323
ARTIFICIAL INTELLIGENCE & EXPERT SYSTEMS

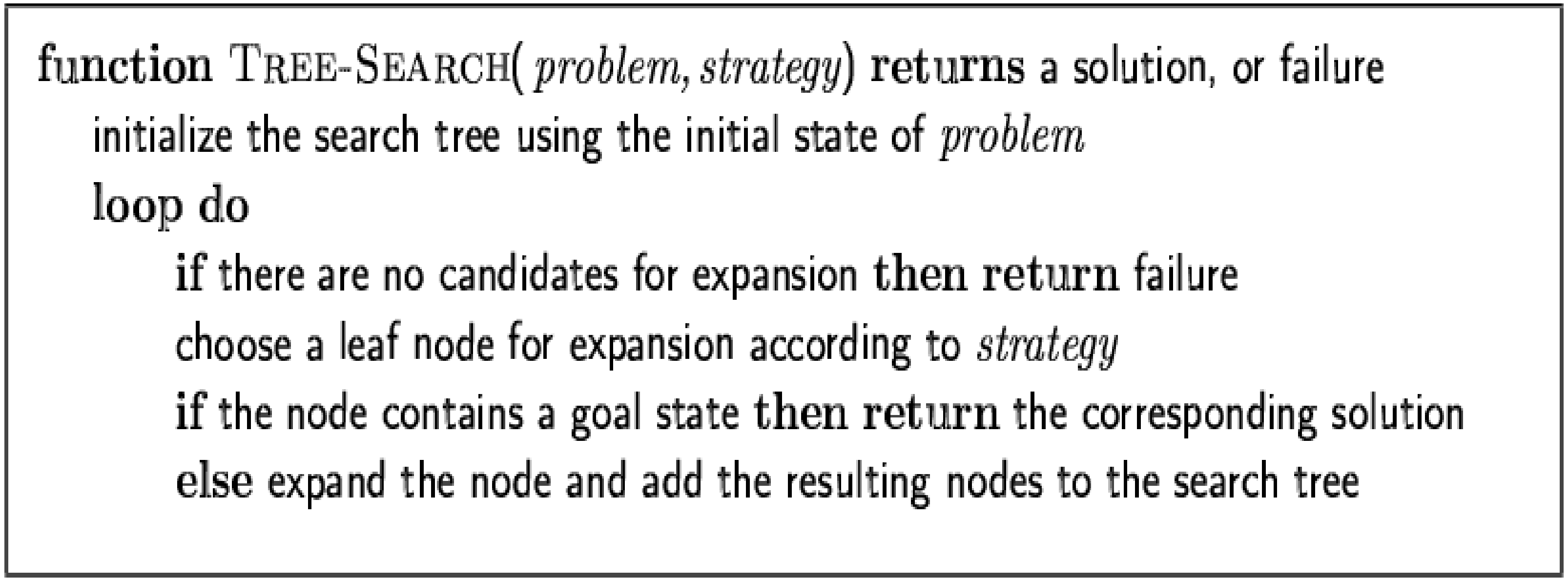
Informed search algorithms

Chapter 4

Outline

- Best-first search
  - Greedy best-first search
  - A^* search
- Heuristics
- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms

Review: Tree search



```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

- A search strategy is defined by picking the order of node expansion.
- Uninformed search strategies can find solutions to problems by generating new states and testing them against the goal.
- But, these algorithms are inefficient in most cases.

Informed search algorithms

- Informed search strategy uses problem-specific knowledge beyond the definition of the problem itself.
- Informed search can find solutions more efficiently than an uninformed search.
- Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm.
  - a node is selected for expansion based on an evaluation function, $f(n)$.
  - The node with the lowest evaluation is selected for expansion, because the evaluation measures distance to the goal.

Best-first search

- There are various Best-first search algorithms with different evaluation functions.
- A key component of these algorithms is a heuristic function:

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

- if *n* is a goal node, then *h(n)=0.*

Best-first search

- Idea: use an evaluation function *f(n)* for each node
  - estimate of "desirability"
  - → Expand most desirable unexpanded node

- Implementation:
  Order the nodes in fringe in decreasing order of desirability

- Special cases:
  - greedy best-first search
  - A^* search

Best-first search

```
1. Start with OPEN containing just the initial state.
2. Until a goal is found or there are no nodes left
   on OPEN do:
   (a) Pick the best node on OPEN.
   (b) Generate its successors.
   (c) For each successor do:
       i. If it has not been generated before, evaluate it,
          add it to OPEN, and record its parent.

