Search (II)
Informed search algorithms

Outline

- Greedy best-first search
- A* search
- Heuristics
- Local search algorithms
- Hill-climbing search

Best-first search

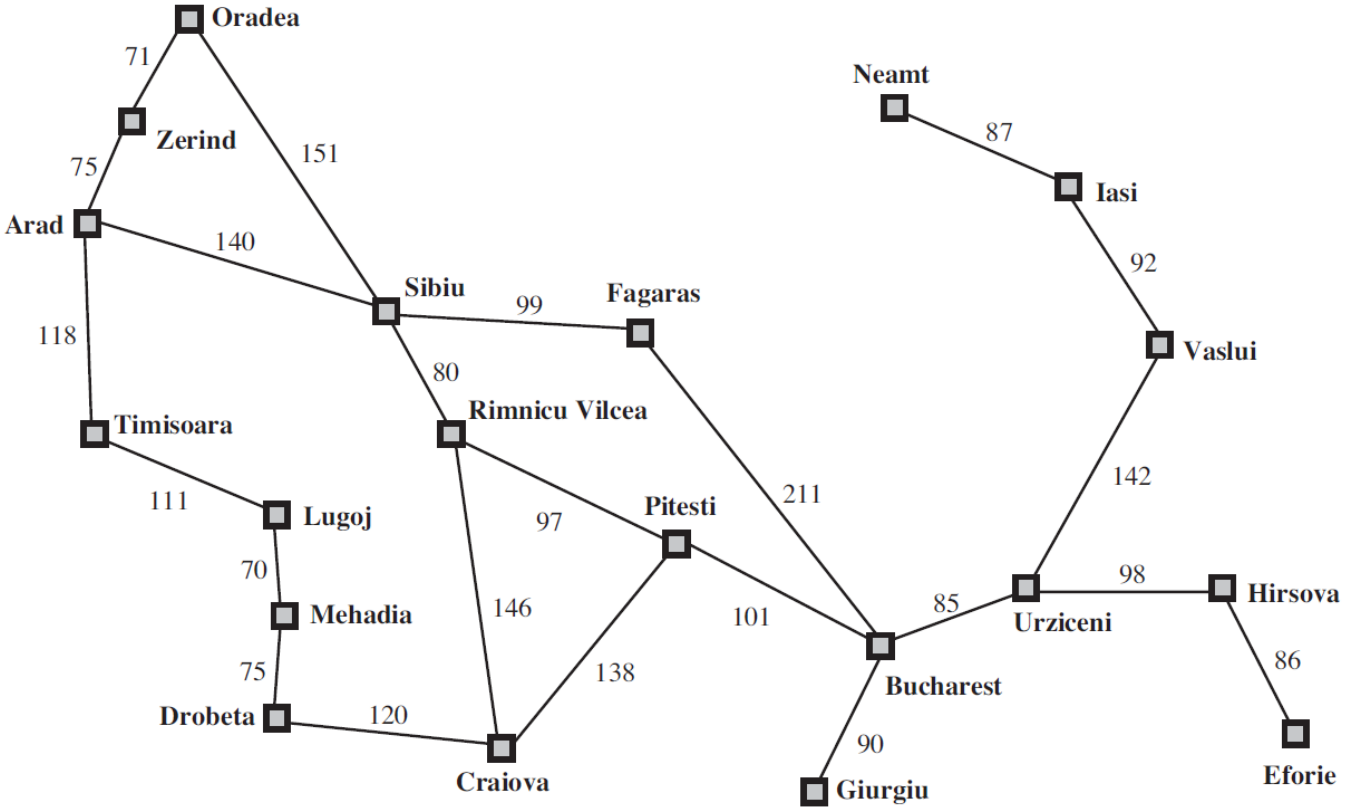
- Idea: use an evaluation function *f(n)* for each node (based on state)
  - estimate of "desirability"
  - → Expand most desirable unexpanded node

- Implementation:

  Order the nodes in frontier in decreasing order of desirability

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

Romania



Straight-line distance

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

Greedy best-first search

- Evaluation function ***f(n) = h(n)*** (heuristic)

  = estimate of cost from ***n*** to ***goal***

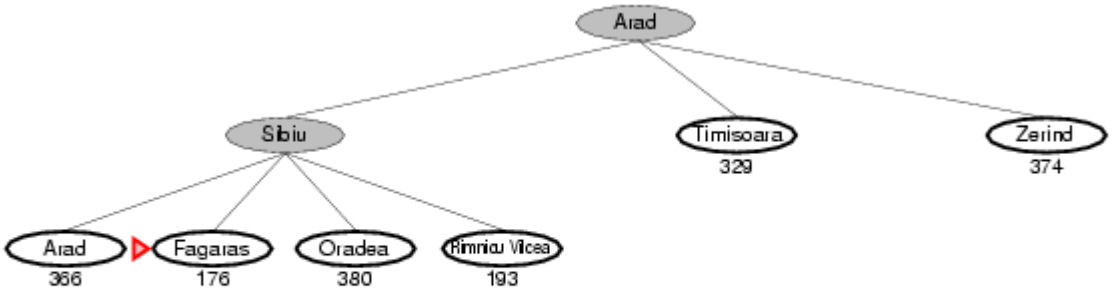
  – e.g., $\boldsymbol{h_{SLD}(n)}$ = straight-line distance from *n* to Bucharest

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

Greedy best-first search example

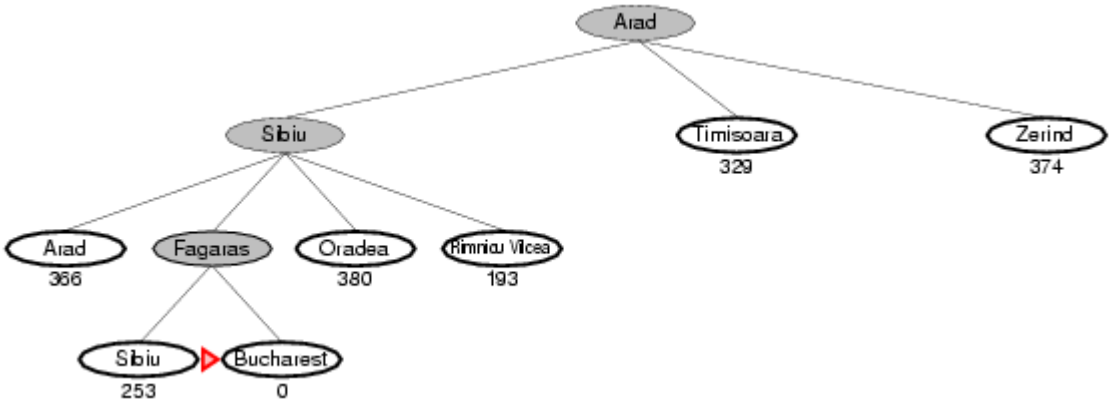
Greedy best-first search example

Greedy best-first search example



Frontier is always the set of leaves.

