Informed Search and Exploration

Berlin Chen 2003

Reference:

1. S. Russell and P. Norvig. Artificial Intelligence: A Modern Approach, Chapter 4
2. S. Russell's teaching materials

Introduction

- Informed Search
  - Also called heuristic search
  - Use problem-specific knowledge
  - Search strategy: a node is selected for exploration based on an evaluation function, $f(n)$
    - Estimate of desirability

- Evaluation function generally consists of two parts
  - The path cost from the initial state to a node *n*, $g(n)$ (optional)
  - The estimated cost of the cheapest path from a node *n* to a goal node, the heuristic function, $h(n)$
    - If the node *n* is a goal state $\rightarrow$ $h(n) = 0$
    - Can't be computed from the problem definition (need experience)

Heuristics

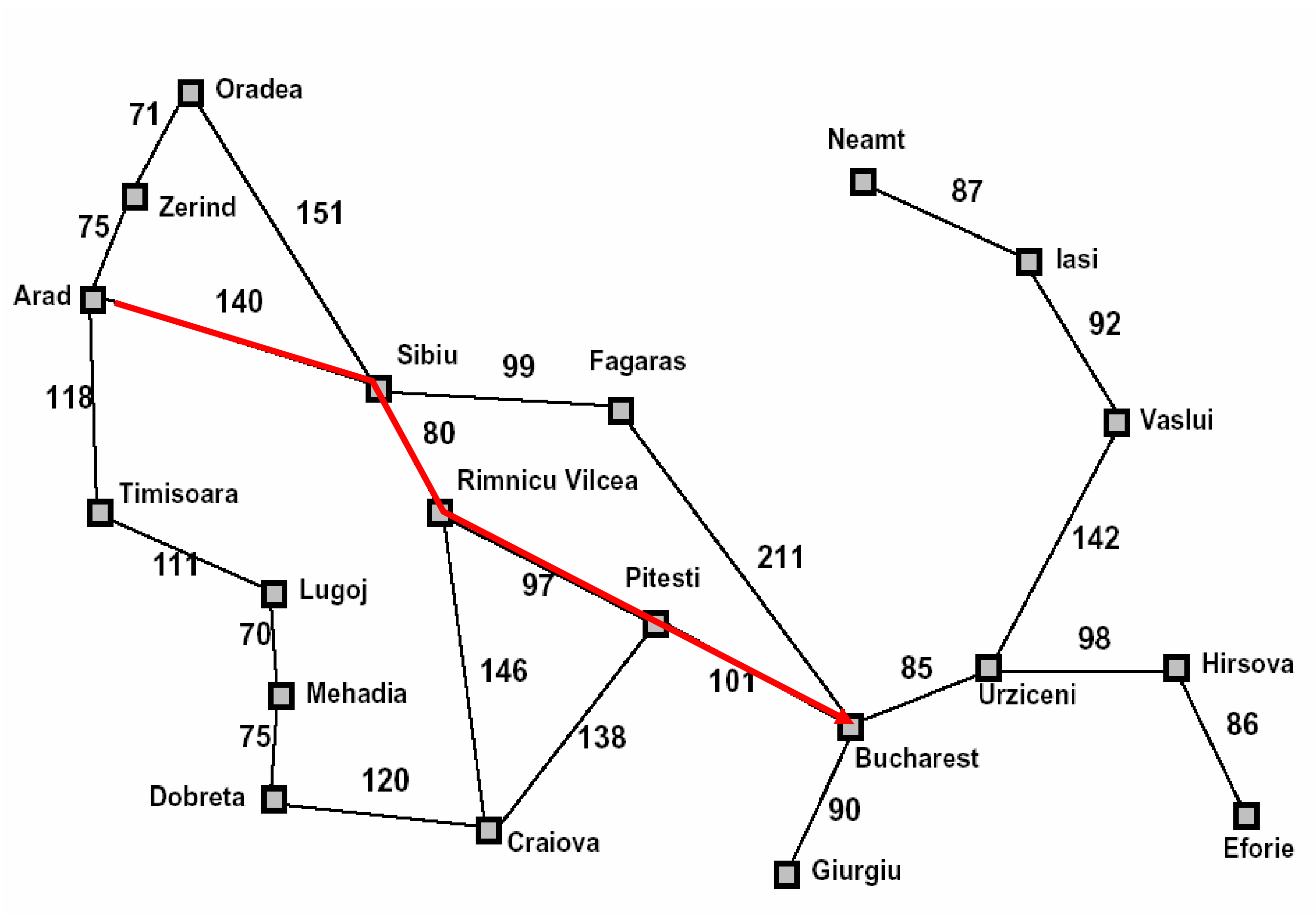
- Used to describe rules of thumb or advise that are generally effective, but not guaranteed to work in every case
- In the context of search, a heuristic is a function that takes a state as an argument and returns a number that is an estimate of the merit of the state with respect to the goal
- Not all heuristic functions are beneficial
  - Should consider the time spent on evaluating the heuristic function
  - Useful heuristics should be computationally inexpensive

Best-First Search

- Choose the most desirable (seemly-best) node for expansion based on evaluation function
  - Lowest cost/highest probability evaluation
- Implementation
  - *Fringe* is a priority queue in decreasing order of desirability
- Several kinds of best-first search introduced
  - **Greedy best-first search**
  - **A* search**
  - Iterative-Deepening A* search
  - Recursive best-first search
  - Simplified memory-bounded A* search

memory-bounded heuristic search

Map of Romania



$h(n)$

Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best-First Search

- Expand the node that appears to be closet to the goal, based on the heuristic function only

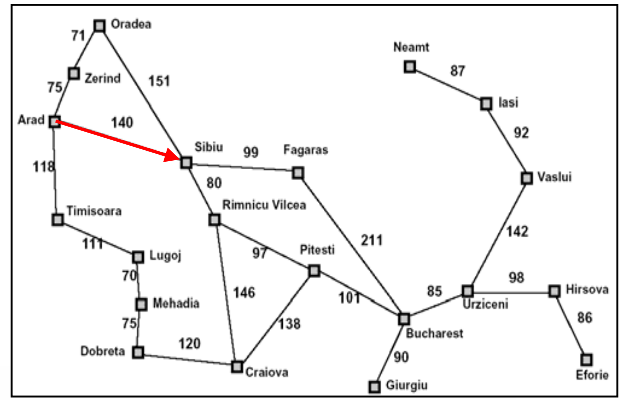
$$f(n) = h(n) = \text{estimate of cost from node } n \text{ to the closest goal}$$

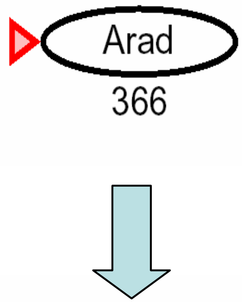
  - E.g., the straight-line distance heuristics h_{SLD} to Bucharest for the route-finding problem
    - $h_{SLD}(In(Arad)) = 366$

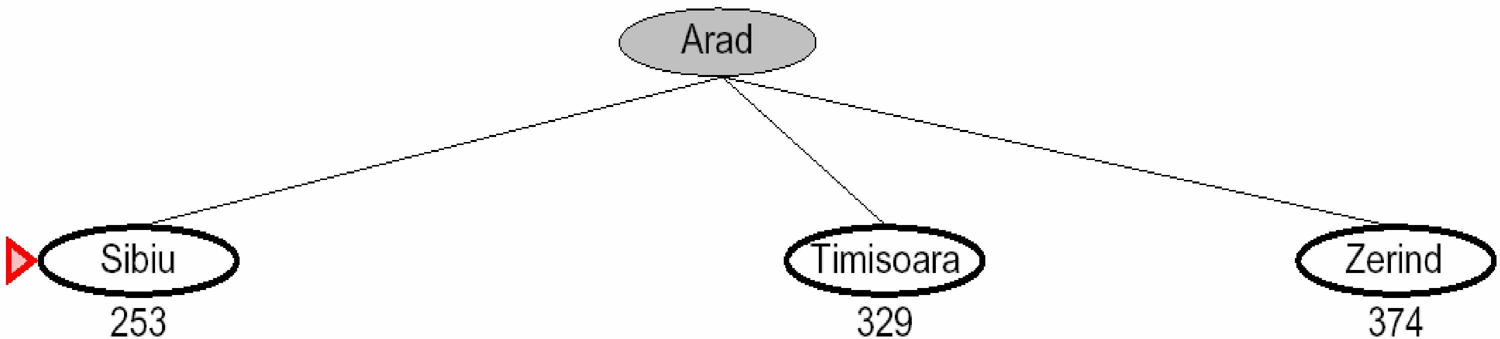
- "greedy" – at each search step the algorithm always tries to get close to the goal as it can

