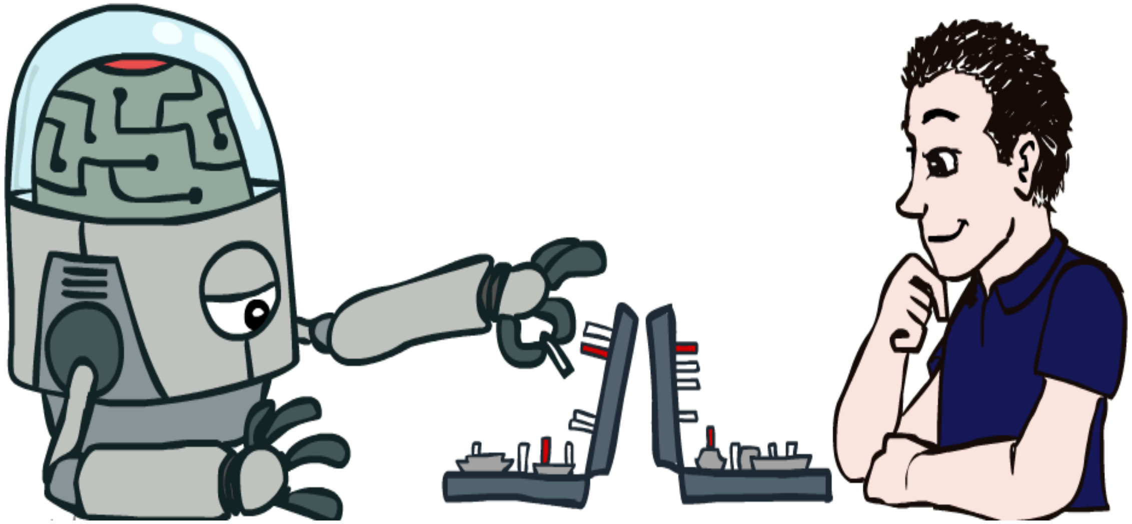
ARTIFICIAL INTELLIGENCE- CS411

Prof. Alaa Sagheer



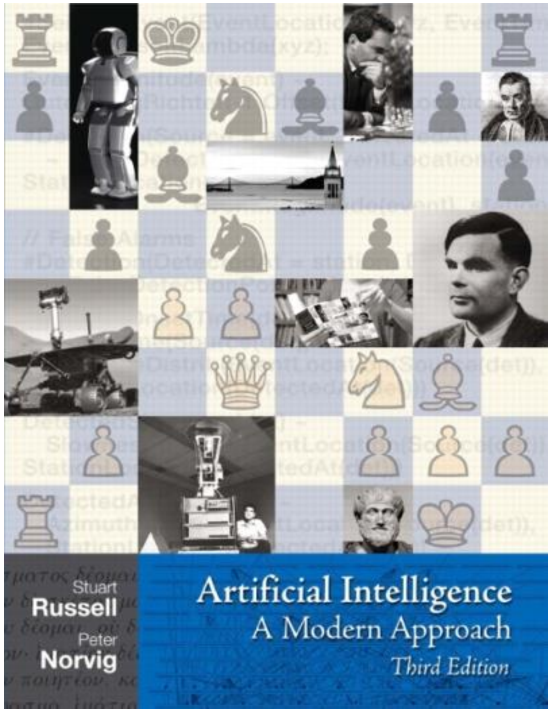
Artificial Intelligence "CS 411"

- **Textbook**:

S. Russell and P. Norvig

Artificial Intelligence: A Modern Approach

Prentice Hall, 2010, *Third Edition*



- **Place:** Online Lectures
- **Grading:**

Class Activity (5%),

Project @ Lab (10%),

Quizzes @ Class (10%),

Quizzes @ Lab (15 %),

Mid-term exam (20%),

Final exam (40%),

Chapter 4: Problem Solving

Informed Search

Informed Search

In which we see how information about the state space can prevent algorithms from blundering about in the dark

Chapter 3 showed that uninformed search strategies can find solutions to problems by systematically generating new states and testing them against the goal. Unfortunately, these strategies are incredibly inefficient in some cases.

Best-First Search

- A node is selected for expansion based on an evaluation function, $f(n)$
  - Traditionally, the node with the ***lowest*** evaluation is selected for expansion, because the evaluation measures distance to the goal.
- Implementation: Order the nodes in fringe in ascending order of f-values
- There is a whole family of BEST-FIRST-SEARCH algorithms with different evaluation functions. A key component of these algorithms is a <u>heuristic function $h(n)$:</u>

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

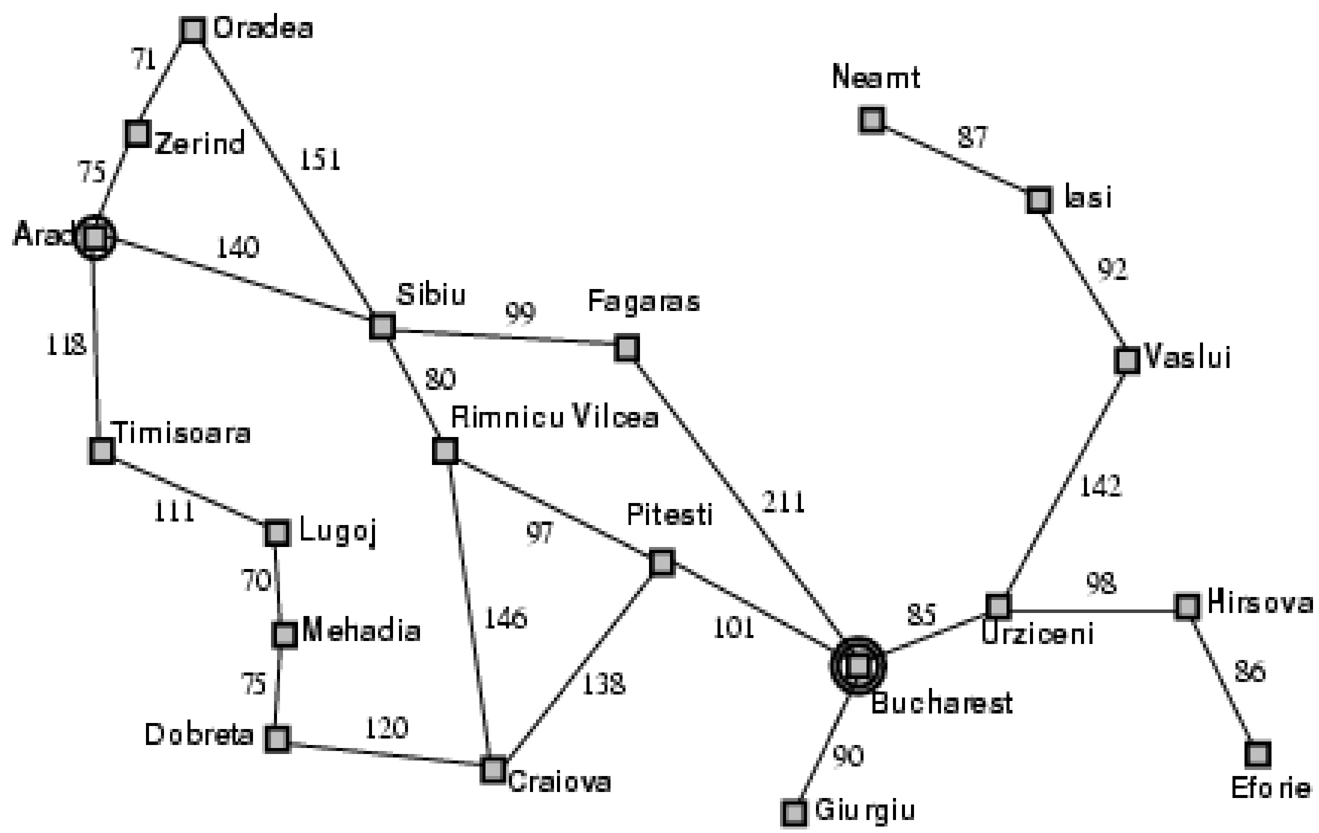
- By heuristic function, additional knowledge of the problem is imparted to the search algorithm.
- if n is a goal node, then $h(n) = 0$.

Greedy Best-First Search (1)

- It is a special case of search by heuristic. <u>It expands the node that is closest to the goal</u> (which leads to a solution very quickly). Evaluates nodes using the heuristic func.:

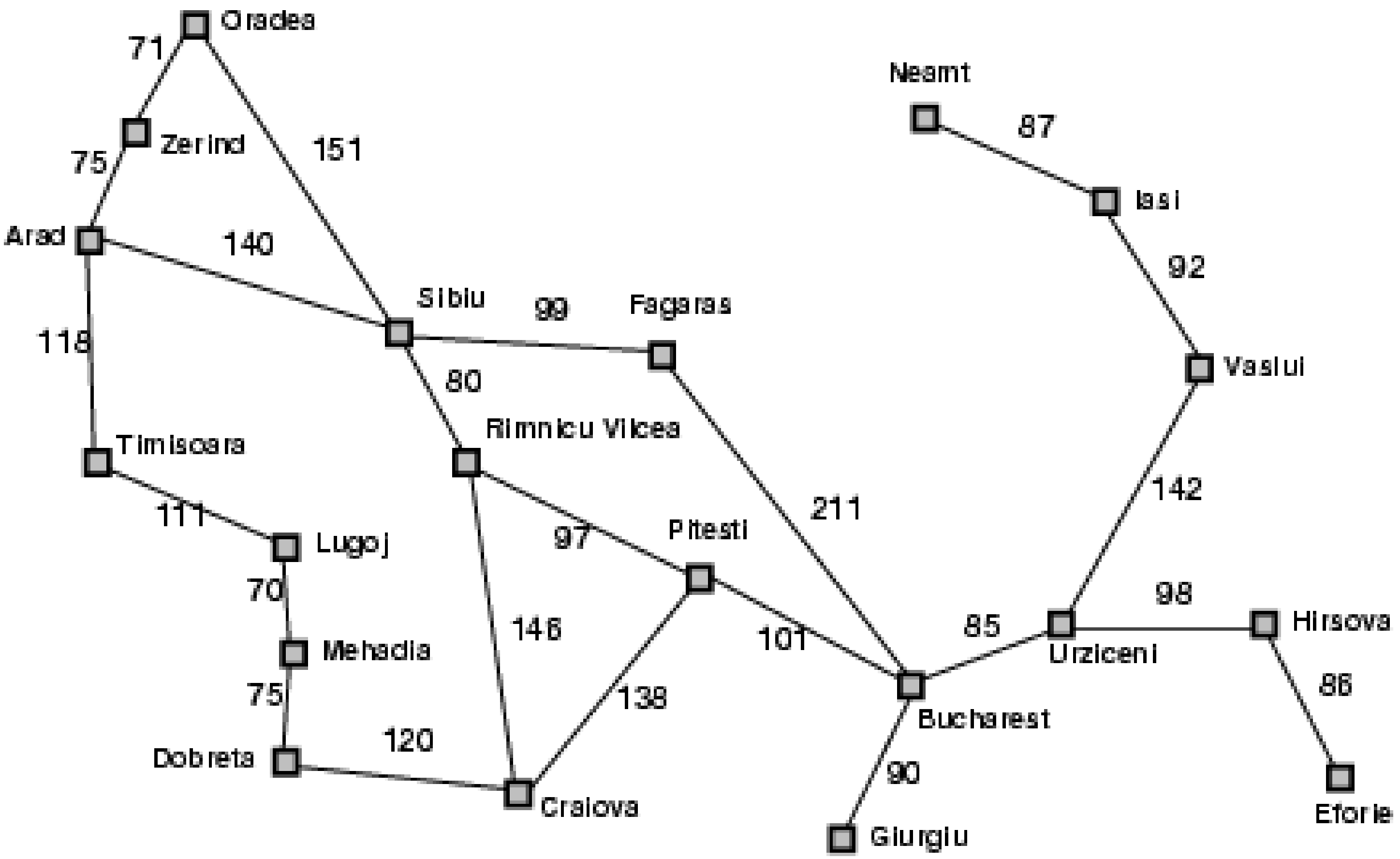
$$f(n) = h(n)$$

Estimate the cost of the cheapest path from Arad to Bucharest via the <u>straight-line distance</u> from Arad to Bucharest.



Greedy Best-First Search (2)

- Using the straight-line distance heuristic h_{SLD},



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

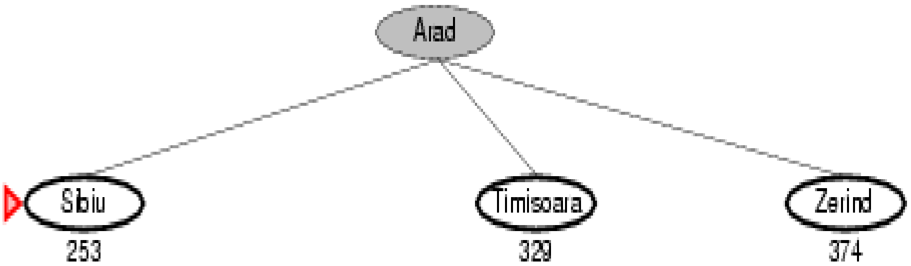
❖ The values of h_{SLD} cannot be computed from the problem description itself

Greedy Best-First Search (3)

- Greedy best-first search expands the node that ***appears*** to be closest to goal.

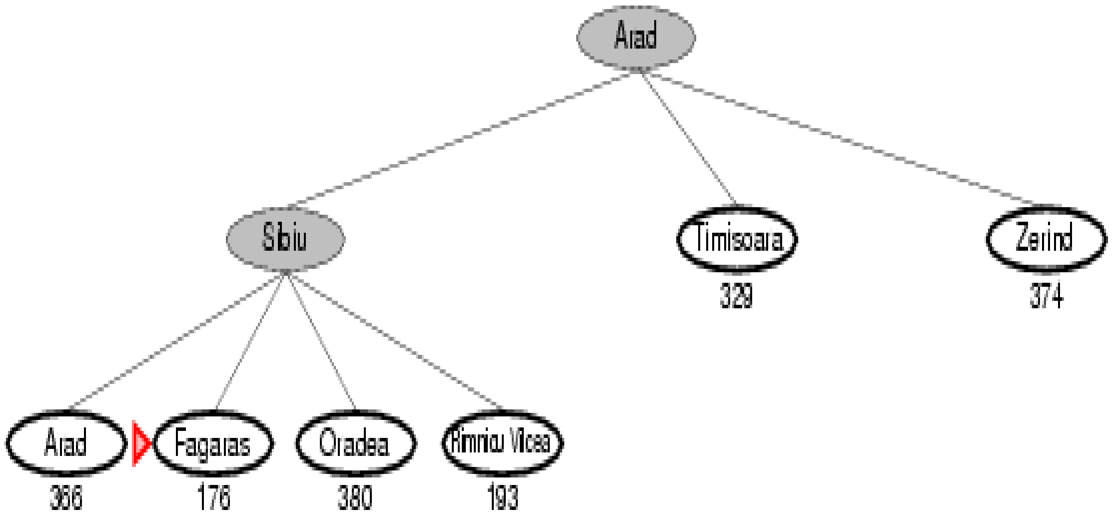
Greedy Best-First Search (4)

- Greedy best-first search expands the node that ***appears*** to be closest to goal.



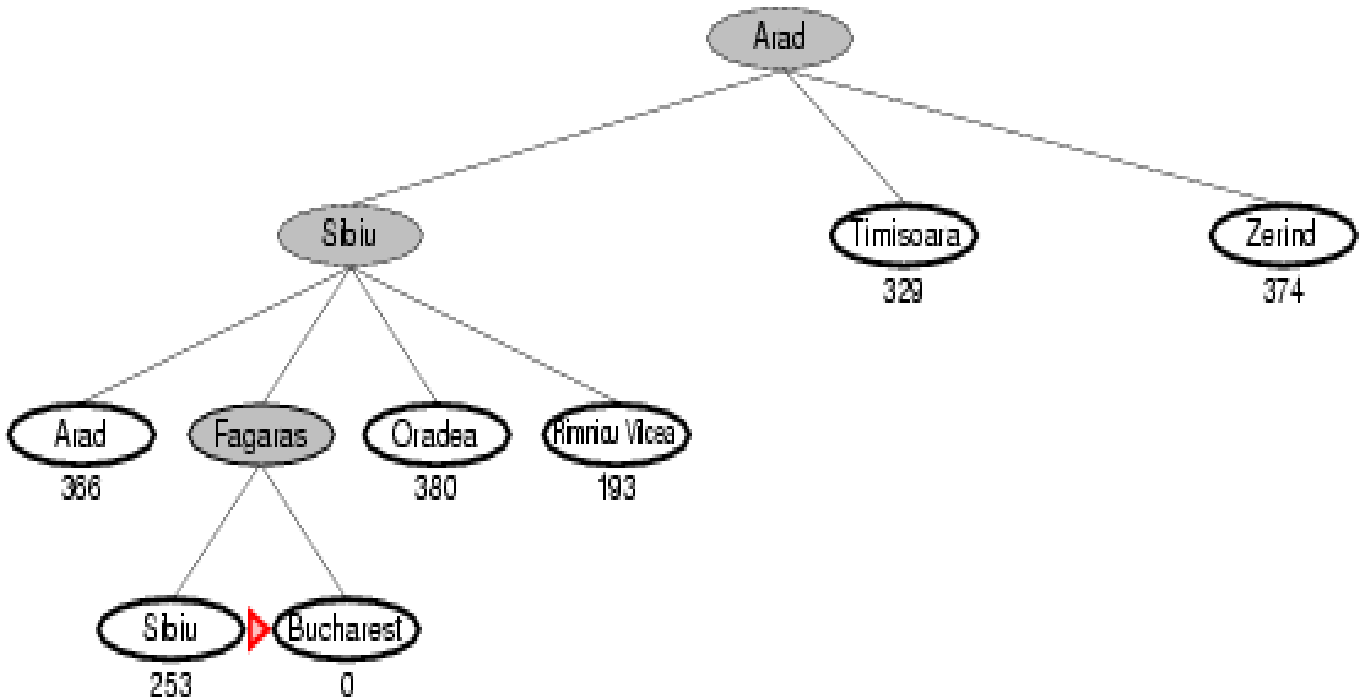
Greedy Best-First Search (5)

- Greedy best-first search expands the node that ***appears*** to be closest to goal.



