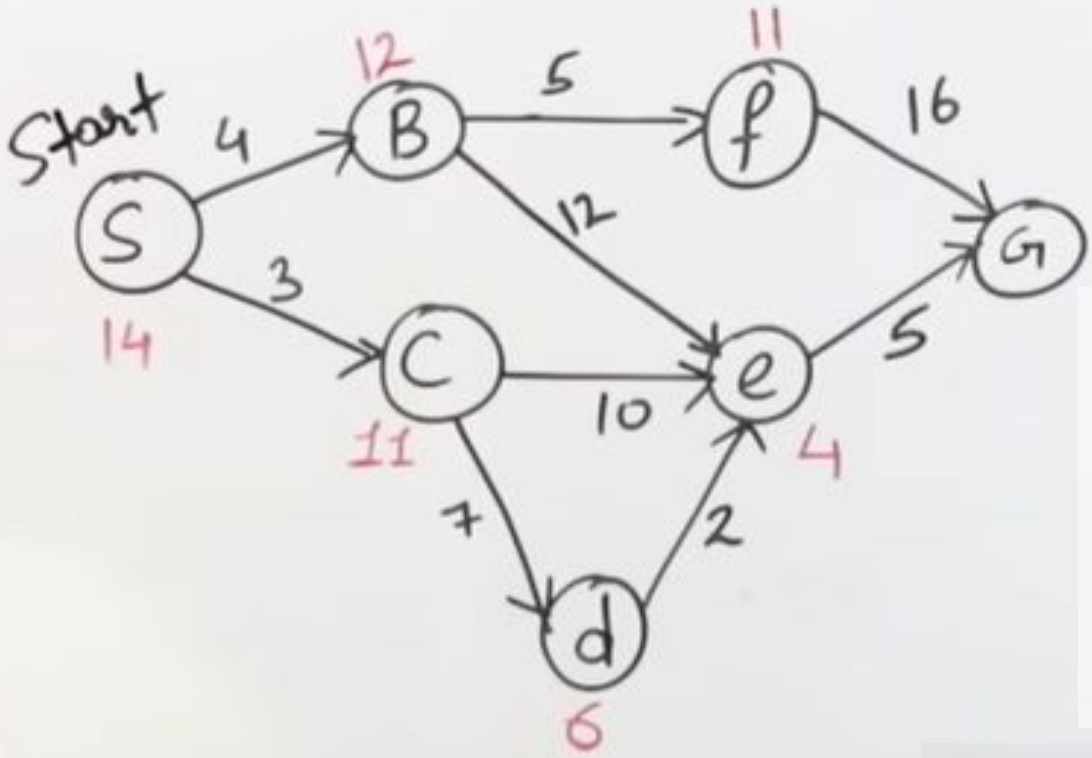
- $g(N)$: Actual Cost from Start node to n
- $h(N)$: Estimation Cost from n to Goal node

$f(S) = 0 + 14 = (14)$

$S \rightarrow B$: $4 + 12 = (16)$

$S \rightarrow C$: $3 + 11 = (14)$

A* algorithm → Informed Searching



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

- $g(N)$: Actual Cost from Start node to n
- $h(N)$: Estimation Cost from n to Goal node

$f(S) = 0 + 14 = (14)$

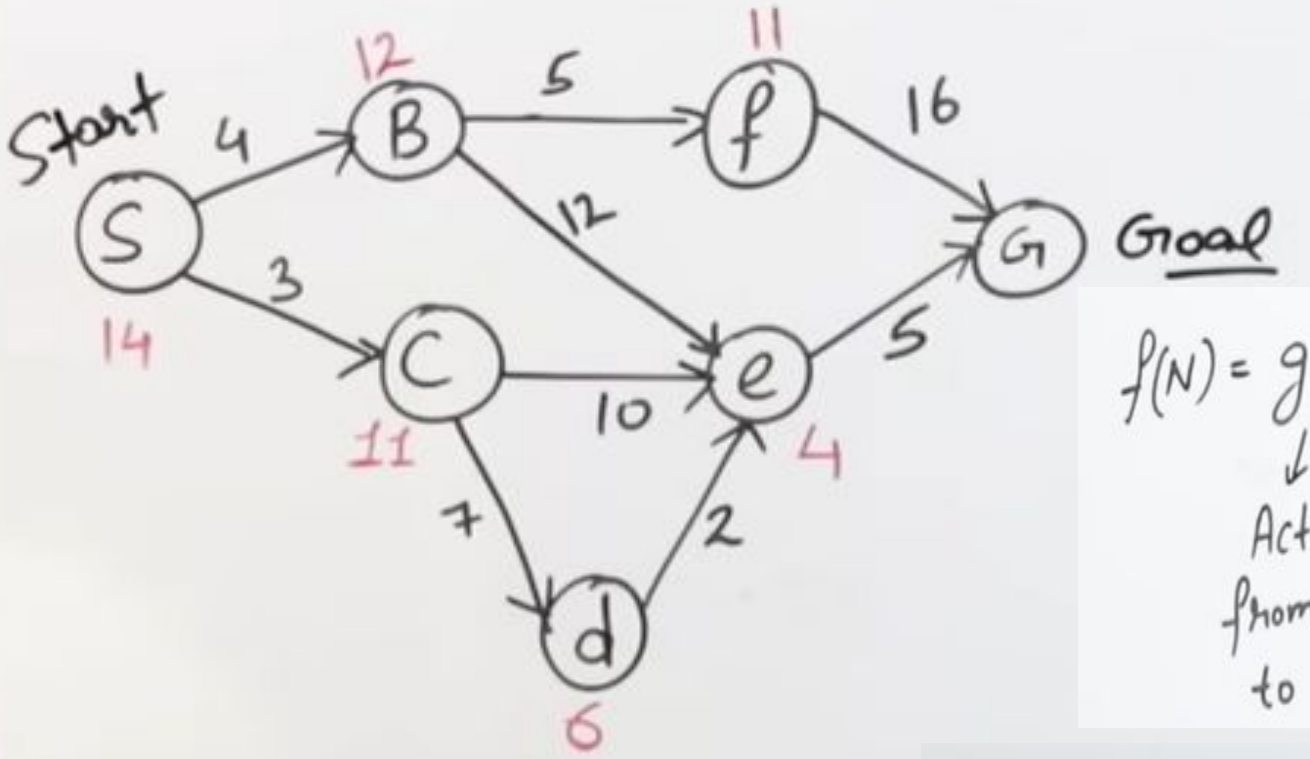
$S \rightarrow B$: $4 + 12 = (16)$

$S \rightarrow C$: $3 + 11 = (14)$

$SC \rightarrow e$: $3 + 10 + 4 = 17$

$SC \rightarrow d$: $3 + 7 + 6 = (16)$ ✓

A* algorithm → Informed Searching



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

- $g(N)$: Actual Cost from Start node to n
- $h(N)$: Estimation Cost from n to Goal node