Greedy best-first search example



This is not the best solution though!

Properties of greedy best-first search

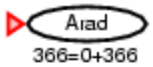
- Complete? No – can get stuck in loops,
  - e.g., suppose we start from Iasi with SLD of 126 and suppose, for the sake of the example, we change the SLD of Neamt is 198 and Vaslui is 227, then

    Iasi → Neamt → Iasi → Neamt →…
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

A* search

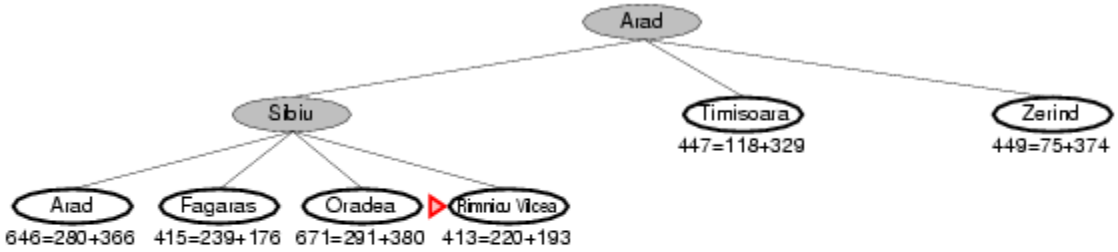
- Idea: avoid expanding paths that are already expensive
- Evaluation function ***f(n) = g(n) + h(n)***
- ***g(n)*** = cost so far to reach *n*
- ***h(n)*** = estimated cost from *n* to goal
- ***f(n)*** = estimated total cost of path through *n* to goal

A* search example

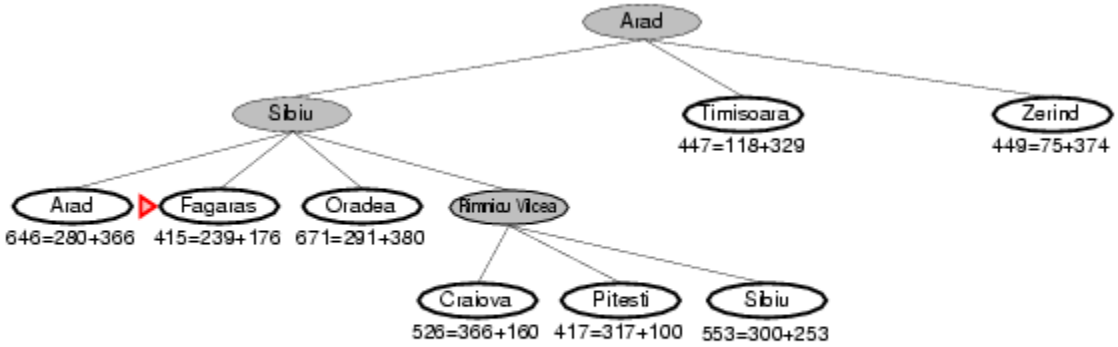


A* search example

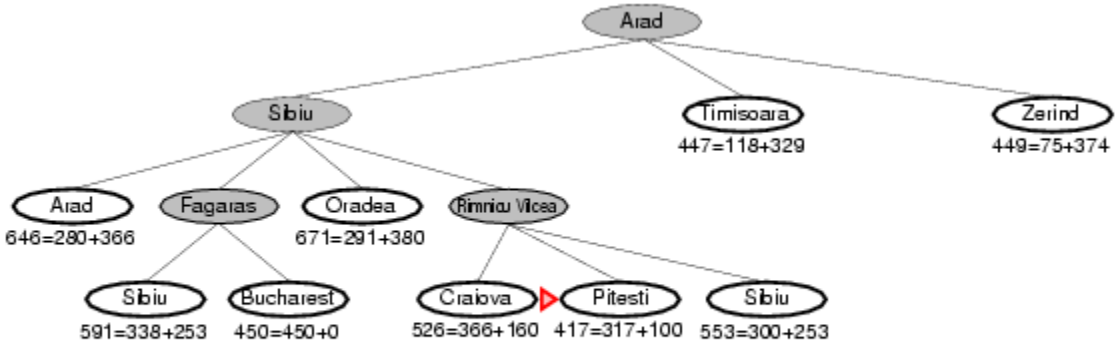
A* search example



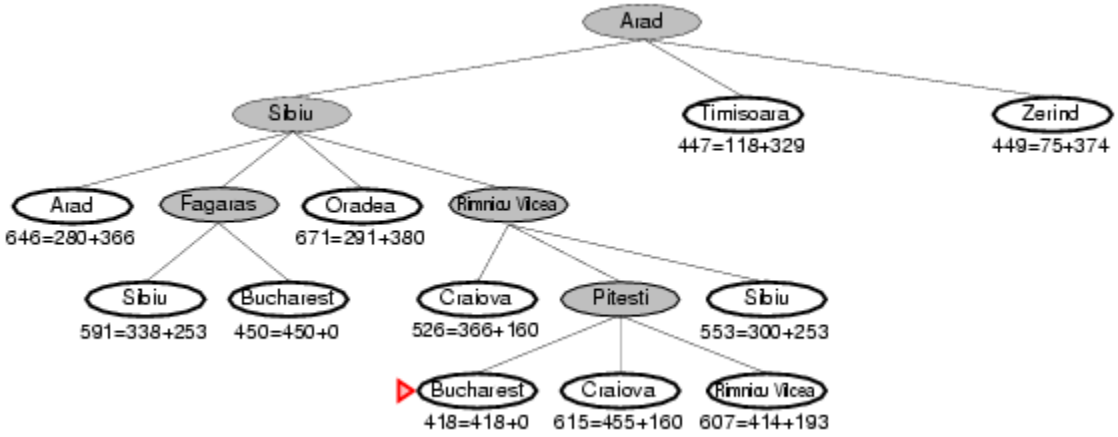
A* search example



A* search example



A* search example



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.