Greedy Best-First Search

- Example 1: the route-finding problem

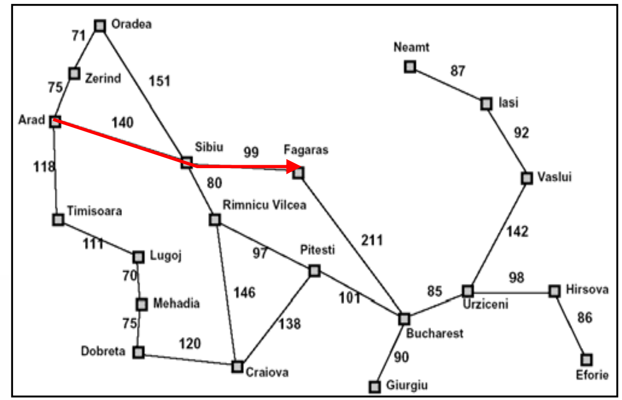


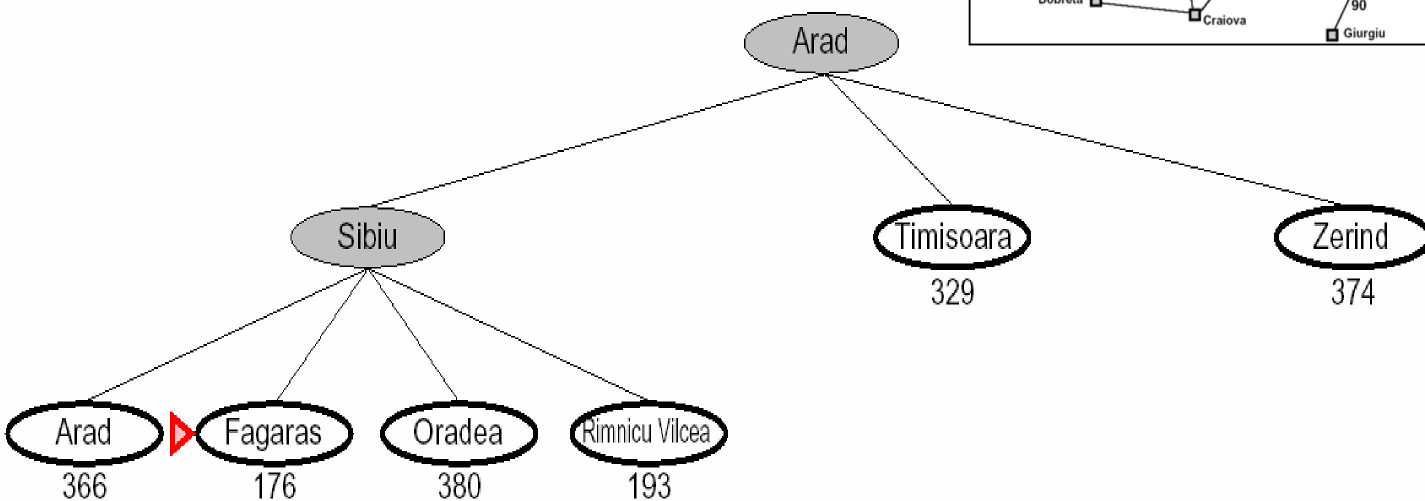




Greedy Best-First Search

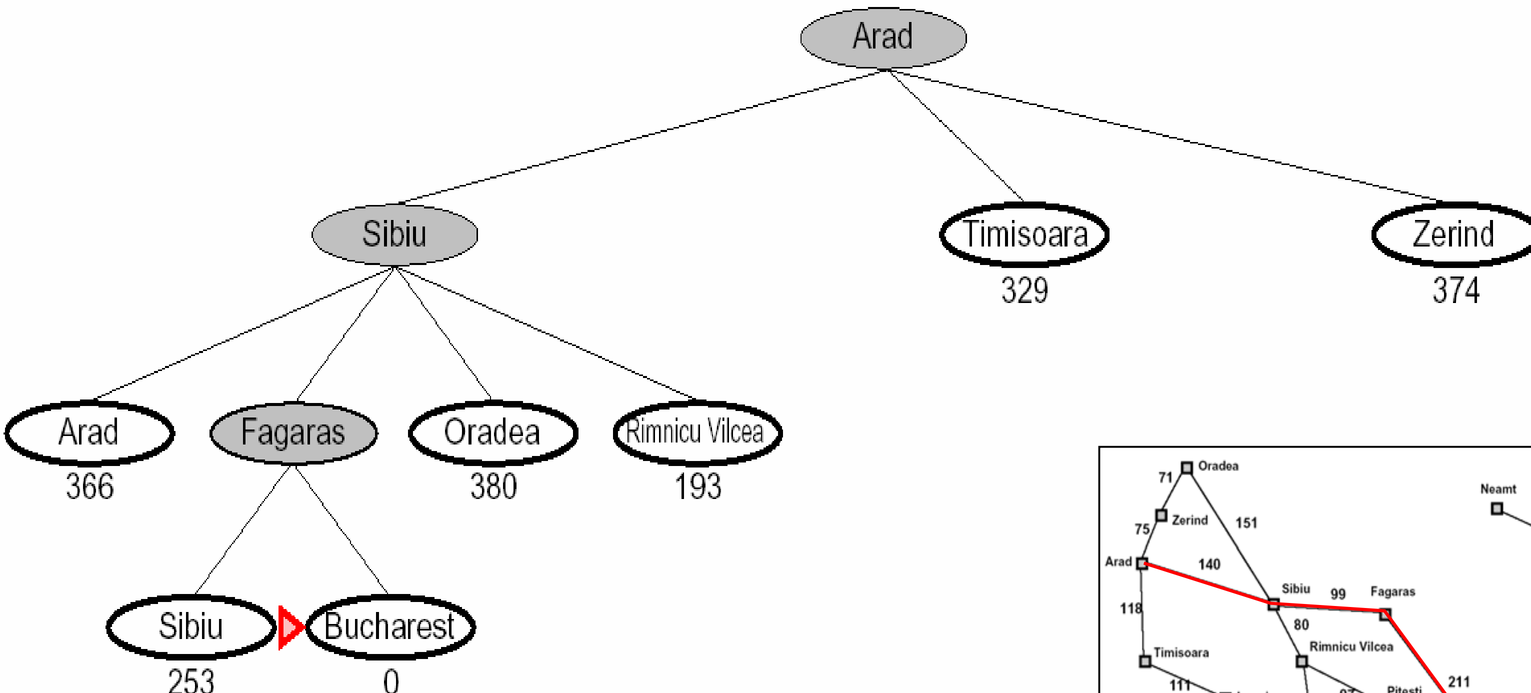
- Example 1: the route-finding problem

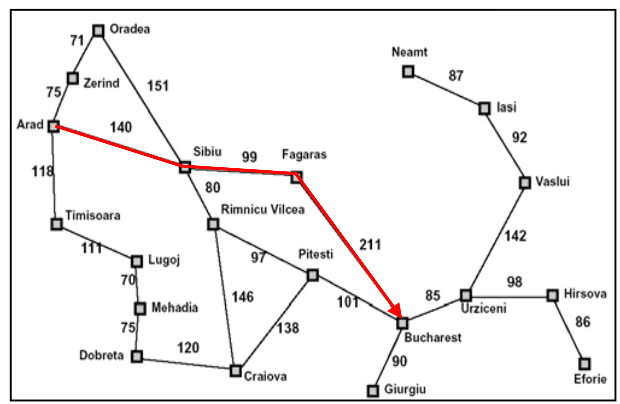




Greedy Best-First Search

- Example 1: the route-finding problem



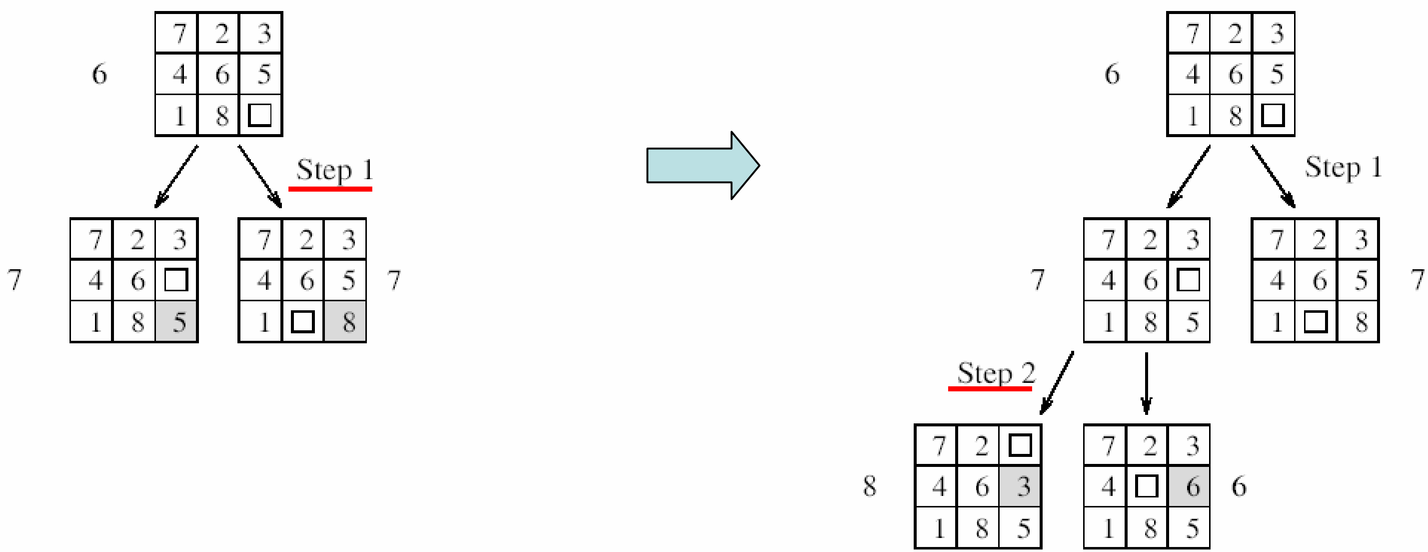


  - The solution is not optimal (?)

Greedy Best-First Search