Greedy Best-First Search (6)

- Greedy best-first search expands the node that ***appears*** to be closest to goal.



Greedy Best-First Search (7)

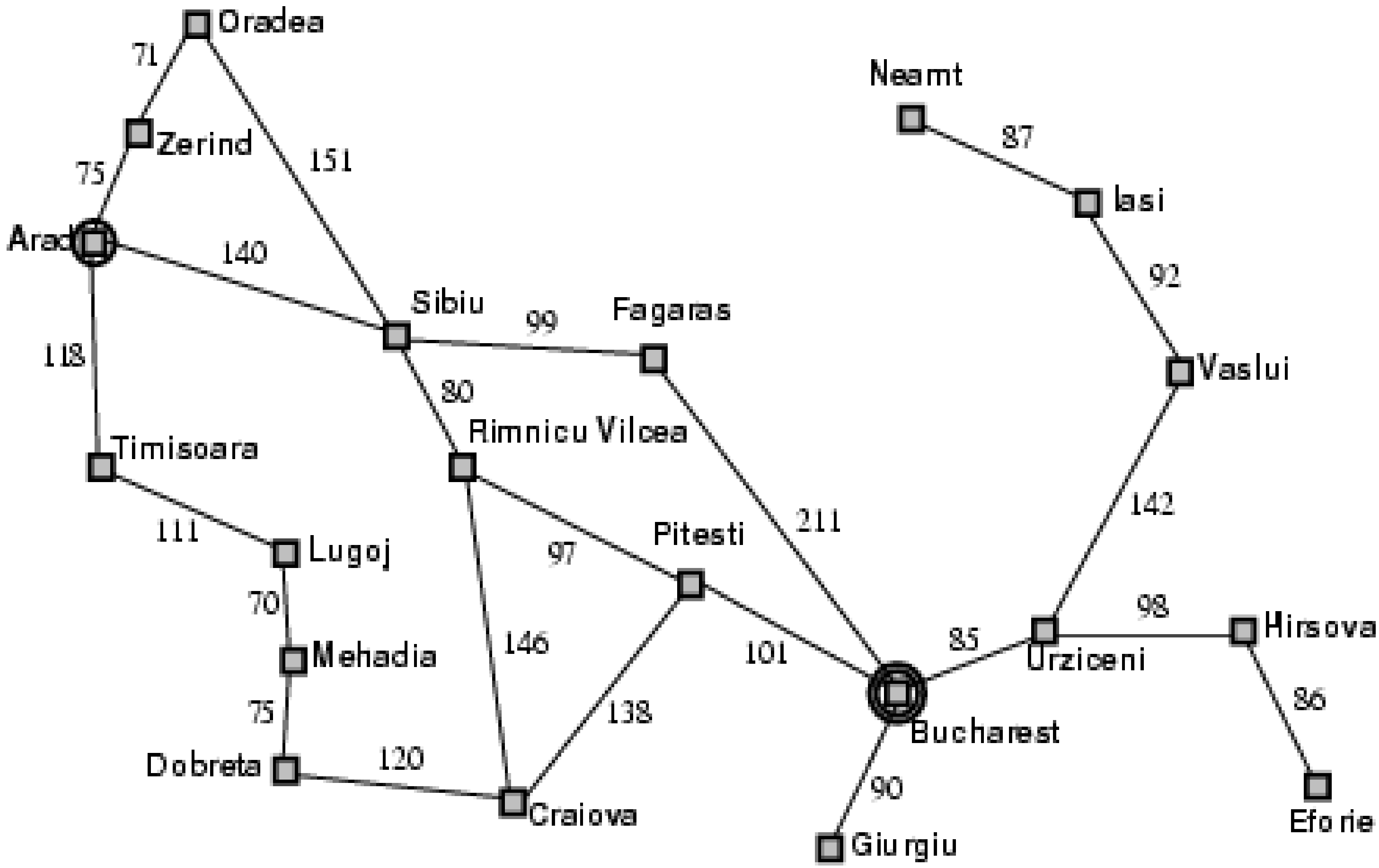
- The first node to be expanded from Arad will be Sibiu..Why?
- The next node to be expanded will be Fagaras..Why?
- Fagaras in turn generates Bucharest, which is the goal. (GooD!)

❖ Greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path. How?
 - Hence, its search cost is minimal.
 - HOWEVER, it is not optimum

❖ The path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.

(Go to the map)

Greedy Best-First Search



Greedy Best-First Search (7)

- The first node to be expanded from Arad will be Sibiu..Why?
- The next node to be expanded will be Fagaras..Why?
- Fagaras in turn generates Bucharest, which is the goal. (GooD!)

❖ Greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path. How?
- Hence, its search cost is minimal.
- HOWEVER, it is not optimum

❖ The path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.

This shows why the algorithm is called "greedy'- at each step it tries to get as close to the goal as it can.

Properties of Greedy Best-First Search

- **Complete?** It depends on start node!

  - Susceptible to false start *

  - if it is not careful to detect repeated states, it can get stuck in oscillated loops, e.g., Iasi → Neamt → Iasi → Neamt →

  - it can start down an infinite path and never return to try other possibilities

- **Optimal?** No!

- **Time?**

  $O(b^m)$, but a good heuristic can give dramatic improvement

- **Space?**

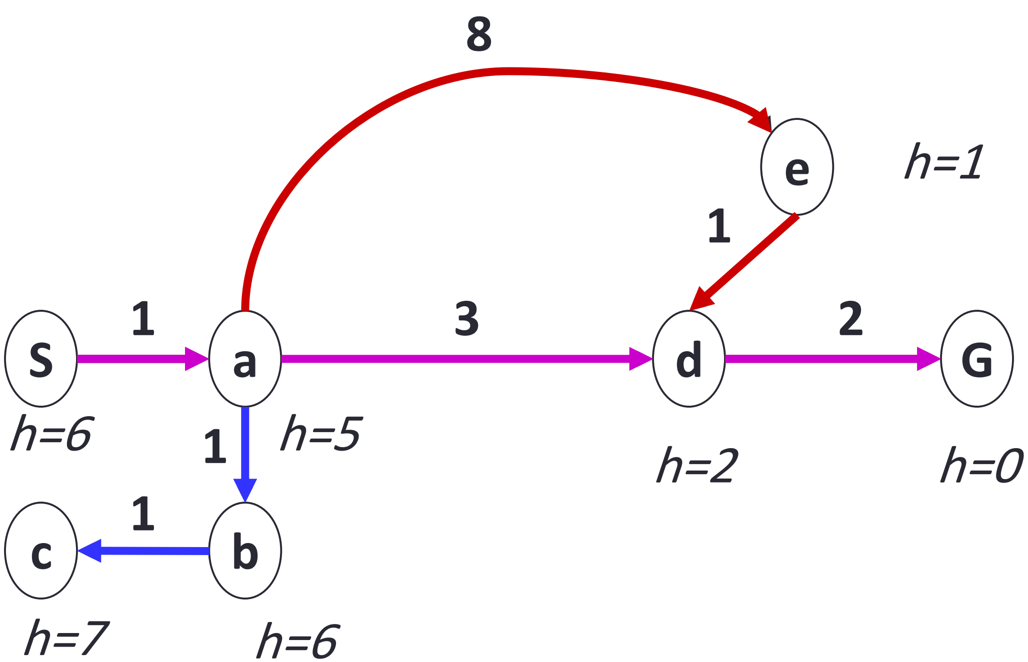
  $O(b^m)$ -- keeps all nodes in memory

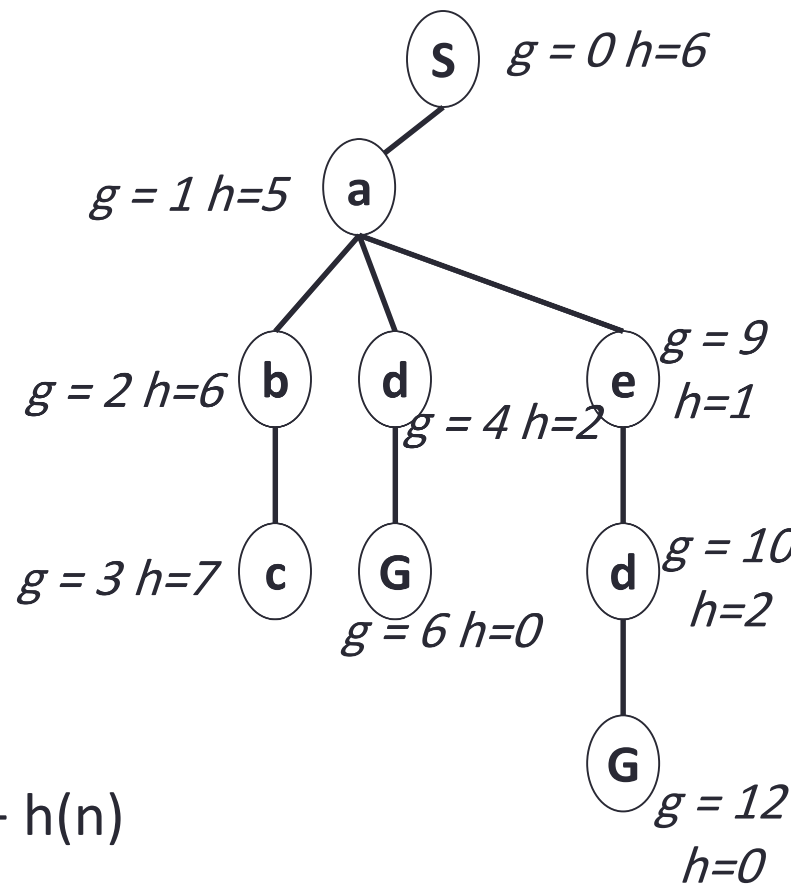
  b : The branching factor (or maximum number of successors of any node)

  m : The maximum depth of the search space.

Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)





- A* Search orders by the sum: $f(n) = g(n) + h(n)$

A*

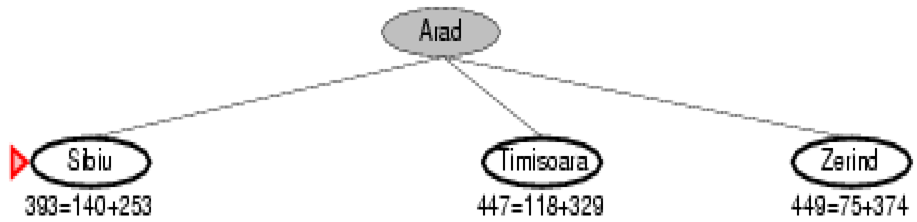
- Idea: avoid expanding paths that are already expensive.
- It evaluates nodes by combining *g(n),* the cost to reach the node, and *h(n),* the cost to get from the node to the goal.

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

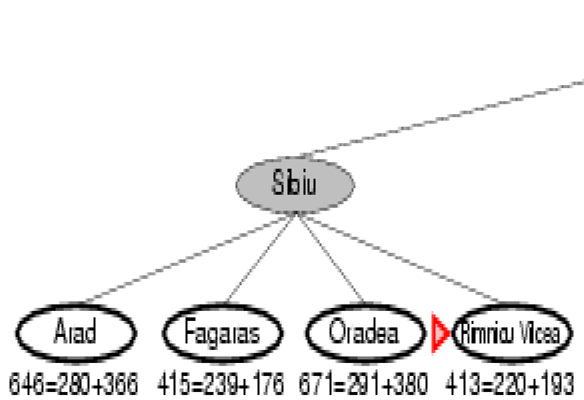
- *g(n)* = the past cost to reach *n* from start node,
- *h(n)* = the estimated cost of the cheapest path from *n* to goal,
- *f(n)* = the estimated total cost of cheapest <u>solution</u> to goal through *n.*
- **To find the cheapest solution, a reasonable thing to try first is the node with the lowest value of *g(n)+h(n)***