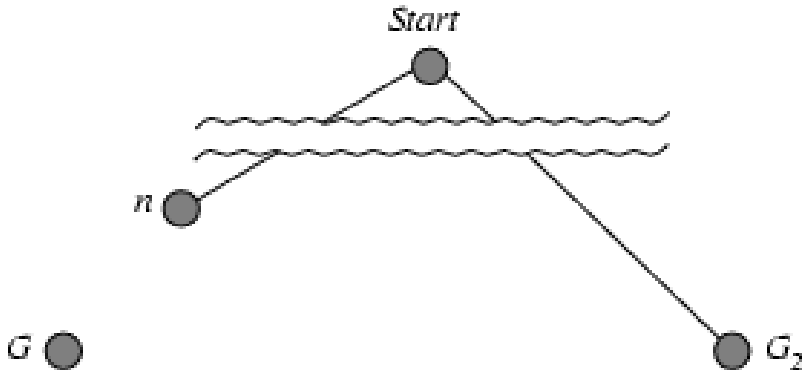
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic

- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)

- 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 frontier. Let *n* be an unexpanded node in the frontier such that *n* is on a shortest path to an optimal goal *G*.



$f(G_2) = g(G_2)$	since $h(G_2) = 0$ (G_2 is a goal)
$> g(G)$	since G_2 is sub-optimal
$= g(n) + h^*(n)$	since there is only one path from root to G (tree)
$\geq g(n) + h(n)$	since h is admissible
$= f(n)$	by definition of f

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

Recall that it is only when a node is picked for expansion that we check if it is goal.

Properties of A*

- Complete? **Yes**
- Optimal? **Yes**

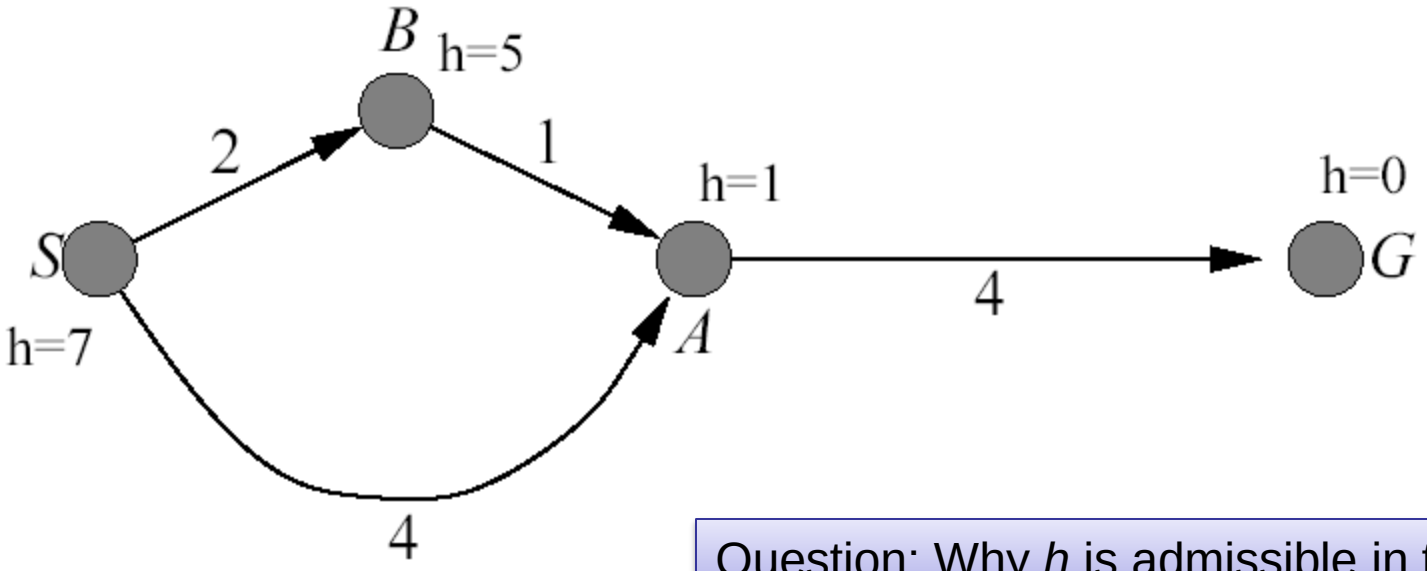
- Time and Space discussion
  - Let **C*** be the cost of the optimal solution.
  - A* expands all nodes with **f(n) < C***
  - A* expands some nodes with **f(n) = C***
  - A* expands no nodes with **f(n) > C***

- The choice of the heuristic function is very important. It can cut the search space significantly.

A* using Graph-Search

- A* using Graph-Search can produce sub-optimal solutions, even if the heuristic function is admissible.

- Sub-optimal solutions can be returned because Graph-Search can discard the optimal path to a repeated state if it is not the first one generated.



Question: Why *h* is admissible in this example?