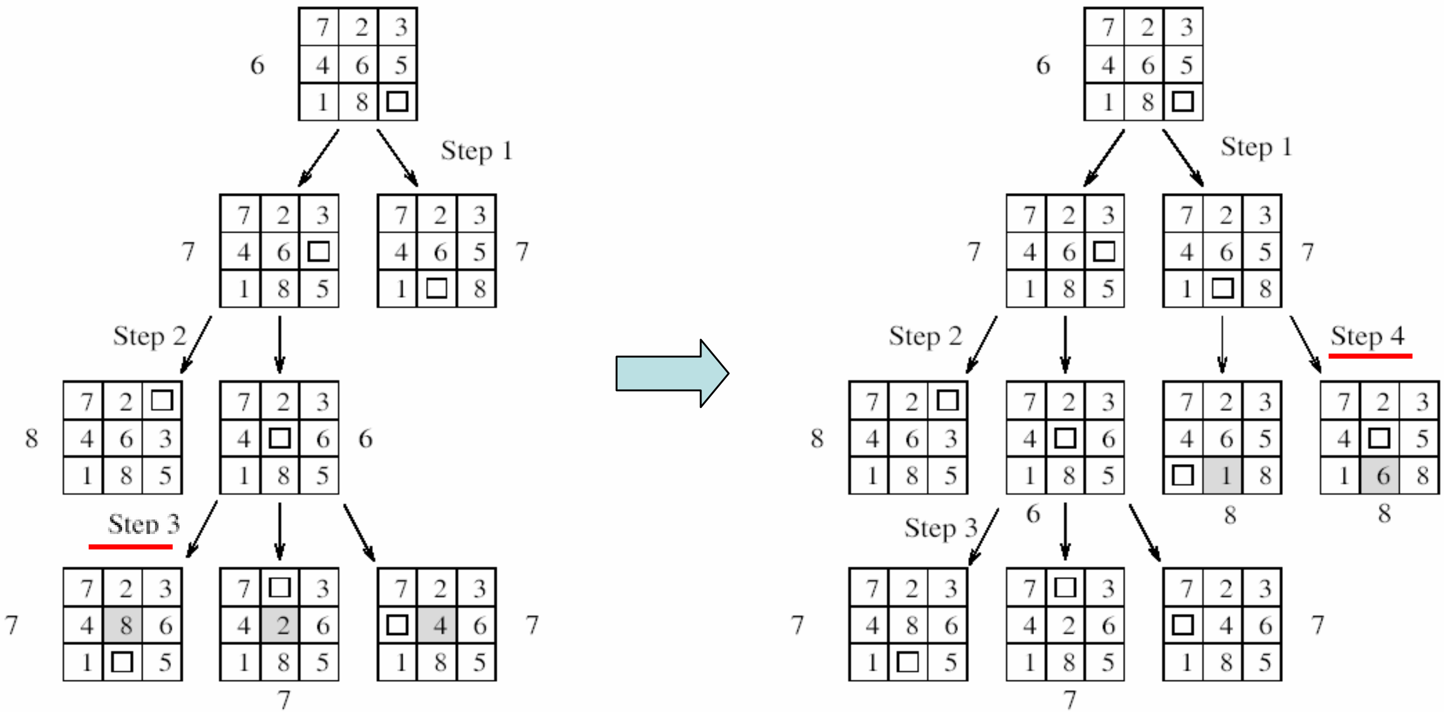
- Example 2: the 8-puzzle problem

2+0+0+0+1+1+2+0=6 (Manhattan distance)



Greedy Best-First Search

- Example 2: the 8-puzzle

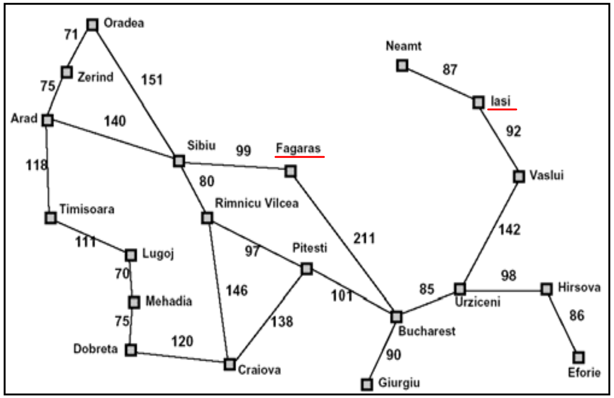


(c)

Figure 11.6 Applying best-first search to the 8-puzzle: (a) initial configuration; (b) final configuration; and (c) states resulting from the first four steps of best-first search. Each state is labeled with its h-value (that is, the Manhattan distance from the state to the final state).