A*

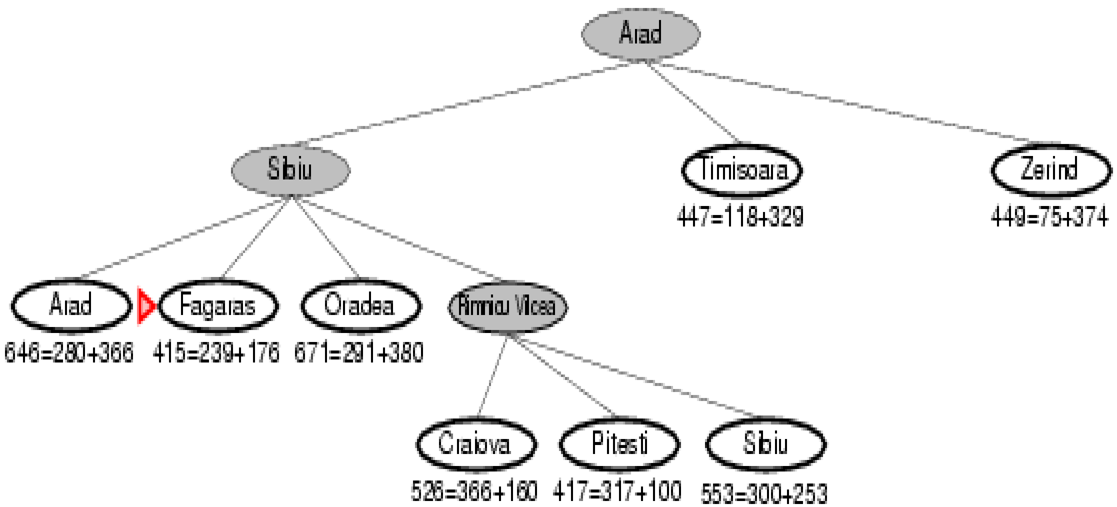
A*


Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

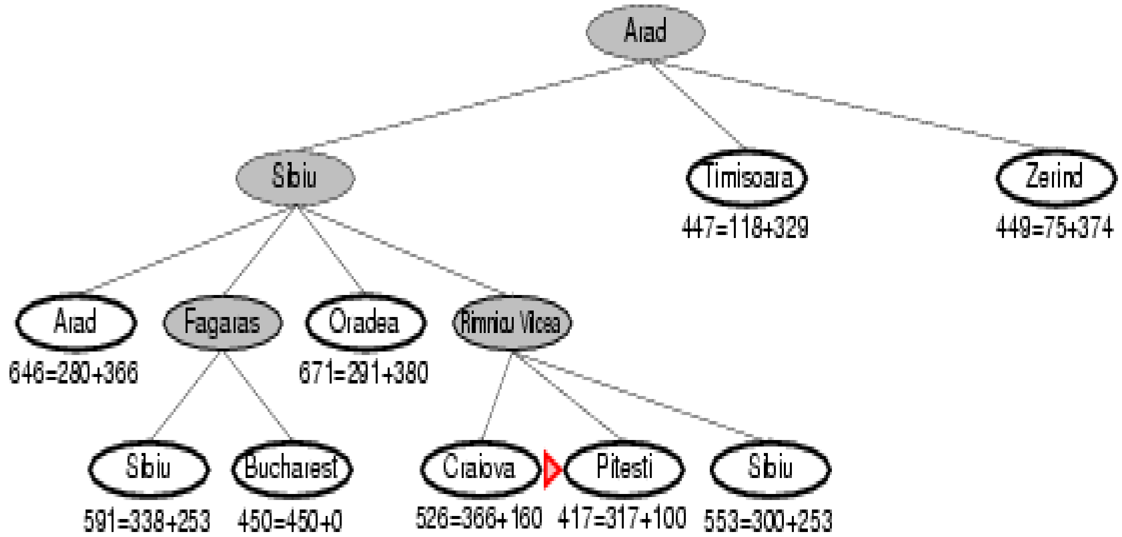
A*


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

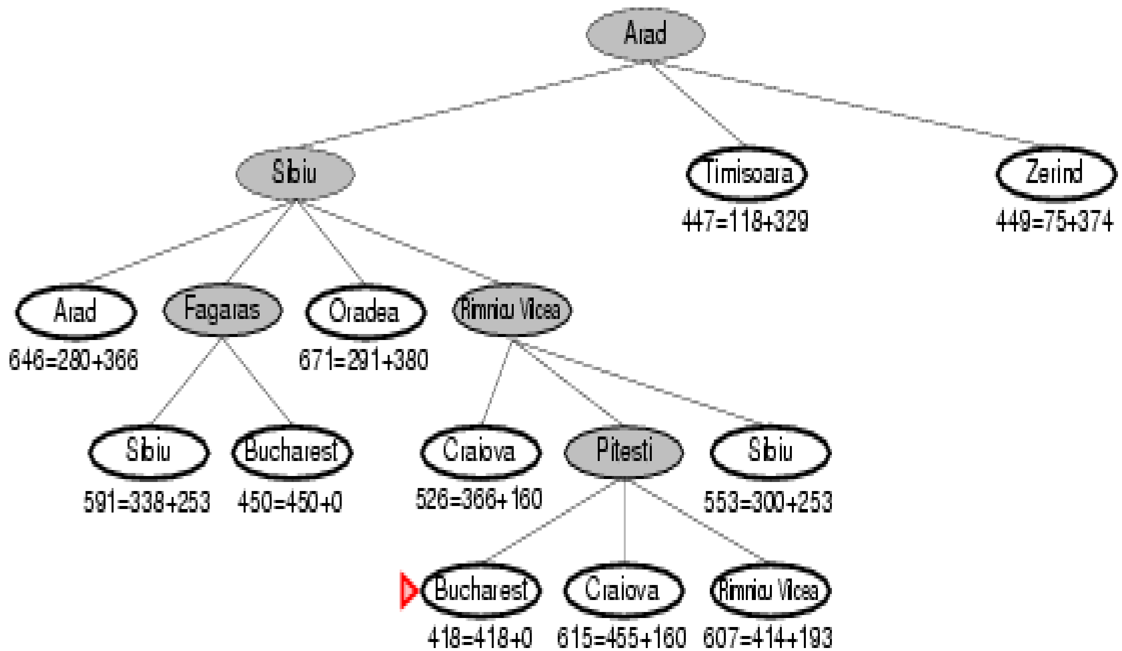
A*


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

A*


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

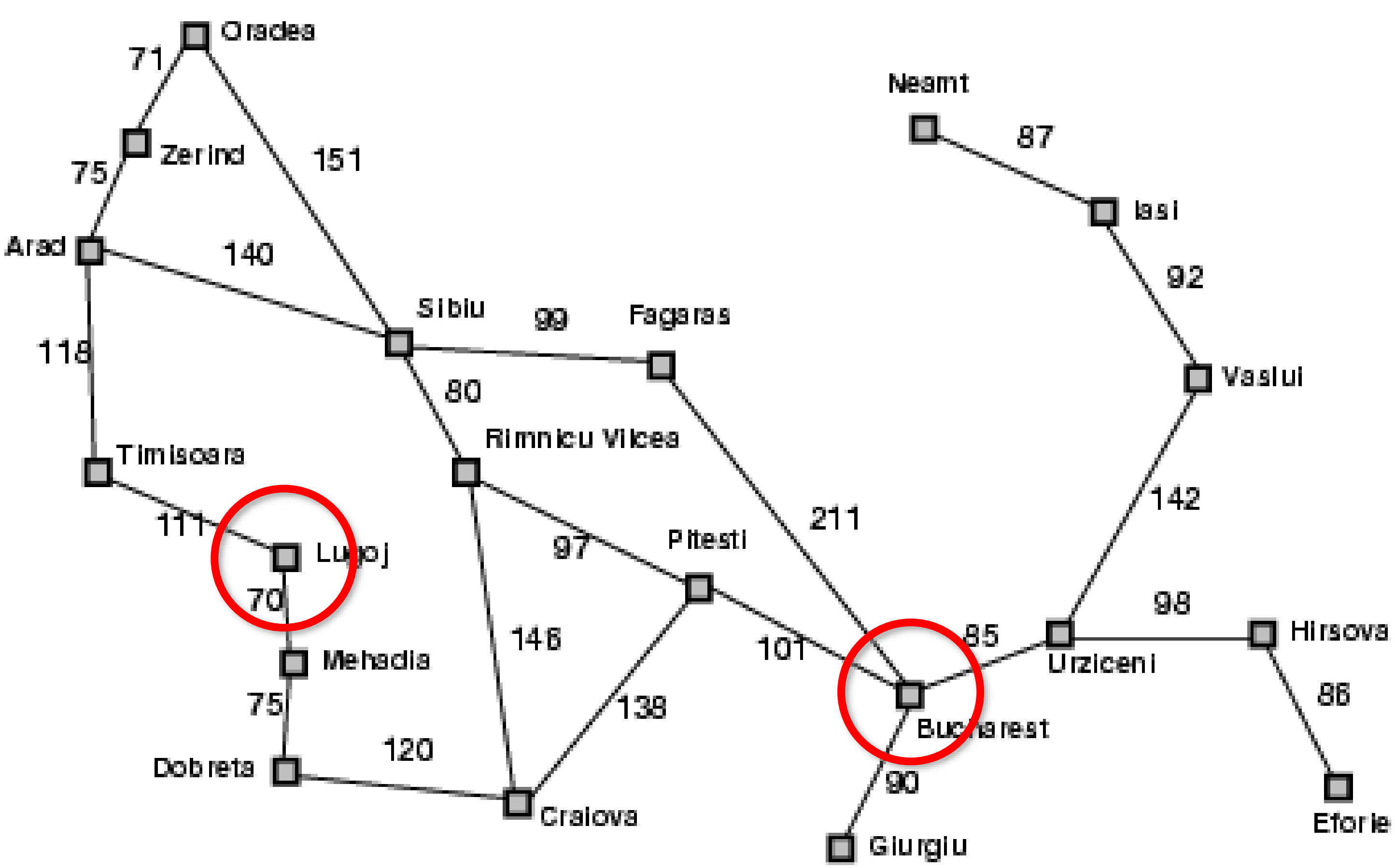
A*


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

Examples

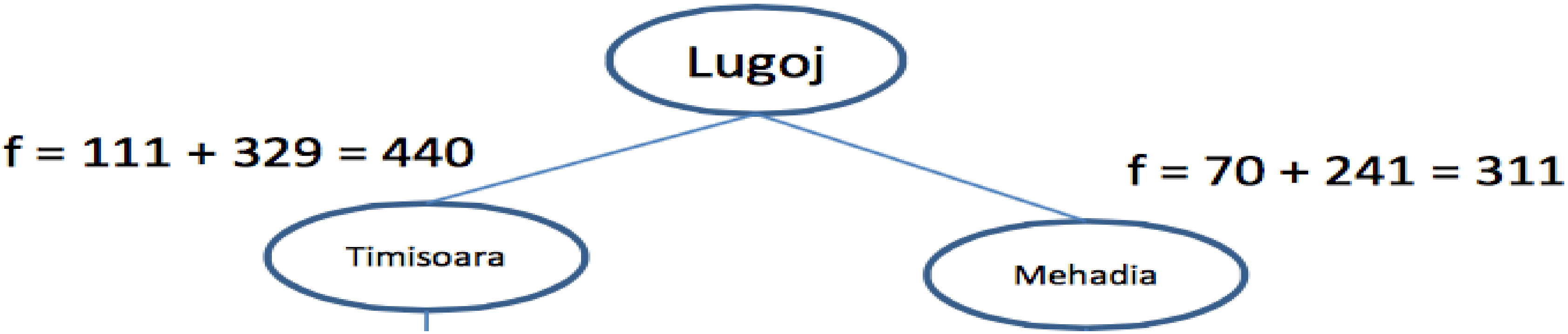
On the Romanian map, **explain** the available paths for a tourist would like to travel from *Lugoj to Bucharest*. **What** are the costs of each path using the A* search algorithm via the given extra information in table? **Which** shortest path the tourist should take?

*- You **SHOULD** enhance your answer with a tree includes all possible paths!*

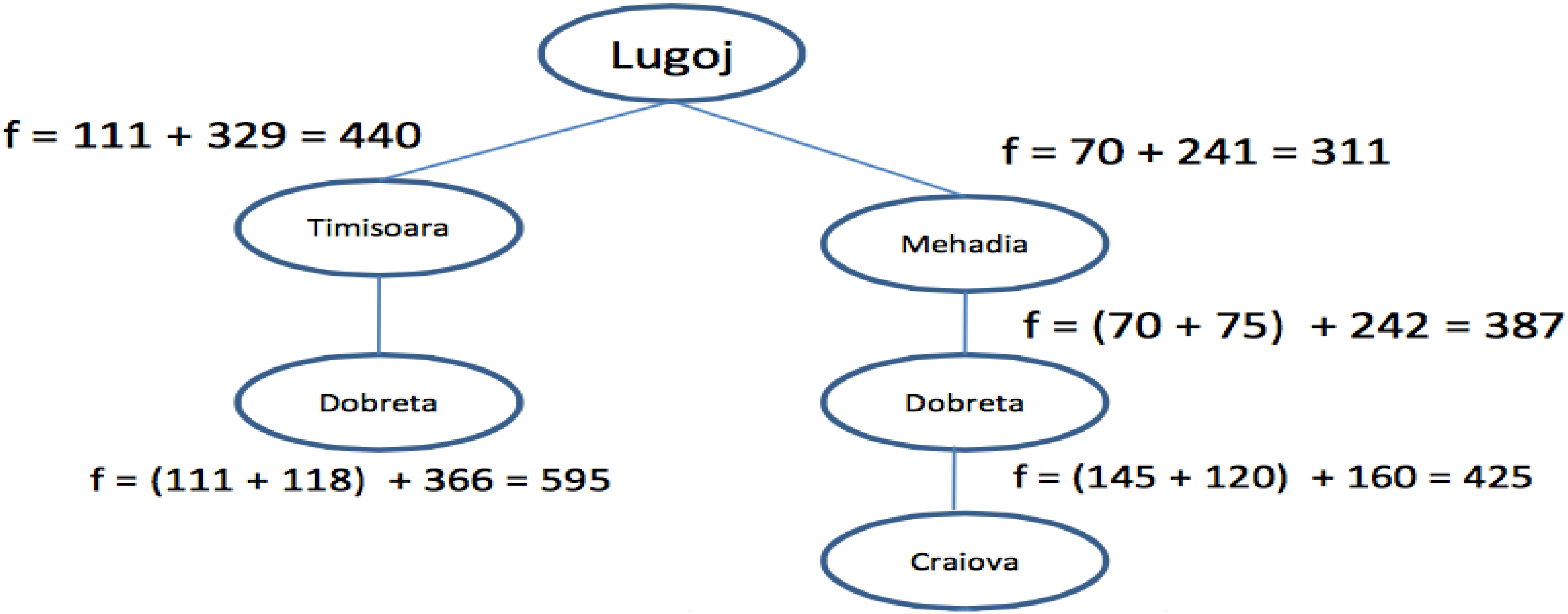


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

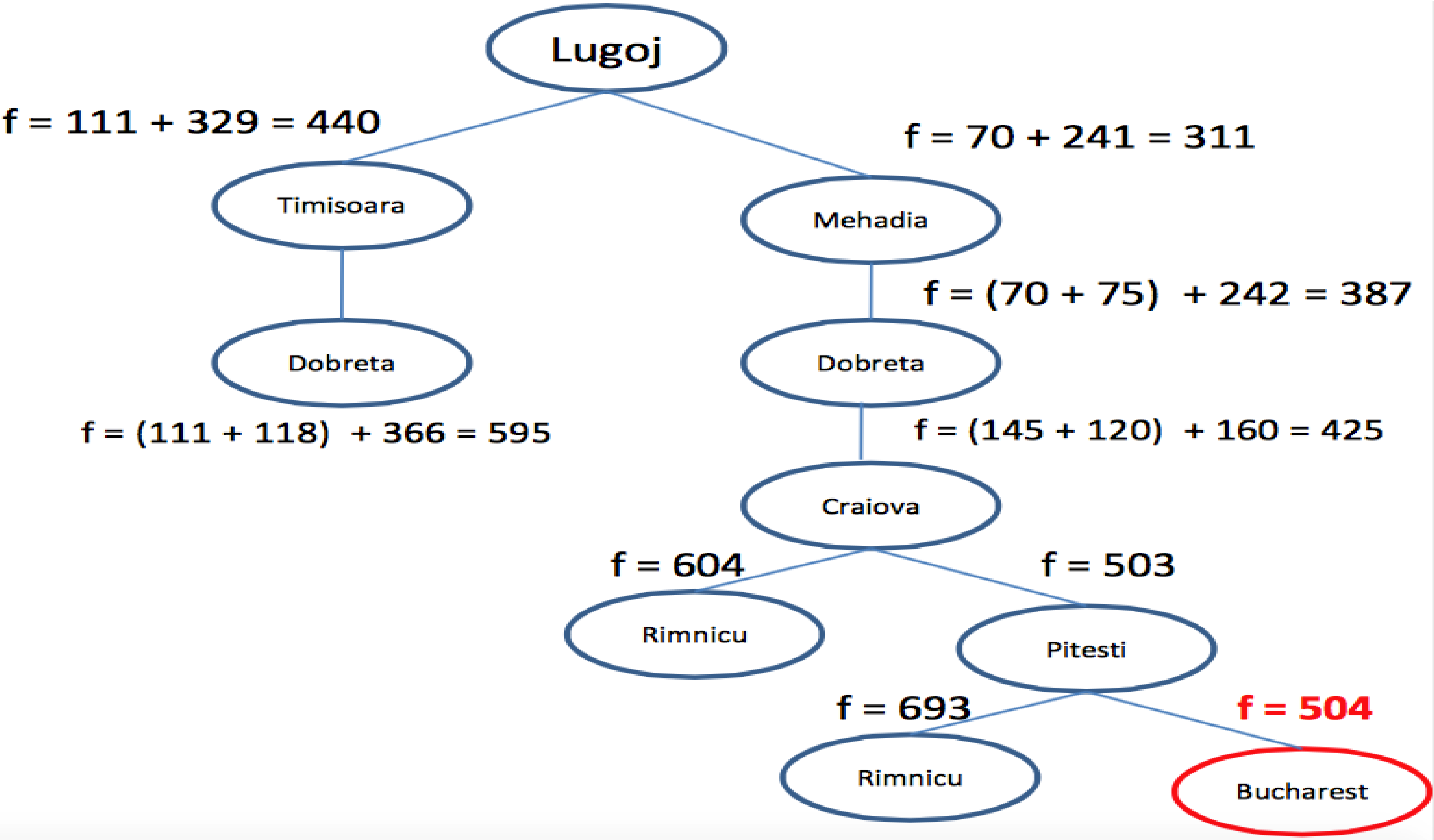
Solution



Solution


Lugoj
f = 111 + 329 = 440
f = 70 + 241 = 311
Timisoara
Mehadia
f = (70 + 75) + 242 = 387
Dobreta
Dobreta
f = (111 + 118) + 366 = 595
f = (145 + 120) + 160 = 425
Craiova

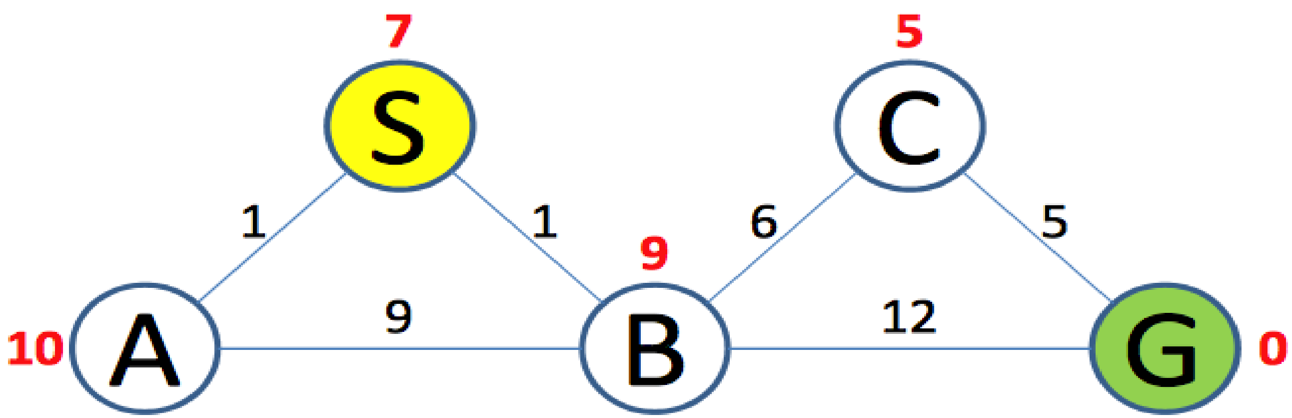
Solution


Lugoj
f = 111 + 329 = 440
f = 70 + 241 = 311
Timisoara
Mehadia
f = (70 + 75) + 242 = 387
Dobreta
Dobreta
f = (111 + 118) + 366 = 595
f = (145 + 120) + 160 = 425
Craiova
f = 604
f = 503
Rimnicu
Pitesti
f = 693
f = 504
Rimnicu
Bucharest

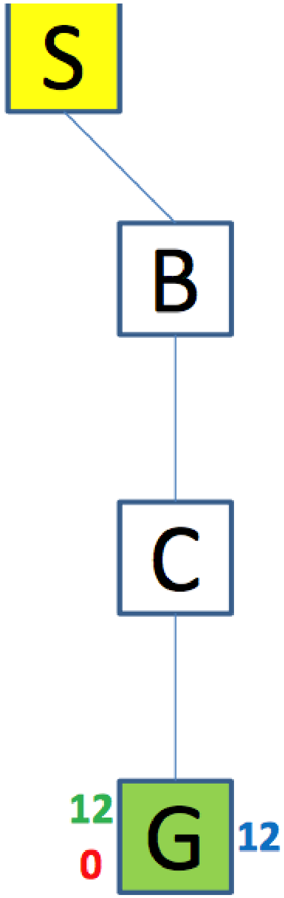
Examples

On the given graph, write the Queue form of this problem. **Which** shortest path from S to G