$f(S) = 0 + 14 = (14)$

$S \rightarrow B$: $4 + 12 = (16)$

$S \rightarrow C$: $3 + 11 = (14)$

$SC \rightarrow e$: $3 + 10 + 4 = 17$

$SC \rightarrow d$: $3 + 7 + 6 = (16)$ ✓

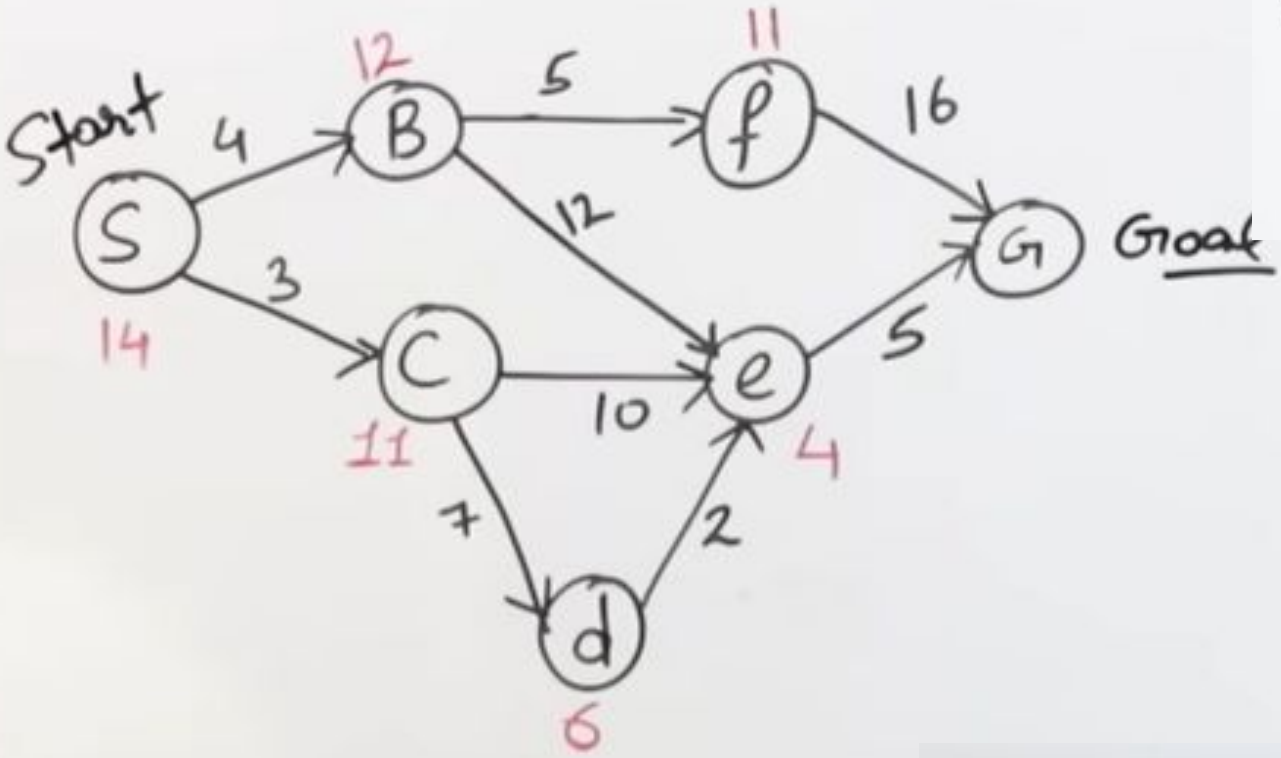
$SB \rightarrow f$: $5 + 4 + 11 = (20)$

$SB \rightarrow e$: $4 + 12 + 4 = (20)$

A* algorithm → Informed Searching

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

- $g(N)$: Actual Cost from Start node to n
- $h(N)$: Estimation Cost from n to Goal node



$f(S) = 0 + 14 = (14)$

$S \rightarrow B$: $4 + 12 = (16)$

$S \rightarrow C$: $3 + 11 = (14)$

$SC \rightarrow e$: $3 + 10 + 4 = 17$

$SC \rightarrow d$: $3 + 7 + 6 = (16)$ ✓

$SB \rightarrow f$: $5 + 4 + 11 = (20)$

$SB \rightarrow e$: $4 + 12 + 4 = (20)$

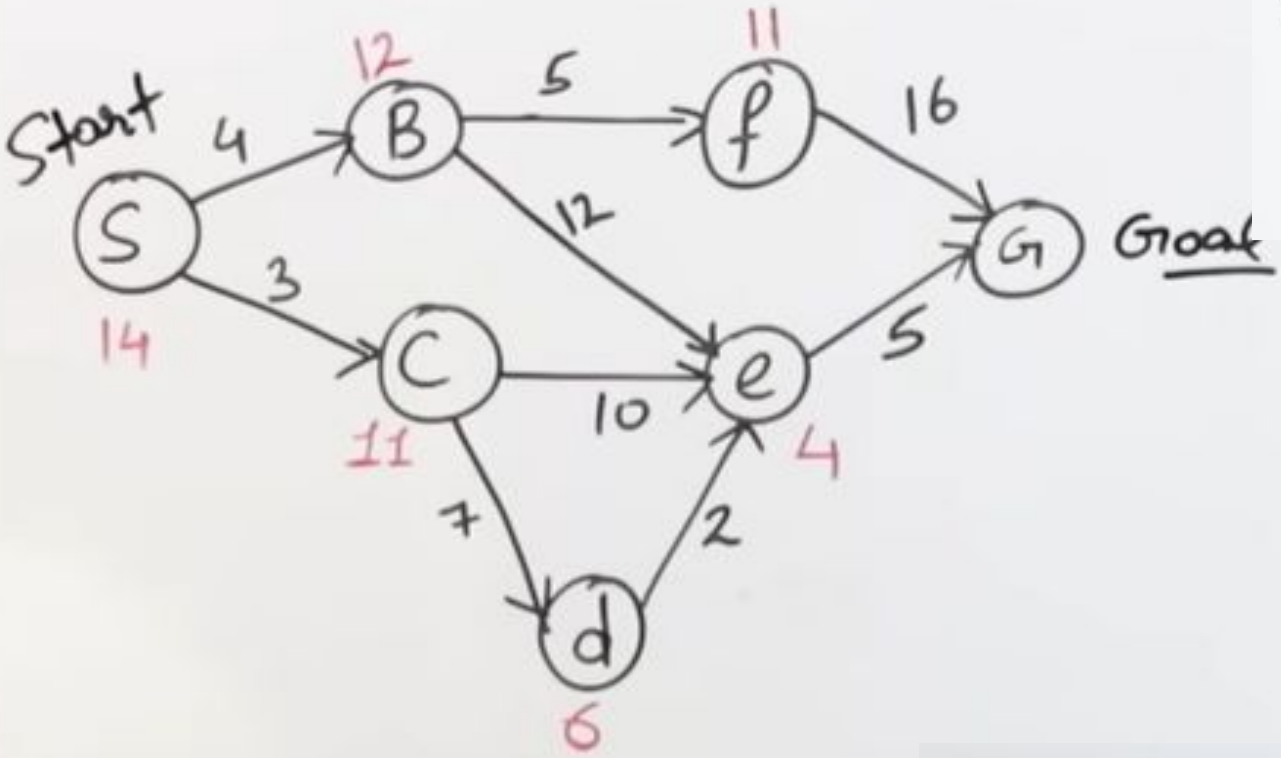
$SC \rightarrow d$: $3 + 7 + 6 = (16)$ ✓

$SC\,d \rightarrow e$: $3 + 7 + 2 + 4 = (16)$

A* algorithm → Informed Searching

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

- $g(N)$: Actual Cost from Start node to n
- $h(N)$: Estimation Cost from n to Goal node



SC → d
3+7+6 = (16) ✓

SC d → e
3+7+2+4 = (16) ✓

f(S) = 0+14 = (14)

S → B
4+12
= (16)

S → C
3+11 = (14)

SC → e
3+10+4
= 17 ✗

SC → d
3+7+6 = (16) ✓

Scde → G
12+5+0
= (17)

SB → f 5+4+11 = (20)

SB → e 4+12+4 = (20)

Properties of A*

- Complete? Yes

  (unless there are infinitely many nodes with $f \leq f(G)$;

  can't happen if step-cost $\geq \varepsilon > 0$)

- Time/Space? Exponential $O(b^d)$

  except if: $|h(n) - h^*(n)| \leq O(\log h^*(n))$

- Optimal? Yes

  (with: Tree-Search, admissible heuristic;

  Graph-Search, consistent heuristic)

- *Optimally Efficient?* Yes

  (no optimal algorithm with same heuristic is guaranteed to expand fewer nodes)

Local Search Algorithms

Example

Local Search

- The *uninformed and informed search algorithms* that we have seen are designed to explore search spaces systematically.
  - They keep one or more paths in memory and by record which alternatives have been explored at each point along the path.
  - When a goal is found, *the path to that goal also constitutes a solution to the problem*.
  - In many problems, however, the path to the goal is irrelevant. (8 QUEEN)

- If ***the path to the goal does not matter***, we might consider a different class of algorithms that do not worry about paths at all.

  local search algorithms

Local Search

- **Local search algorithms** operate using a *single current node* and generally move only to neighbors of that node.
- **Local search algorithms** ease up on ***completeness and optimality*** in the interest of ***improving time and space complexity***
- Although **local search algorithms** are not ***systematic***, they have ***two key advantages***:
  1. They use ***very little memory*** (usually a constant amount), and
  2. They can often find ***reasonable solutions*** in large or infinite (continuous) state spaces.
- In addition to finding goals, **local search algorithms** are useful for solving pure **optimization problems**, in which the aim is ***to find the best state*** according to an ***objective function***.
  - In optimization problems, ***the path to goal is irrelevant*** and ***the goal state itself is the solution.***
  - In some optimization problems, the ***goal is not known*** and ***the aim is to find the best state***.