Properties of Greedy Best-First Search

- Prefer to follow a single path all the way to the goal, and will back up when dead end is hit (like DFS)
  - Also have the possibility to go down infinitely
- Is neither optimal nor complete
  - Not complete: *could get suck in loops*
    - E.g., finding path from *Iasi* to *Fagars*



- Time and space complexity
  - Worse case: $O(b^m)$
  - But a good heuristic function could give dramatic improvement

A* Search

- Pronounced as “A-star search”

- Expand a node by evaluating the path cost to reach itself, $g(n)$, and the estimated path cost from it to the goal, $h(n)$
  - Evaluation function

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

$$g(n) = \text{path cost so far to reach } n$$
$$h(n) = \text{estimated path cost to goal from } n$$
$$f(n) = \text{estimated total path cost through } n \text{ to goal}$$

  - Uniform-cost search + greedy best-first search ?
  - Avoid expanding nodes that are already expansive

A* Search

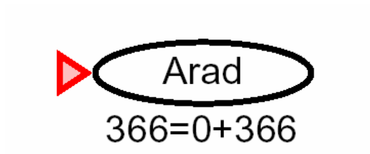
- A* is optimal if the heuristic function $h(n)$ never overestimates
  - Or say "if the heuristic function is admissible"
  - E.g. the straight-line-distance heuristics are admissible

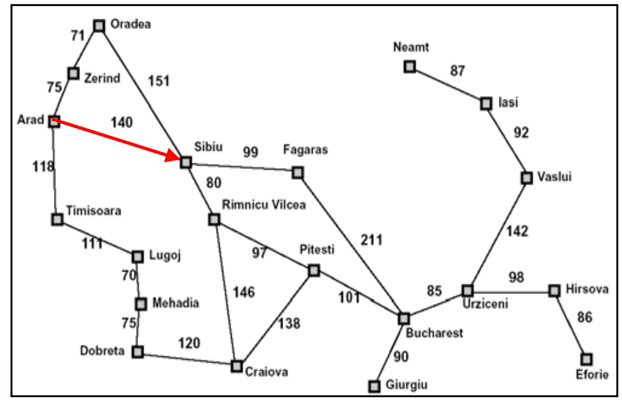
$$\boxed{\begin{array}{l} h(n) \le h^*(n), \\ \text{where } h^*(n) \text{ is the true path cost from } n \text{ to goal} \end{array}}$$