A* Graph-Search

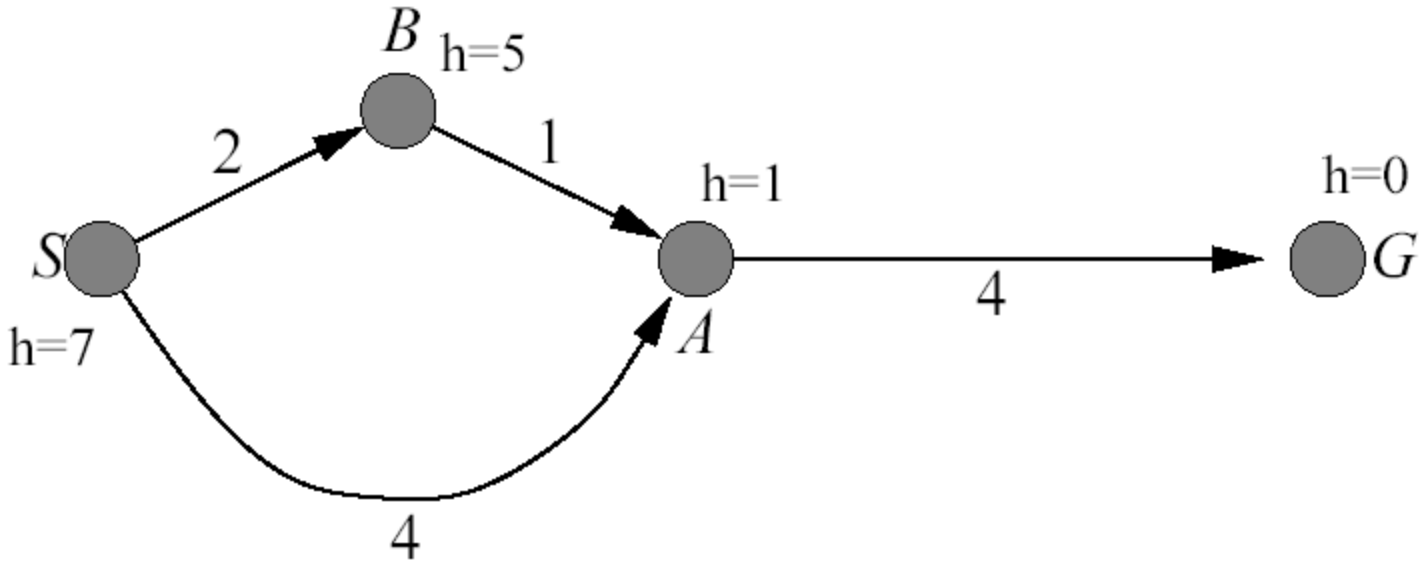
- The problem is that sub-optimal solutions can be returned because **Graph-Search** can discard the optimal path to a repeated state if it is not the first one generated.

- We can still use **Graph-Search** if we discard the longer path. We store in a map (dictionary) pairs **(statekey, pathcost)**

```
Graph-Search(problem, frontier)
    Insert(frontier, MakeNode(problem.initialstate) )
    explored = empty key-value map //not a set anymore
    Loop do
        if ( Empty(frontier) ) return null; //failure
        node = Remove(frontier);
        if (problem.GoalTest(node.state)) return node
        cost = explored.get(node.state)
        if (cost == null || cost > node.pathcost)
            explored.put(node.state, node.pathcost)
            InsertAll (frontier, Expand(node, problem) )
```

A* using Graph-Search

- Let's try it here

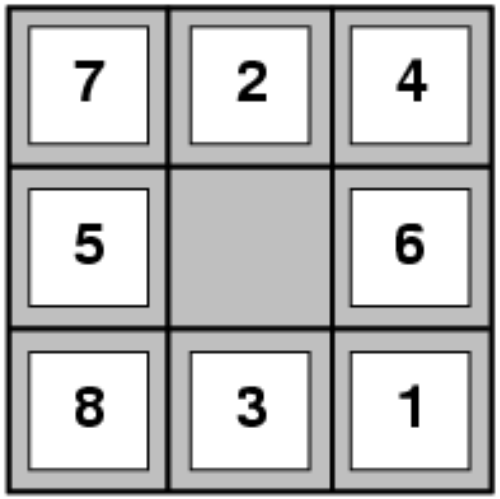


Admissible heuristics

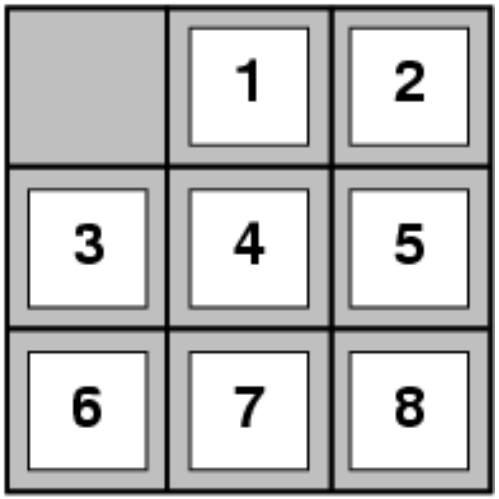
E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance $|x_1-x_2|+|y_1+y_2|$

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



Start State



Goal State

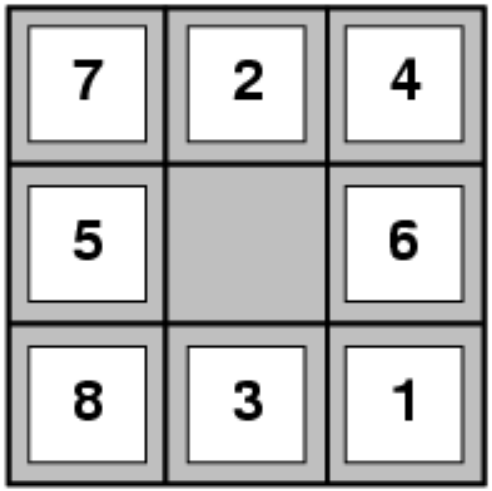
- $h_1(S) = ?$
- $h_2(S) = ?$

Admissible heuristics

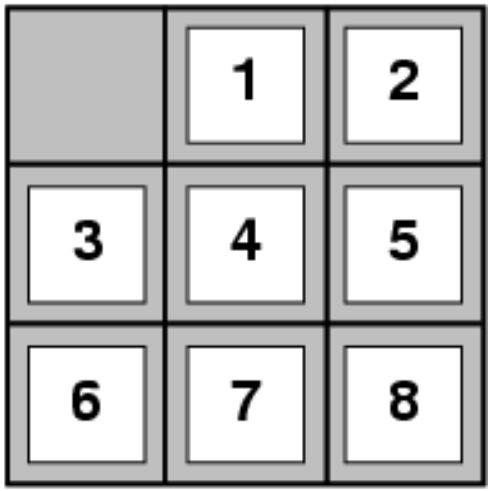
E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance $\mathbf{|x_1-x_2|+|y_1+y_2|}$

(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

Why are they **admissible**?

- **Misplaced tiles**: No move can get more than one misplaced tile into place,so this measure is a guaranteed underestimate and hence admissible.

- **Manhattan**: In fact, each move can at best decrease by one the rectilinear distance of a tile from its goal.