Hill Climbing Search

- It is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem.
- It terminates when it reaches a peak value where no neighbor has a higher value.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Hill Climbing Search

- At each iteration, the **hill-climbing search algorithm** moves to the ***best successor*** of the ***current node*** according to an ***objective function.***
  - Best successor is the successor with best value (highest or lowest) according to an objective function.
  - If no successors have better value than the current value, it returns.
  - It moves in direction of uphill (hill climbing).
  - It terminates when it reaches a "peak" where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- **Hill climbing** does not look ahead beyond the immediate neighbors of the current state.
- **Hill climbing** is sometimes called ***greedy local search*** because it grabs a good neighbor state without thinking ahead about where to go next.
  - Greedy algorithms often perform quite well and
  - Hill climbing often makes rapid progress toward a solution.

Algorithm for Simple Hill Climbing

- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:

  If it is goal state, then return success and quit.

  Else if it is better than the current state then assign new state as a current state.

  Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

Hill Climbing Example

8-puzzle

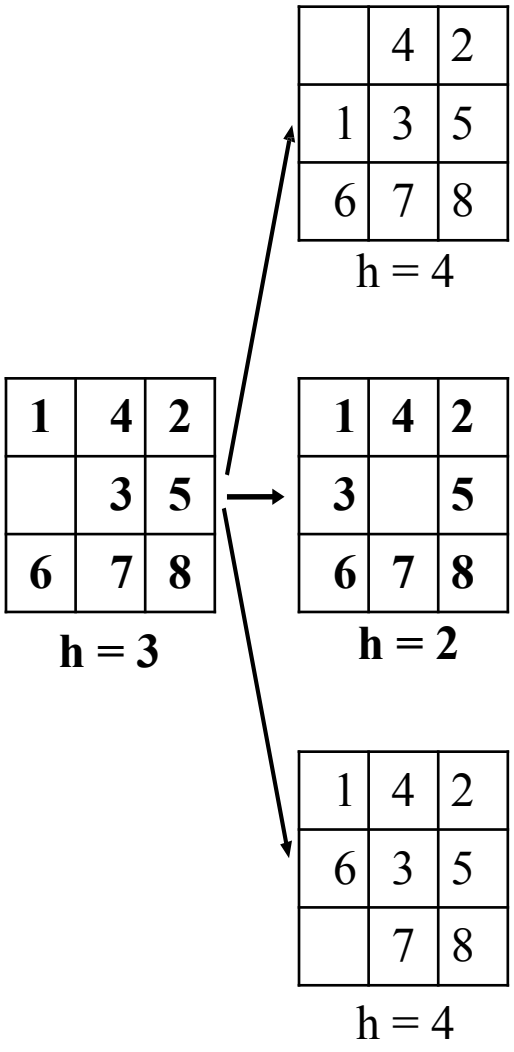
Heuristic function is Manhattan Distance