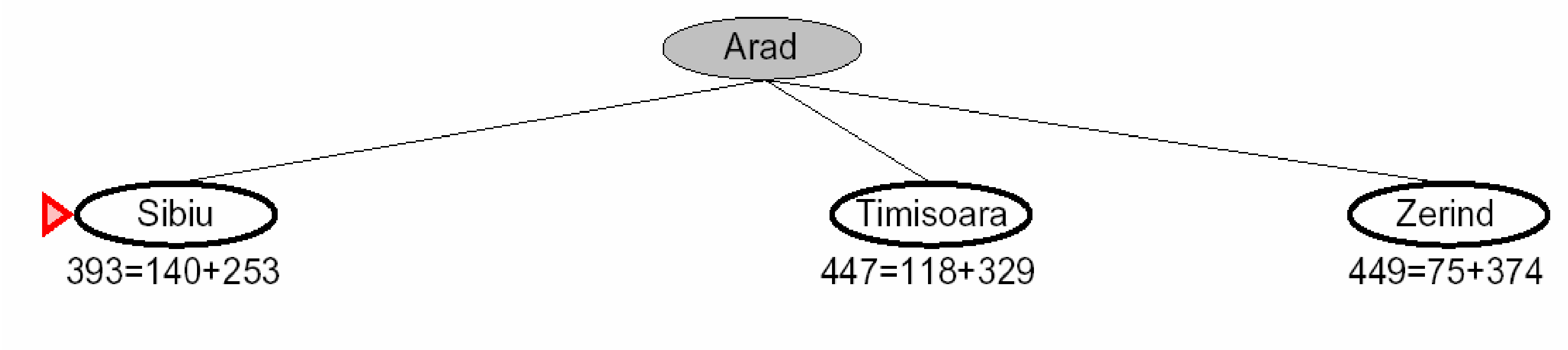
Finding the shortest-path goal

A* Search

- Example 1: the route-finding problem

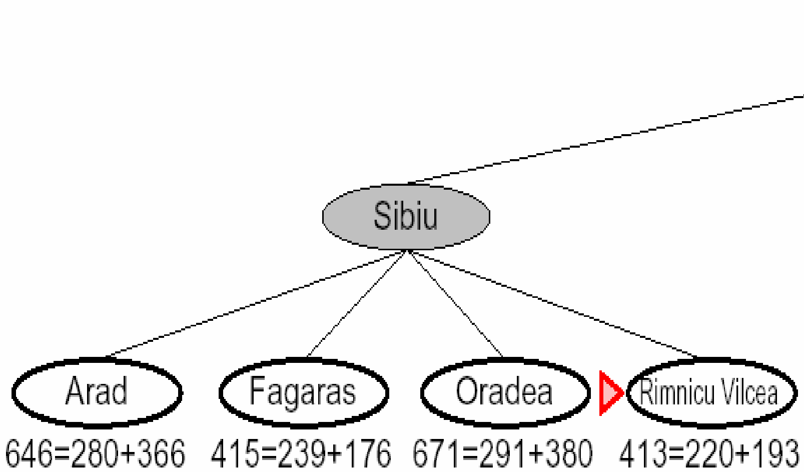


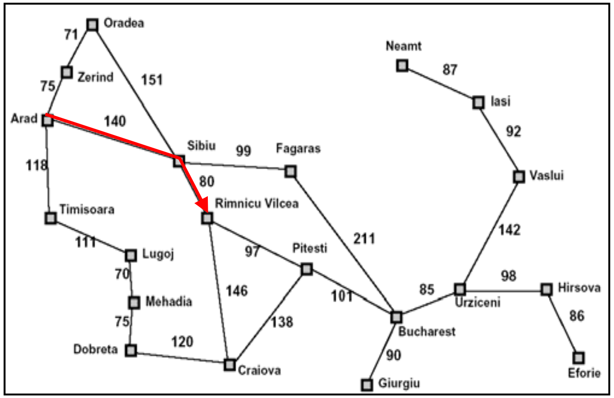




A* Search

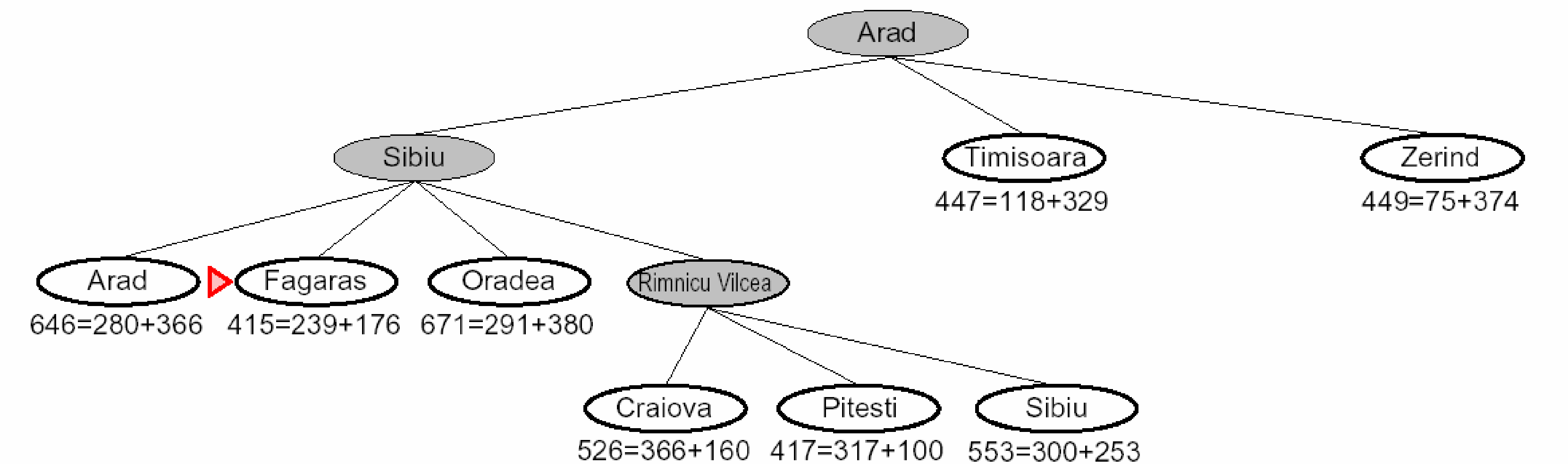
- Example 1: the route-finding problem

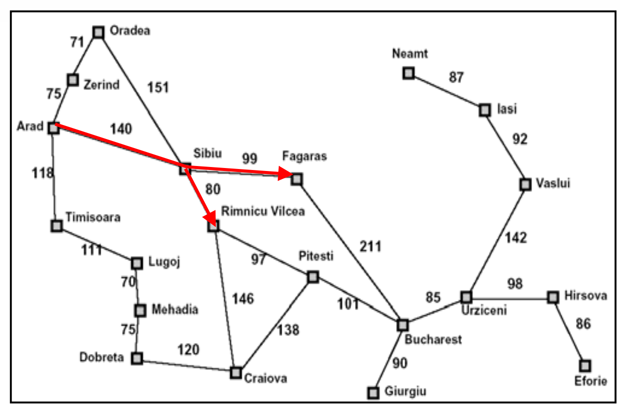




A* Search

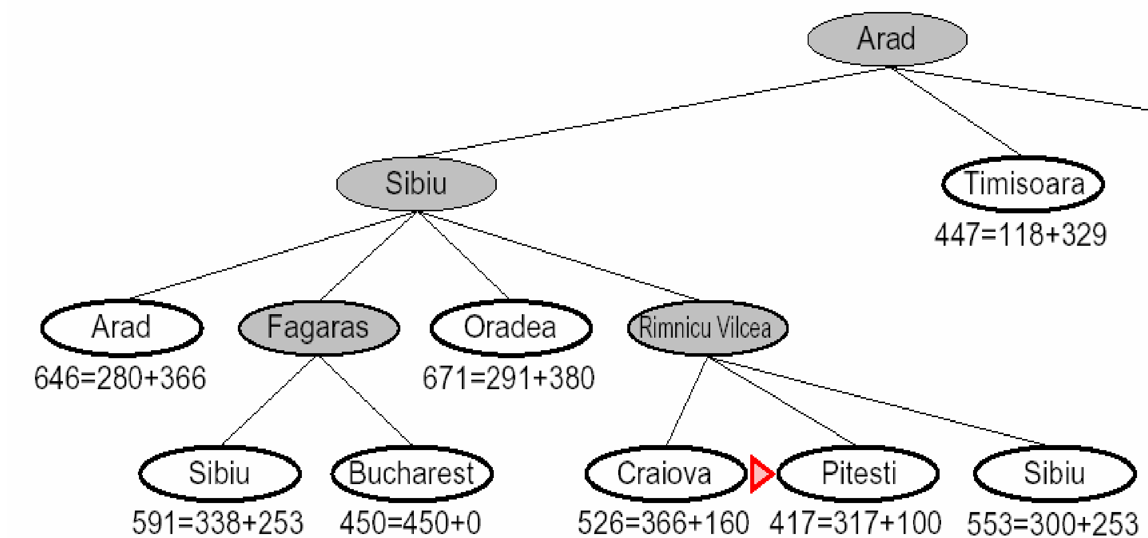
- Example 1: the route-finding problem

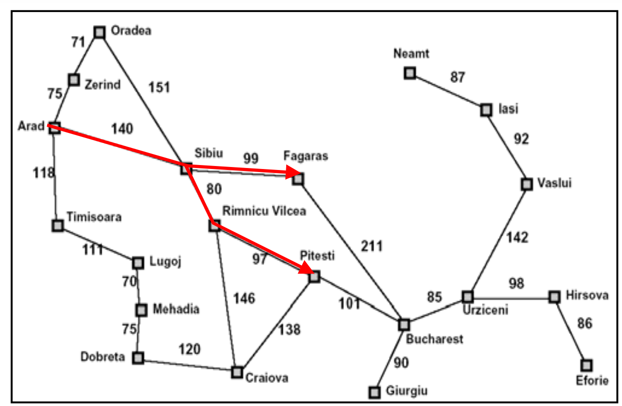




A* Search

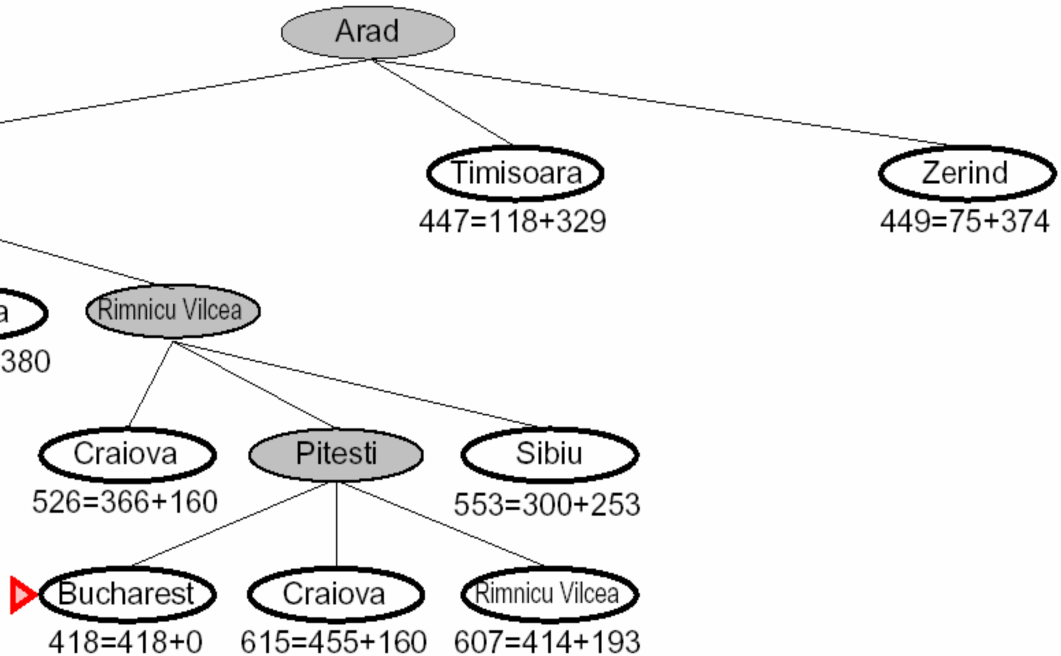
- Example 1: the route-finding problem

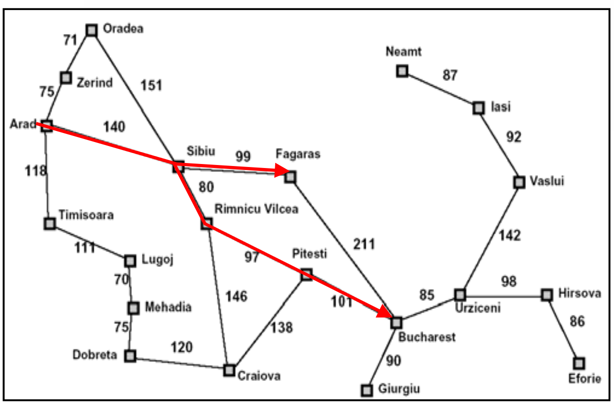




A* Search

- Example 1: the route-finding problem



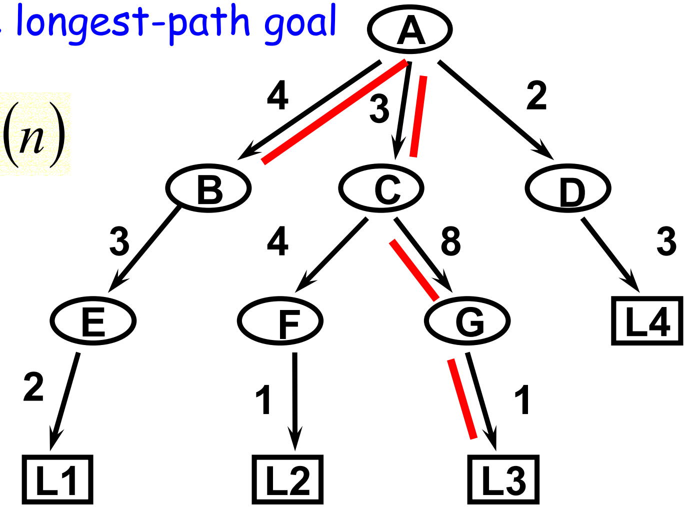


A* Search

- Example 2: the state-space just represented as a tree

Finding the longest-path goal

$h(n) \geq h^*(n)$



Evaluation function of node n:

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

Node	g(n)	h(n)	f(n)
A	0	15	15
B	4	9	13
C	3	12	15
D	2	5	7
E	7	4	11
F	7	2	9
G	11	3	14
L1	9	0	9
L2	8	0	8
L3	12	0	12
L4	5	0	5

Fringe *(sorted)*

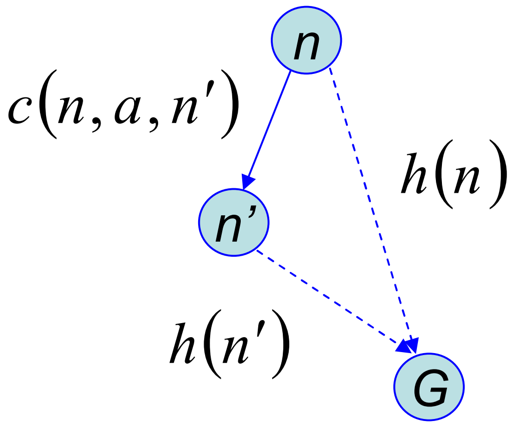
Fringe Top	*Fringe Elements*
A(15)	A(15)
C(15)	C(15), B(13), D(7)
G(14)	G(14), B(13), F(9), D(7)
B(13)	B(13), L3(12), F(9), D(7)
L3(12)	L3(12), E(11), F(9), D(7)

Consistency of A* Heuristics

- A heuristic *h* is *consistent* if

$$h(n) \leq c(n, a, n') + h(n')$$

  - A stricter requirement on *h*



Finding the shortest-path goal

, where *h*(•) is the straight-line distance to the nearest goal

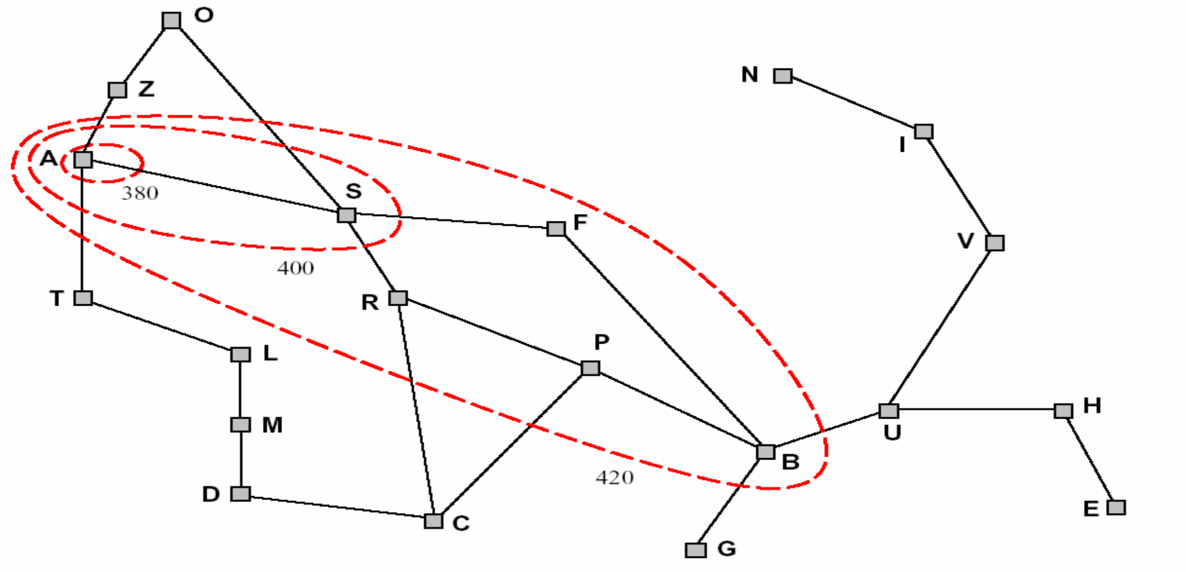
- If *h* is consistent

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

  - I.e., $f(n)$ is nondecreasing along any path during search

Contours of the Evaluation Functions

- Fringe (leaf) nodes expanded in concentric contours



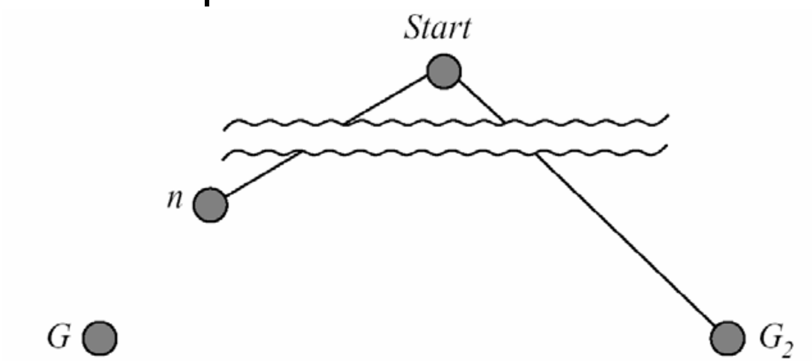
- Uniformed search ($\forall n, h(n) = 0$)
  - Bands circulate around the initial state
- A* search
  - Bands stretch toward the goal and is narrowly focused around the optimal path if more accurate heuristics were used

Contours of the Evaluation Functions

- If G is the optimal goal
  - A* search expands all nodes with $f(n) < f(G)$
  - A* search expands some nodes with $f(n)=f(G)$
  - A* expands no nodes with $f(n) > f(G)$

Optimality of A* Search

- A* search is optimal
- Proof
  - Suppose some suboptimal goal G_2 has been generated and is in the fringe (queue)
  - Let n be an unexpanded node on a shortest path to an optimal goal G



$f(G_2) = g(G_2)$	since $h(G_2) = 0$
$> g(G) \, (= g(n) + h^*(n))$	since G_2 is suboptiomal
$\geq f(n) \; (= g(n) + h(n))$	since h is admissible $(h(n) \leq h^*(n))$

  - A* will never select G_2 for expansion since $f(G_2) > f(n)$

Optimality of A* Search

- Another proof
  - Suppose when algorithm terminates, G_2 is a complete path (a solution) on the top of the fringe and a node n that stands for a partial path presents somewhere on the fringe. There exists a complete path G passing through n, which is not equal to G_2 and is optimal (with the lowest path cost)

1. G is a complete which passes through node n, $f(G)>=f(n)$
2. Because G_2 is on the top of the fringe ,
   $f(G_2)<=f(n)<=f(G)$
3. Therefore, it makes contrariety !!

- A* search optimally efficient
  - For any given heuristic function, no other optimal algorithms is guaranteed to expand fewer nodes than A*

Completeness of A* Search

- A* search is complete
  - If every node has a finite branching factor
  - If there are finitely many nodes with $f(n) \leq f(G)$
    - Every infinite path has an infinite path cost

Proof:
Because A* adds bands (expands nodes) in order of increasing f, it must eventually reach a band where f is equal to the path to a goal state.