QUEUE: <S>

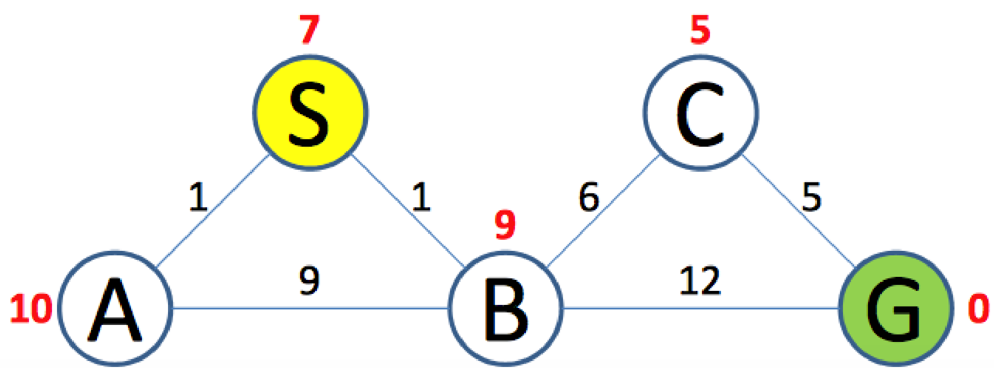


Solution



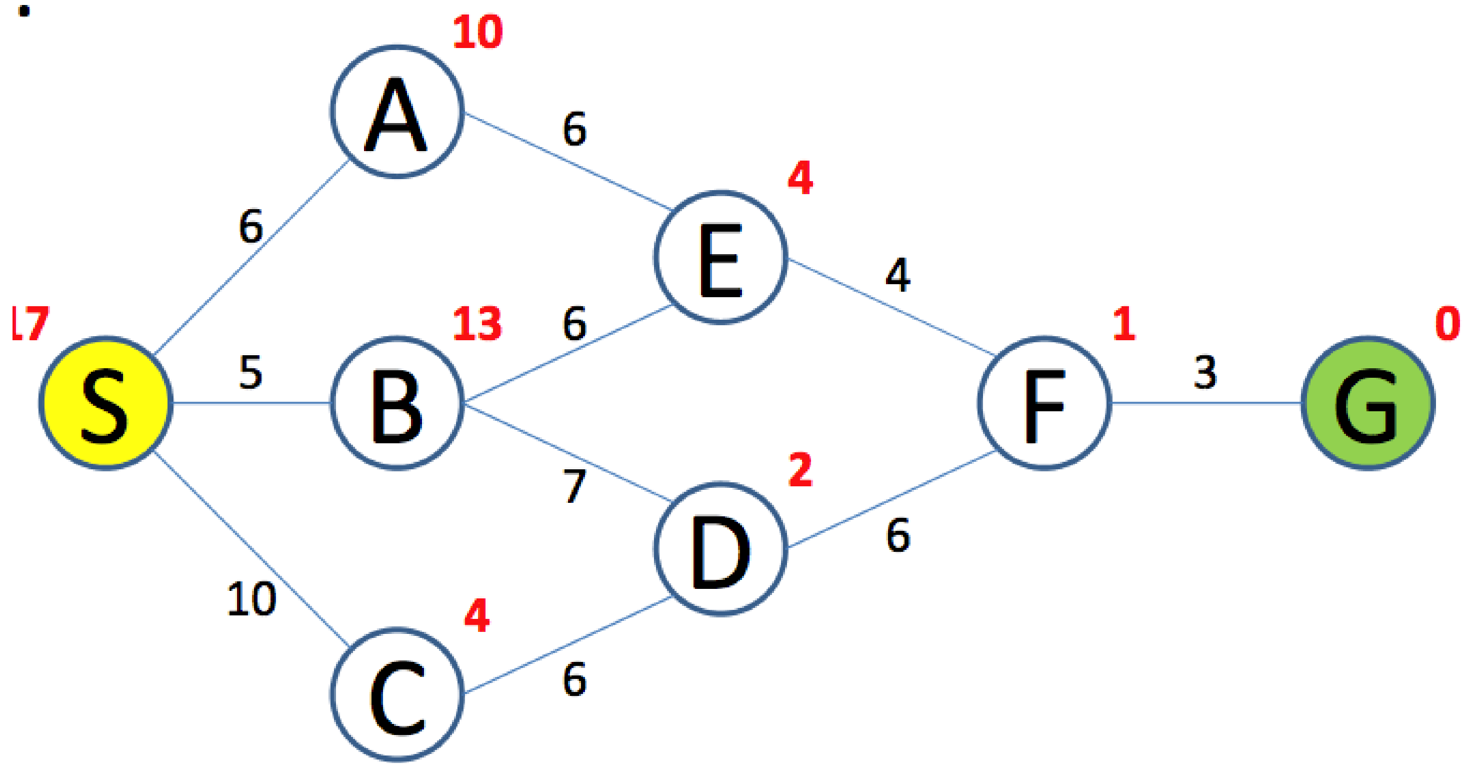
SUCCESS

QUEUE: <SBCG>

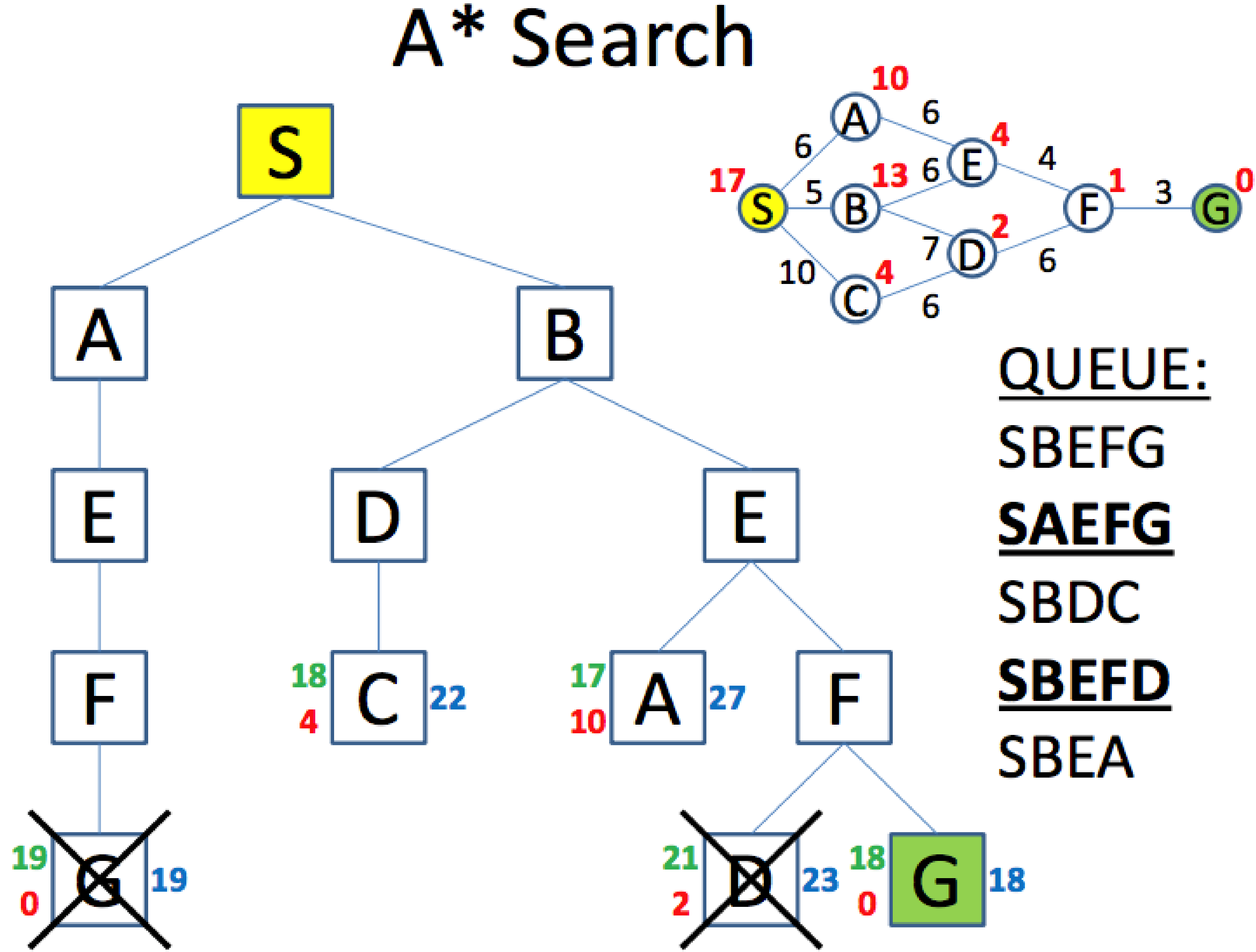


Examples

On the given graph, write the Queue form of this problem. **Which** shortest path from S to G

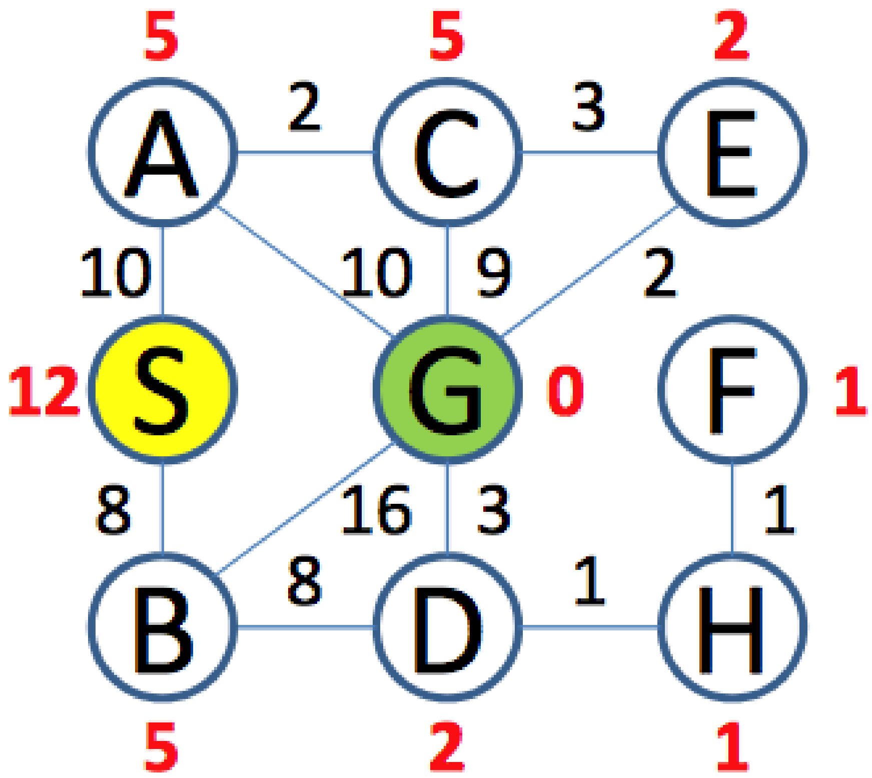


Solution


A* Search
S
A
B
E
D
E
F
18
4
C
22
17
10
A
27
F
19
0
G
19
21
2
D
23
18
0
G
18
10
A
6
6
4
E
4
17
S
5
13
B
6
1
F
3
G
0
7
2
D
6
10
4
C
6
QUEUE:
SBEFG
SAEFG
SBDC
SBEFD
SBEA

Example

On the given graph, write the Queue form of this problem. **Which** shortest path from S to G



Admissible Heuristics

- A heuristic $h(n)$ is admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true/actual cost to reach the goal state from n.

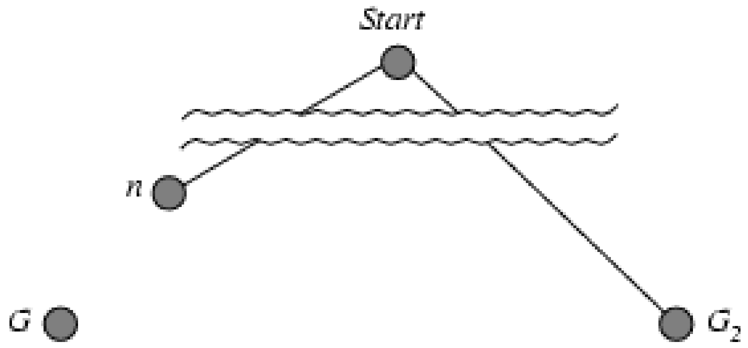
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimum.
  - Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is,
  - Example: h_{SLD}(n) never overestimates the actual road distance (SLD is admissible because the shortest path between any two points is a straight line, so it can not be an overestimate)

Theorem: If $h(n)$ is admissible, A* using `TREE-SEARCH` is optimal

Optimality of A* (proof)

- Suppose some suboptimal goal path G_2 has been generated and is in the frontier. Let n be an unexpanded node in the frontier such that n is on a shortest path to an optimal goal G.

Prove that G_2 can NOT be chosen?

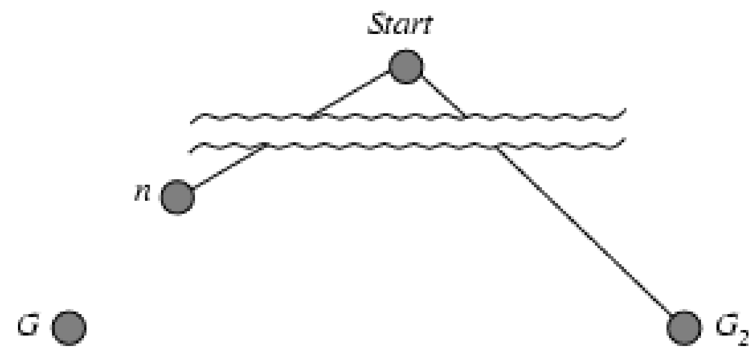


- $f(G_2) = g(G_2)$ since $h(G_2) = 0$ because h is admissible
- $g(G_2) > g(G)$ since G_2 is suboptimal, cost of reaching G is less.
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above

Optimality of A* (proof)

- Suppose some suboptimal goal path G_2 has been generated and is in the frontier. Let n be an unexpanded node in the frontier such that n is on a shortest path to an optimal goal G.

- **Prove that G_2 can NOT be chosen?**



- h(n) $\leq$ h*(n) since h is admissible, h* is minimal distance.
- g(n) + h(n) $\leq$ g(n) + h^*(n)
- f(n) $\leq$ f(G)

Hence $f(G_2) > f(n)$, and A^* will never select G_2 for expansion, since A* is based on lowest evaluation function is chosen for expansion.

Properties of A*

- **Complete?** Yes (unless there are infinitely many nodes with f $\leq f(G)$)
-
- **Optimal?** Yes
-
- **Time?** Exponential
-
- **Space?** Keeps all nodes in memory

A* Applications

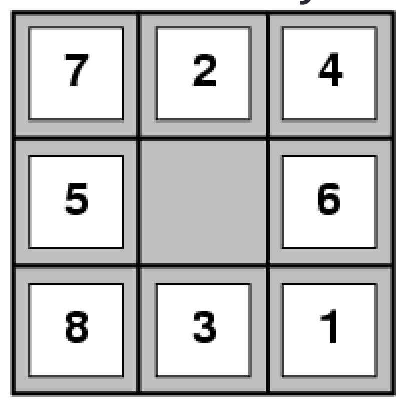
- Video games
- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …

Heuristic Functions

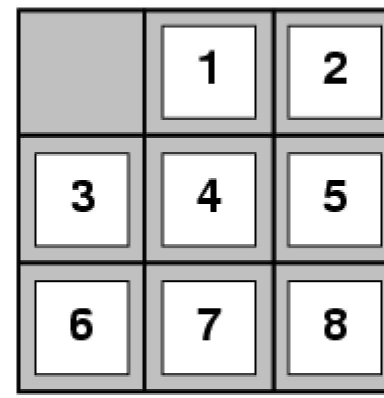
- The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3 *.
- This means that an exhaustive search to depth 22 would look at about

$$3^{22} = 3.1 \times 10^{10} \text{ states}$$

-
- By tracking repeated states, we could cut this down by a factor of about 170,000.
- In case of 15-puzzle is roughly 10^{13}



Start State



Goal State

If we want to find the shortest solutions by using A^*, we need a heuristic function that never overestimates the number of steps to the goal

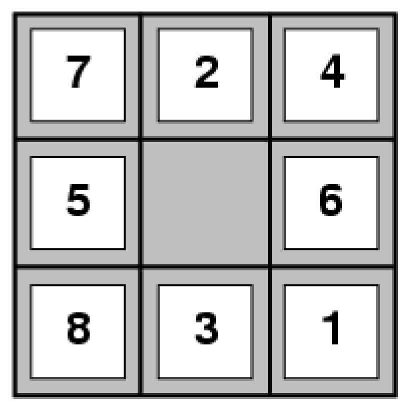
Heuristic Functions

For this target, two ways for solution (finding *h*) we have:

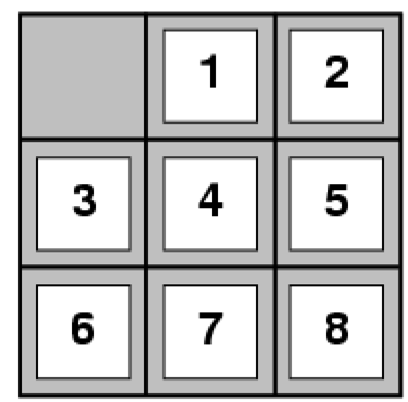
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance *

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

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



Start State



Goal State

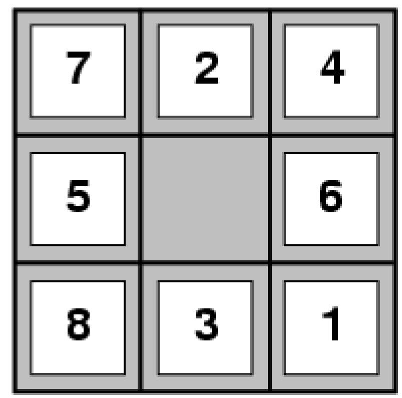
Admissible Heuristic

Two ways for solution (finding *h*)

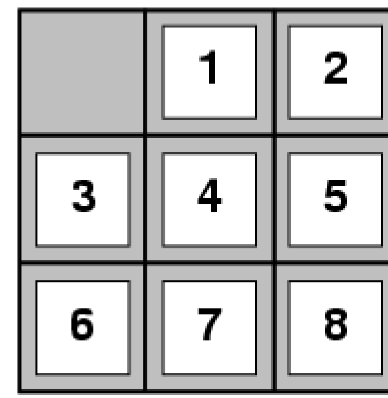
- $h_1(n)$ = number of misplaced tiles *
- $h_2(n)$ = total Manhattan distance

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

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



Start State



Goal State

Neither of these overestimates the true solution cost, which is 26. Then both are admissible!

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)

  Then h_2 dominates h_1. Or h_2 is better for search

- Domination translates directly into efficiency: A* using h_2 will never expand more nodes than ***A**** using $h_{1.}$

- Hence, it is always better to use a heuristic function with higher values, provided it does not overestimate and that the computation time for the heuristic is not too large.

Dominance

We generated 1200 random problems of 8-puzzle with solution lengths from 2 to 24 (100 time for each even *d*). Here the average number of nodes generated

	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- If $h_2(n) >= h_1(n)$ for all *n* (both admissible)…then h_2 dominates h_1 and is better for search

-Solution with length 12, ***A**** with h_2 is 50,000 times more efficient than uninformed iterative deepening search.

Inventing Admissible Heuristics

- One problem with generating new heuristic functions is that one often fails to get one "clearly best" heuristic.

- If a collection of admissible heuristics $h_1 \ldots h_m$ is available for a problem, and none of them dominates any of the others, **which should we choose?**

- Here, we need not make a choice. We can have the best of all worlds, by defining:

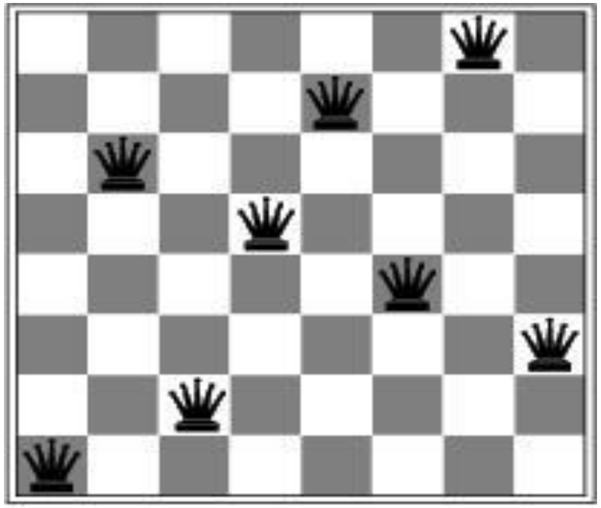
$$h(n) = \max \{h_1(n), h_2(n), \ldots, h_m(n)\}$$

Local Search Algorithms

- Till now: Systematic exploration of search space.