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

h = 3

Hill Climbing Example

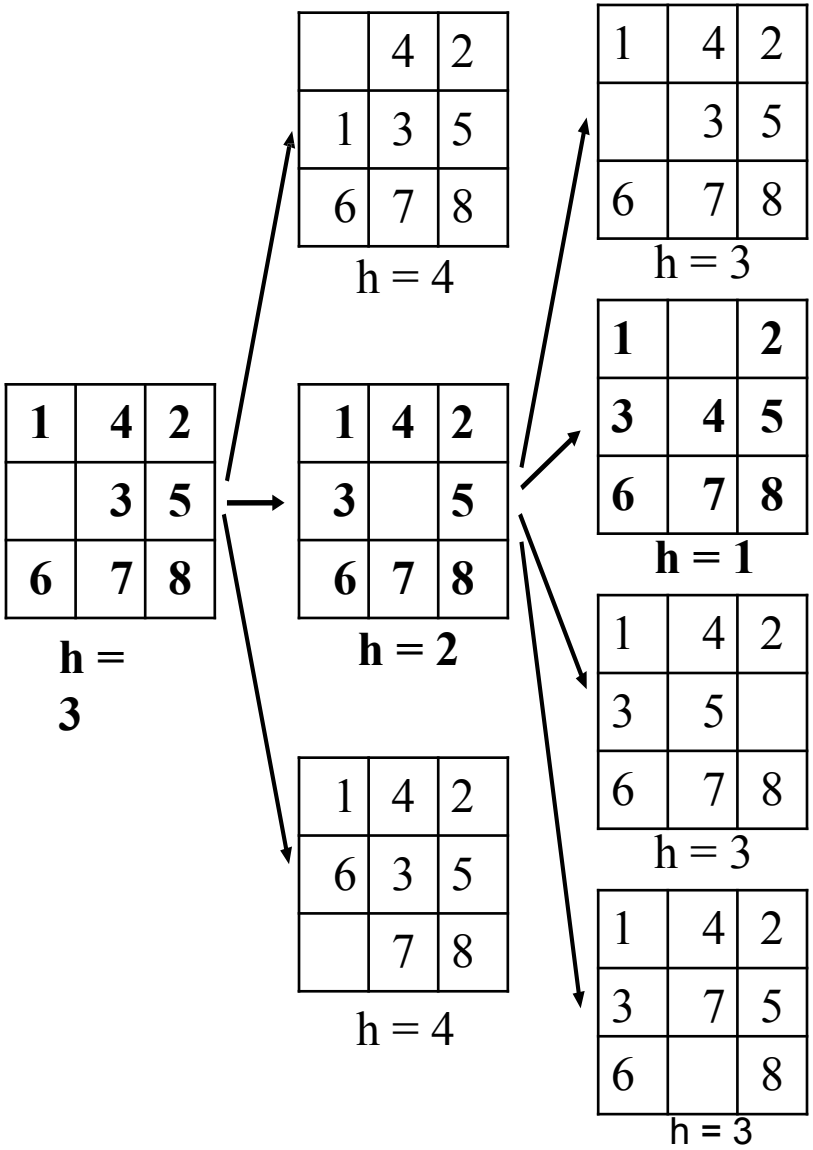
8-puzzle



Heuristic function is Manhattan Distance

Hill Climbing Example

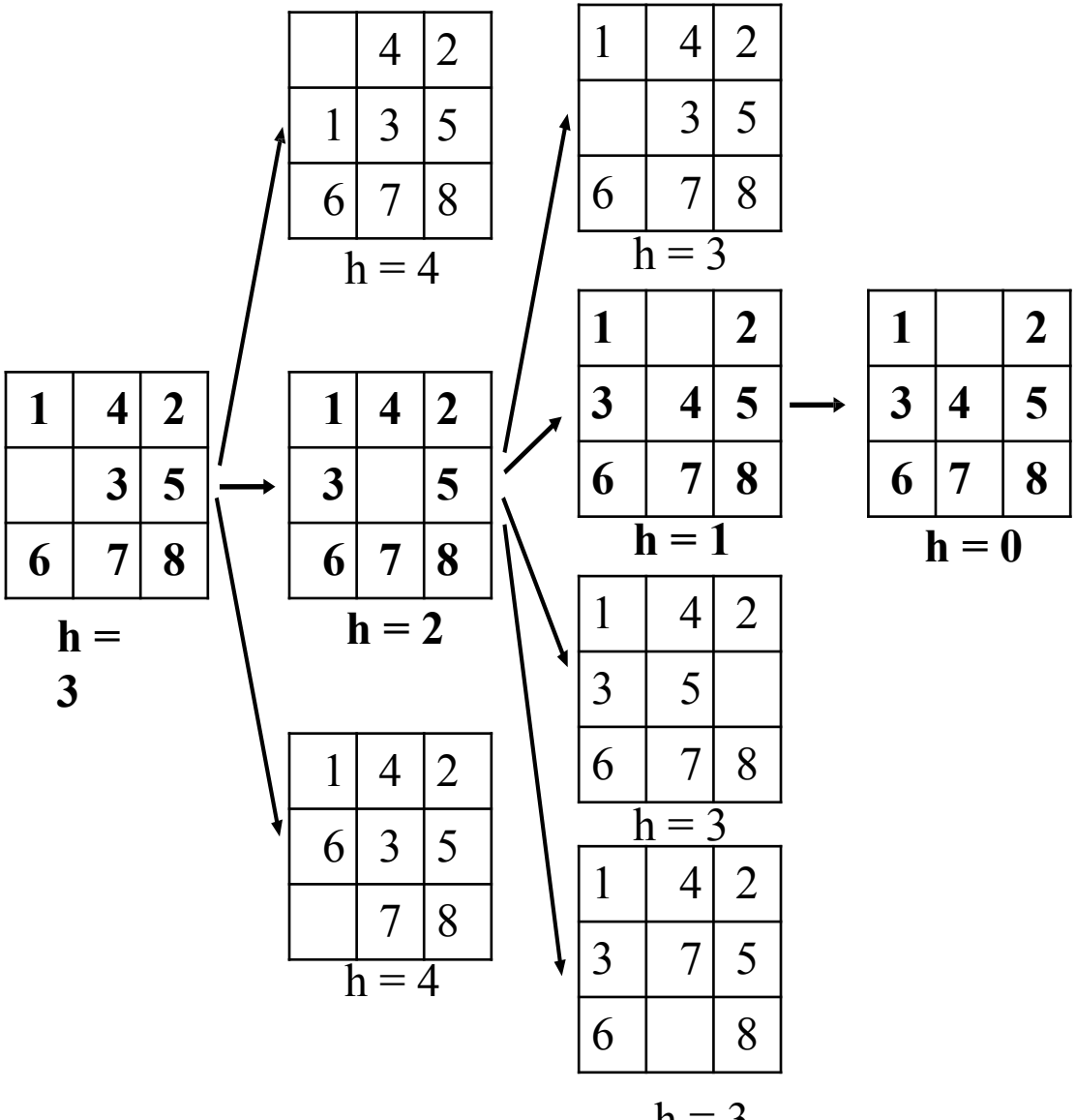
8-puzzle



Heuristic function is Manhattan Distance

Hill Climbing Example