       ii. If it has been generated before, change the parent
           if this new path is better than the previous one. In
           that case, update the cost of getting to this node
           and to any successors that this node may already
           have.
```

Greedy best-first search

- Greedy best first search tries to expand the node that is closest to the goal.
- It evaluates nodes by using just the heuristic function: $f(n) = h(n)$ (heuristic)
  - e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest
  - one might estimate the cost of the cheapest path from n to Bucharest via the SLD.

Greedy best-first search

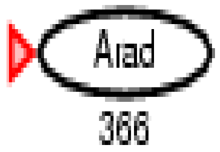
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Values of h_{SLD}-straight line distances to Bucharest

NOTE: the values of $\mathrm{h}_{\mathrm{SLD}}$ cannot be computed from the problem description itself.

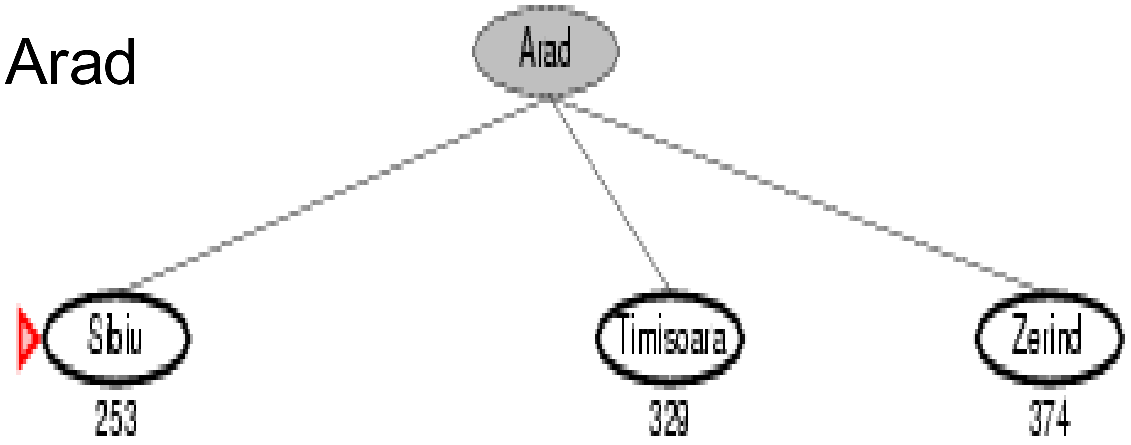
Greedy best-first search example

(a) The initial state



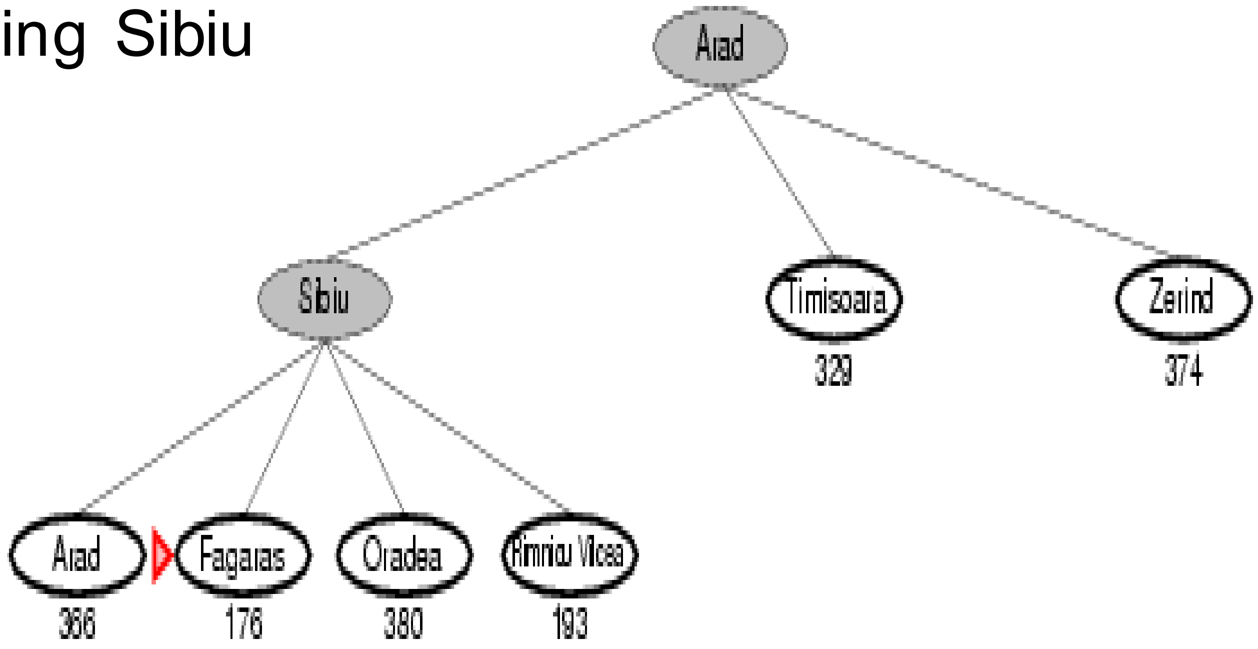
Greedy best-first search example

(b) After expanding Arad



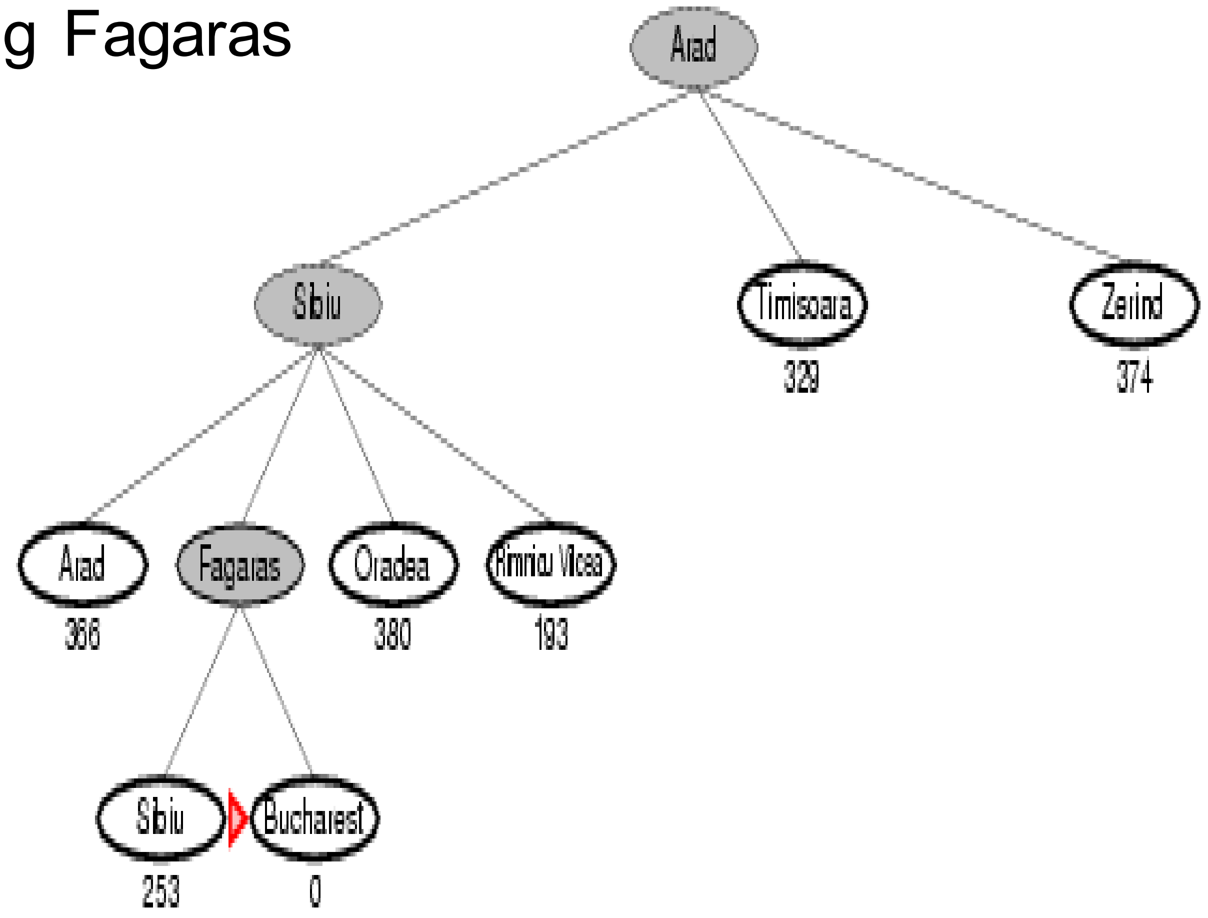
Greedy best-first search example

(c) After expanding Sibiu



Greedy best-first search example

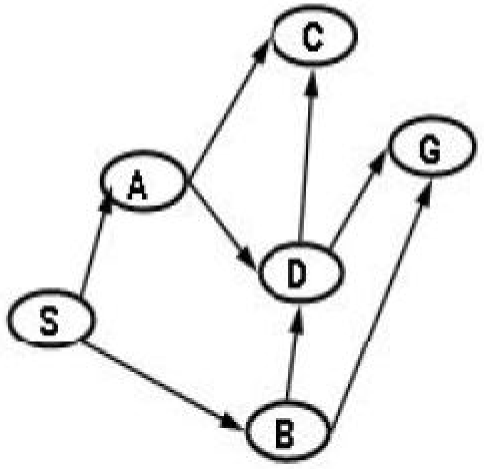
(d) After expanding Fagaras



GREEDY BEST-FIRST SEARCH – ANOTHER EXAMPLE

Pick "best " (by heuristic value) element of Q; Add path extensions anywhere in Q

	Q	Visited
1	(10 S)	S
2		
3		
4		
5		

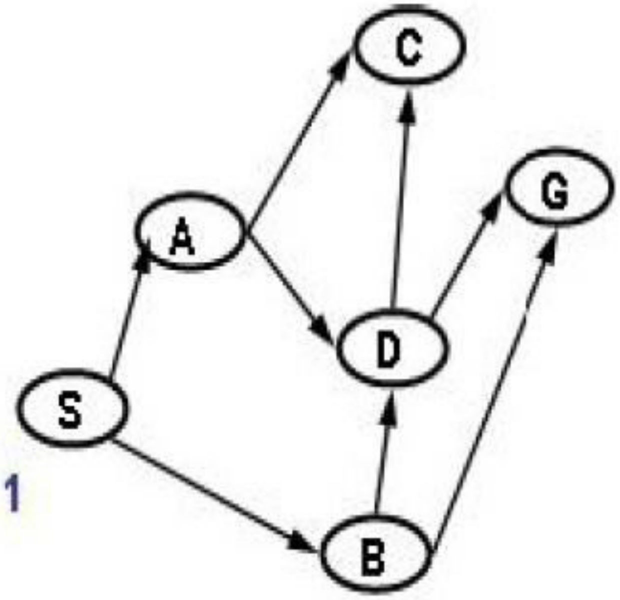


Heuristic Values

A=2	C=1	S=10
B=3	D=4	G=0

Added paths in blue; heuristic value of node's state is in front. We show the paths in reversed order; the node's state is the first entry.

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3		
4		
5		

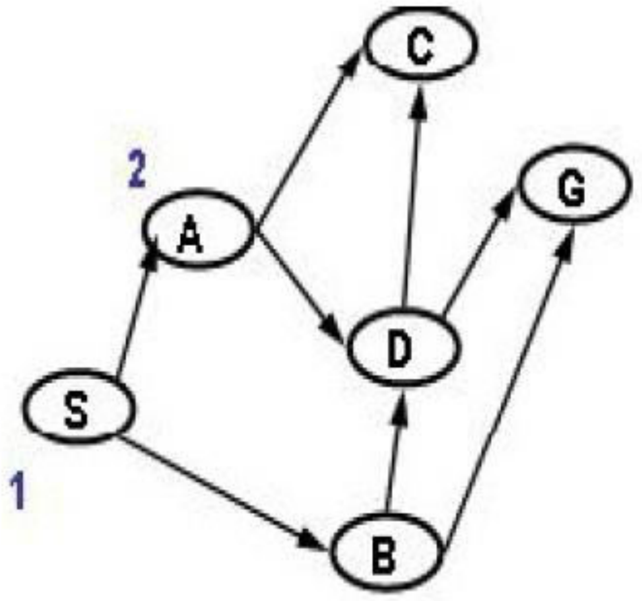


1

Heuristic Values

A=2	C=1	S=10
B=3	D=4	G=0

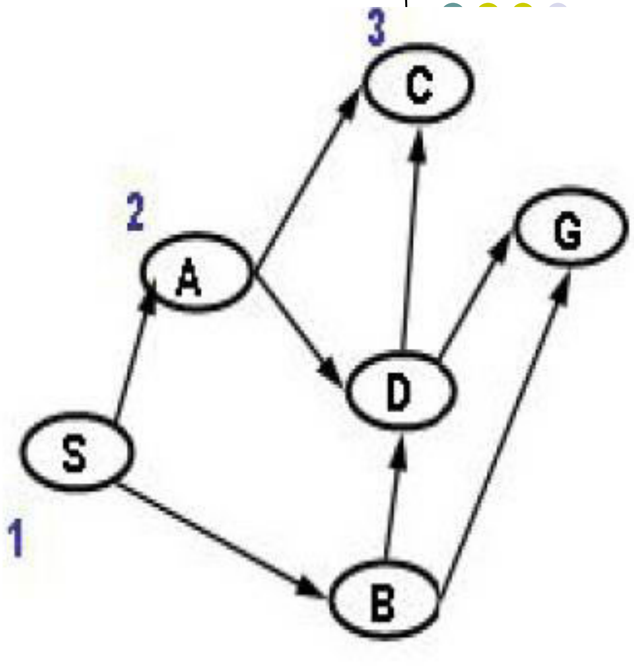
	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4		
5		



Heuristic Values

A=2	C=1	S=10
B=3	D=4	G=0

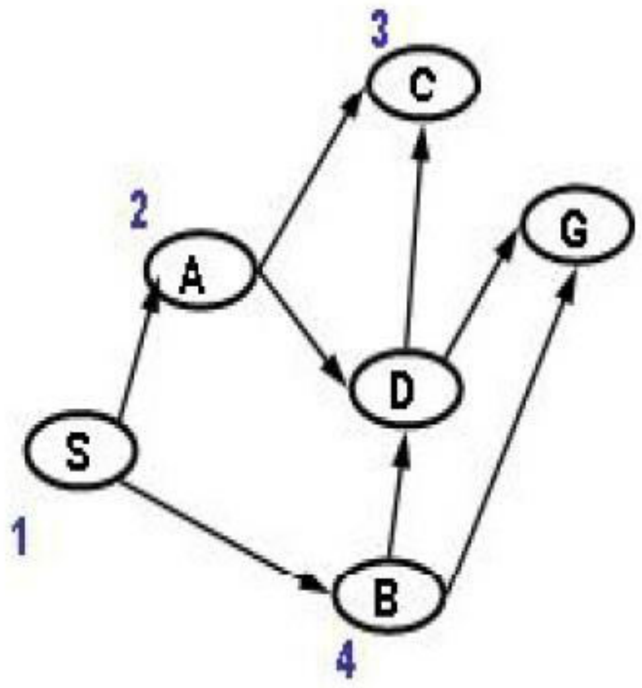
	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	(3 B S) (4 D A S)	C,D,B,A,S
5		



Heuristic Values

A=2	C=1	S=10
B=3	D=4	G=0

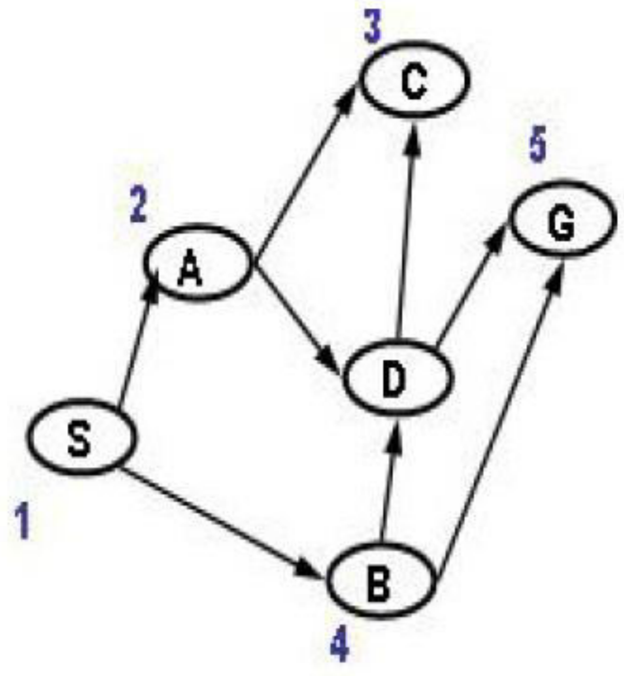
	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	(3 B S) (4 D A S)	C,D,B,A,S
5	(0 G B S) (4 D A S)	G,C,D,B,A,S



Heuristic Values

A=2	C=1	S=10
B=3	D=4	G=0

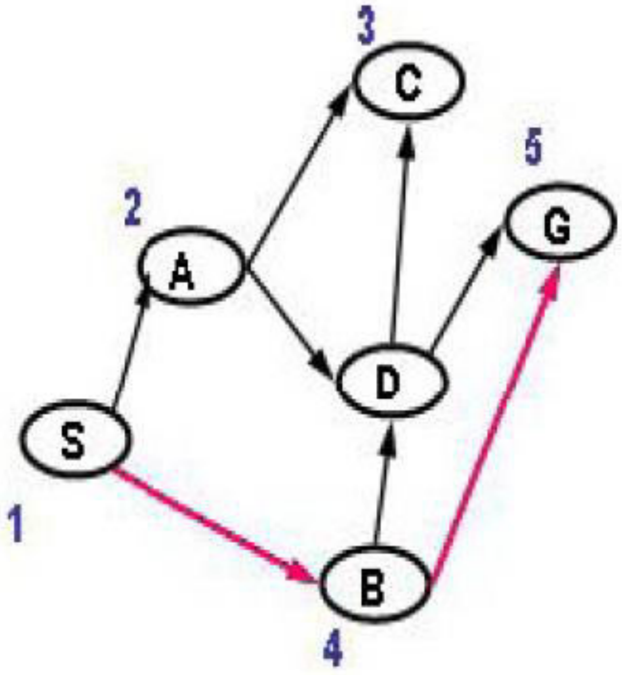
	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	(3 B S) (4 D A S)	C,D,B,A,S
5	(0 G B S) (4 D A S)	G,C,D,B,A,S



Heuristic Values

A=2	C=1	S=10
B=3	D=4	G=0

	Q	Visited
1	(10 S)	S
2	(2 A S) (3 B S)	A,B,S
3	(1 C A S) (3 B S) (4 D A S)	C,D,B,A,S
4	(3 B S) (4 D A S)	C,D,B,A,S
5	$\boxed{\text{(0 G B S)}}$ (4 D A S)	G,C,D,B,A,S



Heuristic Values

A=2	C=1	S=10
B=3	D=4	G=0

Properties of greedy best-first search

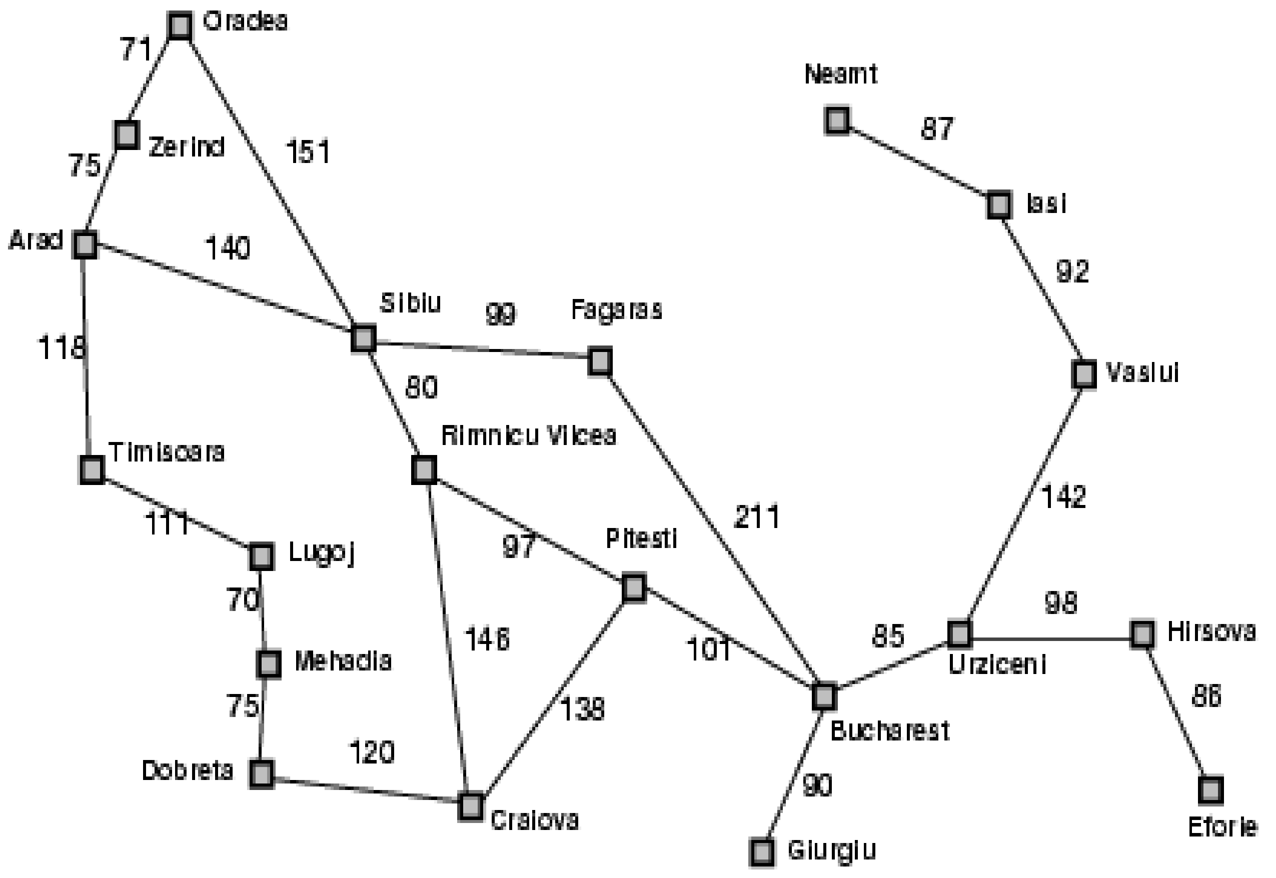
- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →

  Complete in finite space with repeated-state checking
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement – *m: max. depth*