  - Path to goal is solution to problem

- In many optimization problems, the path to the goal is irrelevant; i.e. we do not care about path to solution, we need the solution itself.
- The goal state itself is the solution, just want solution.
- Examples:
  - Placing queens on a chessboard,
  - How many airline flights to have to where?
  - Any 8-puzzle state,
- In such cases, we can use local search algorithms



Local Search Algorithms

- It operate by keeping a single "current" state, then try to improve it gradually or iteratively!
- (Another): A local search algorithm starts from a candidate solution and then iteratively moves to a neighbor solution.

- Although LSA are not systematic, two key advantages:
  - Use very little memory, (no paths are retained)
  - Find often reasonable solutions in large or infinite state spaces.

Hill-Climbing Search

- It is simply a loop that continually moves in the direction of increasing value, that is, uphill,
- It terminates when it reaches a "peak" where no neighbor has a higher value, (i.e. greedy local search)
- Does not maintain a search tree, so the current node data structure need only to record the state and its objective function value
- Hill-climbing does not look ahead beyond the immediate neighbors of the current state.

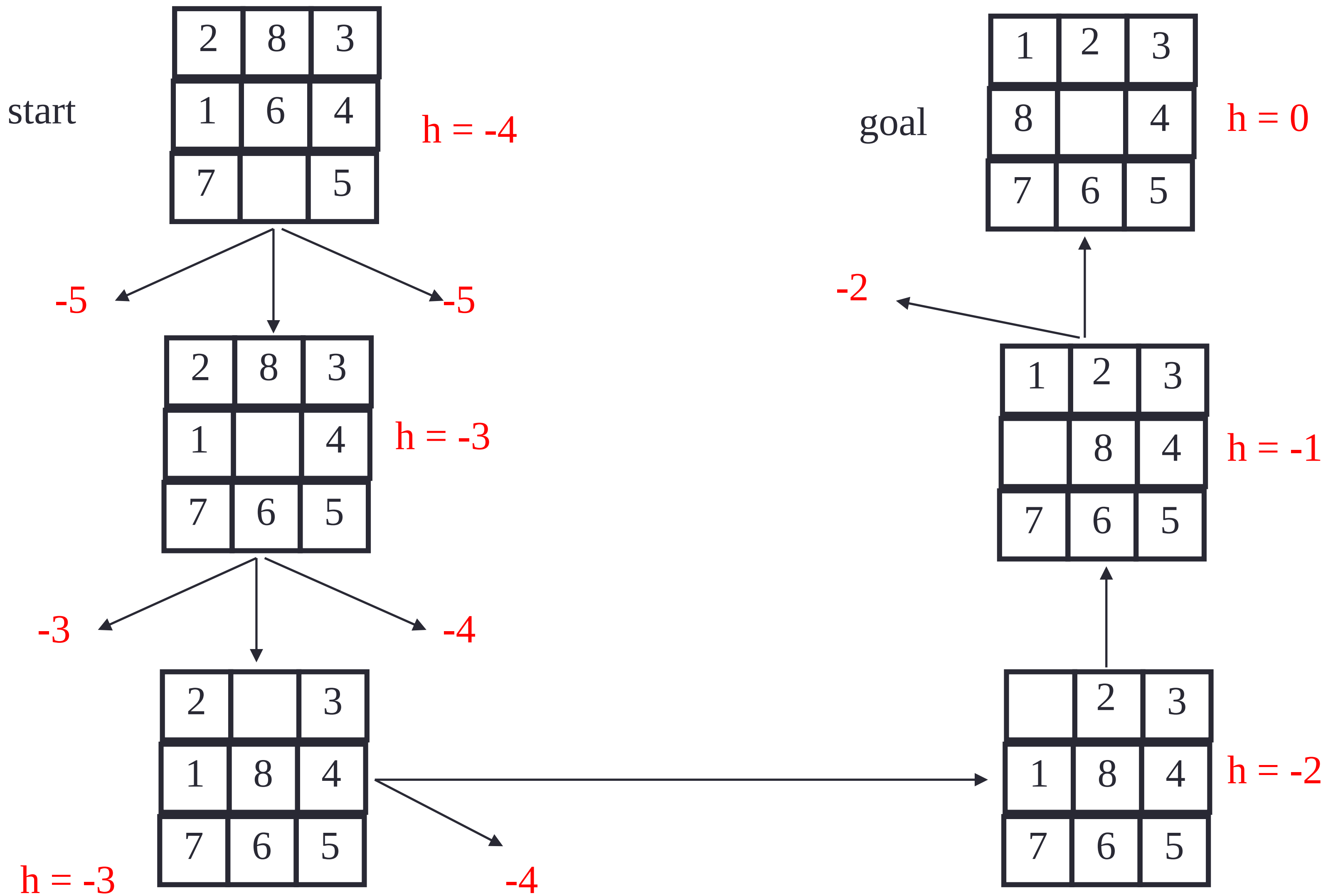
Hill-Climbing Search

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: 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
```

- HCS chooses randomly among the set of best successors, if there is more than one,
- HCS likes *greedy local search* because it grabs a good neighbor state without thinking ahead about where to go next!

HCS Example

h(n) = - (number of tiles out of place)



HCS Example 8-queens problem

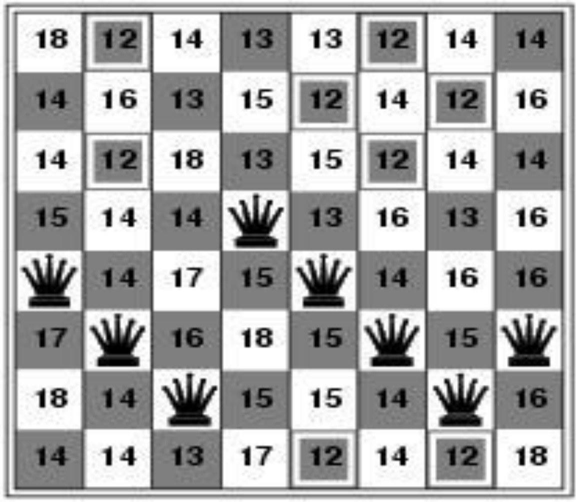
- Put n queens on an n x n board with no two queens on the same row, column, or diagonal

- Successor function – returns all possible states generated by moving a single queen to another square in the same column – each state has 8 x 7 = 56 successors.

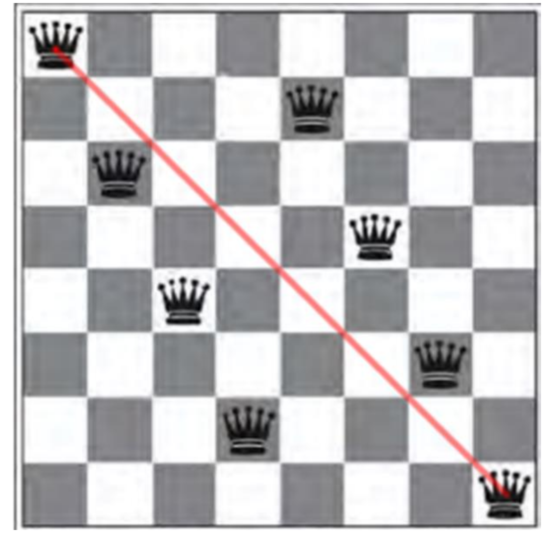
- Heuristic function *h(n)*: the number of pairs of queens that are attacking each other (directly or indirectly) when at least two queens are on the same row or column or diagonal.

HCS Example 8-queens problem

a)



b)

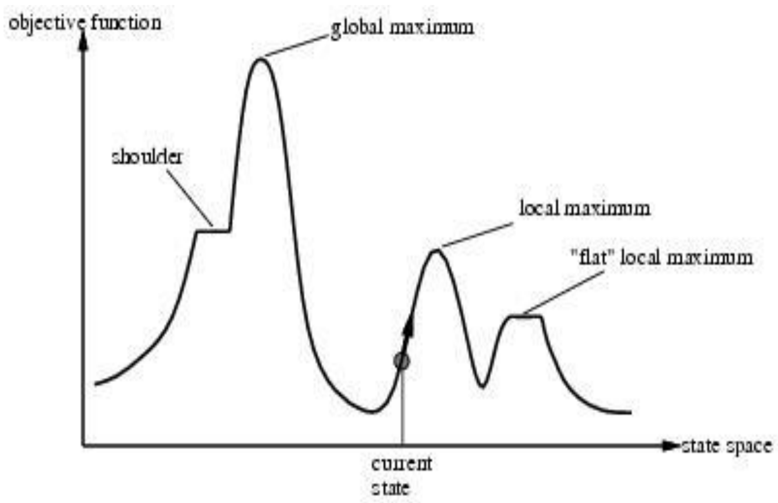


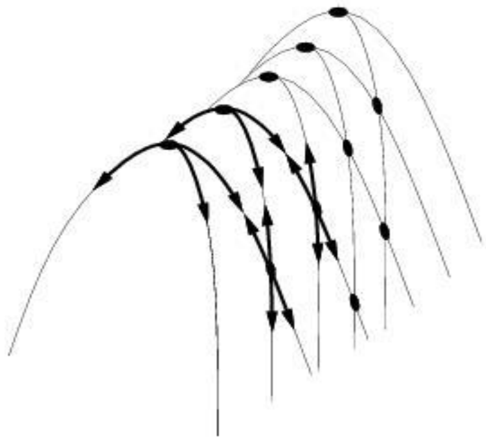
a) shows a state of *h*=17 and the *h*-value for each possible successor.

b) A local minimum in the 8-queens state space (h=1).

- HCS often makes very rapid progress towards a solution, because it is usually quite easy to improve a bad state. *

HCS Drawbacks





- Local Maxima is a peak that is higher than each of its neighboring states, but lower than the global maximum – figure shows a local Maxima *
- Ridge = sequence of local maxima difficult for greedy algorithms to navigate
- Plateaux = an area of the state space where the evaluation function is flat – no uphill exit seems to exist, i.e. gives no direction (random walk)

In each case, the algorithm reaches a point at which no progress is being made.

- Remedy:
- Introduce randomness!

Genetic Algorithm

Genetic Algorithm (GA)

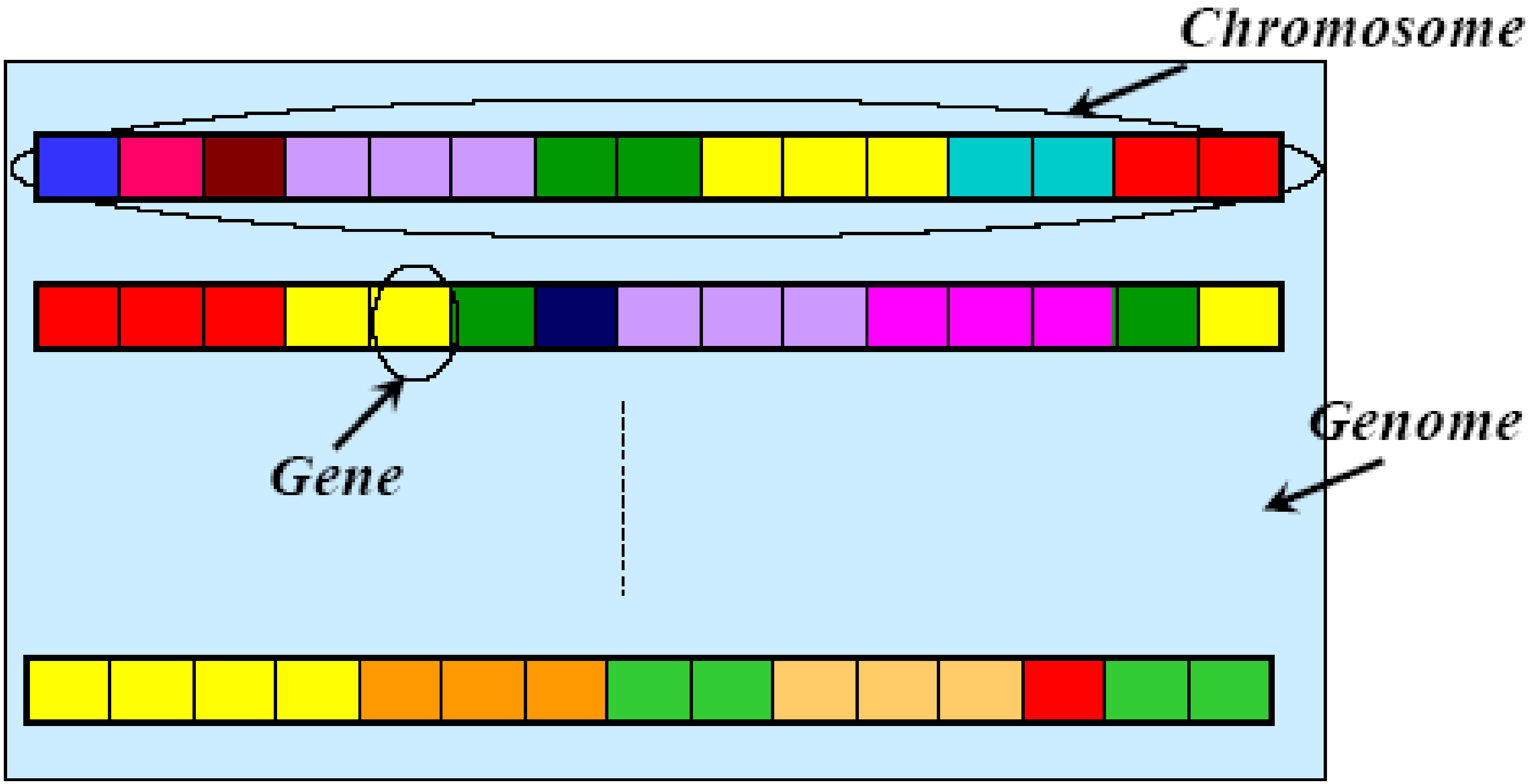
- It is based on the mechanics of biological evolution
- Widely-used today in business, scientific and engineering circles
- In GA successor states are generated by combing two parent states rather than by modifying a single state.