- To Summarize again *If G* is the optimal goal

$$\text{A}^* \text{ expands all nodes with } f(n) < f(G)$$

$$\text{A}^* \text{ expands smoe nodes with } f(n) = f(G)$$

$$\text{A}^* \text{ expands no nodes with } f(n) > f(G)$$

Complexity of A* Search

- Time complexity: $O(b^d)$
- Space complexity: $O(b^d)$
  - Keep all nodes in memory
  - Not practical for many large-scale problems
- Theorem
  - The search space of A* grows exponentially unless the error in the heuristic function grows no faster than the logarithm of the actual path cost

$$\left|h(n) - h^*(n)\right| \le O\left(\log h^*(n)\right)$$

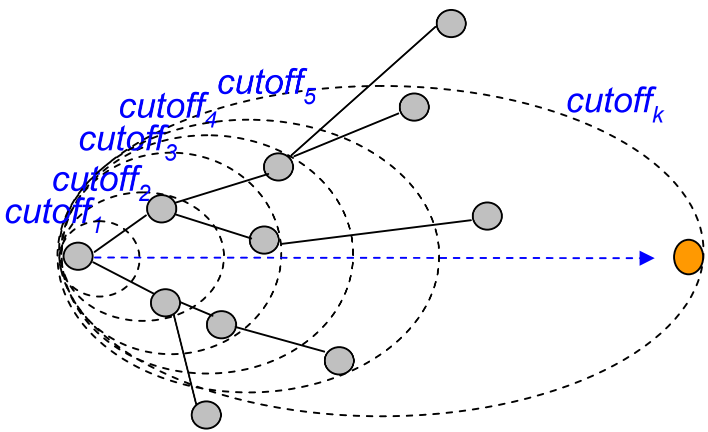
Memory-bounded Heuristic Search

- Iterative-Deepening A* search
- Recursive best-first search
- Simplified memory-bounded A* search

Iterative Deepening A* Search (IDA*)

- The idea of iterative deepening was adapted to the heuristic search context to reduce memory requirements

- At each iteration, DFS is performed by using the f-cost ($g + h$) as the cutoff rather than the depth
  - E.g., the smallest f-cost of any node that exceeded the cutoff on the previous iteration



Iterative Deepening A* Search

```
function IDA*(problem) returns a solution sequence
   inputs: problem, a problem
   static: f-limit, the current f- COST limit
           root, a node

   root ← MAKE-NODE(INITIAL-STATE[problem])
   f-limit ← f- COST(root)
   loop do
      solution, f-limit ← DFS-CONTOUR(root, f-limit)
      if solution is non-null then return solution
      if f-limit = ∞ then return failure; end
________________________________________________________________
function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f- COST limit
   inputs: node, a node
           f-limit, the current f- COST limit
   static: next-f, the f- COST limit for the next contour, initially ∞

   if f- COST[node] > f-limit then return null, f- COST[node]
   if GOAL-TEST[problem](STATE[node]) then return node, f-limit
   for each node s in SUCCESSORS(node) do
      solution, new-f ← DFS-CONTOUR(s, f-limit)
      if solution is non-null then return solution, f-limit
      next-f ← MIN(next-f, new-f); end
   return null, next-f
```

Properties of IDA*

- IDA* is complete and optimal
- Space complexity: $O(bf(G)/\delta) \approx O(bd)$
  - δ : the smallest step cost
  - $f(G)$: the optimal solution cost
- Time complexity: $O(\alpha b^d)$
  - α : the number of distinct f values small than the optimal goal
- Between iterations, IDA* retains only a single number – the f -cost
- IDA* has difficulties in implementation when dealing with real-valued cost

Recursive Best-First Search (RBFS)

- Attempt to mimic best-first search but use only linear space
  - Can be implemented as a recursive algorithm
  - Keep track of the f -value of the best alternative path from any ancestor of the current node
  - It the current node exceeds the limit, the recursion unwinds back to the alternative path
  - As the recursion unwinds, the f -value of each node along the path is replaced with the best f -value of its children

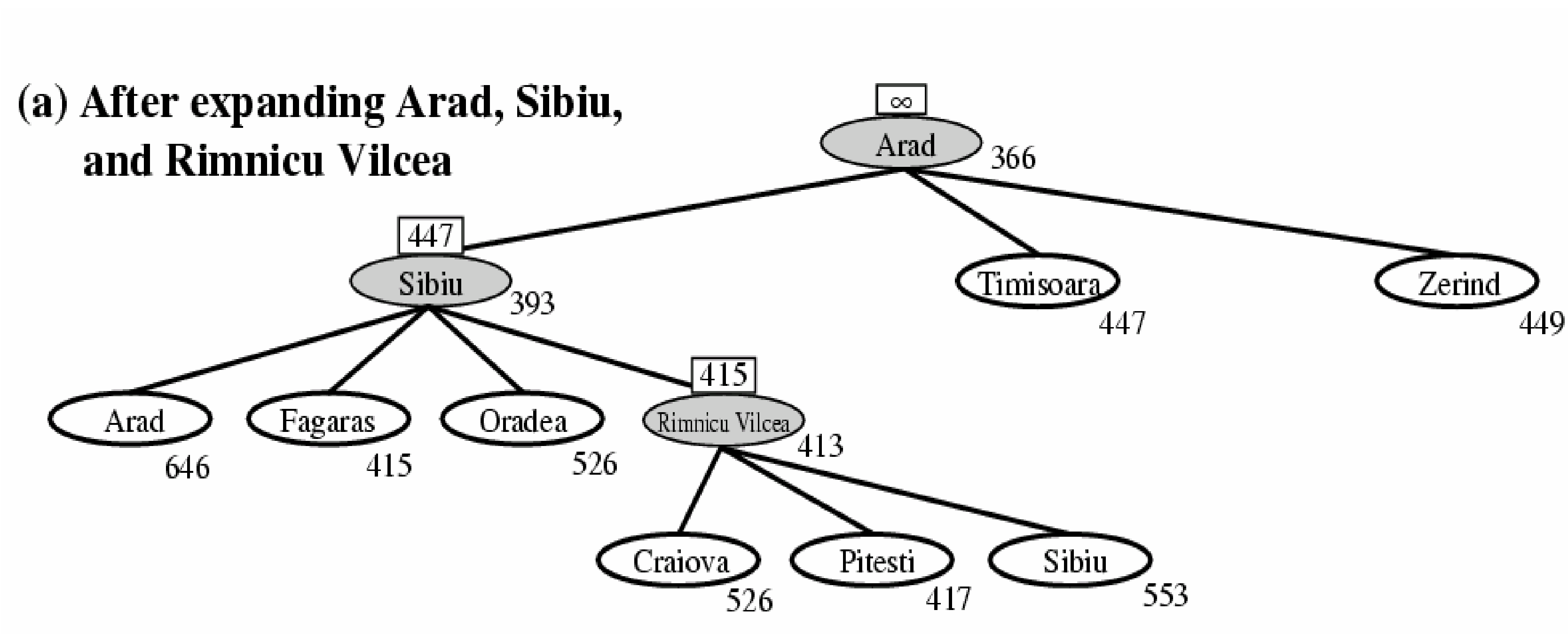
Recursive Best-First Search (RBFS)

- Algorithm

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
   RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
   if GOAL-TEST[problem](state) then return node
   successors ← EXPAND(node, problem)
   if successors is empty then return failure, ∞
   for each s in successors do
      f[s] ← max(g(s) + h(s), f[node])
   repeat
      best ← the lowest f-value node in successors
      if f[best] > f_limit then return failure, f[best]
      alternative ← the second-lowest f-value among successors
      result, f[best] ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