- Space? $O(b^m)$ – keeps all nodes in memory
- Optimal? No – it can start down an infinite path and never return to try other possibilities.

A^* search

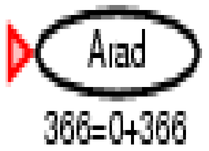
- Idea: minimizing the total estimated solution cost.
- Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = the cost to reach the node n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal

Romania with step costs in km

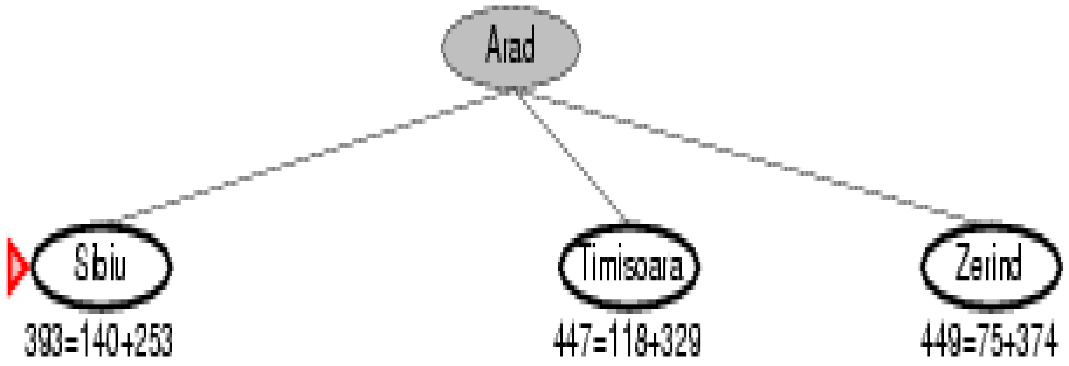


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

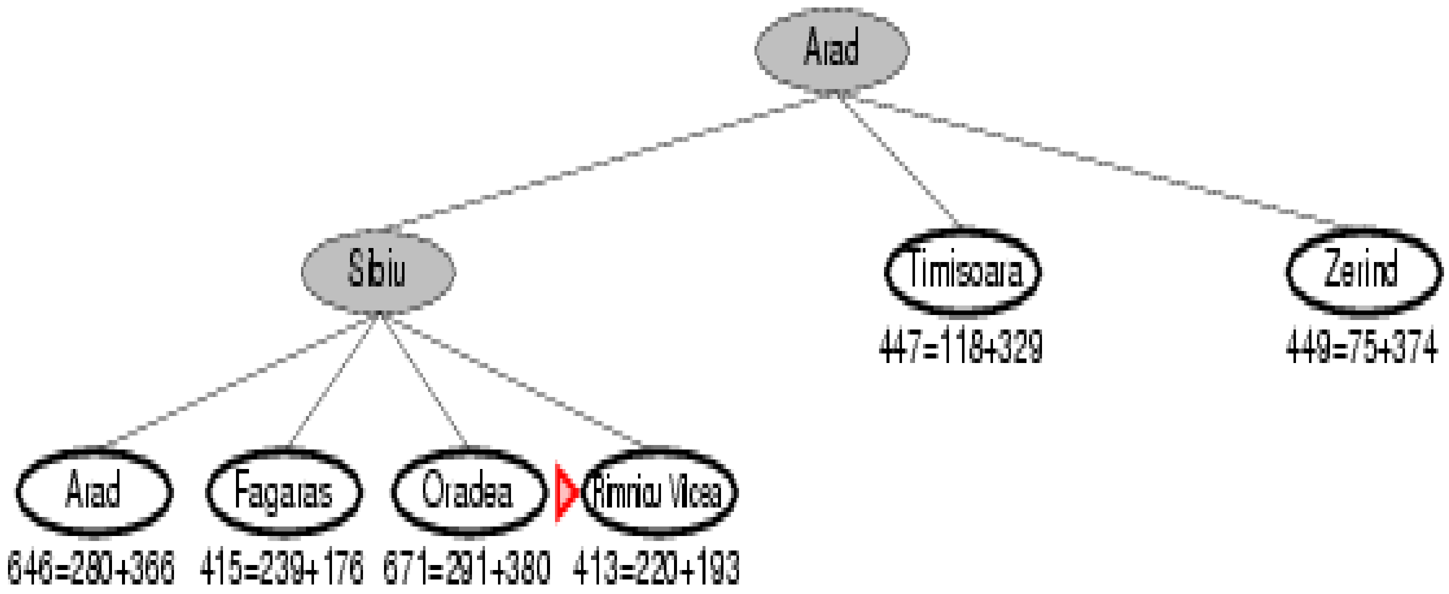
A* search example



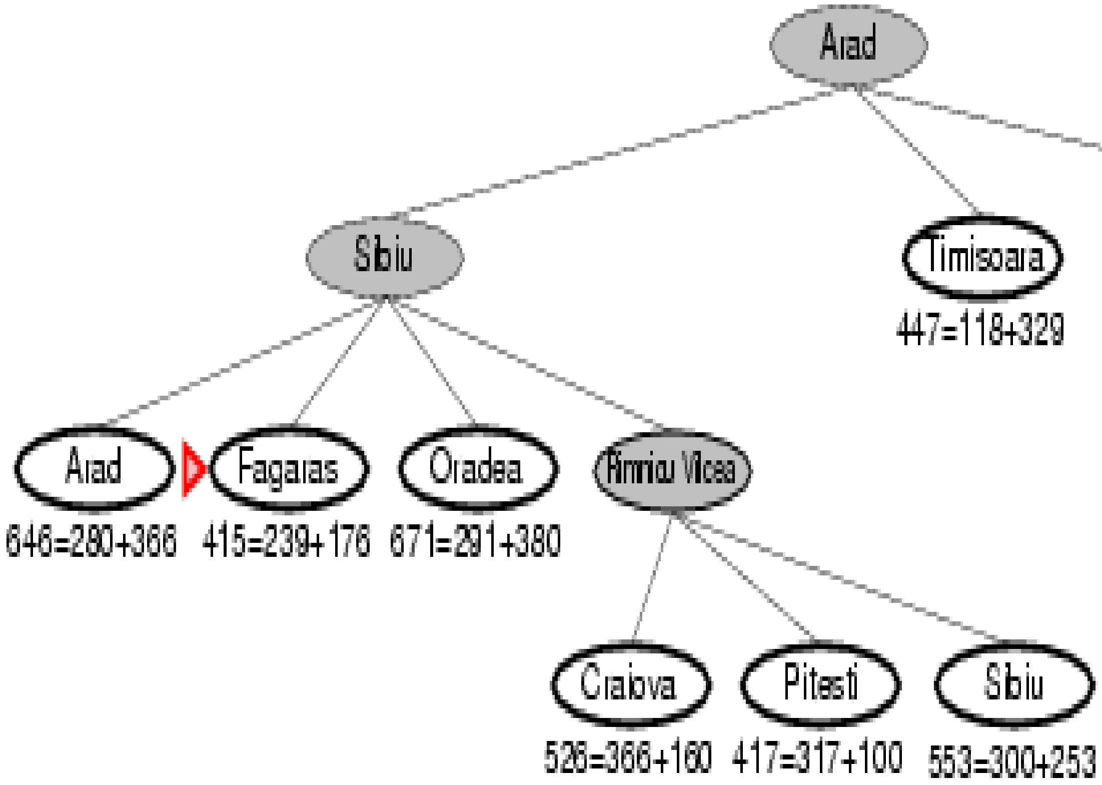
A* search example



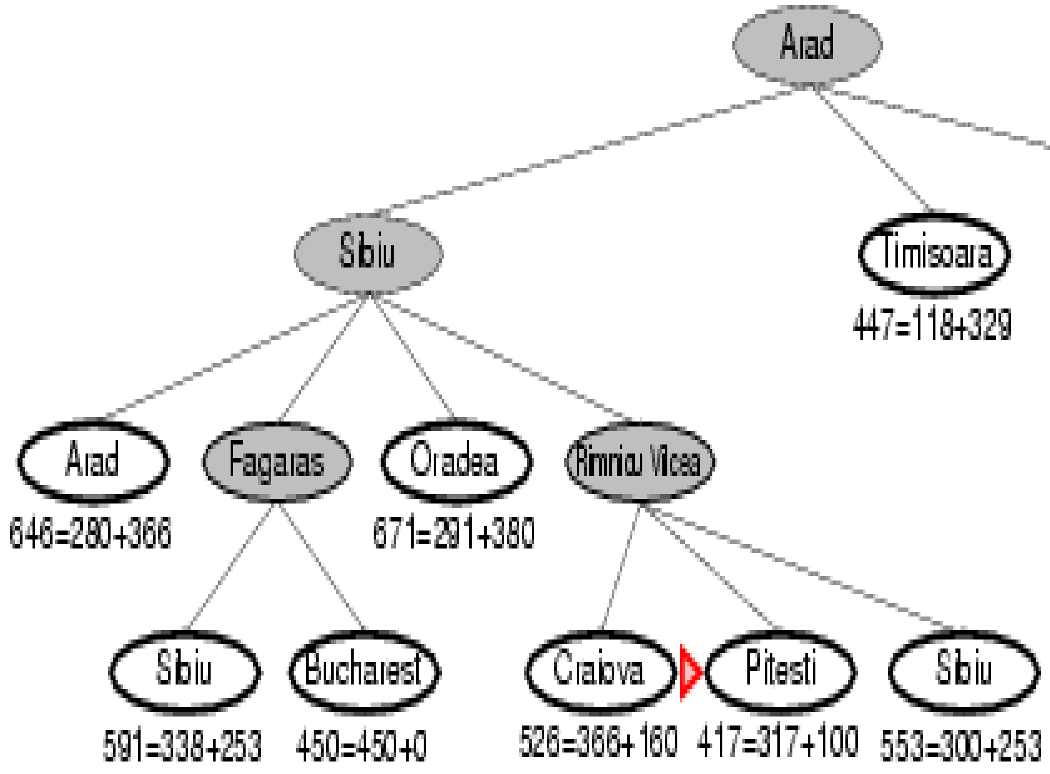
A* search example



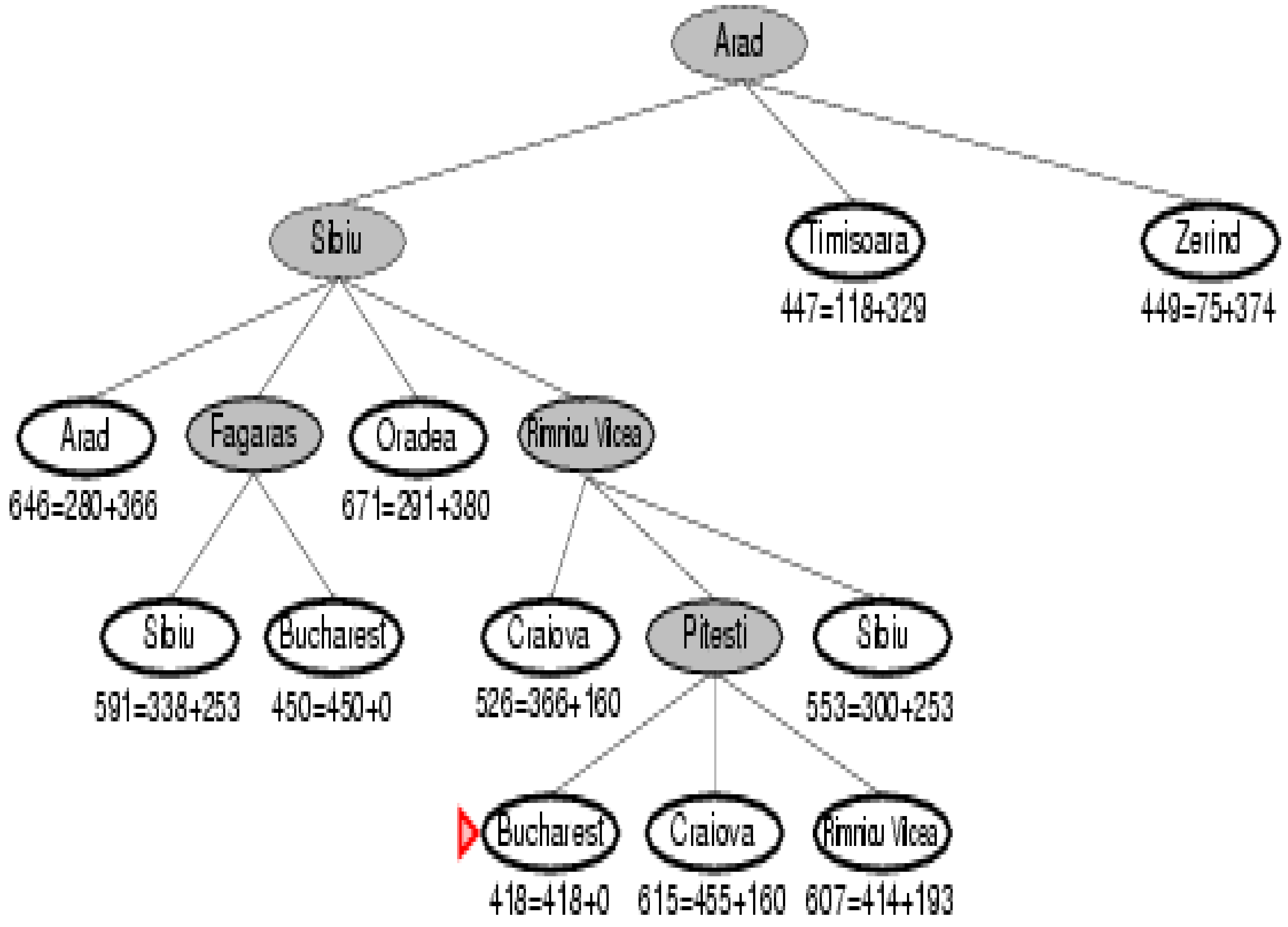
A* search example



A^* search example



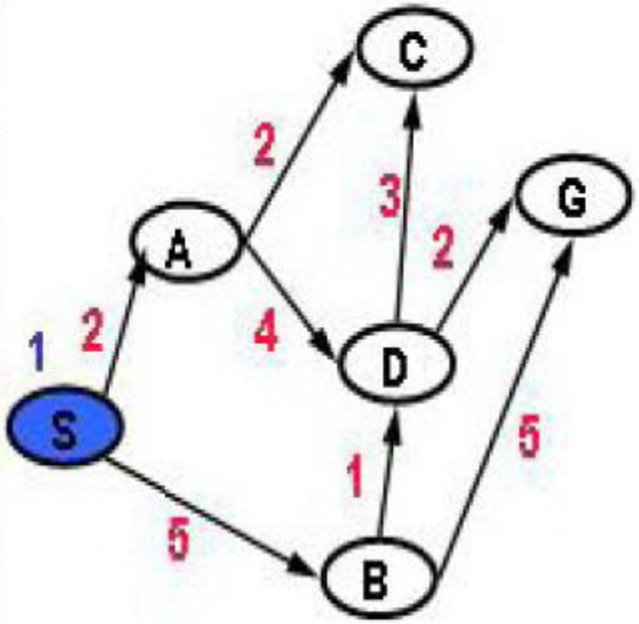
A* search example



A* SEARCH – ANOTHER EXAMPLE

Pick best (by path length+heuristic) element of Q; Add path extensions anywhere in Q

	Q
1	(0 S)

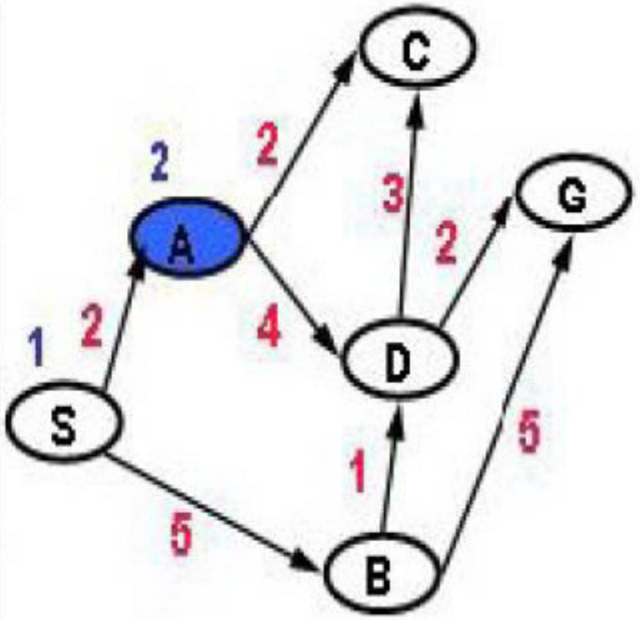


Heuristic Values

A=2	C=1	S=0
B=3	D=1	G=0

Added paths in blue; underlined paths are chosen for extension.
We show the paths in reversed order; the node's state is the first entry.

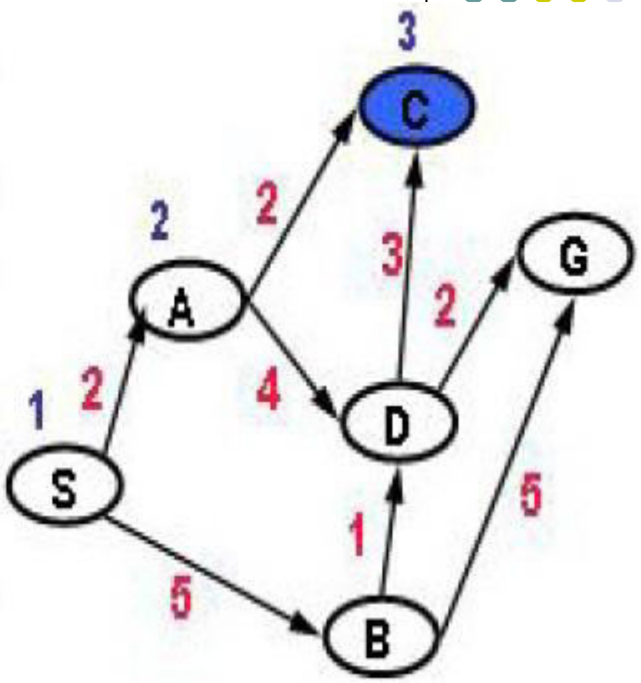
	Q
1	(0 S)
2	(4 A S) (8 B S)



Heuristic Values

A=2	C=1	S=0
B=3	D=1	G=0

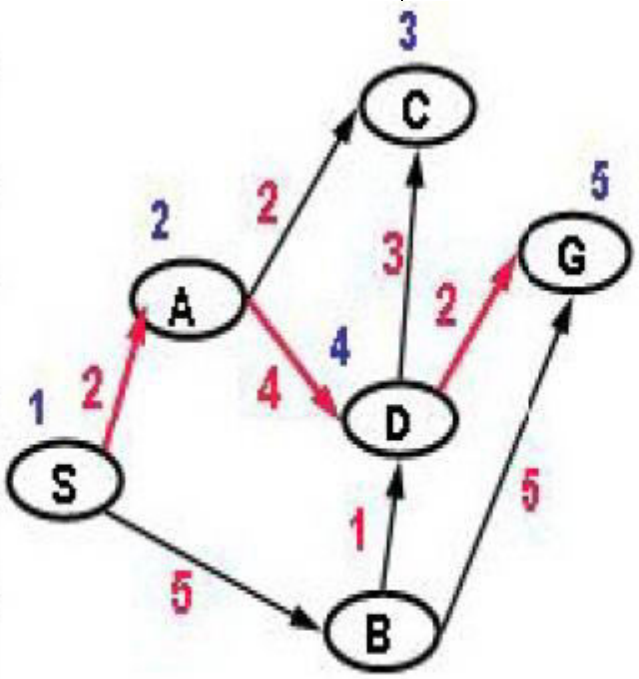
	Q
1	(0 S)
2	(4 A S) (8 B S)
3	(5 C A S) (7 D A S) (8 B S)



Heuristic Values

A=2	C=1	S=0
B=3	D=1	G=0

	Q
1	(0 S)
2	(4 A S) (8 B S)
3	(5 C A S) (7 D A S) (8 B S)
4	(7 D A S) (8 B S)
5	(8 G D A S) (10 C D A S) (8 B S)



Heuristic Values

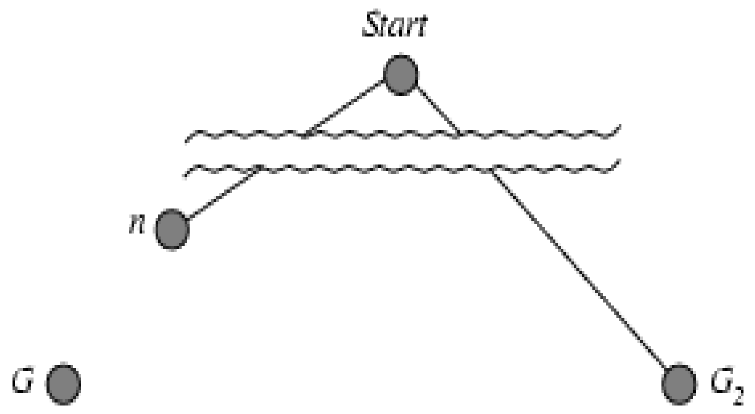
A=2	C=1	S=0
B=3	D=1	G=0

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 (thinks that the cost of solving the problem is less than it actually is)
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance – because the shortest path between any two points is a straight line.)
- Theorem: If $h(n)$ is admissible, A* using **TREE-SEARCH** is optimal

Optimality of A* (proof)

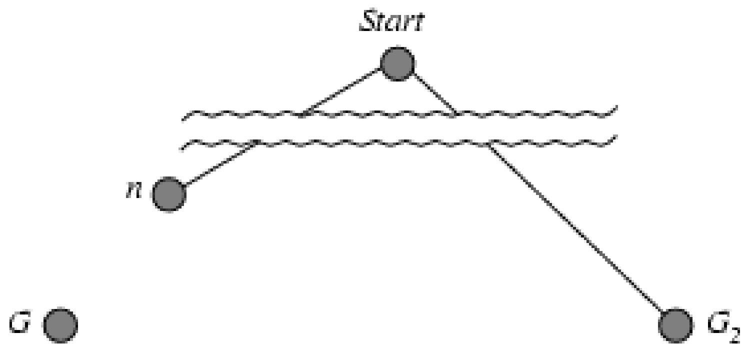
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*.



- $f(G_2) = g(G_2)$ since $h(G_2) = 0$ (true for any goal node)
- $g(G_2) > g(G)$ since G_2 is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above

Optimality of A* (proof)

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



- **$f(G_2)$ $> f(G)$** from above
- $h(n) \le h^*(n)$ since h is admissible
- $g(n) + h(n) \le g(n) + h^*(n)$
- **$f(n) \le f(G)$**

Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion

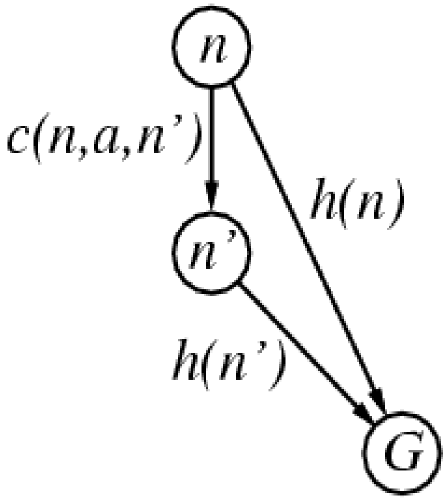
Consistency

- If we use the GRAPH-SEARCH algorithm instead of TREE-SEARCH, then this proof breaks down.
- GRAPH-SEARCH can discard the optimal path if it is not the first one generated.
- So, suboptimal solutions can be returned.
- To fix this problem, the solution is to ensure that the optimal path to any repeated state is always the first one followed – as in the case with uniform-cost search.
  - we impose an extra requirement on $h(n)$ – consistency (monotonicity)