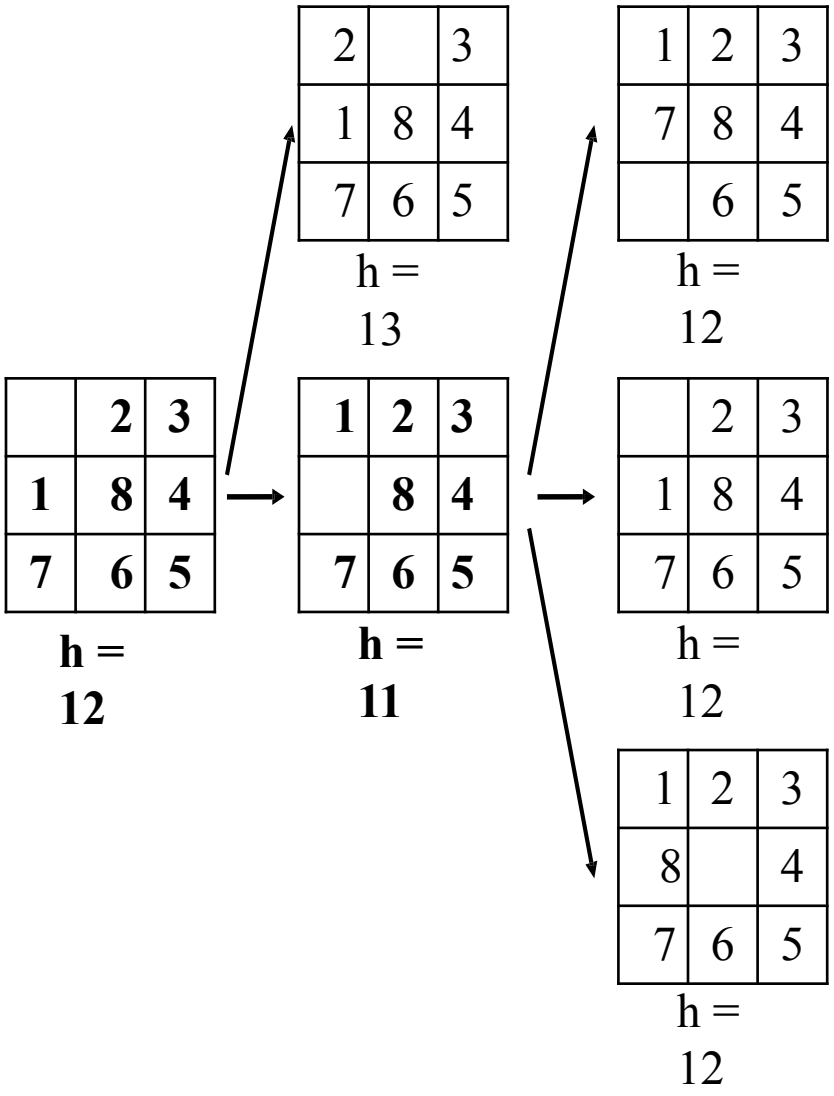
8-puzzle: a solution case



Heuristic function is Manhattan Distance

Hill Climbing Example

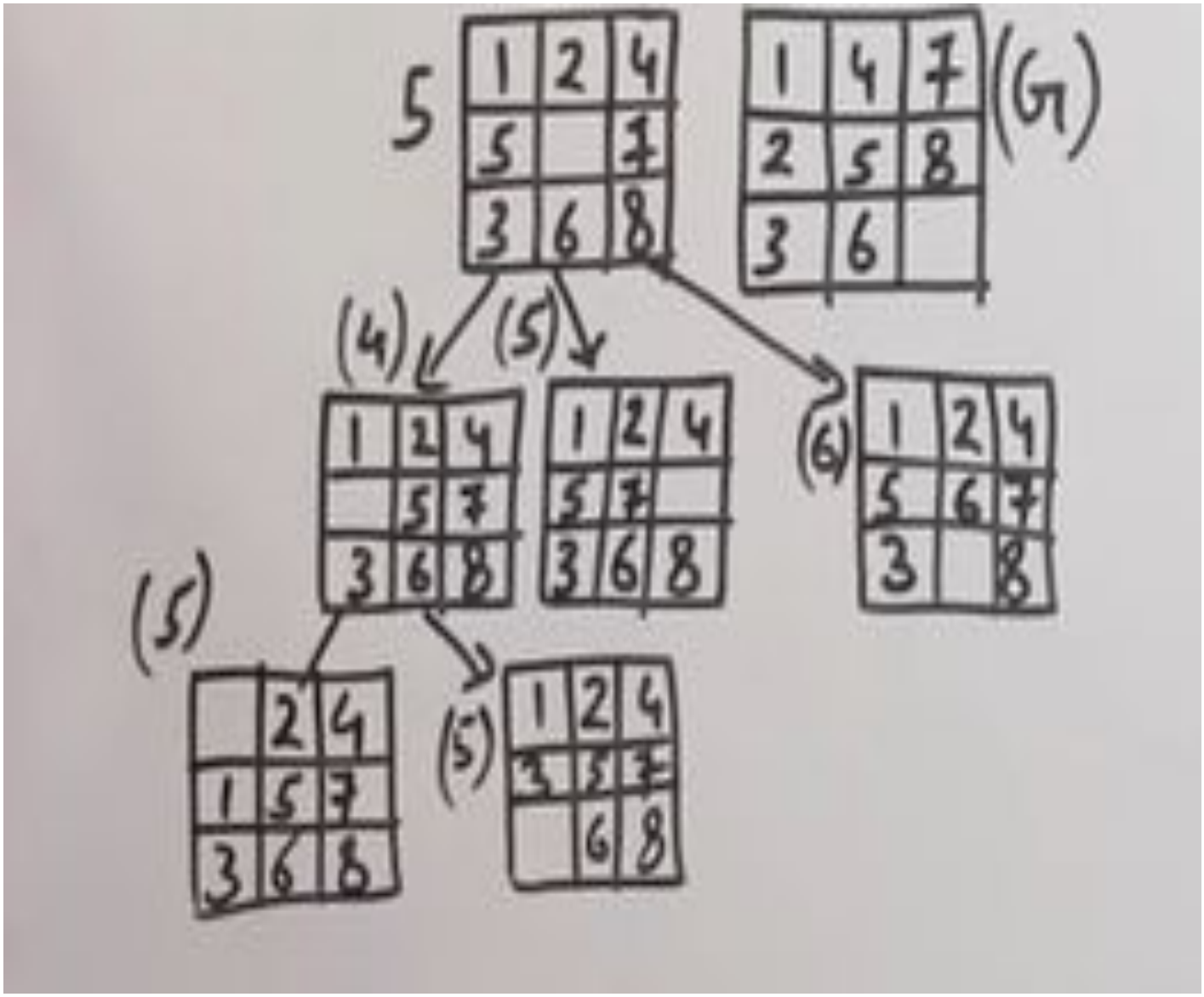
8-puzzle: stuck at local maximum



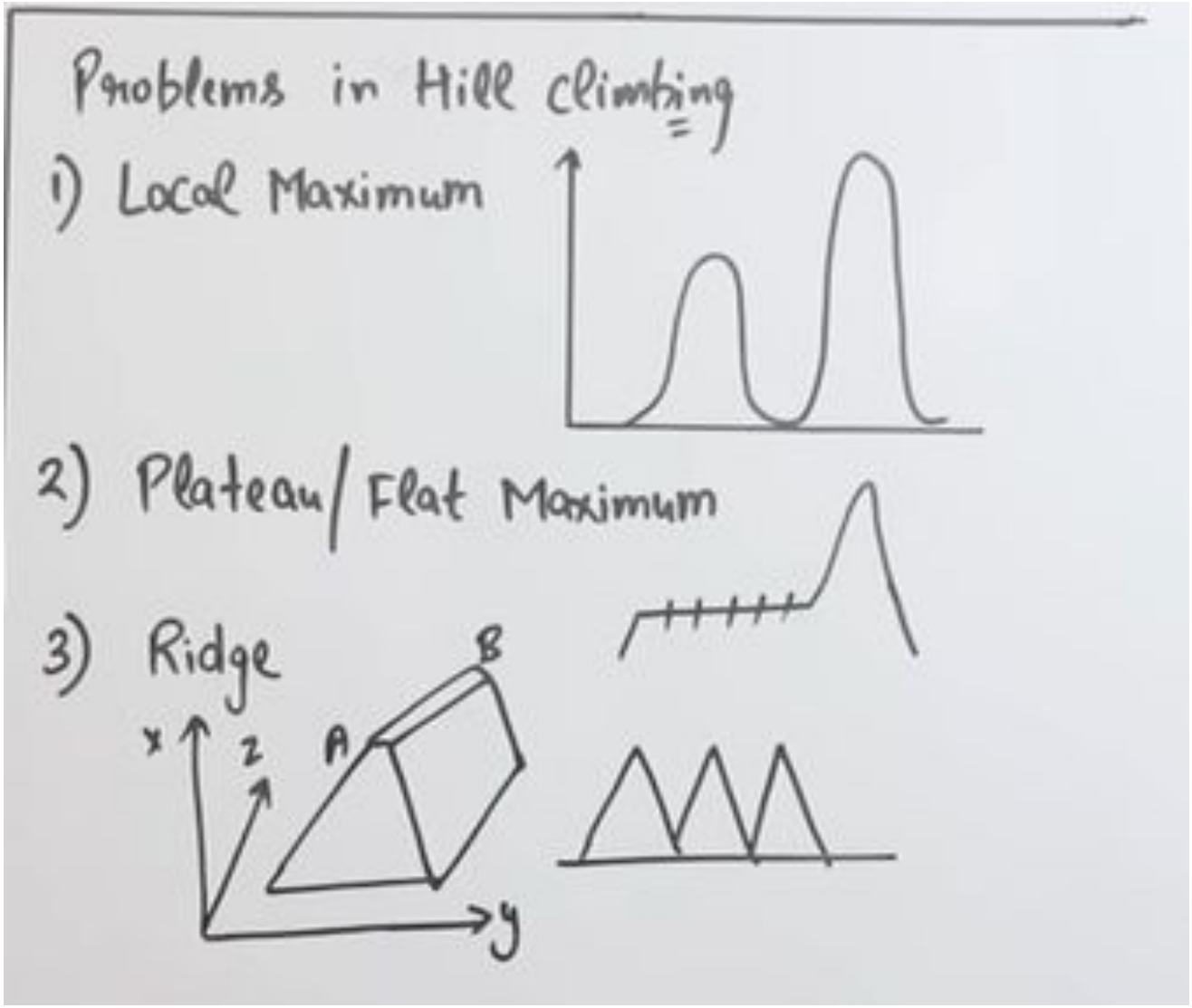
Heuristic function is Manhattan Distance

We are stuck with a local maximum.