- **General idea** is to find a solution by iteratively selecting fittest individuals from a population and breeding them until either a threshold on iterations or fitness is hit.
- This is similar to the natural selection in genes and therefore called the genetic algorithms.

Genetic Algorithm

- It is implemented as a computer simulation in which a population of abstract representations (called chromosomes or the genotype or the genome) of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem evolves toward better solutions.

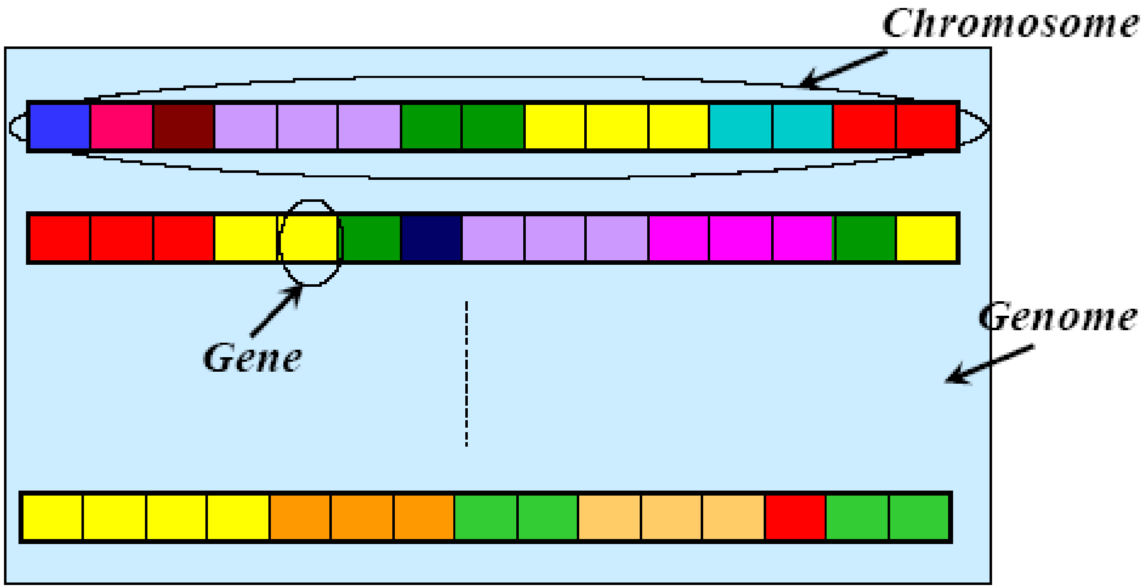
- Traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible.

Genetic Algorithm

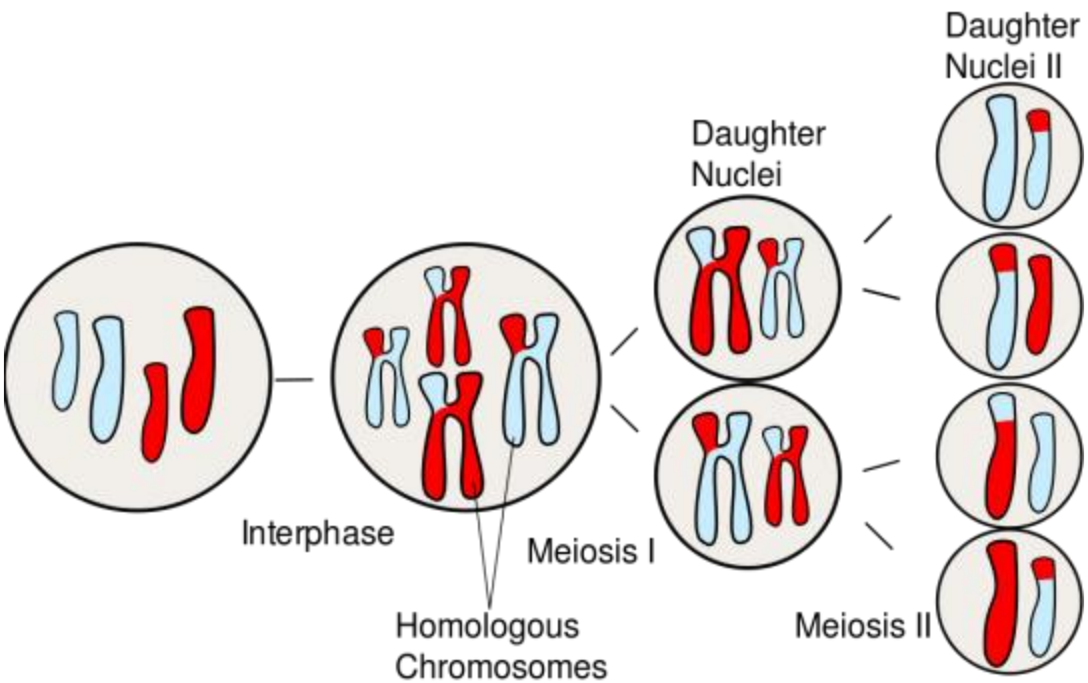


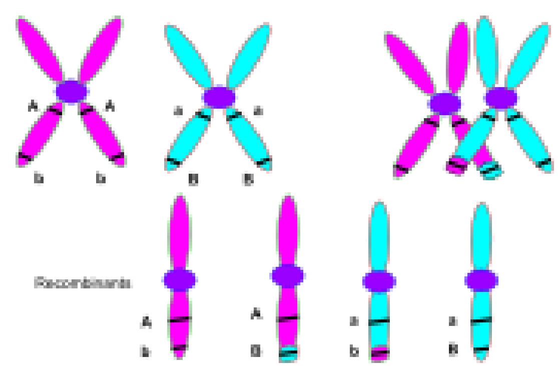
Genetic Algorithm

- An individual state is represented by a sequence of "genes".
- The selection strategy is randomized with probability of selection proportional to "fitness".
- Individuals selected for reproduction are randomly paired, certain genes are crossed-over, and some are mutated.



Genetic Algorithm-A bit of Biology





Key terms

- **Individual** - Any possible solution
- **Population** - Group of all *individuals*
- **Search Space** - All possible solutions to the problem
- **Chromosome** - Blueprint for an *individual*
- **Trait** - Possible aspect (*features)* of an *individual*
- **Allele** - Possible settings of trait (black, blond, etc.)
- **Locus** - The position of a *gene* on the *chromosome*
- **Genome** - Collection of all *chromosomes* for an *individual*

Genetic Algorithm Scenario

- Population:
  - Start with K randomly generated states
  - Each state is a representation of strings (or array of bits)
- Fitness Function:
  - objective function to evaluate the best states
  - It is a problem dependent
- Selection:
  - Select n states out of the K state population based on fitness function
- Crossover
- Mutation

Genetic Algorithm Procedure

1. A population is created with a group of individuals created randomly.
2. The individuals in the population are then evaluated.
3. The evaluation function is provided by the programmer and gives the individuals a score based on how well they perform at the given task.
4. Two individuals are then selected based on their fitness, the higher the fitness, the higher the chance of being selected.
5. These individuals then "reproduce" to create one or more offspring, after which the offspring are mutated randomly.
6. This continues until a suitable solution has been found or a certain number of generations have passed, depending on the needs of the programmer.

General Algorithm for GA

Initialization

- Initially many (hundreds/thousands) individual solutions are generated to form an initial population.
- Traditionally, the population is generated randomly, covering the entire range of possible solutions (the search space).

Selection

- A proportion of the existing population is selected to breed a new generation.
- Individual solutions are selected through a fitness-based process, where fitter solutions are typically more likely to be selected.
- Certain selection methods rate the fitness of each solution and preferentially select the best solutions.
- The known selection method is **roulette wheel selection**

General Algorithm for GA

❖ Reproduction

- Generate a second generation population of solutions from those selected through genetic operators: **crossover** and **mutation**.

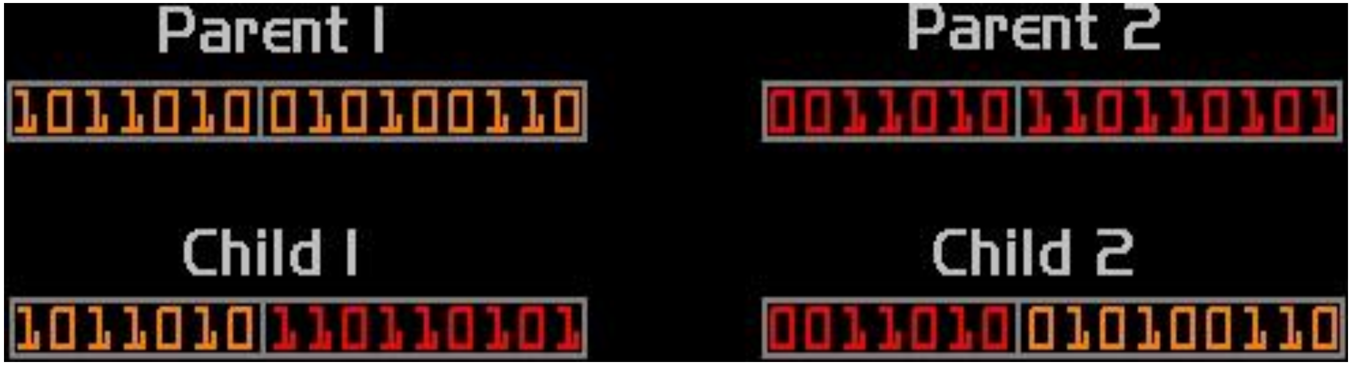