Consistent heuristics

- A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*,
  $h(n) \le c(n,a,n') + h(n')$ (triangle inequality)

(each side of a triangle cannot be longer than the sum of the other two sides.)



- If *h* is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\ge g(n) + h(n) \\ &= f(n) \end{aligned}$$

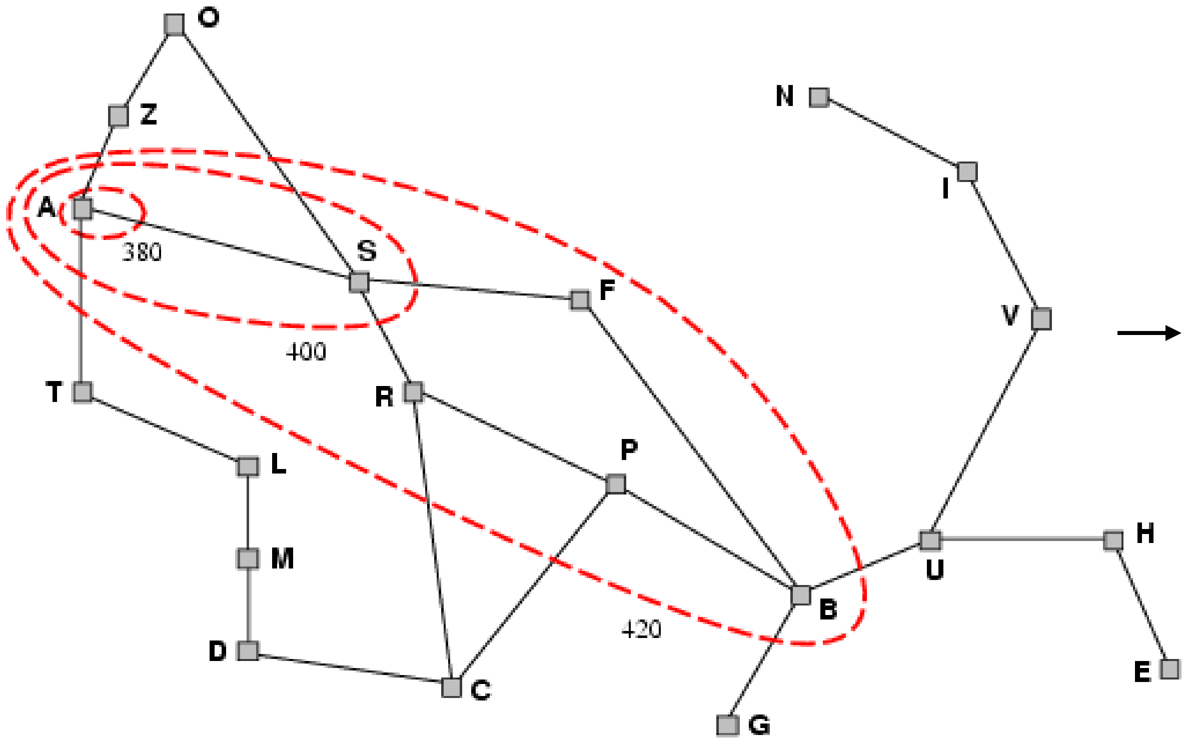
- i.e., *f(n)* is non-decreasing along any path.
- Theorem: If *h(n)* is consistent, A* using `GRAPH-SEARCH` is optimal

Consistent heuristics

- Consistency is a stricter requirement than addmissibility.
  - all the admissible heuristics we discussed so far are also consistent.
  - e.g. h_{SLD} - general triangle inequality is satisfied when each side is measured by SLD.
- Because the sequence of nodes expanded by A* using GRAPH-SEARCH is in nondecreasing order of $f(n)$, the first goal node selected for expansion must be an optimal solution.

Optimality of A*

- A* expands nodes in order of increasing *f*-values.
- Gradually adds "*f*-contours" of nodes in the state space.
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$



→ Map of Romania showing contours at *f=380, f=400, f=420*

Nodes inside a given contour have f-costs less than or equal to the contour value.

Properties of A*

- Complete? Yes
- Time? Exponential
- Space? Keeps all nodes in memory
- Optimal? Yes

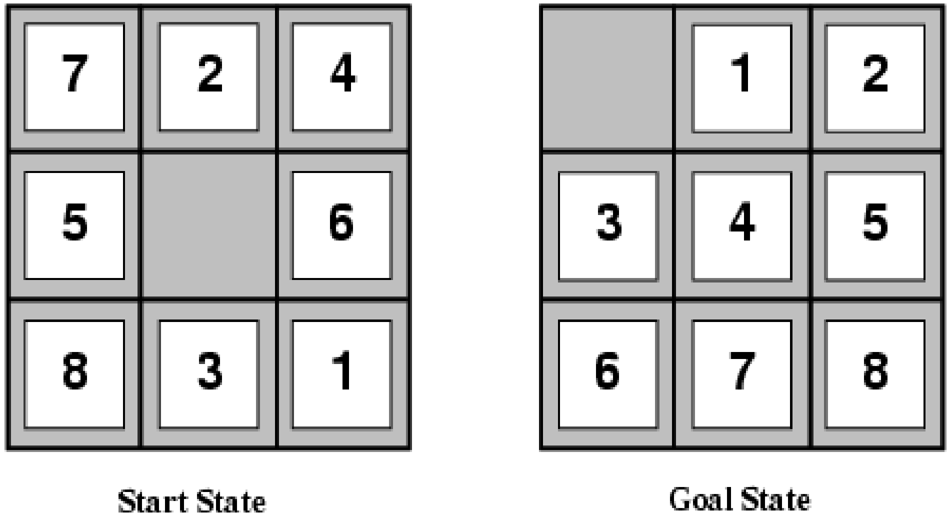
Heuristic Functions

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)

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



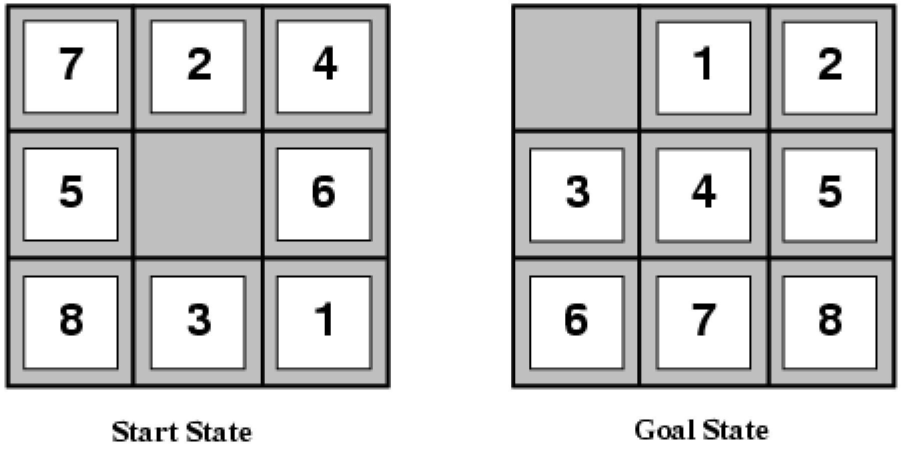
The solution is 26 steps long.

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)



- $h_1(S) = ?$ 8
- $h_2(S) = ?$ 3+1+2+2+3+2+2+3 = 18
- Neither of these overestimates the true solution cost.

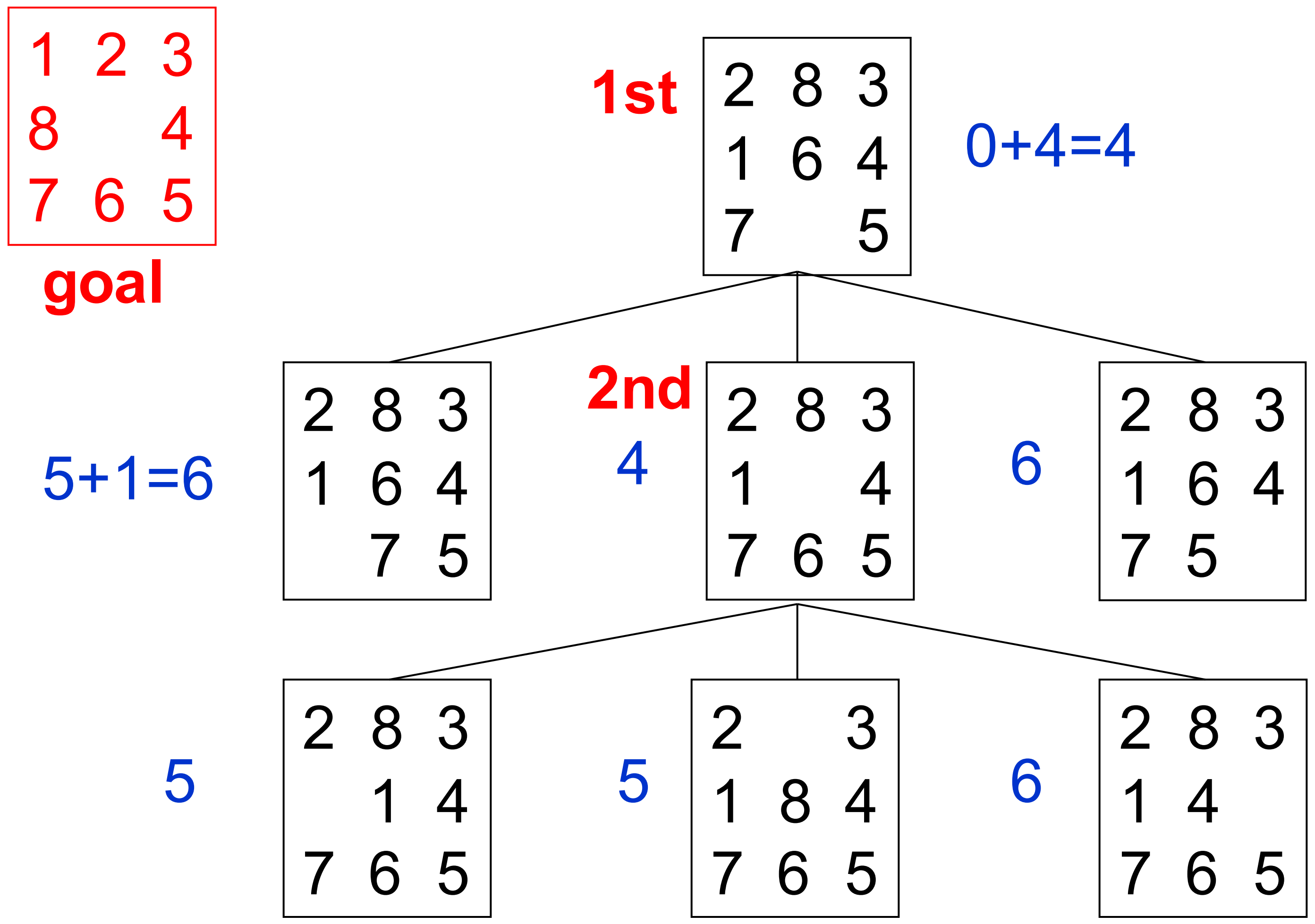
Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)
- then h_2 dominates h_1
- h_2 is better for search

- Typical search costs (average number of nodes expanded):

- $d=12$ IDS = 3,644,035 nodes
  $A^*(h_1)$ = 227 nodes
  $A^*(h_2)$ = 73 nodes
- $d=24$ IDS = too many nodes
  $A^*(h_1)$ = 39,135 nodes
  $A^*(h_2)$ = 1,641 nodes

8 Puzzle Using Number of Misplaced Tiles with A* Search



Relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution
- **Key Point** : the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem.

Inventing admissible heuristic functions

- If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically (ABSOLVER)
- If 8-puzzle is described as
  - A tile can move from square A to square B if
  - A is horizontally or vertically adjacent to B and B is blank

Inventing admissible heuristic functions

- A relaxed problem can be generated by removing one or both of the conditions

  (a) A tile can move from square A to square B if A is adjacent to B

  (b) A tile can move from square A to square B if B is blank

  (c) A tile can move from square A to square B
- h2 can be derived from (a) – h2 is the proper score if we move each tile into its destination
- h1 can be derived from (c) – it is the proper score if tiles could move to their intended destination in one step