Dominance

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

- For the 8-puzzle, h_2 is better for search

  - Let **C*** be the cost of the optimal solution.
  - A* expands all nodes with **f(n) < C***
  - A* expands some nodes with **f(n) = C***
  - A* expands no nodes with **f(n) > C***
  - Hence, we want **f(n)** (g(n)+h(n)) to be as big as possible. Since we can't do anything about **g(n)** we are interested in having **h(n)** as big as possible.

	Search Cost (nodes generated)			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

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d.

The **max** of heuristics

- 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, we create a compound heuristic as:
  - $h(n) = \max \{h_1(n), \ldots, h_m(n)\}$

- Is it admissible?
- Yes, because each component is admissible, so h won't over-estimate the distance to the goal.

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

Subproblems

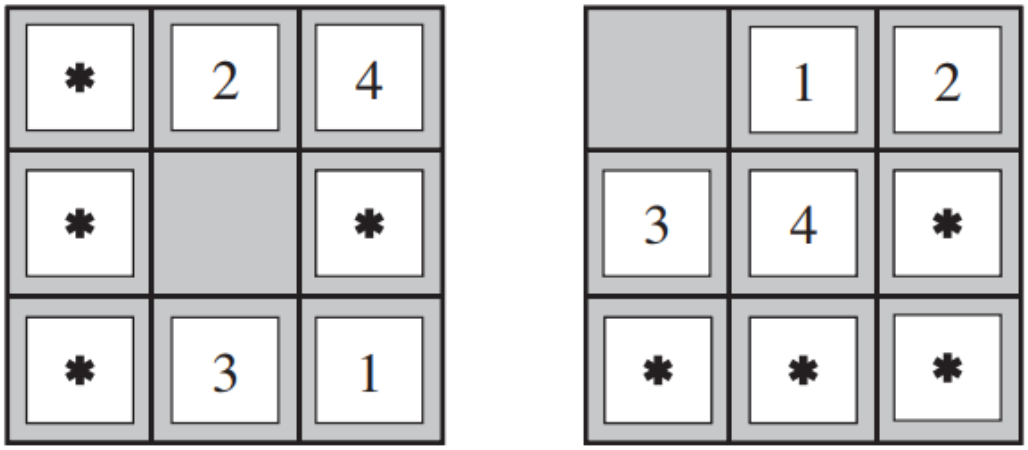
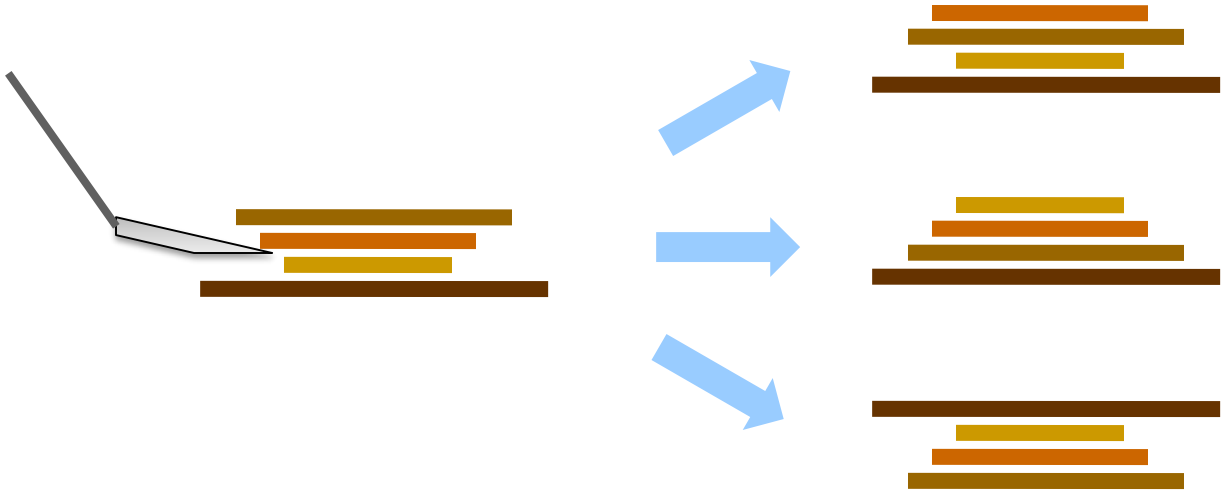


Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

Pattern databases

- Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem.
  - E.g. see fig. The subproblem involves getting tiles 1, 2, 3, 4 into their correct positions.
  - Clearly, the cost of the optimal solution of this subproblem is a lower bound on the cost of the complete problem.
  - More accurate than Manhattan distance.
- Pattern databases: store these exact solution costs for every possible subproblem instance—
  - in our example, every possible configuration of the four tiles and the blank.
  - Then compute an admissible heuristic for each complete state encountered during a search simply by looking up the corresponding subproblem configuration in the database.

Example: Pancake Problem



Cost: Number of pancakes flipped

Example: Pancake Problem

BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES
Microsoft, Albuquerque, New Mexico