Hill Climbing Search



Hill Climbing Search

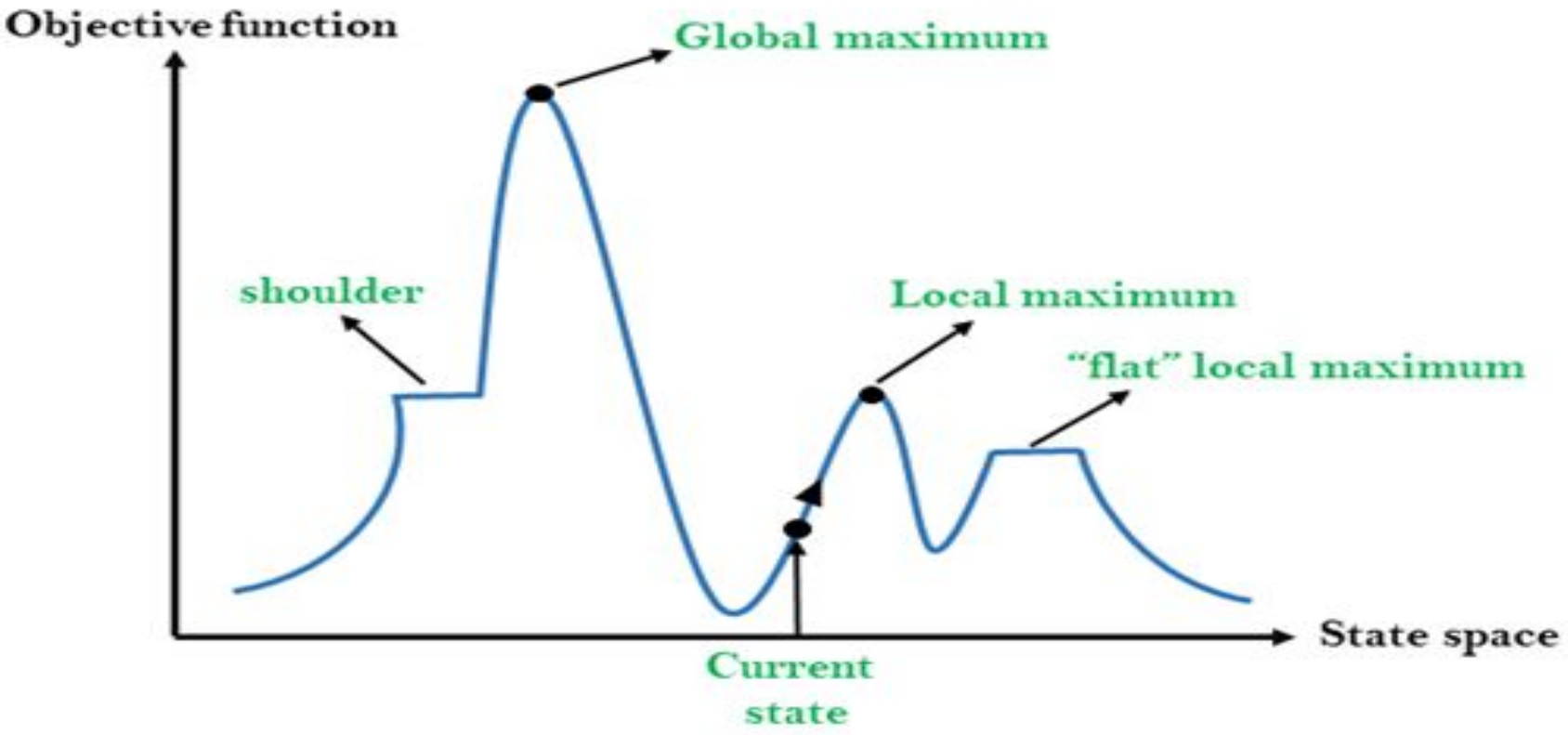


Features of Hill Climbing:

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.
- **Deterministic Nature:**Hill Climbing is a deterministic optimization algorithm, which means that given the same initial conditions and the same problem, it will always produce the same result. There is no randomness or uncertainty in its operation.
- **Local Neighborhood:**Hill Climbing is a technique that operates within a small area around the current solution. It explores solutions that are closely related to the current state by making small, gradual changes. This approach allows it to find a solution that is better than the current one although it may not be the global optimum.

State-space Diagram for Hill Climbing:

- **Y-axis** we have taken the function which can be an **objective function** or **cost function**, and **state-space** on the **x-axis**.
- If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



Hill Climbing Search

(Steepest Ascent/Descent)

```
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
```

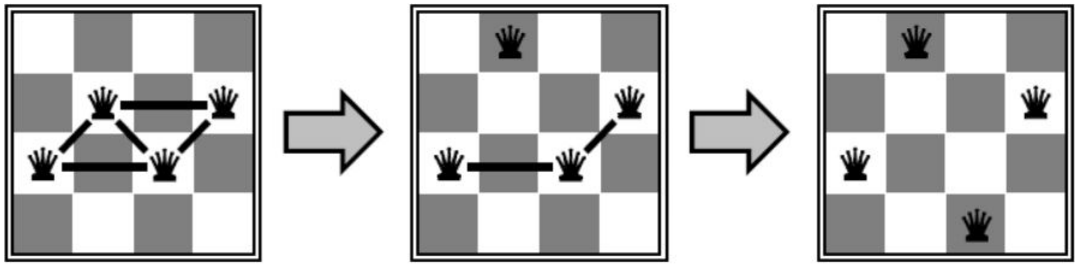
best successor

Hill Climbing Example

n-queens

- Put n queens on an n × n board with no two queens on the same row, column, or diagonal
- Move a queen to reduce number of conflicts.

  Objective function: number of conflicts (no conflicts is global minimum)

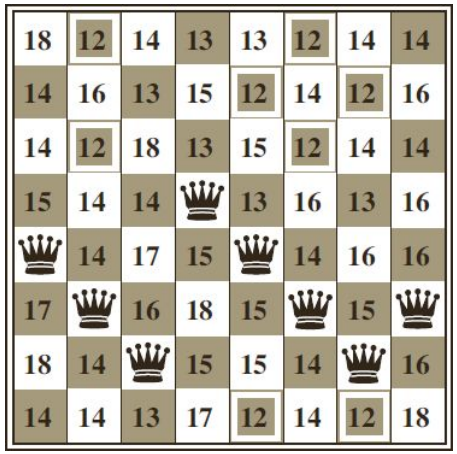


- The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has n*(n-1) successors).
- The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly.
- The global minimum of this function is zero, which occurs only at perfect solutions.

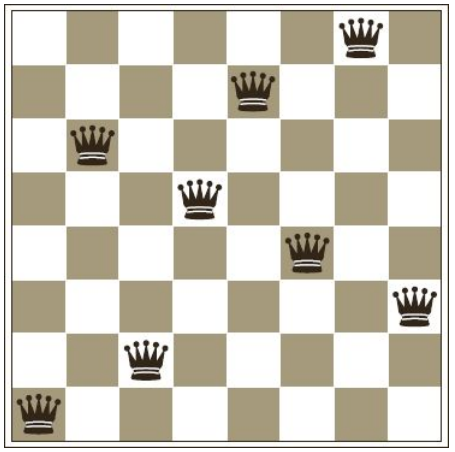
Hill Climbing Example

n-queens

- Hill Climbing may NOT reach to a goal state for n-queens problem.



 five steps to reach this state



- An 8-queens state with heuristic cost estimate h=17
- The value of h for each possible successor obtained by moving a queen within its column.
- The best moves are marked with value 12.

- A local minimum in the 8-queens state space; the state has h=1
- but every successor has a higher cost.
- Hill Climbing will stuck here

Hill Climbing Example

n-queens

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

- The Hill Climbing algorithm halts if it reaches a **plateau.**
  - One possible solution is to allow **sideways move** in the hope that the **plateau** is really a **shoulder**.
  - If we always allow sideways moves when there are no uphill moves, an infinite loop will occur whenever the algorithm reaches a flat local maximum that is not a shoulder.
  - One common solution is to put a limit on the number of consecutive sideways moves allowed.
  - For example, we could allow up to 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill climbing from 14% to 94%.

Hill Climbing

- Hill Climbing is **NOT complete**.
- Hill Climbing is **NOT optimal**.

- **Why use local search?**
  - Low memory requirements – usually constant
  - Effective – Can often find good solutions in extremely large state spaces
  - Randomized variants of hill climbing can solve many of the drawbacks in practice.

- Many variants of hill climbing have been invented.

Stochastic Hill Climbing

having a random probability distribution or pattern that may be analyzed statistically but may not be predicted precisely.

- **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**.
- For 8-queens instances with no sideways moves allowed, $p \approx 0.14$, so we need roughly 7 iterations to find a goal (6 failures and 1 success).
  - For 8-queens, then, random-restart hill climbing is very effective indeed.
  - Even for three million queens, the approach can find solutions in under a minute.
- 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.
  - NP-hard problems typically have an ***exponential number of local maxima*** to get stuck on.

Simulated Annealing

- A **hill-climbing algorithm** that never makes "downhill" moves toward states with lower value (or higher cost) is guaranteed to be **incomplete**, because ***it can get stuck on a local maximum***.
  - In contrast, a **purely random walk**—that is, moving to a successor chosen uniformly at random from the set of successors—is **complete but extremely inefficient**.
  - Therefore, it seems reasonable to ***combine hill climbing with a random walk in some way that yields both efficiency and completeness***.
- **Idea:** ***escape local maxima by allowing some "bad" moves but gradually decrease their size and frequency.***
- The **simulated annealing algorithm**, a version of ***stochastic hill climbing*** where some downhill moves are allowed.
- **Annealing:** the process of gradually cooling metal to allow it to form stronger crystalline structures

Simulated Annealing

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE – current.VALUE
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

- **Downhill moves** are accepted readily early in the annealing schedule and then less often as time goes on.