SUMMARY

- Heuristic functions estimate costs of shortest paths.
- Good heuristics can dramatically reduce search cost.
- Greedy best-first search expands lowest h.
  - incomplete and not always optimal.
- A* search expands lowest g + h.
  - complete and optimal.
  - also optimally efficient.
- Admissible heuristics can be derived from exact solution of relaxed problems.

Local search algorithms

- The search algorithms we have seen so far are systematic.
- This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path and which have not.
- When a goal is found, the path to that goal is also a part of a solution to the problem.
- In many problems, the path to the goal is irrelevant.
- If the path to the goal does not matter, we might consider a different class of algorithms – local search algorithms.

Local search algorithms

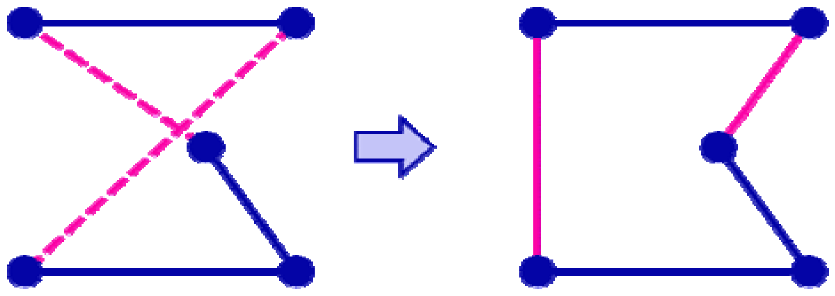
- Local search algorithms work by keeping in memory just one current state (or perhaps a few), moving around the state space based on purely local information.
- The paths followed by the search are not retained.
- Local search algorithms are not systematic,
- They have two advantages:

  (1) they use very little memory—usually a constant amount;

  (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

Example: Travelling Salesperson Problem

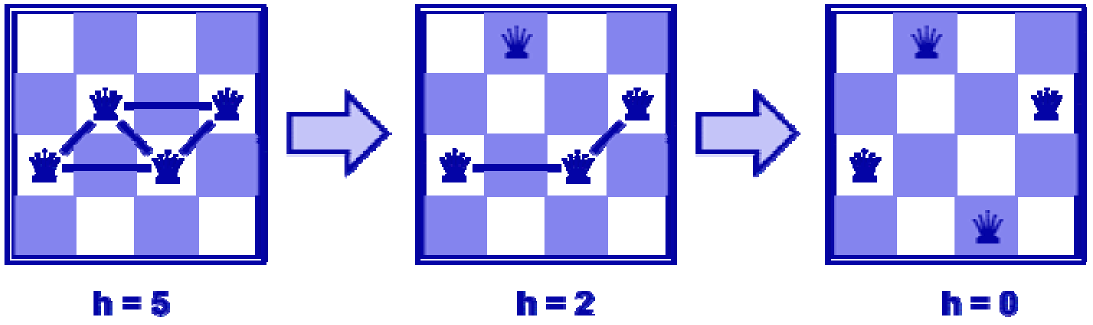
- In addition to finding goals, local search algorithms are useful for solving optimization problems.
  - the aim is to find the best state according to an objective function.
  - start with any complete tour, perform pairwise exchanges



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

Example: *n*-queens

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal
- What matters is the final configuration of queens, not the order in which they are added.
- Local search: start with all *n*, move a queen to reduce conflicts



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

Generate-and-Test

Algorithm

1. Generate a possible solution.
2. Test to see if this is actually a solution.
3. Quit if a solution has been found. Otherwise, return to step 1.

Hill Climbing Search

- Hill Climbing search is simply a loop that continually moves in the direction of increasing value – uphill.
- It terminates when it reaches a "peak" where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the current node only records the state and objective function value.

Hill Climbing

- Searching for a goal state = Climbing to the top of a hill
- Generate-and-test + direction to move.
- Heuristic function to estimate how close a given state is to a goal state.

Simple Hill Climbing

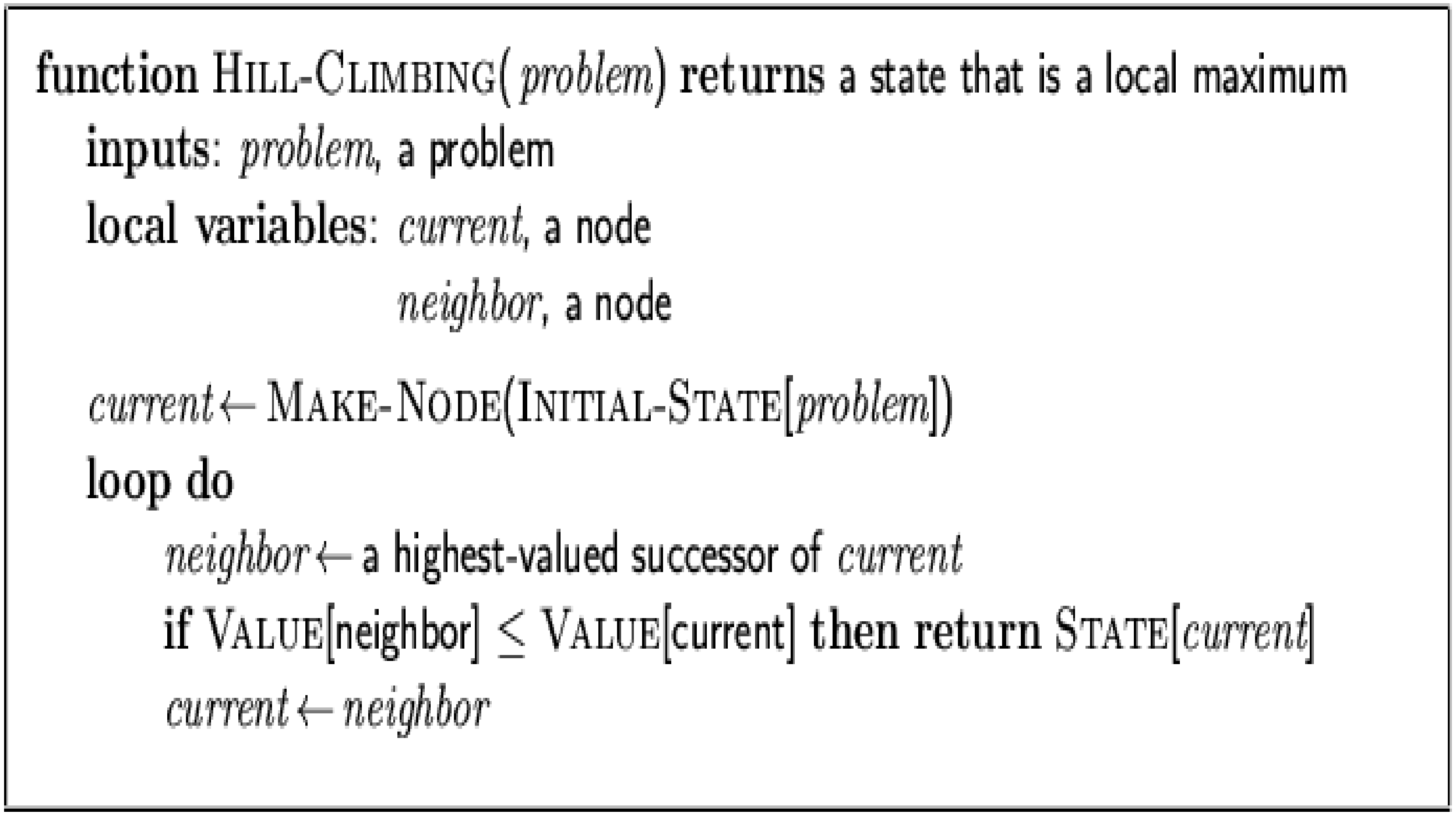
Algorithm

1. Evaluate the initial state.
2. Loop until a solution is found or there are no new operators left to be applied:
   - Select and apply a new operator
   - Evaluate the new state:

     goal $\rightarrow$ quit

     better than current state $\rightarrow$ new 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
```