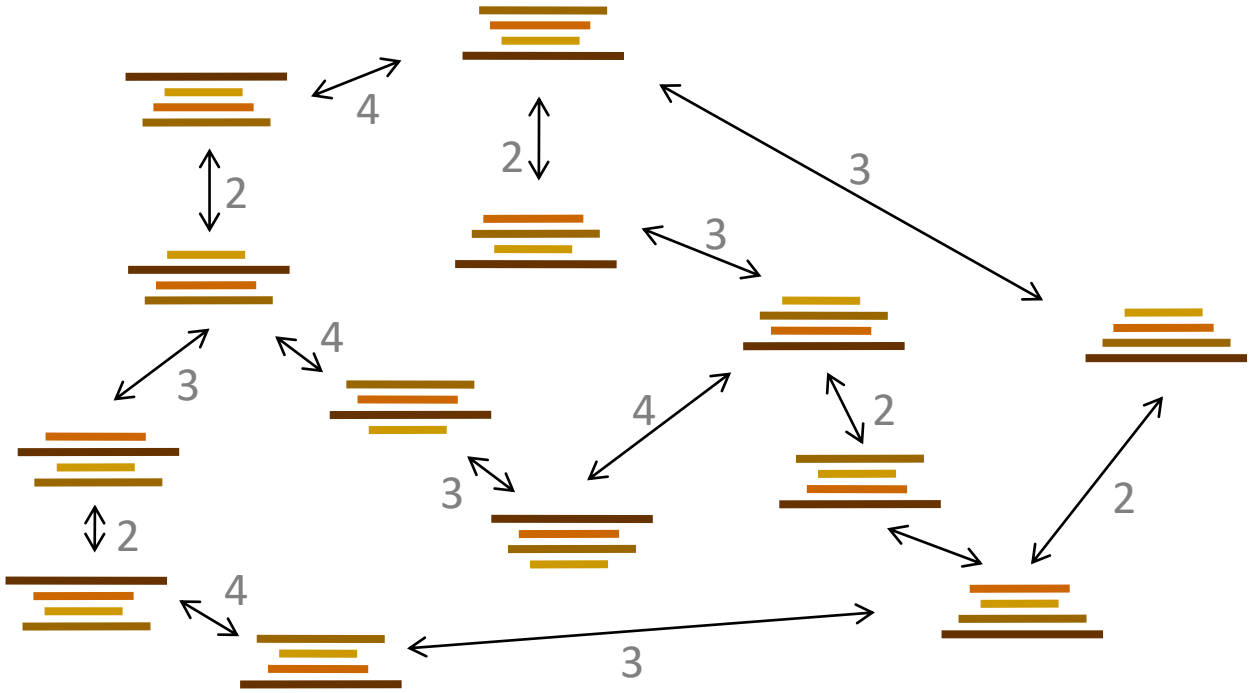
Christos H. PAPADIMITRIOU*†
Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978
Revised 28 August 1978

For a permutation σ of the integers from 1 to n, let $f(\sigma)$ be the smallest number of prefix reversals that will transform σ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all σ in (the symmetric group) S_n. We show that $f(n) \leq (5n+5)/3$, and that $f(n) \geq 17n/16$ for n a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2 - 1 \leq g(n) \leq 2n+3$.

Example: Pancake Problem

State space graph with costs as weights



Heuristic?

- Position of biggest pancake.

4

Local search algorithms

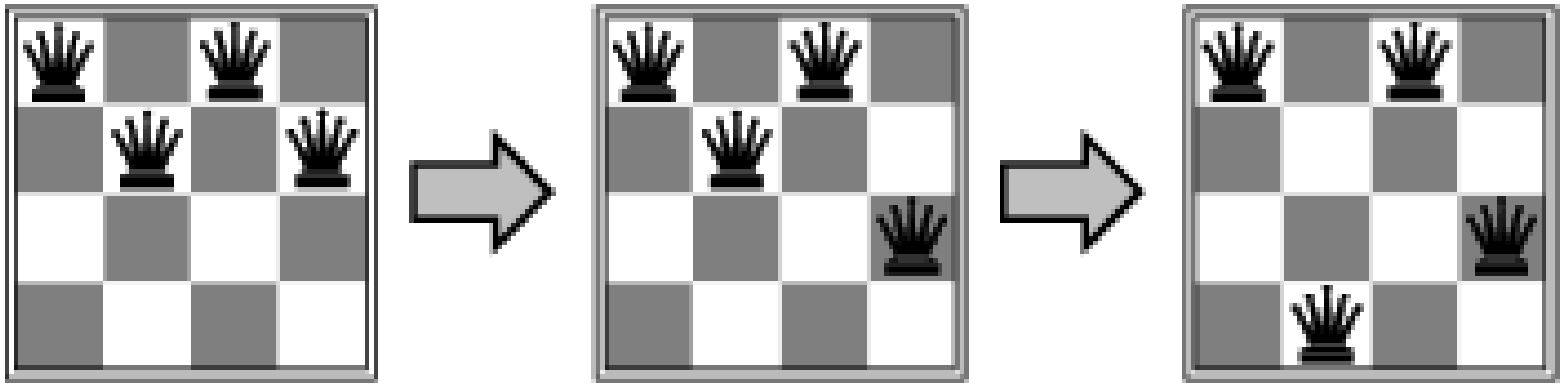
- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution

- State space = set of "complete" configurations
- Find configuration satisfying constraints,
  e.g., *n*-queens

- In such cases, we can use local search algorithms, which keep a single "current" state, and try to improve it.

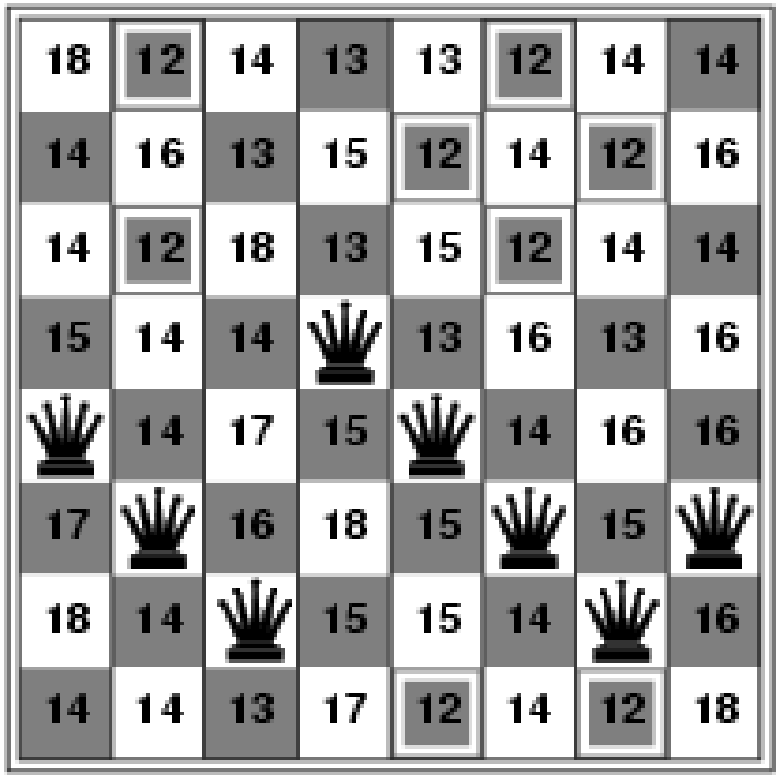
Example: *n*-queens

- **Problem definition**: Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal



- How many successors from each state?
- (n-1)*n (changing the position of one queen at a time)