- The **schedule** input determines the value of the temperature T as a function of time.

Simulated Annealing

- Instead of picking the best move, Simulated Annealing picks a random move.
  - If the move improves the situation, it is always accepted.
  - Otherwise, the algorithm accepts the move with some probability less than 1.
- The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened.
- The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.
- **If the schedule lowers T slowly enough, the algorithm will find a best state with probability approaching 1.**

- Simulated Annealing is widely used in VLSI layout and airline

Local Beam Search

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

- The **local beam search algorithm** is **not** the same as ***k searches run in parallel***!
  - In a **local beam search**, searches that find good states recruit other searches to join them.
  - In a random-restart search, each search process runs independently of the others.

Stochastic Beam Search

- **Problem:** quite often, all k states end up on same local hill in ***local beam search algorithm***
- **Idea:** choose k successors randomly, biased towards good ones

  Stochastic Beam Search

- Instead of choosing the best k from the pool of candidate successors, **stochastic beam search** chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.

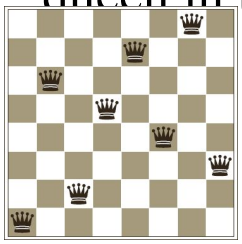
- **Stochastic beam search** bears some resemblance to the process of natural selection, whereby the "successors" (offspring) of a "state" (organism) populate the next generation according to its "value" (fitness).

Genetic Algorithms

- A **genetic algorithm (GA)** is a variant of ***stochastic beam search*** in which successor states are generated by combining two parent states rather than by modifying a single state.
- Like beam searches, **GAs** begin with a set of ***k randomly generated states***, called the **population**.
- Each state, or **individual**, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s.
  - An 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so it requires $8 \times \log_2 8 = 24$ bits.
  - Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8.
- Each state is rated by an ***objective function***, or (in GA terminology) the **fitness function**.
- Pairs of individuals are randomly selected for ***reproduction***.
- From each pair new offsprings (children) are generated using **crossover** operation.

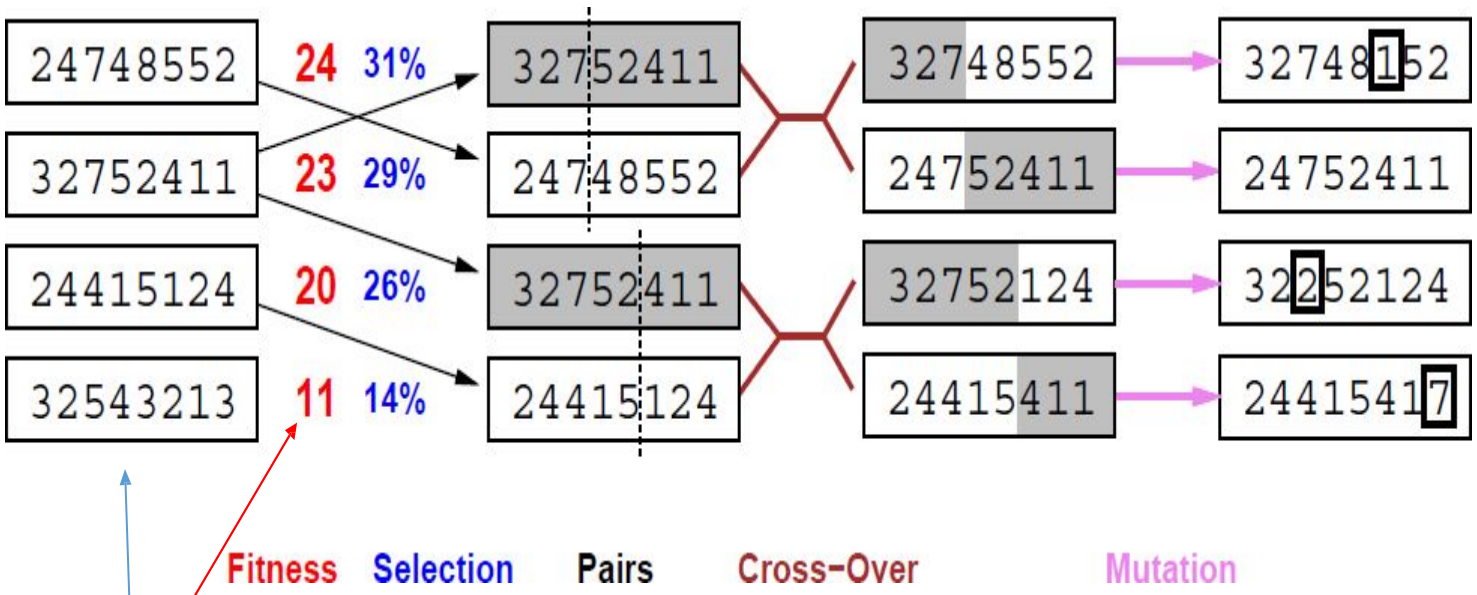
Genetic Algorithms

- A state can be represented using a 8 digit string.
- Each digit in the range from 1 to 8 to indicate the position of the queen in that column.