At each step the current node is replaced by the best neighbor → the neighbor with the highest value.

Simple Hill Climbing

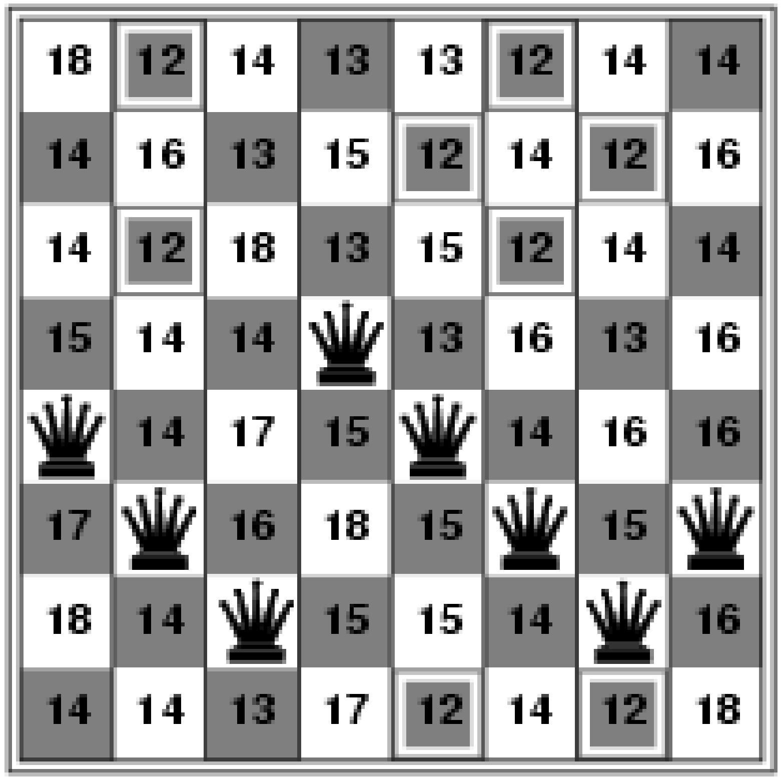
Example: coloured blocks

Heuristic function: the sum of the number of different colours on each of the four sides (solution = 16).

Hill-climbing search: 8-queens problem

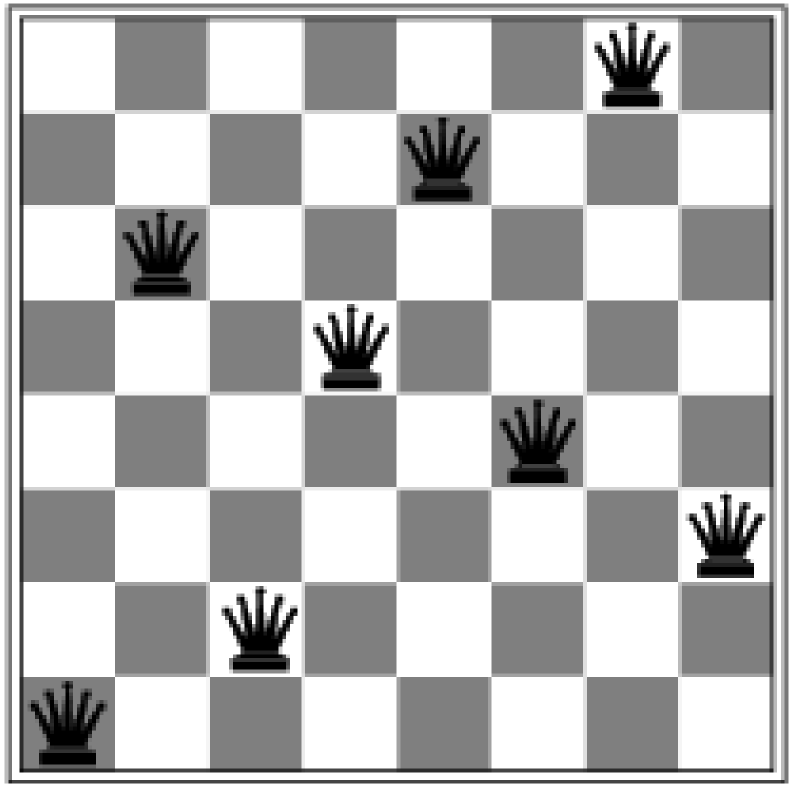
- Each state has 8 queens on the board, one per column.
- The successor function returns all possible states generated by moving a single queen to another square in the same column.
    - so, each state has 8x7=56 successors.
- *h* = number of pairs of queens that are attacking each other, either directly or indirectly
- The global minimum of this function is zero.

Hill-climbing search: 8-queens problem



- $h = 17$ for the above state
- The figure also shows the values of all its successors, with the best successors $h=12$.

Hill-climbing search: 8-queens problem



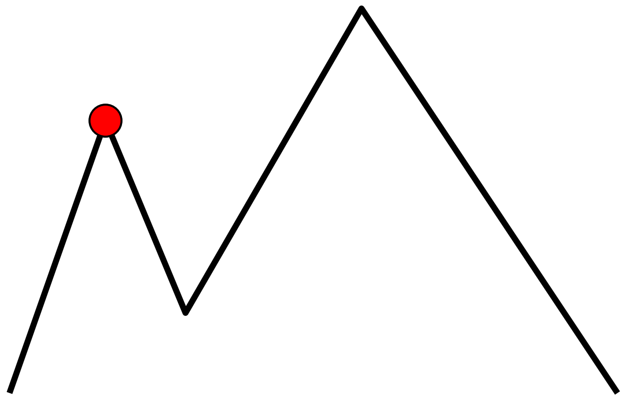
Hill-climbing algorithms choose randomly among the set of best successors, if there is more than one.

- A local minimum with *h* = *1* but every successor has a higher cost.

Hill Climbing: Disadvantages

Local maximum

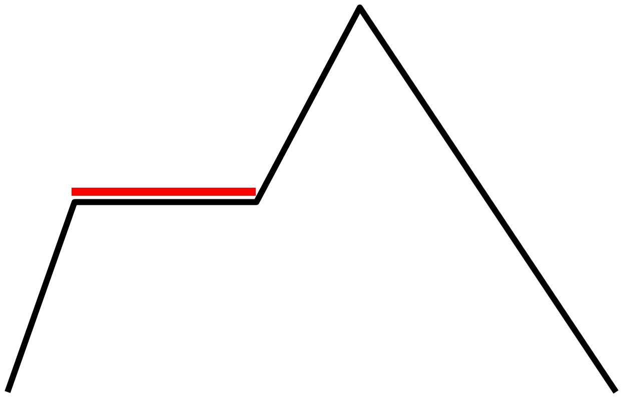
A state that is better than all of its neighbours, but not better than some other states far away.



Hill Climbing: Disadvantages

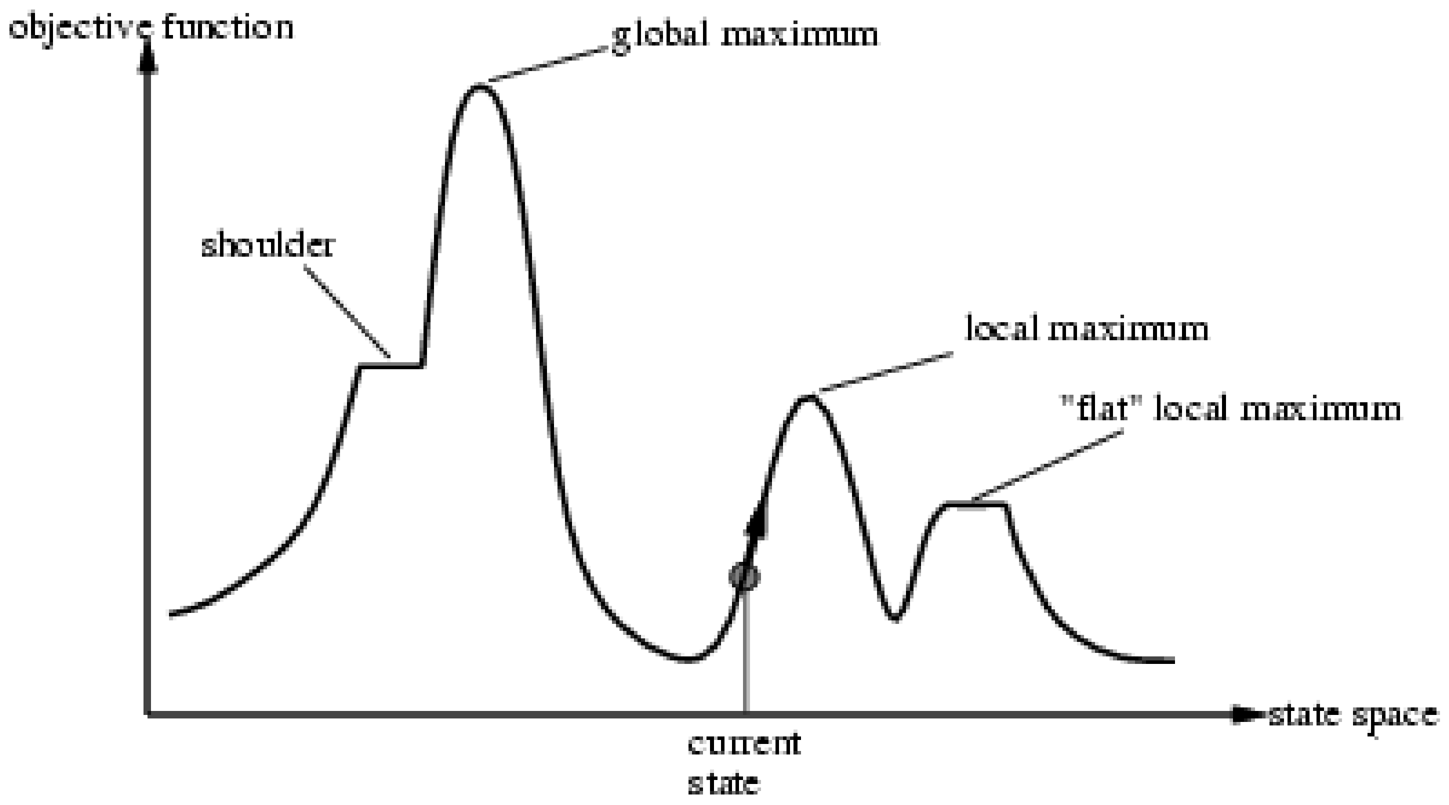
Plateau

A flat area of the search space in which all neighbouring states have the same value.



Hill-climbing search

- Hill-climbing search modifies the current state to try to improve it, as shown by the arrow
- Problem: depending on initial state, can get stuck in local maxima



Hill Climbing: Disadvantages

Ways Out

- Backtrack to some earlier node and try going in a different direction.

- Make a big jump to try to get in a new section.

- Moving in several directions at once.