- For each new solution to be produced, a pair of "parent" solutions is selected for breeding from the pool selected previously.

- By producing a "child" solution using the above methods of crossover and mutation, a new solution is created which typically shares many of the characteristics of its "parents".

- The process continues until a new population of solutions of appropriate size is generated.

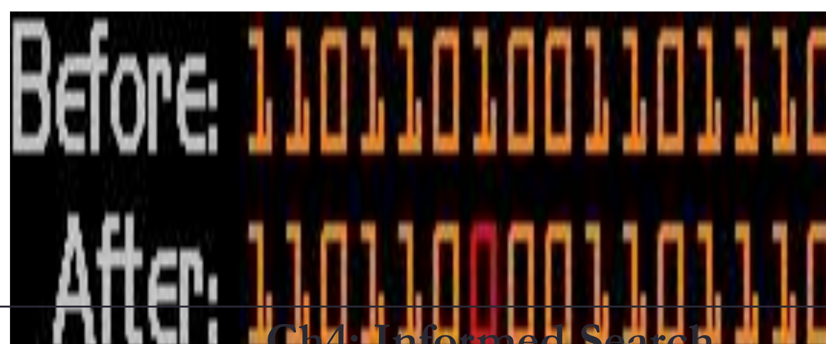
Crossover

- The most common type is <u>**single point crossover**</u>.
- In single point crossover, you choose a locus at which you swap the remaining alleles from on parent to the other.
- The children take **one** section of the chromosome from each parent.
- The point at which the chromosome is **broken** depends on the **randomly** selected crossover point.
- Crossover does not always occur, however. Sometimes, based on a set probability, **no crossover occurs** and the parents are copied directly to the new population. The probability of crossover occurring is usually <u>60% to 70%.</u>



Mutation

- After selection and crossover, you now have a new population.
- Some are directly **copied**, and others are produced by **crossover**.
- <u>In order to ensure that the individuals are not all exactly the same, you allow for a small chance of mutation.</u>
- You loop through all the alleles of all the individuals, and if that allele is selected for mutation, you can either change it by a small amount or replace it with a new value.
- The probability of mutation is usually between <u>10-20 % of the pop</u>.
- Mutation is fairly simple. You just change the selected alleles based on what you feel is necessary and move on. Mutation is, however, vital to ensuring genetic diversity within the population.



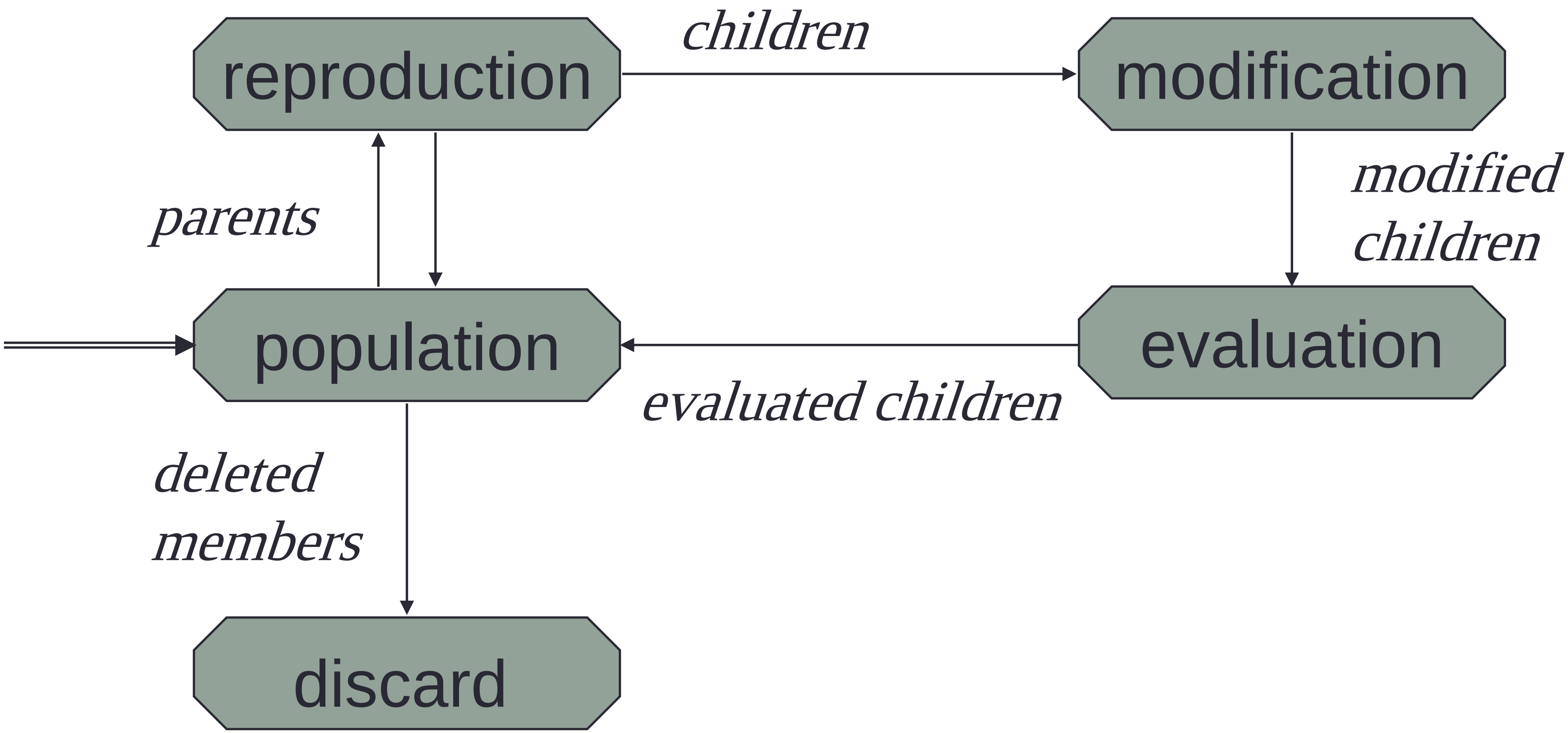
Crossover & Mutation

P1	(0 1 $\boxed{1}$ 0 1 0 0 0)	$\longrightarrow$	(0 1 $\boxed{0}$ 0 1 0 0 0)	C1
P2	(1 1 $\boxed{0}$ 1 1 0 1 0)		(1 1 $\boxed{1}$ 1 1 0 1 0)	C2

Before:	(1 0 1 $\boxed{1}$ 0 1 1 0)
After:	(0 1 1 $\boxed{0}$ 0 1 1 0)

Before:	(1.38 $\boxed{-69.4}$ 326.44 0.1)
After:	(1.38 $\boxed{-67.5}$ 326.44 0.1)

The GA Cycle of Reproduction

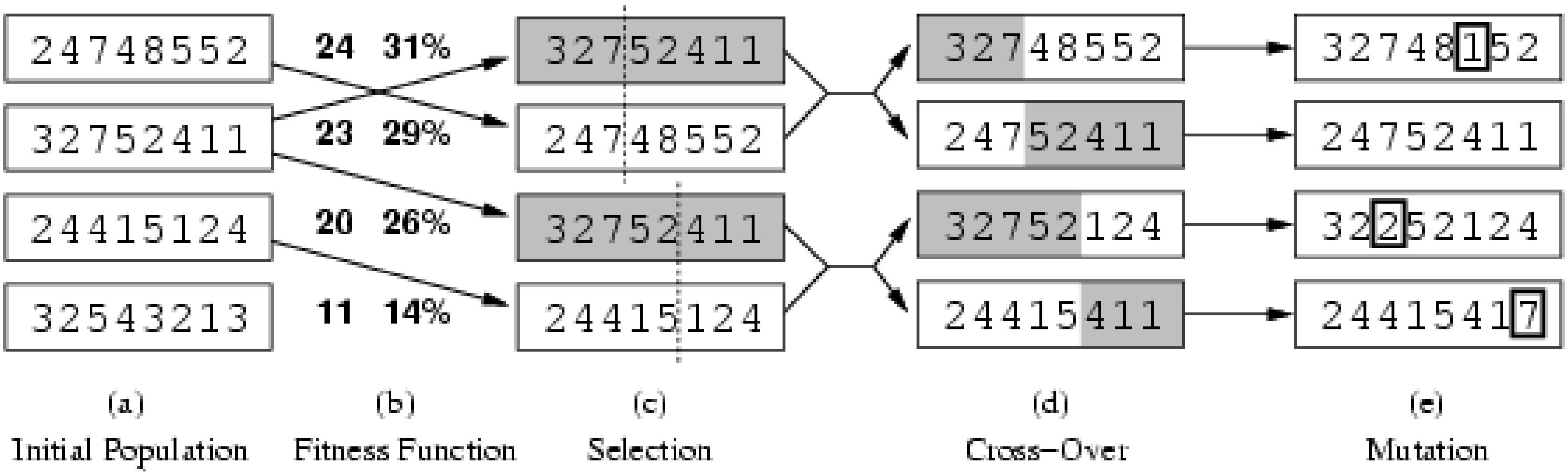


General Algorithm for GA

❖ Termination

- This process is repeated until a termination condition has been reached.
- Common terminating conditions are:
  - ✓ A solution is found that satisfies minimum criteria
  - ✓ Fixed number of generations reached
  - ✓ Allocated budget (computation time/money) reached
  - ✓ Manual inspection

8-Queens by Genetic Algorithm



4 states for 8-queens problem

2 pairs of 2 states randomly selected based on fitness. Random crossover points selected

New states after crossover

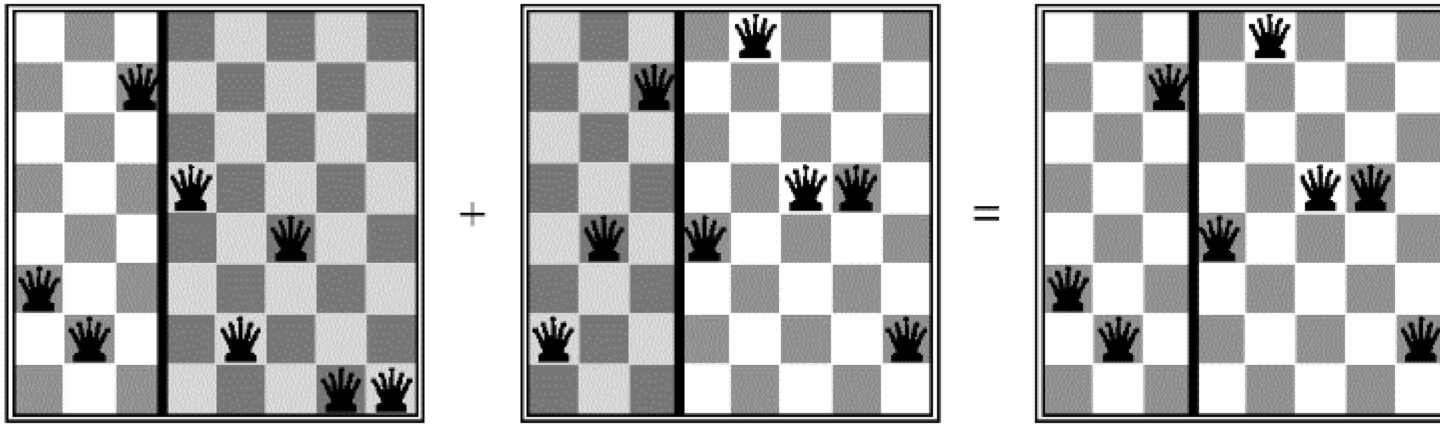
Random mutation applied

Fitness function: number of non-attacking pairs of queens (min = 0, max = $8 \times 7/2 = 28$)

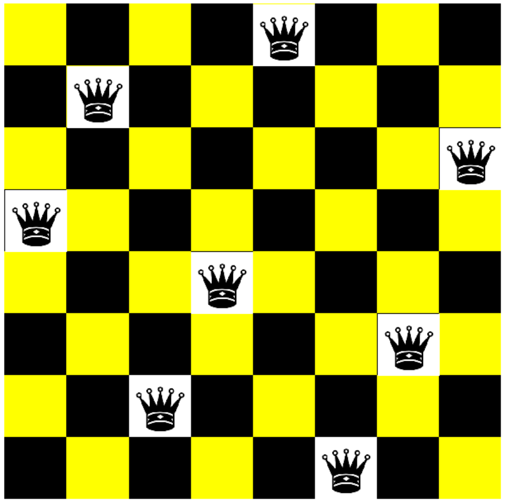
$24/(24+23+20+11) = 31\%$

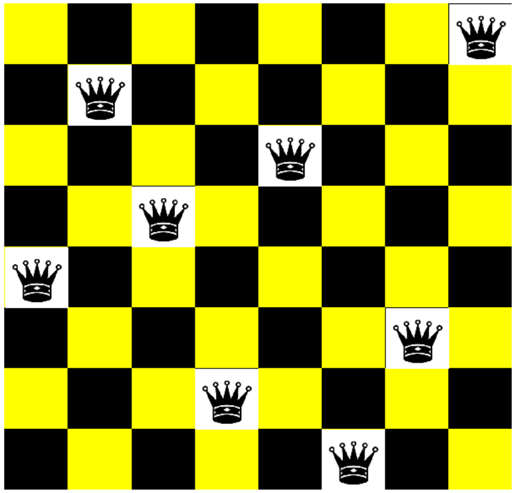
8-Queens by Genetic Algorithm

The Selection step

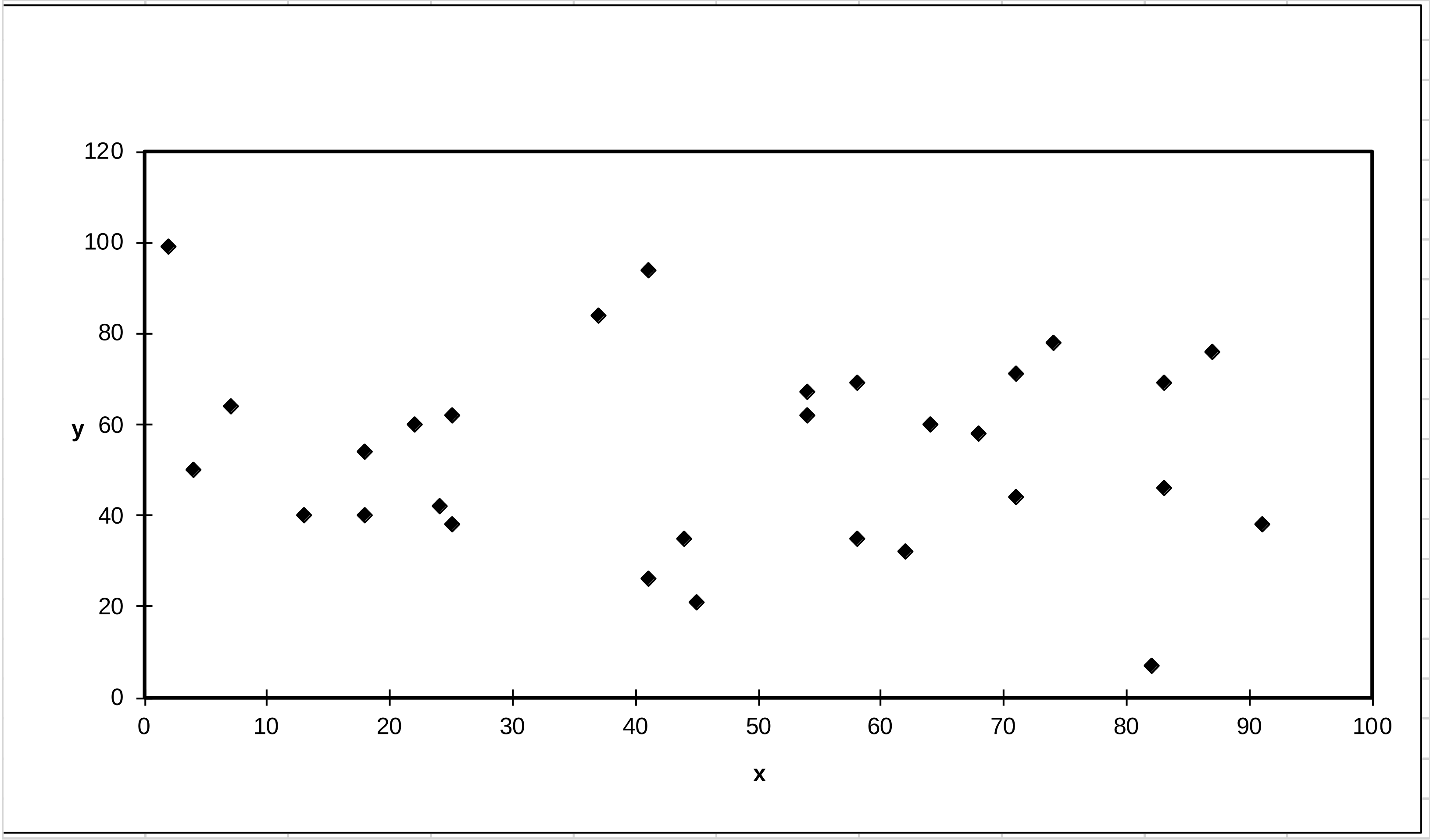


Many Solutions

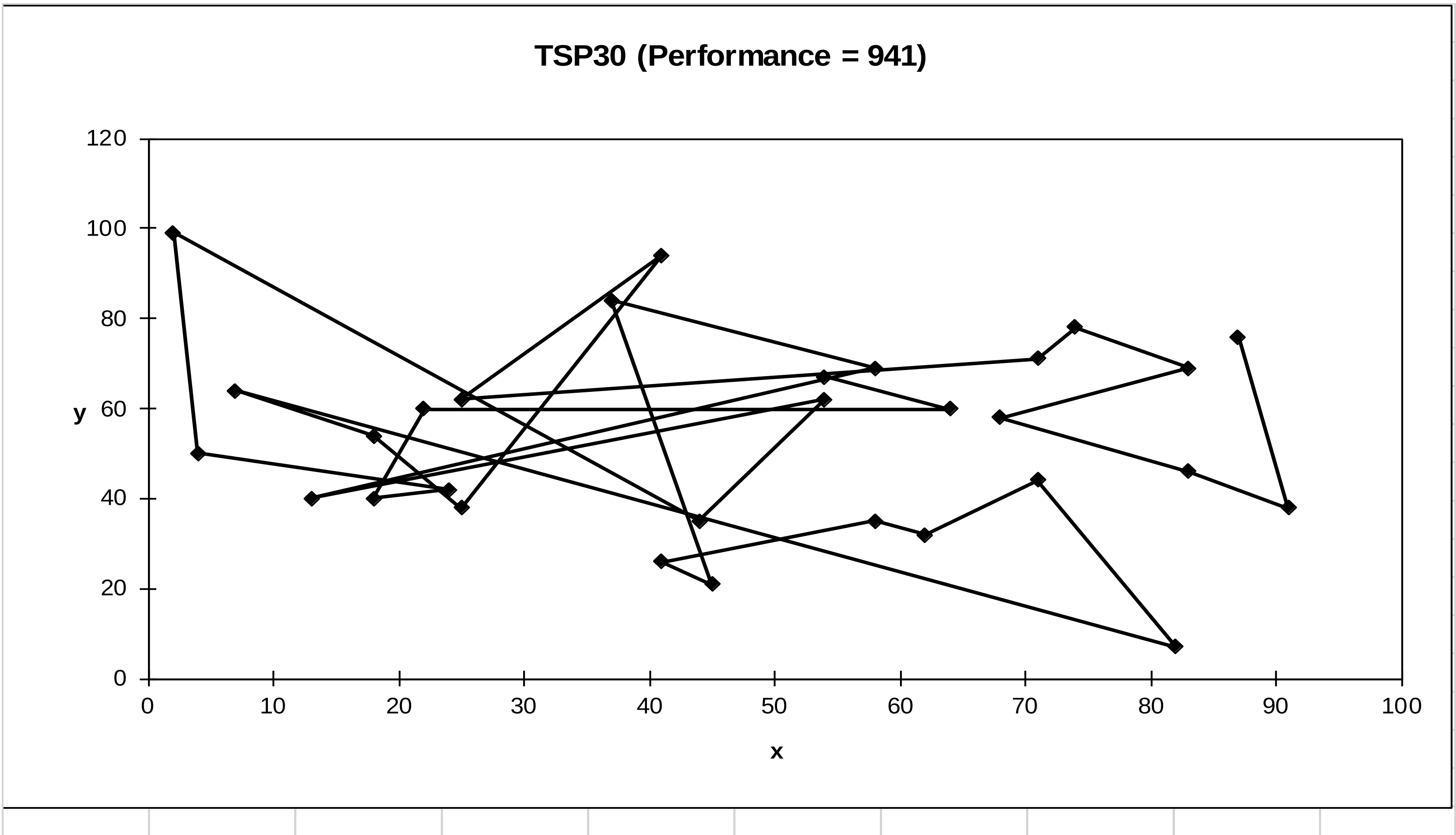




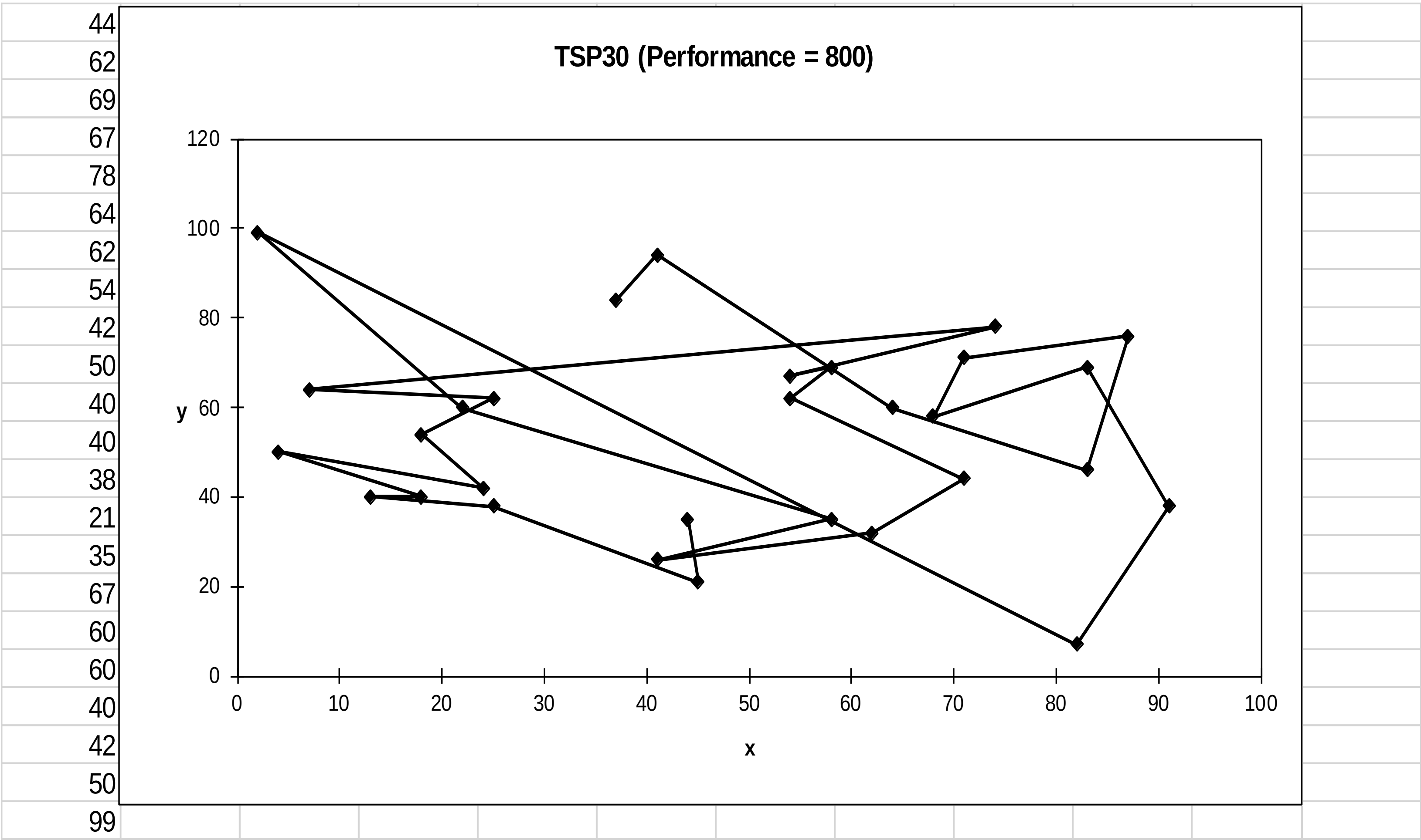
Example: TSP of 30 Cities



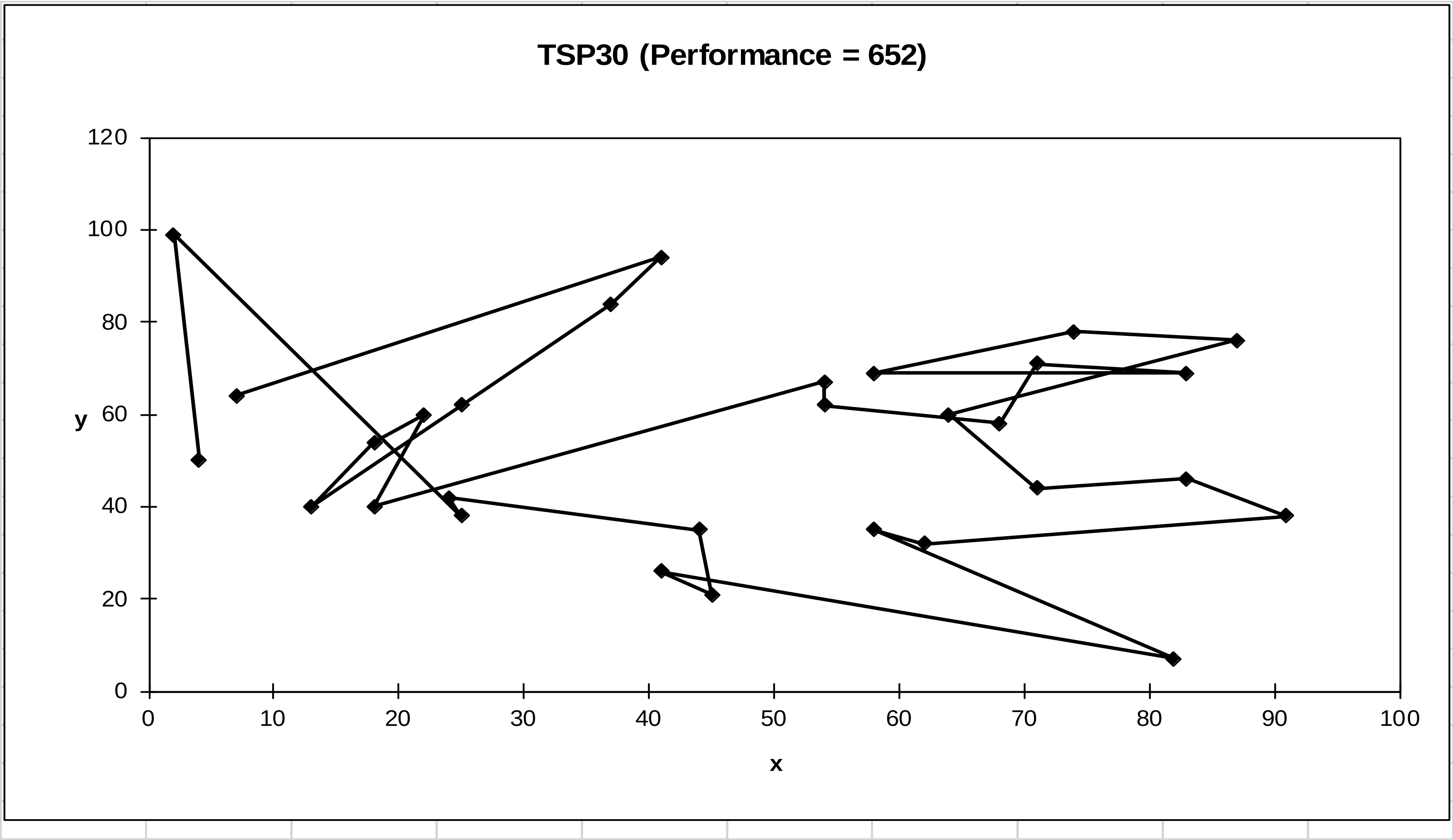
Solution 1 (Distance = 941)



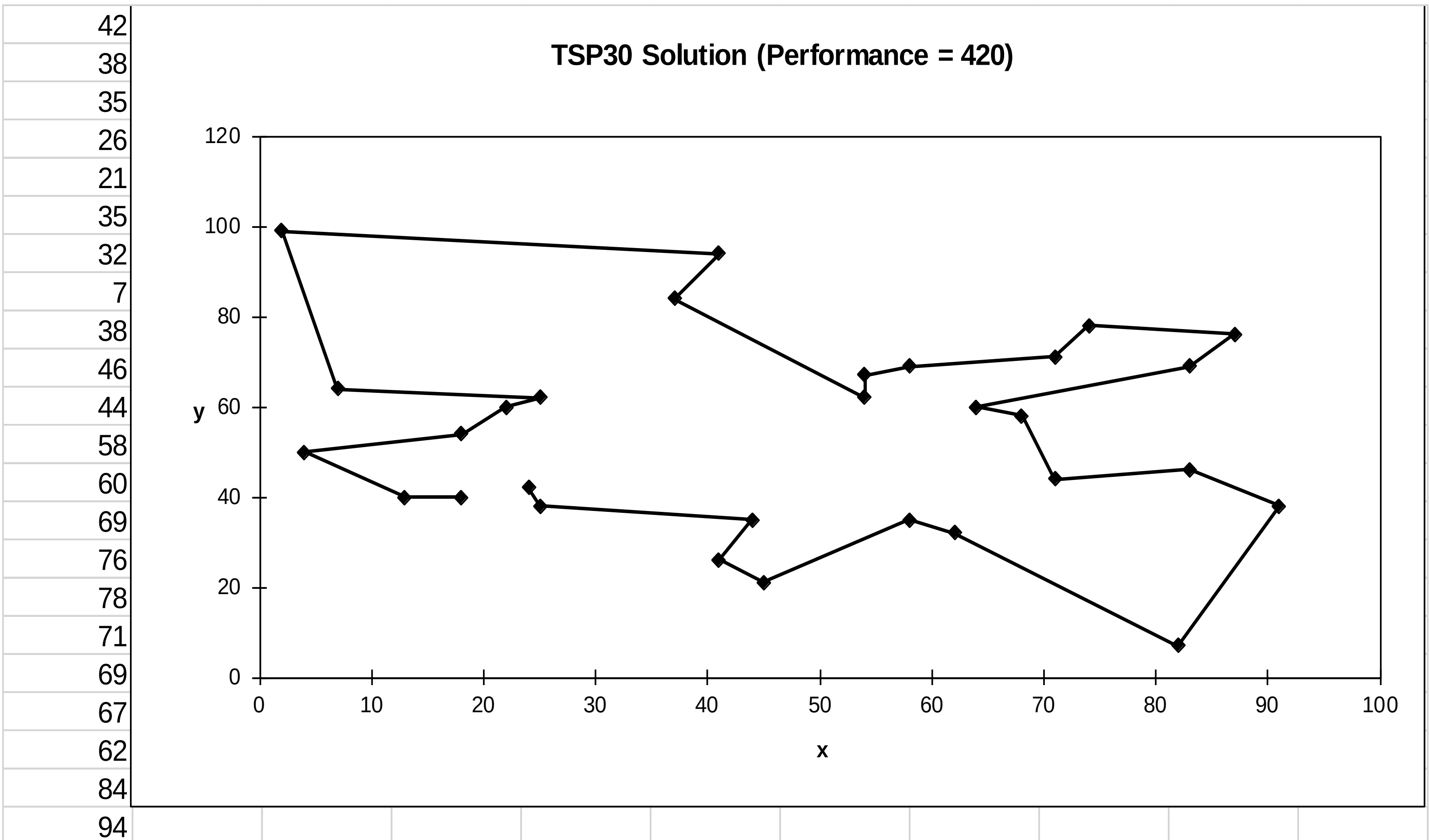
Solution 2 (Distance = 800)



Solution 3 (Distance = 652)



Best Solution 4 (Distance = 420)



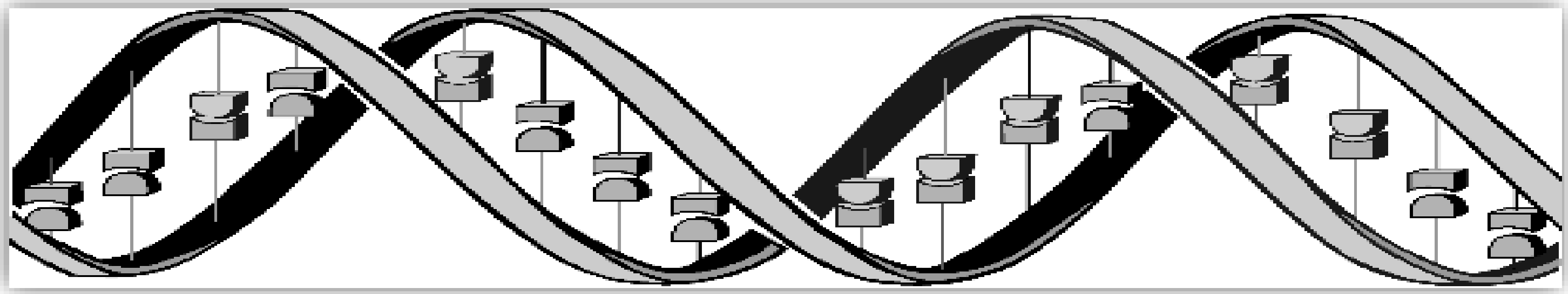
Genetic Algorithm