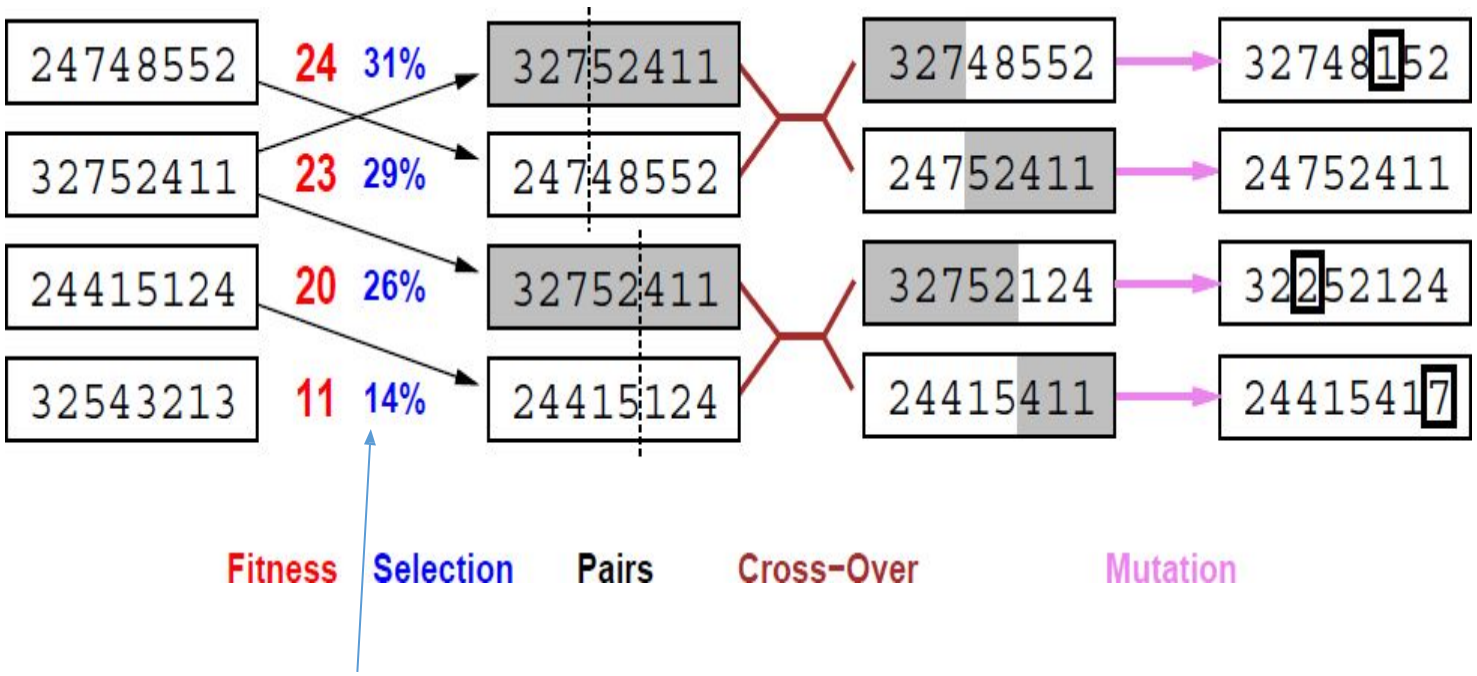
 16257483

Genetic Algorithms



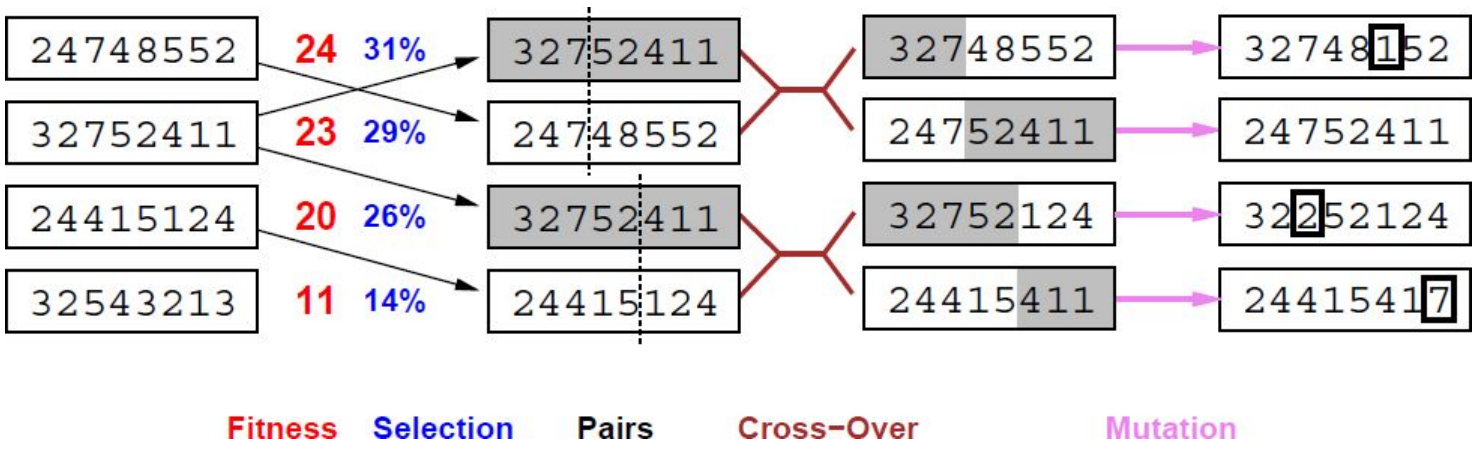
- The **initial population** has 4 states.
- A **fitness function** should return higher values for better states, so, for the 8-queens problem we use
  the number of non-attacking pairs of queens, which has a value of 28 for a solution.
  - The values of the four states are 24, 23, 20, and 11.

Genetic Algorithms



- The probability of being chosen for reproducing is directly proportional to the fitness score.
- Two pairs are selected at random for reproduction, in accordance with the probabilities.
  - Notice that one individual is selected twice and one not at all.

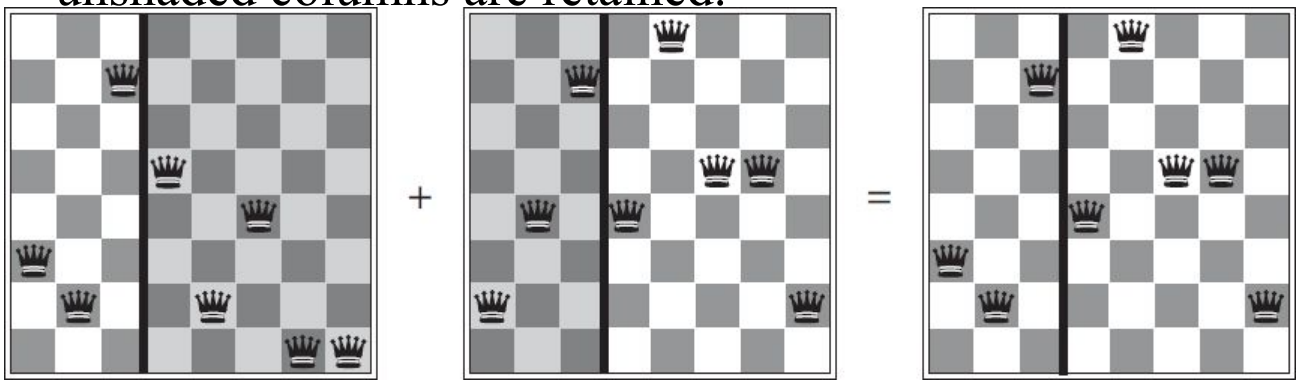
Genetic Algorithms



- The **crossover points** are after third digit in first pair and after fifth digit in second pair.
- The first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent.

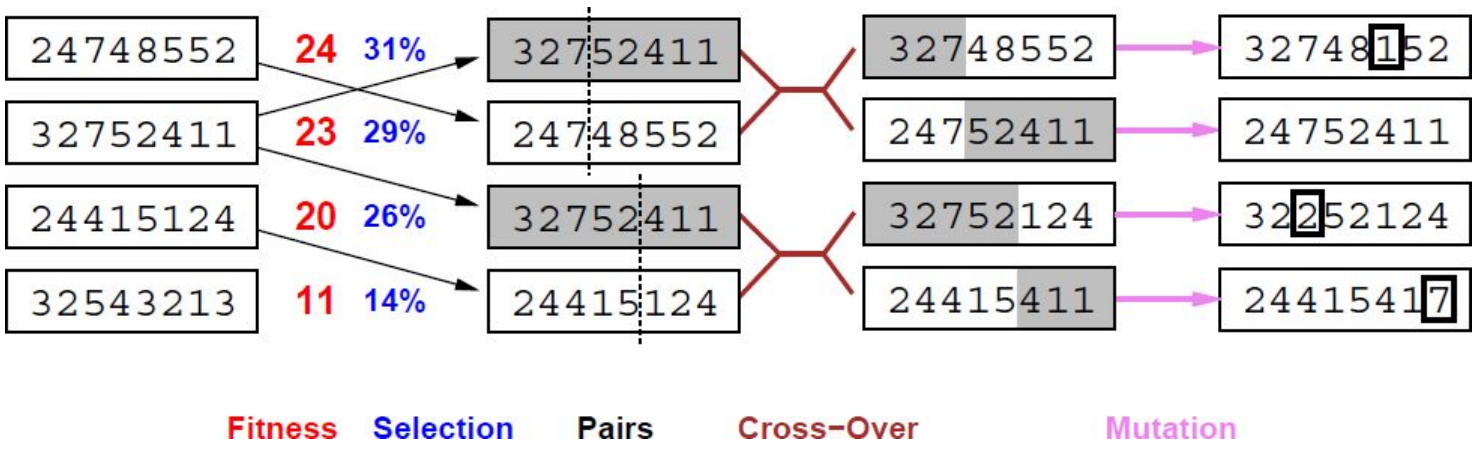
Genetic Algorithms

- Crossover helps iff substrings are meaningful components.
- The shaded columns are lost in the crossover step and the unshaded columns are retained.



32752411 + 24748552 = 32748552

Genetic Algorithms



- One digit was **mutated** in the first, third, and fourth offspring.
- In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column.

Genetic Algorithms

A genetic algorithm: *each mating of two parents produces only*

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
      new_population ← empty set
      for i = 1 to SIZE(population) do
          x ← RANDOM-SELECTION(population, FITNESS-FN)
          y ← RANDOM-SELECTION(population, FITNESS-FN)
          child ← REPRODUCE(x, y)
          if (small random probability) then child ← MUTATE(child)
          add child to new_population
      population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```