Simulated annealing search

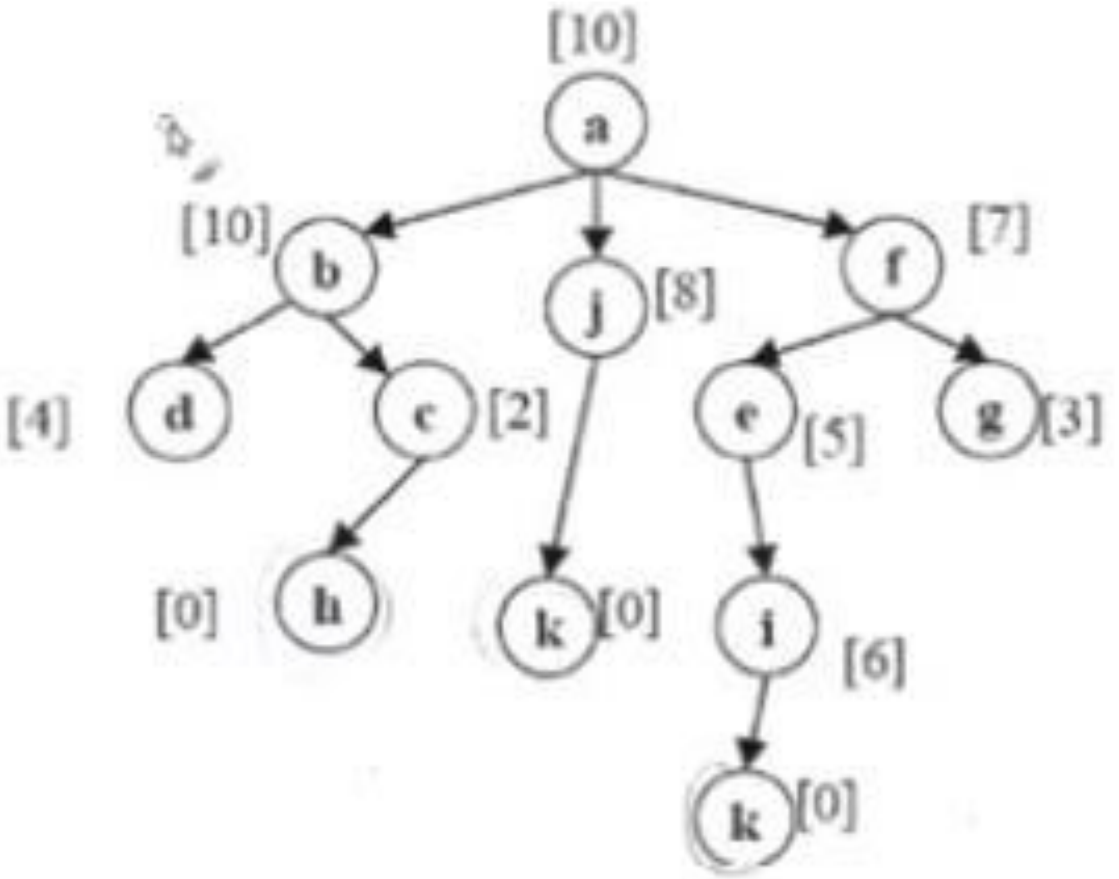
- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency.
- e.g. task of getting a ping-pong ball into the deepest crevice in a bumpy surface.
  - If we just let the ball roll, it will come to rest at a local minimum.
  - If we shake the surface, we can bounce the ball out of the local minimum.
- The trick is to shake just hard enough to bounce the ball out of local minima, but not hard enough get it from the global minimum.

Simulated annealing search

- The principle behind SA is similar to what happens when metals are cooled at a controlled rate
- The slowly decrease of temperature allows the atoms in the molten metal to line themselves up to form a regular crystalline structure that possesses a low density and a low energy.
- One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1.
- Widely used in VLSI layout, airline scheduling, etc

Local beam search

- Keep track of *k* states rather than just one
- Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated
- If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat.

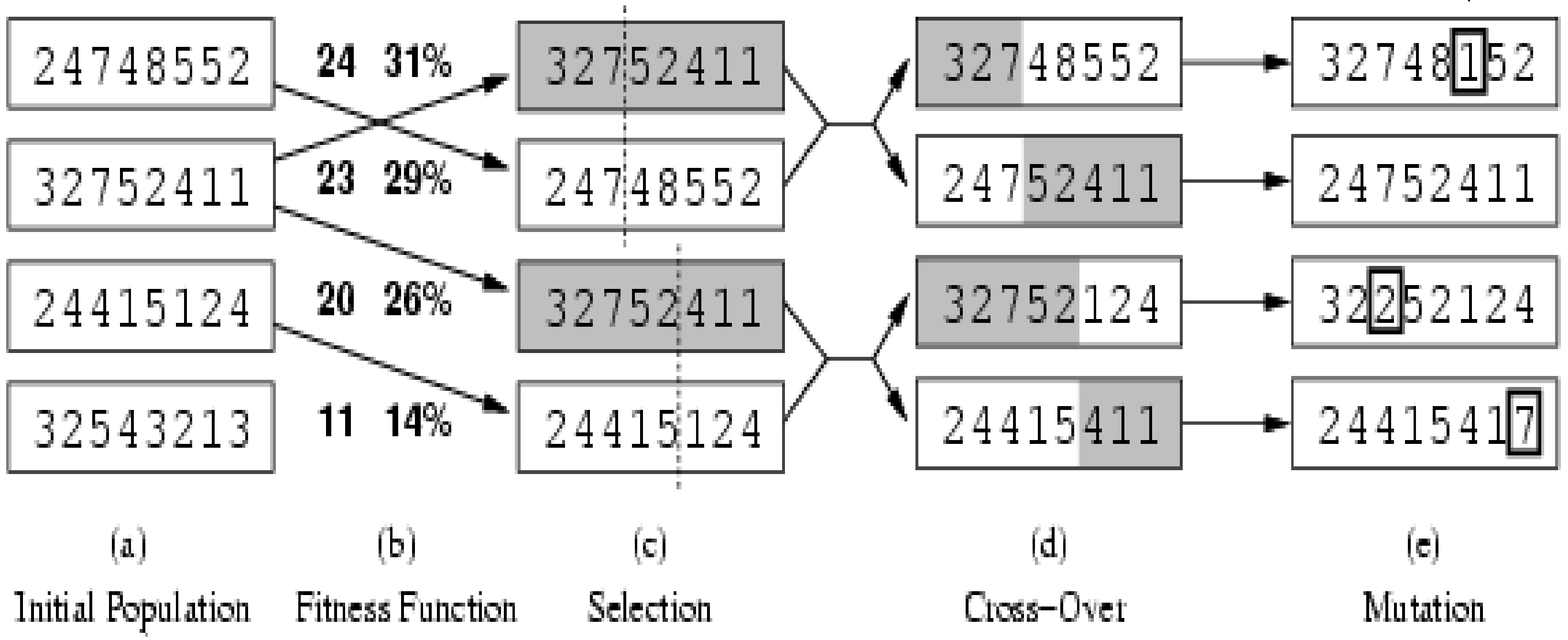


Local Beam Search

Genetic algorithms

- A successor state is generated by combining two parent states rather than modifying a single state.
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

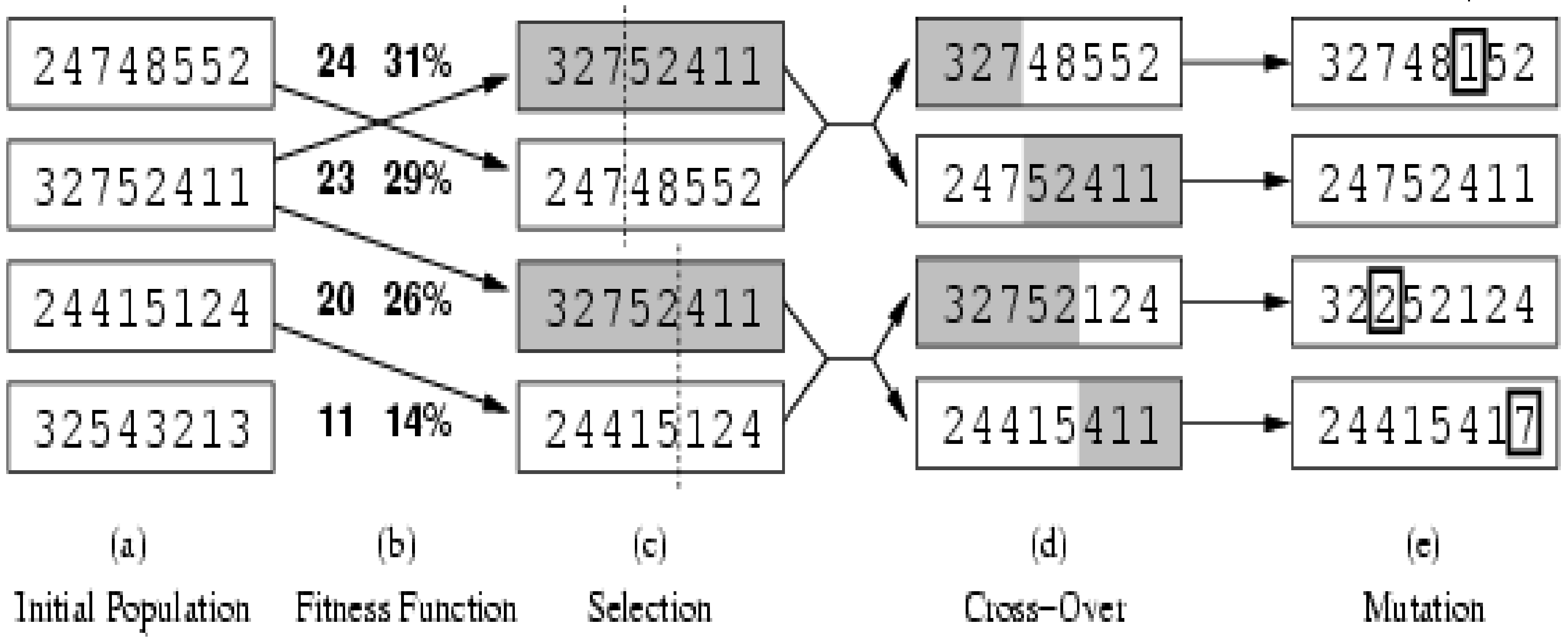
Genetic algorithms



(a) shows a population of four 8-digit strings representing 8-queens states, each range from 1 to 8.

The production of the next generation of states is shown in Figure (b) – (e).

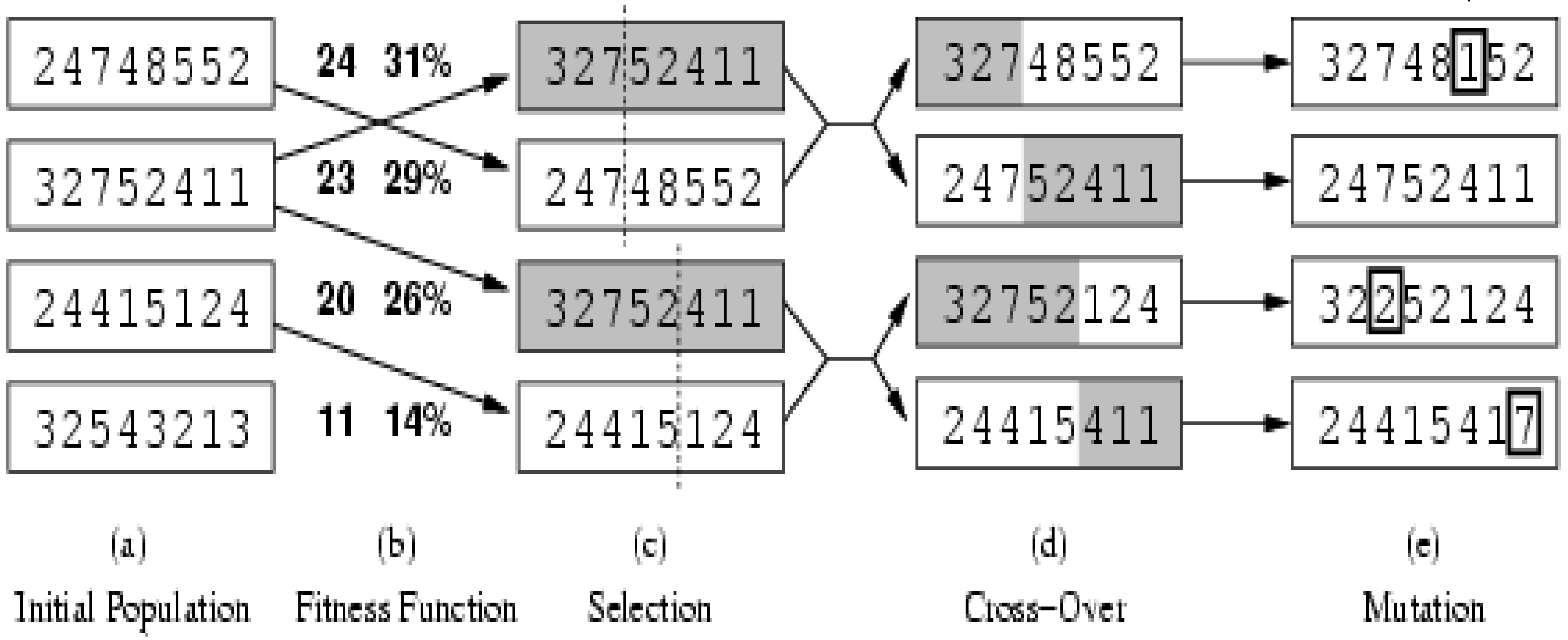
Genetic algorithms



In (b), each state is rated by the evaluation function – fitness function. We use the number of nonattacking pairs of queens – 28 for a solution.

The values of the four states are 24, 23, 20 and 11.