```
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))
```

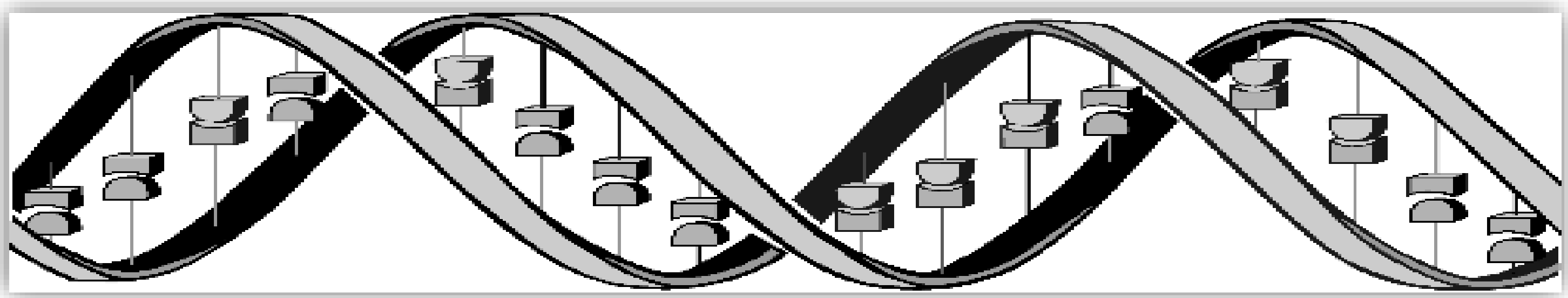
Benefits of Genetic Algorithms

- Concept is easy to understand
- Modular, separate from application
- Supports multi-objective optimization
- Good for “noisy” environments
- Always an answer; answer gets better with time
- Inherently parallel; easily distributed



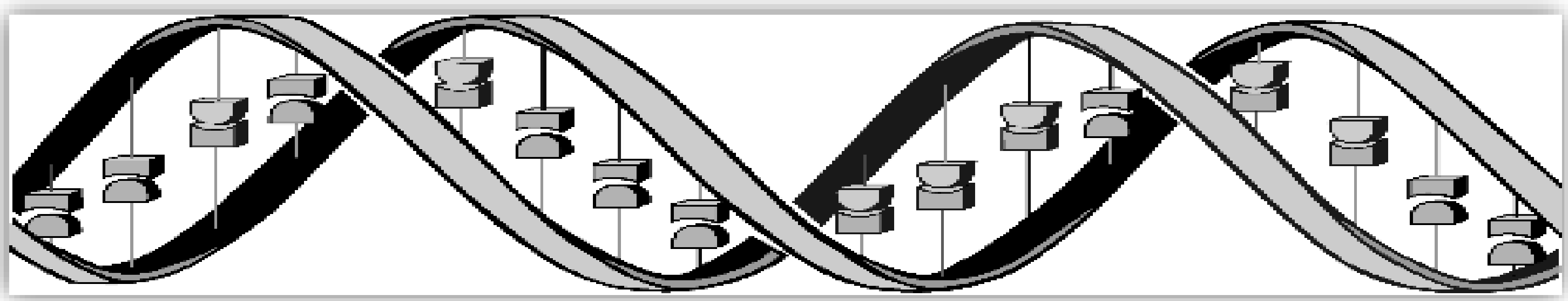
Benefits of Genetic Algorithms

- Many ways to speed up and improve a GA-based application as knowledge about problem domain is gained
- Easy to exploit previous or alternate solutions
- Flexible building blocks for hybrid applications
- Substantial history and range of use



When to Use a GA

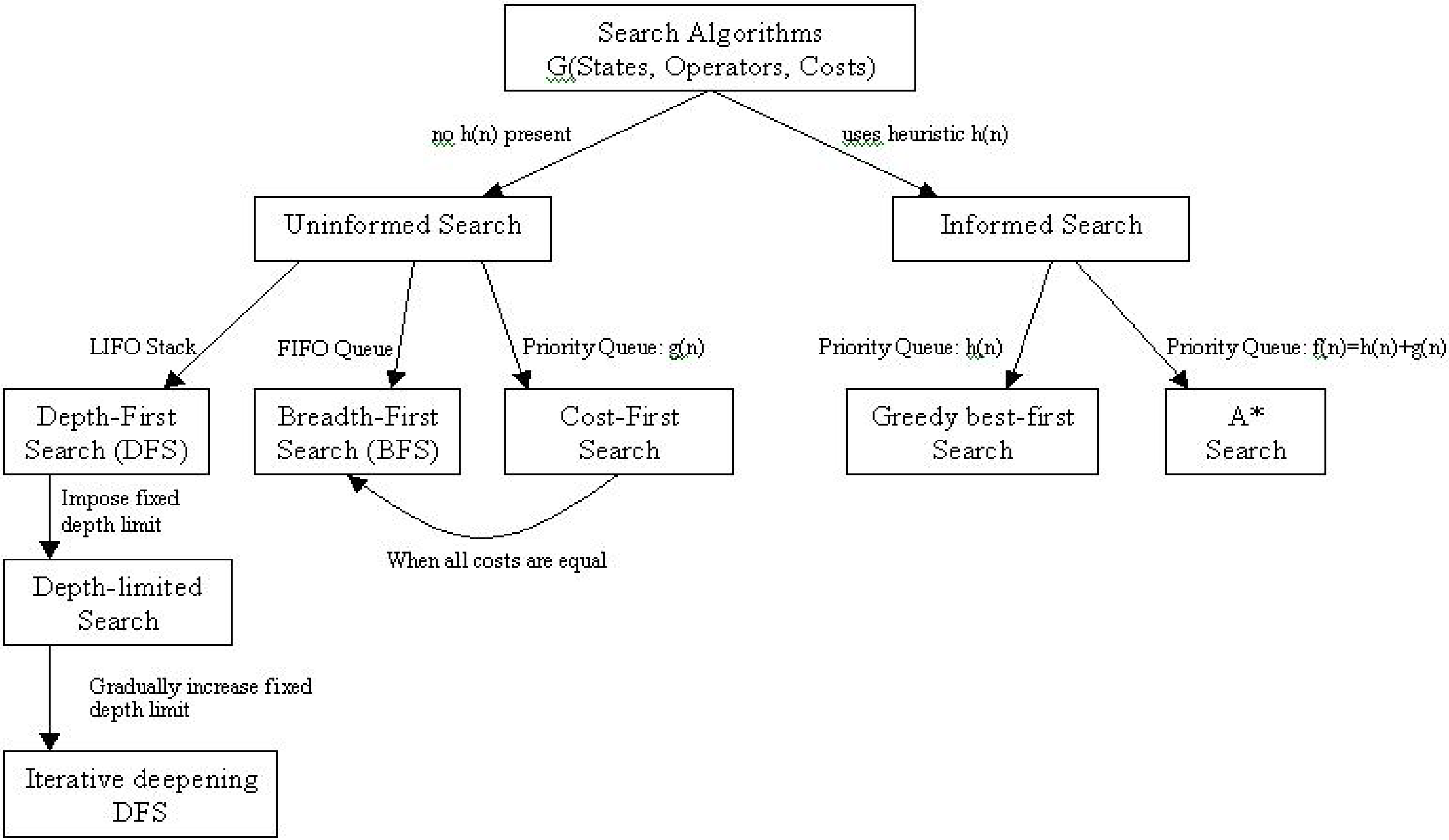
- Alternate solutions are too slow or overly complicated
- Need an exploratory tool to examine new approaches
- Problem is similar to one that has already been successfully solved by using a GA
- Want to hybridize with an existing solution
- Benefits of the GA technology meet key problem requirements



Genetic Algorithm

- Genetic algorithms have been applied to a wide range of problems.
- Results are sometimes very good and sometimes very poor.
- The technique is relatively easy to apply and in many cases it is beneficial to see if it works before thinking about another approach.

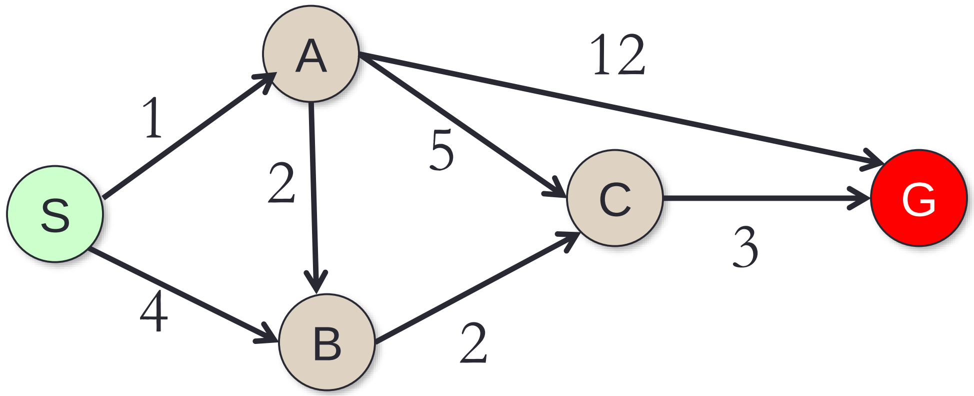
Taxonomy of Search Algorithms



Applications

Find the shortest path? And then judge, does the attached heuristic admissible?

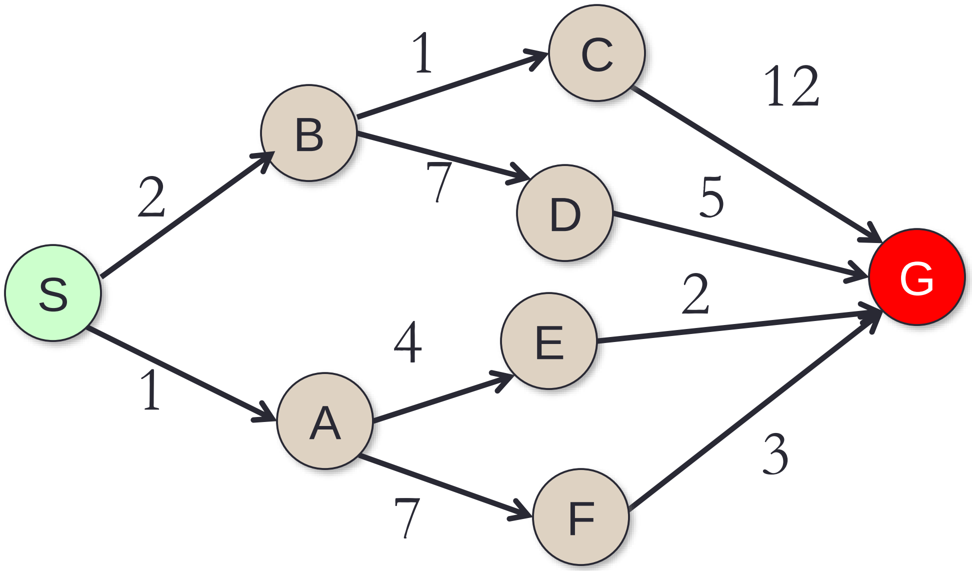
State	*h*
S	7
A	6
B	2
C	1
G	0



Applications

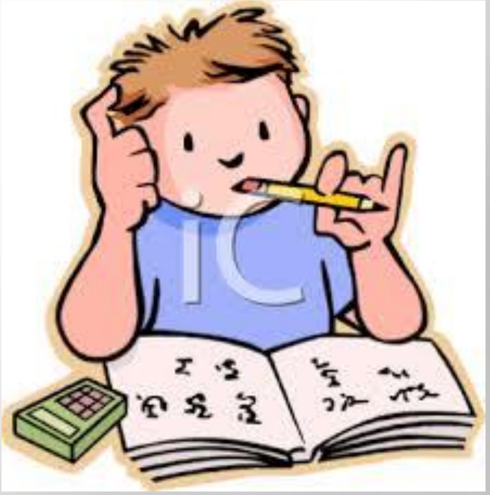
Find the shortest path? And then judge, does the attached heuristic admissible?

State	*h*
A	5
B	6
C	4
D	15
E	5
F	8



EXERCISES

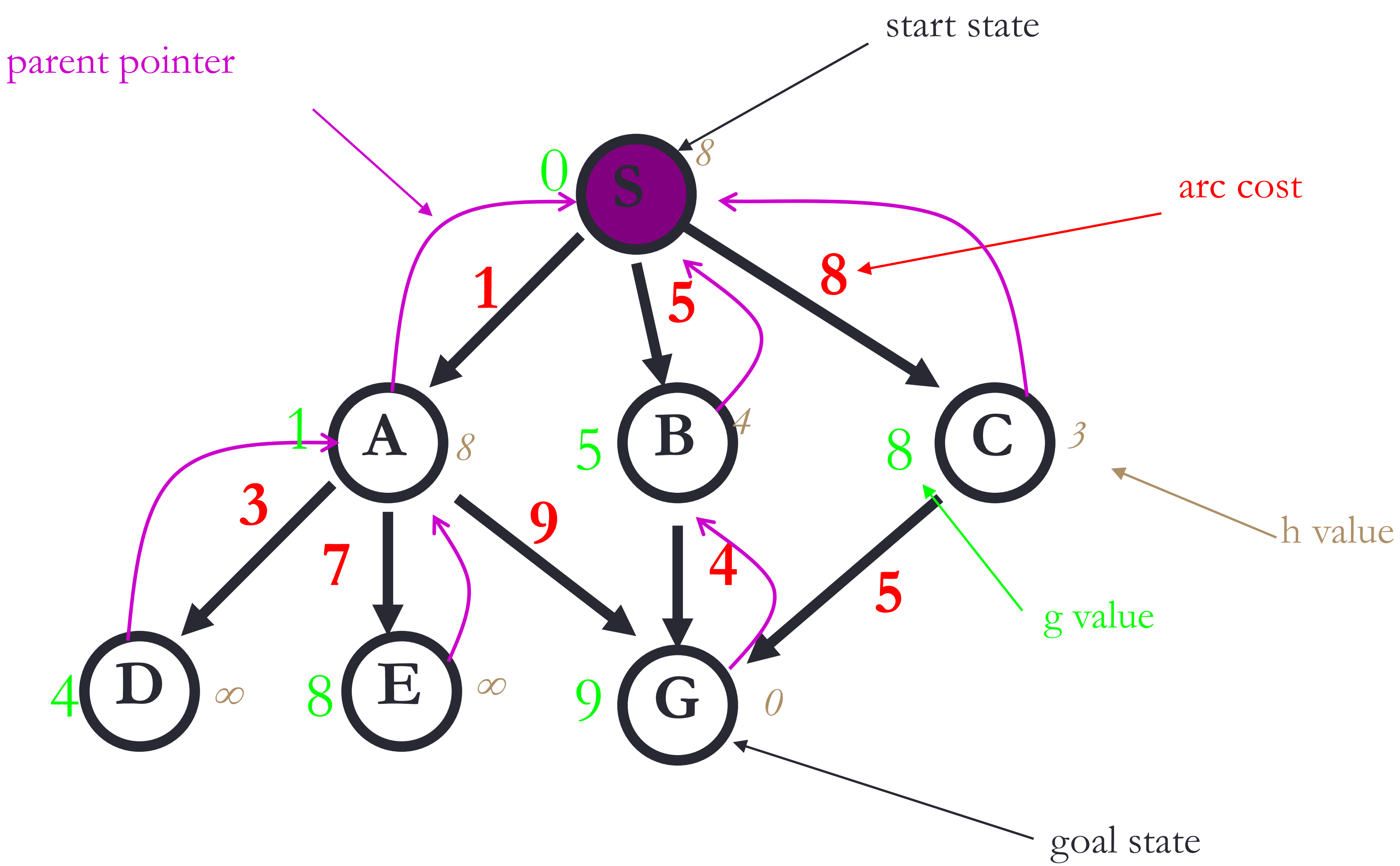
- ❖ Please try to solve the following exercises:
  - 4.1
  - 4.3
  - 4.5



- ❖ Program Task:
  - The 8-Queens Puzzle


Any questions?

Example


parent pointer
start state
arc cost
h value
g value
goal state
0
S
8
1
5
8
1
A
8
5
B
4
8
C
3
3
7
9
4
5
4
D
∞
8
E
∞
9
G
0