8-queens problem



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

- *h = 17* for the above state

Hill-climbing search

- "Like climbing Everest in thick fog with amnesia"

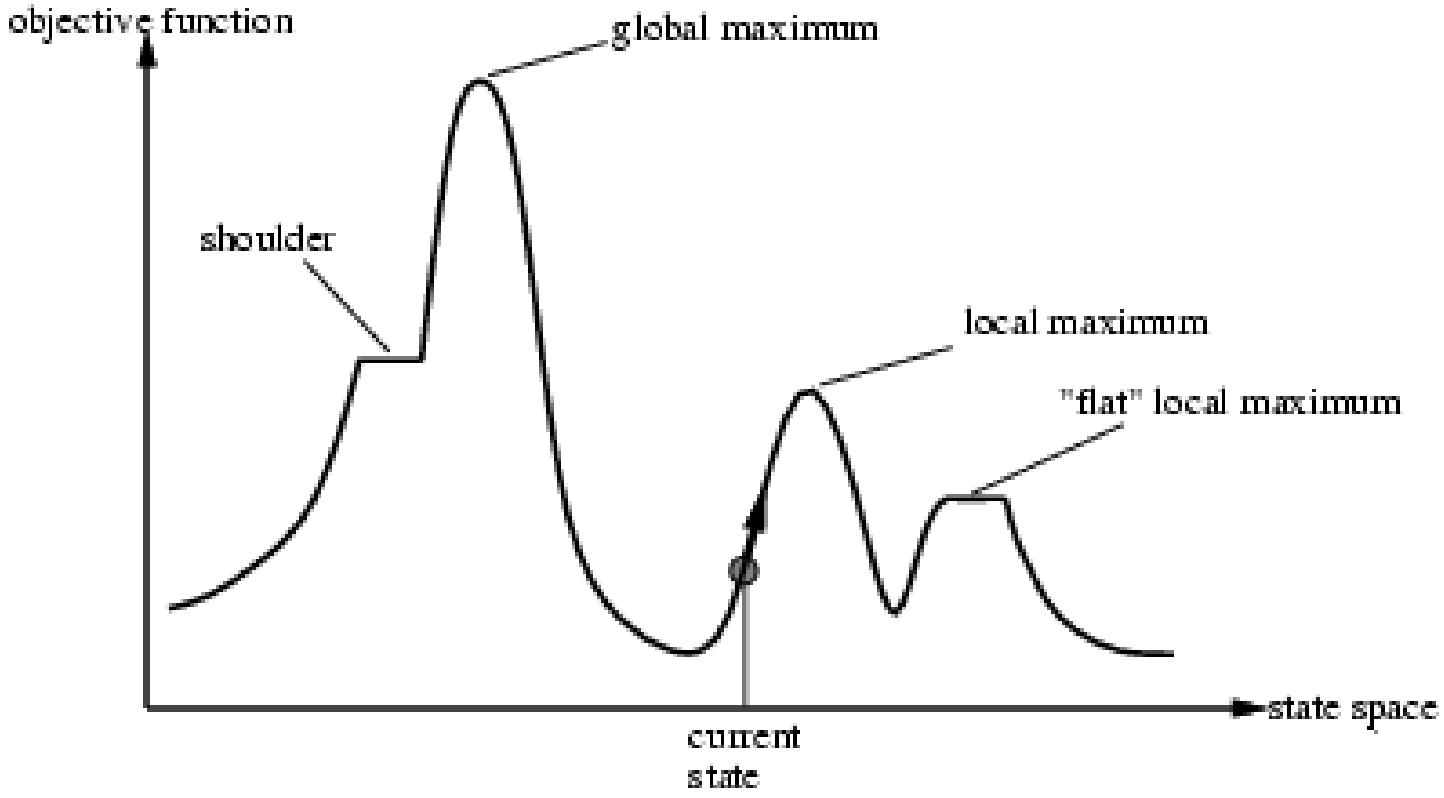
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   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
```

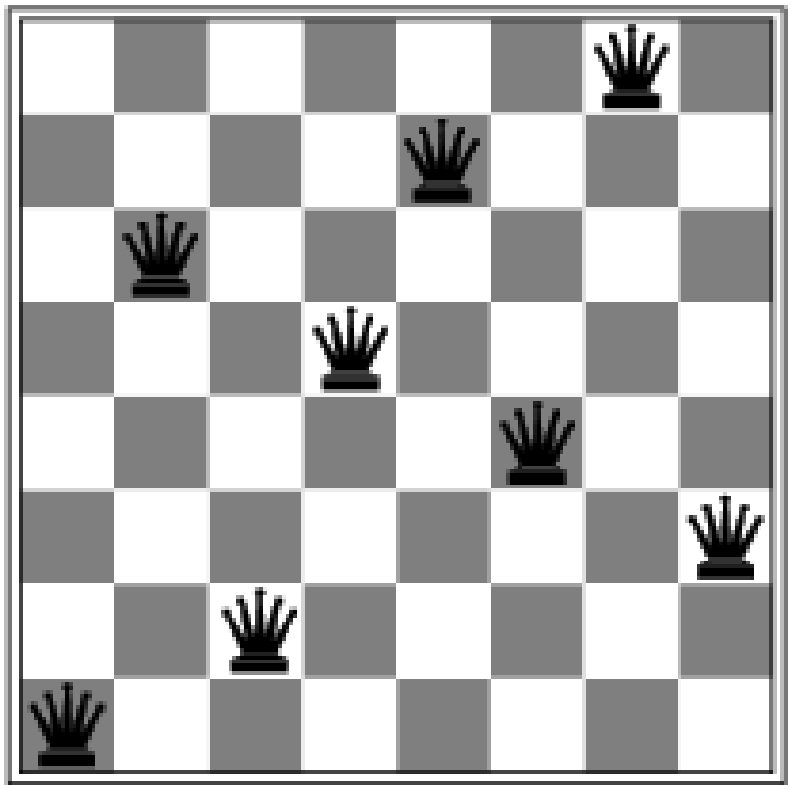
- Hill-Climbing chooses randomly among the set of **best** successors, if there is more than one.

Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima



Hill-climbing search: 8-queens problem



- A local minimum with *h = 1*

Local Minima

- Starting from a randomly generated state of the 8-queens, hill-climbing gets stuck 86% of the time, solving only 14% of problems.

- It works quickly, taking just 4 steps on average when it succeeds, and 3 when it gets stuck – not bad for a state space with $8^8 \cong 17$ million states.
- Memory is constant, since we keep only one state.

- Since it is so attractive, what can we do in order to not get stuck?

Random-restart hill climbing

- Well known adage: “If at first you don’t succeed, try, try again.”