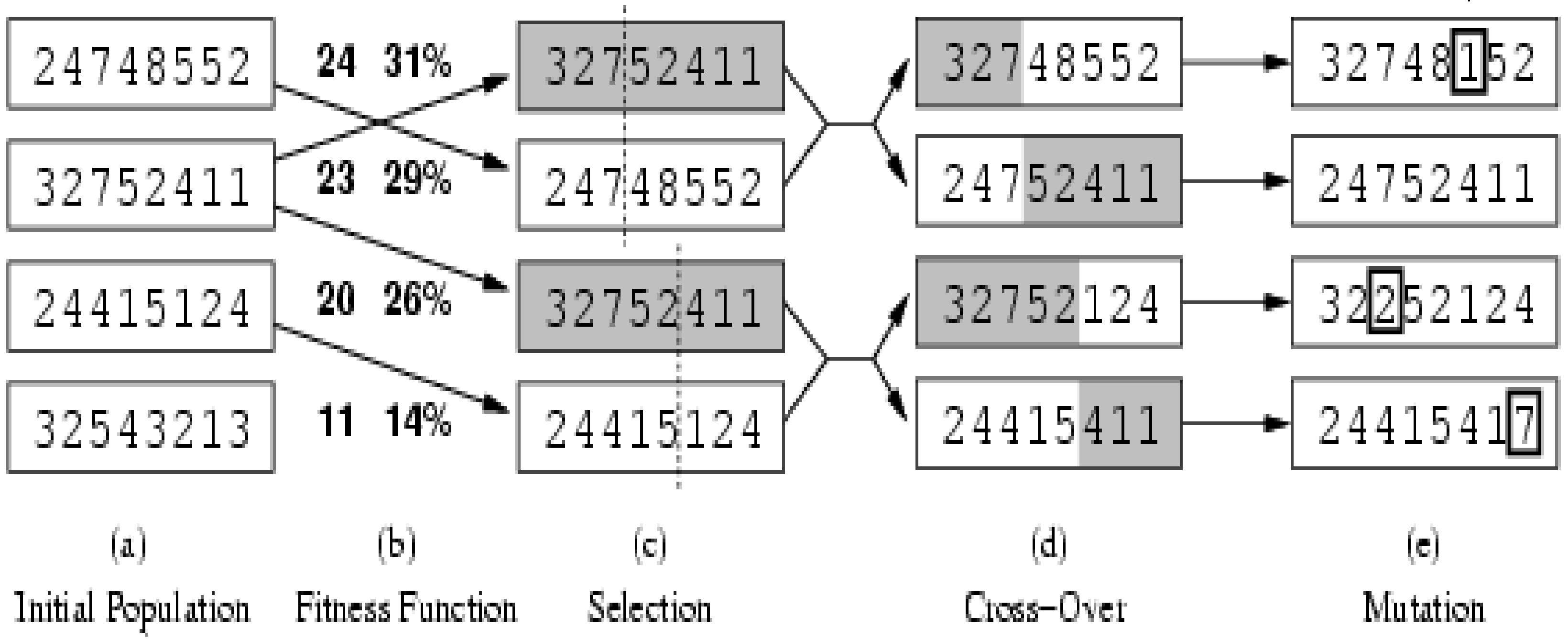
Genetic algorithms



The probability of being chosen for reproducing is proportional to the fitness score :

- ✓ 24/(24+23+20+11) = 31%
- ✓ 23/(24+23+20+11) = 29% etc.

Genetic algorithms

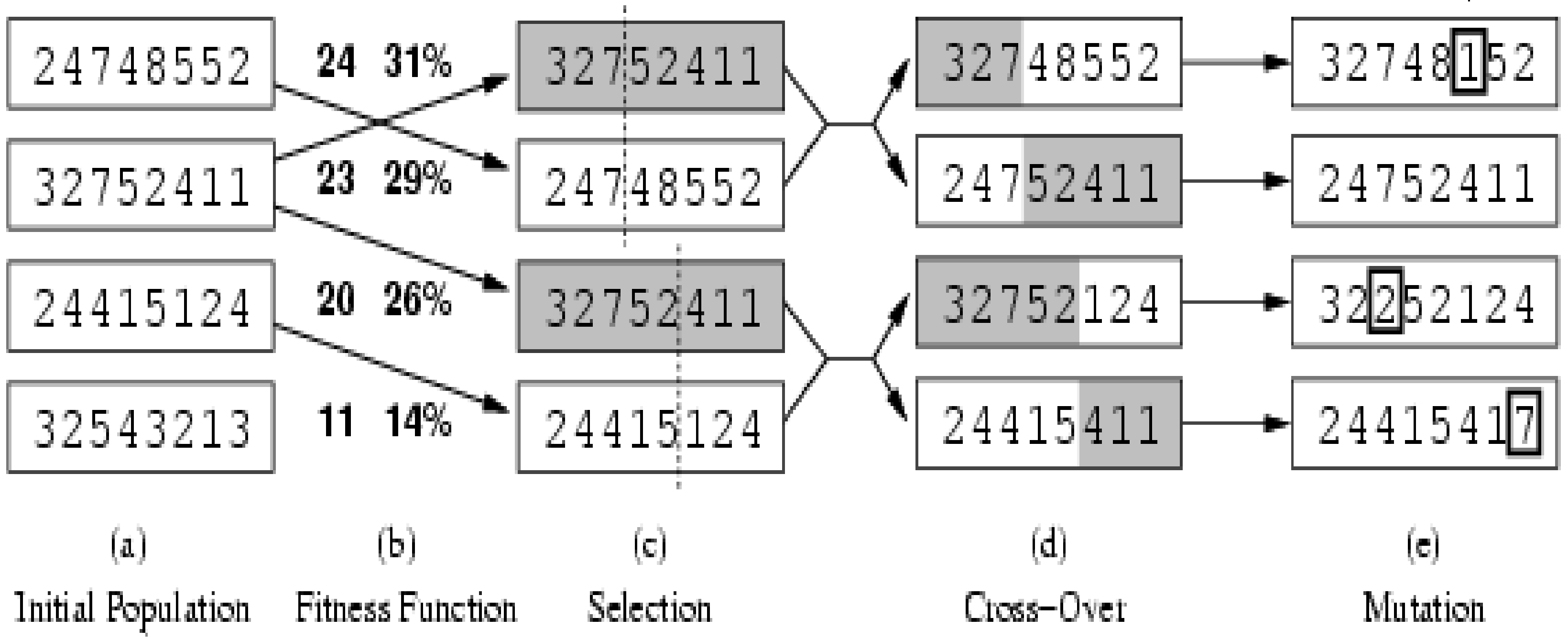


In (c), a random choice of two pairs is selected for reproduction, in accordance with the probabilities in (b).

- one individual is selected twice and one not at all.

For each pair, a croossover point is randomly chosen.

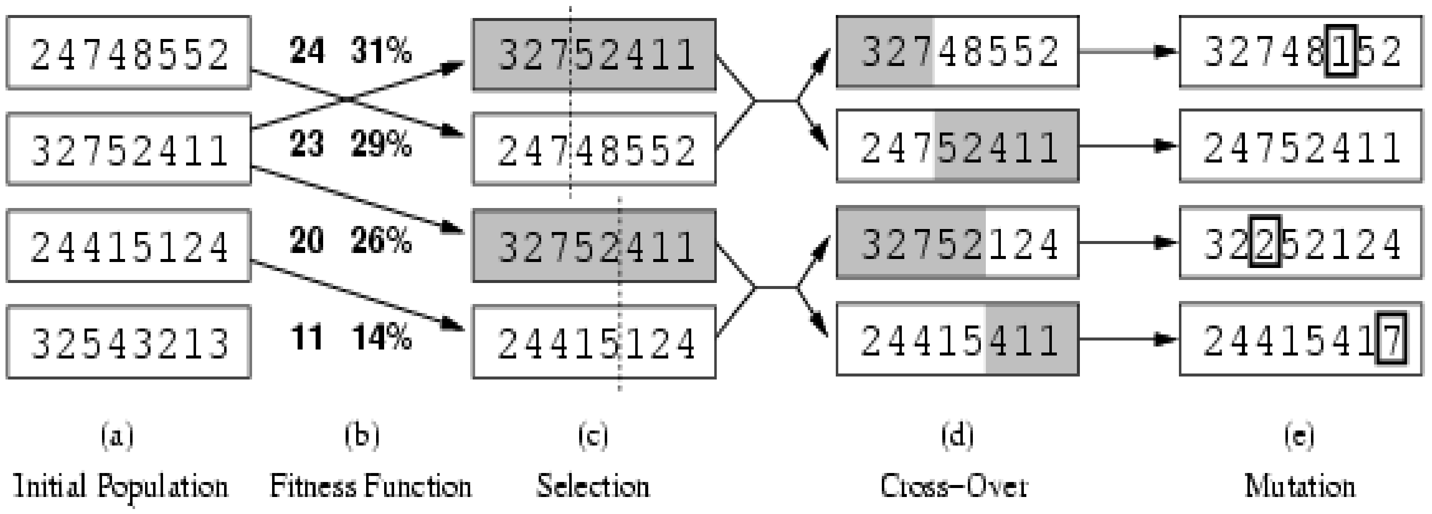
Genetic algorithms

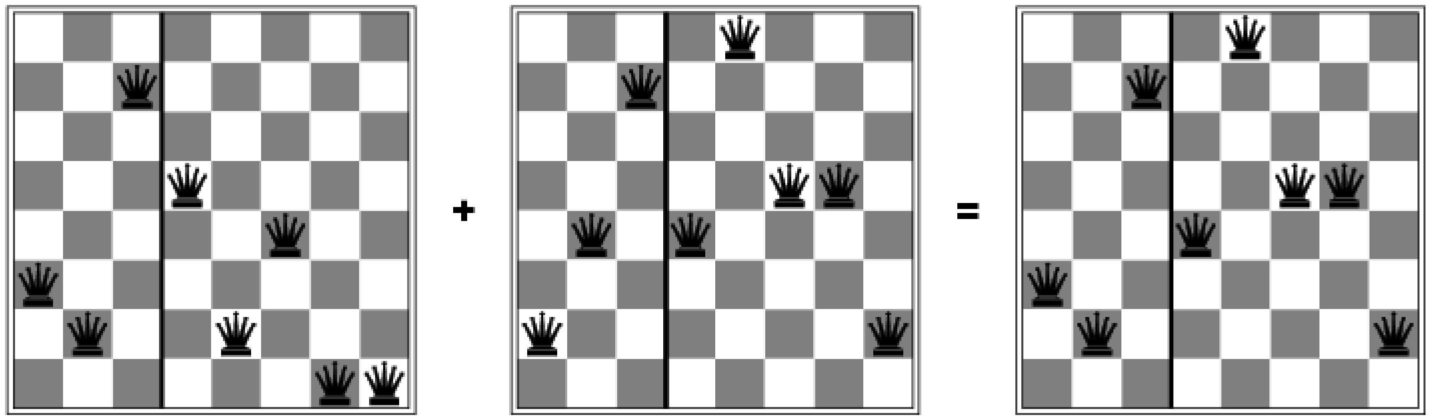


In (d), the offsprings are created by crossing over the parent strings at the crossover point.

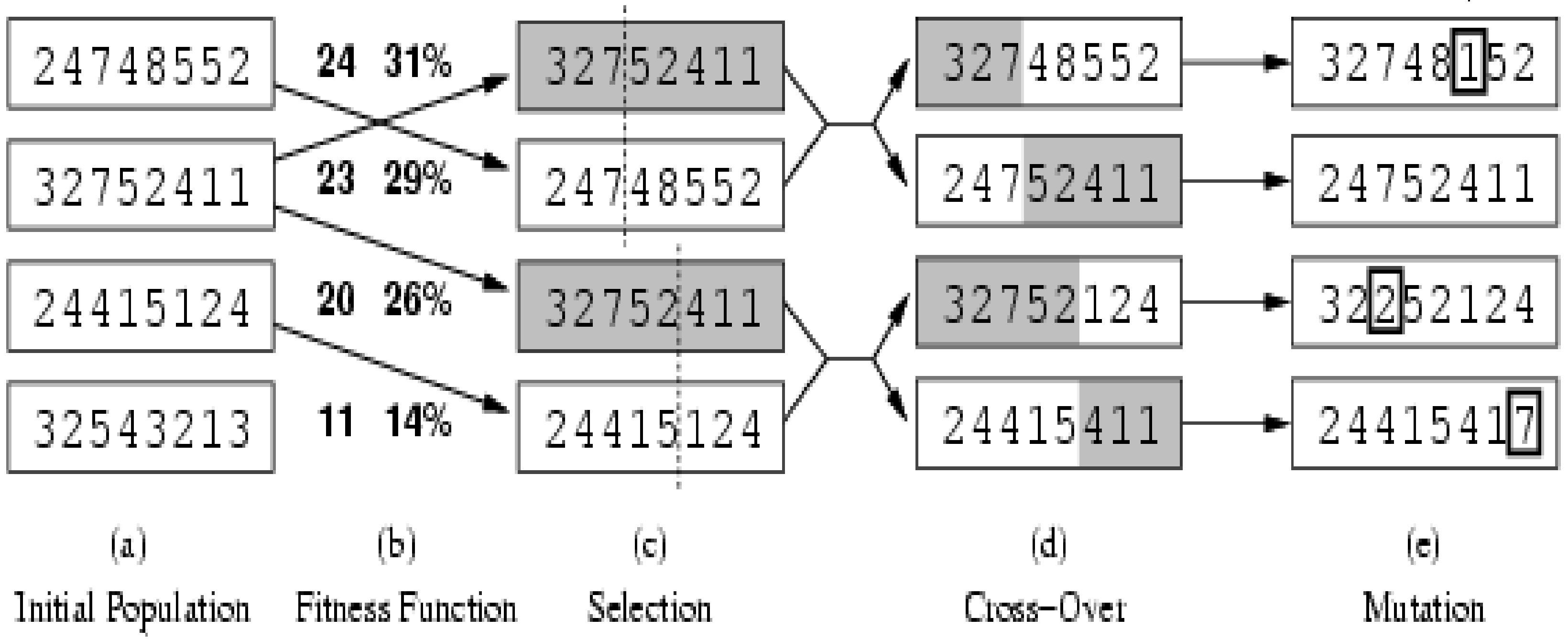
- e.g. the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent.

Genetic algorithms





Genetic algorithms



Finally, in (e), each location is subject to random mutation with probability.

- e.g. in the 8-queens problem, choose a queen at random and move it to a random square in its column.

Problem Example

Problem Example

Consider a GA with chromosomes consisting of six genes $x_i = abcdef$, and each gene is a number between 0 and 9. Suppose we have the following population of four chromosomes:

$$x_1 = 4\,3\,5\,2\,1\,6 \qquad x_2 = 1\,7\,3\,9\,6\,5$$
$$x_3 = 2\,4\,8\,0\,1\,2 \qquad x_4 = 9\,0\,8\,1\,2\,3$$

and let the fitness function be $f(x) = (a + c + e) - (b + d + f)$.

1. Sort the chromosomes by their fitness
2. Do one-point crossover in the middle between the 1st and 2nd fittest, and two-points crossover (points 2, 4) for the 2nd and 3rd.
3. Calculate the fitness of all the offspring

NEXT WEEK

- Constraint Satisfaction Problems (CSP)
- Backtracking search for CSPs
- Local search for CSPs