- It conducts a series of hill-climbing searches from randomly generated initial states, stopping when a goal is found.

- If each hill-climbing search has a probability ***p*** of success, then the expected number of restarts is **1/*p***.
- For 8-queens, p=0.14, so we need roughly 7 iterations to find a goal, i.e. ≅ 22 steps (3 steps for failures and 4 for success)

Genetic Algorithms

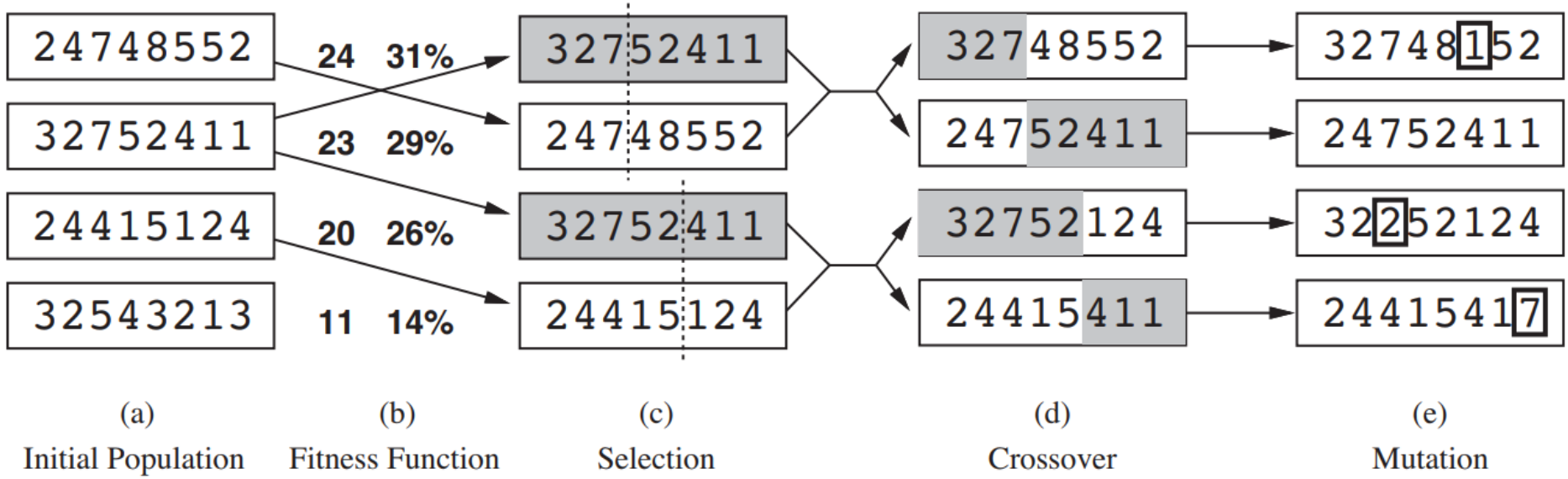


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

Each state is rated by the **fitness function**, which returns higher values for better states.
For the 8-queens use the number of nonattacking pairs of queens, which has a value of 28 for a solution.
The values of the four states shown are 24, 23, 20, and 11.
The probability of being chosen for reproducing is directly proportional to the fitness score.

Illustration

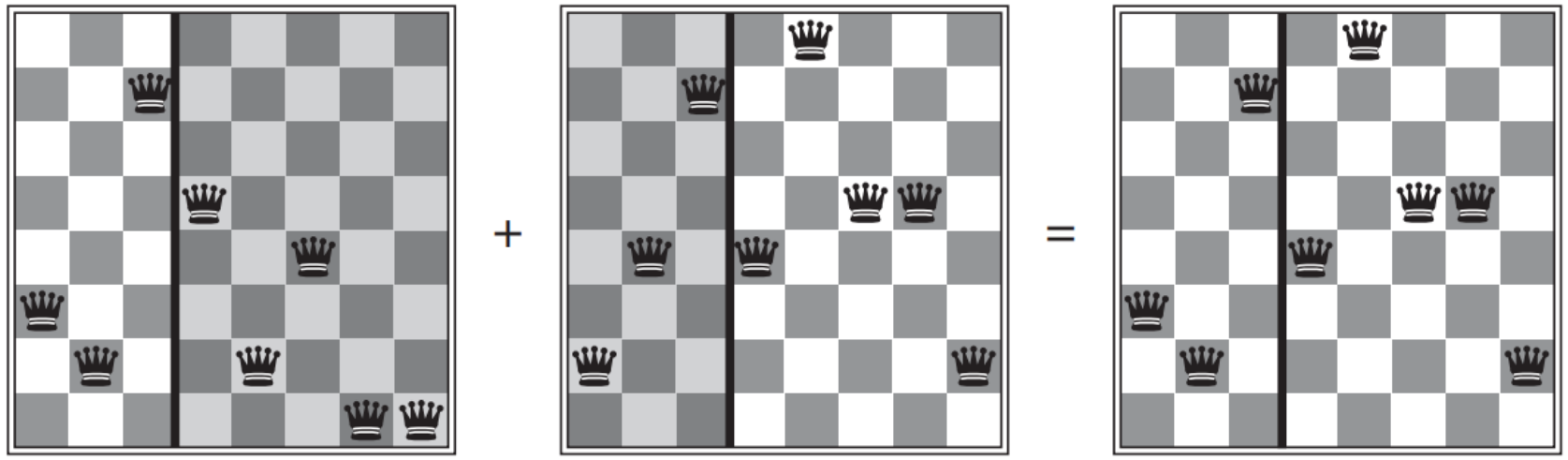


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

GA Pseudocode

```
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
```

GA begins with a set of k randomly generated states, called the population. Each state, or individual, is represented as a string over a finite alphabet