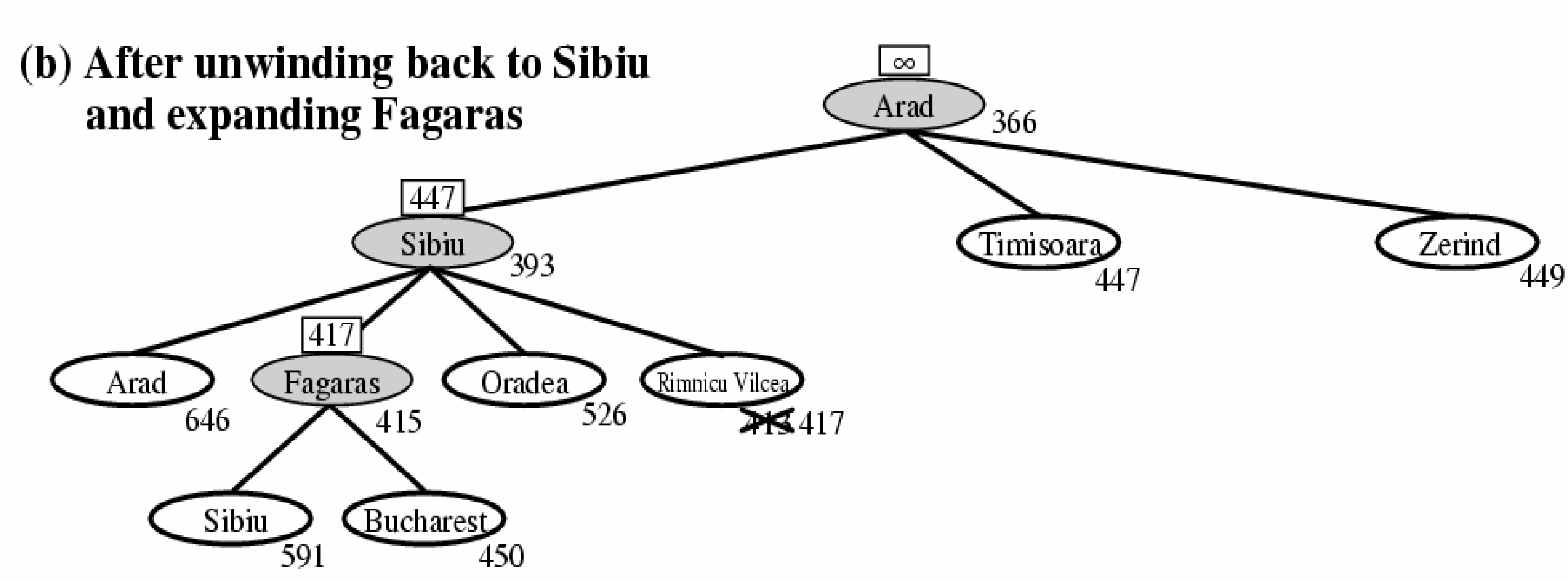
Recursive Best-First Search

- Example: the route-finding problem



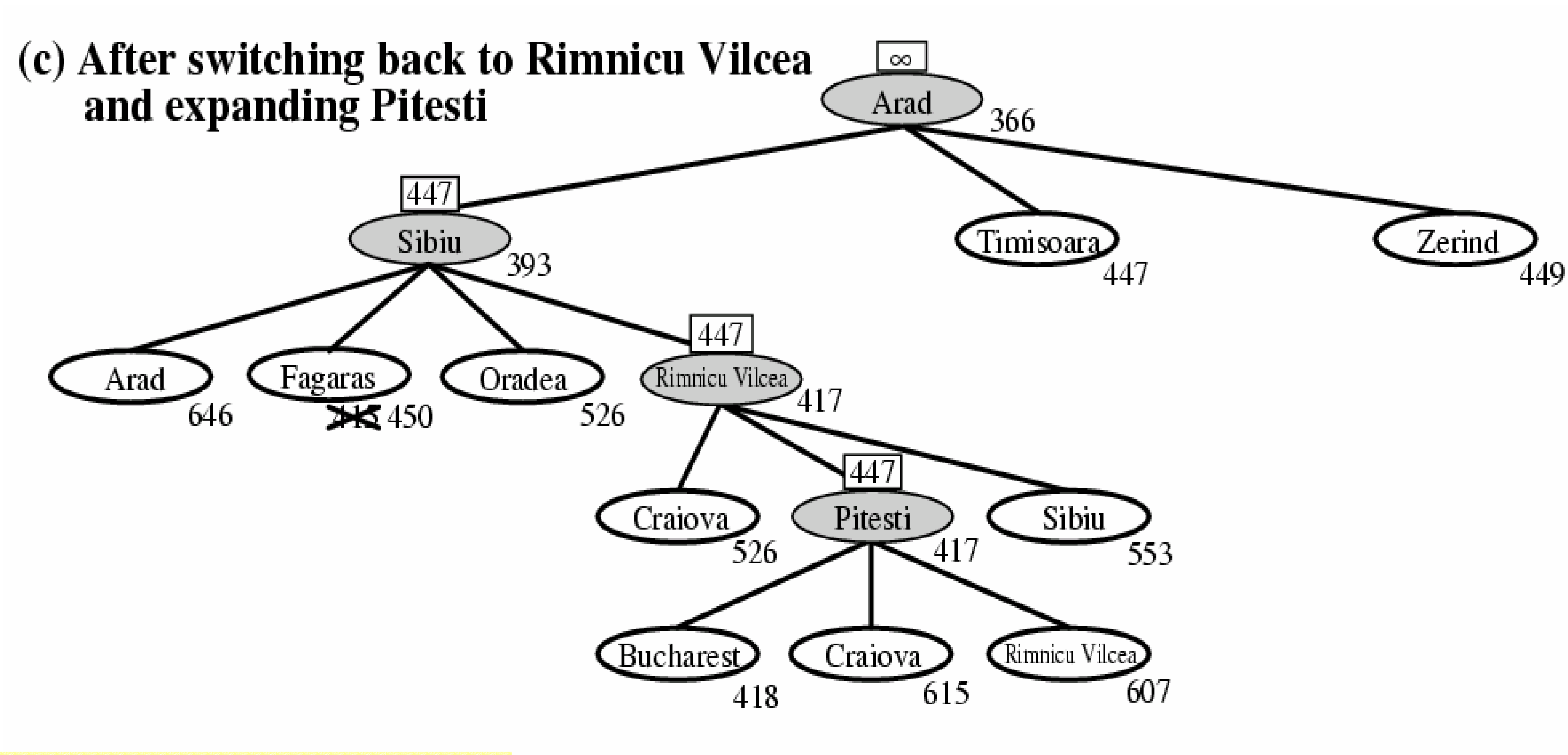
Recursive Best-First Search

- Example: the route-finding problem



Recursive Best-First Search

- Example: the route-finding problem



Re-expand the forgotten nodes
(subtree of Rimnicu Vilcea)

Properties of RBFS

- RBFS is complete and optimal
- Space complexity: $O(bd)$
- Time complexity : worse case $O(b^d)$
  - Depend on the heuristics and frequency of “mind change”
  - The same states may be explored many times

Simplified Memory-Bounded A* Search (SMA*)

- Make use of all available memory *M* to carry out A*
- Expanding the best leaf like A* until memory is full
- When full, drop the worst leaf node (with highest f -value)
  - Like RBFS, backup the value of the forgotten node to its parent if it is the best among the subtree of its parent
  - When all children nodes were deleted/dropped, put the parent node to the fringe again for further expansion

Simplified Memory-Bounded A* Search

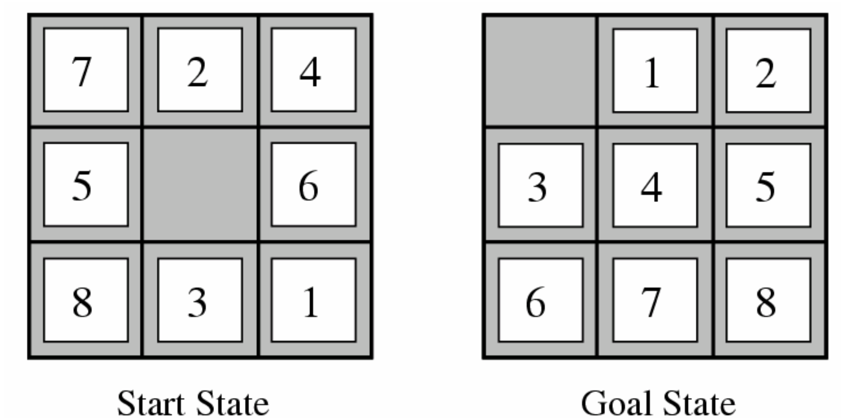
```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
     if Queue is empty then return failure
     n ← deepest least-f-cost node in Queue
     if GOAL-TEST(n) then return success
     s ← NEXT-SUCCESSOR(n)
     if s is not a goal and is at maximum depth then
        f(s) ← ∞
     else
        f(s) ← MAX(f(n), g(s)+h(s))
     if all of n's successors have been generated then
        update n's f-cost and those of its ancestors if necessary
     if SUCCESSORS(n) all in memory then remove n from Queue
     if memory is full then
        delete shallowest, highest-f-cost node in Queue
        remove it from its parent's successor list
        insert its parent on Queue if necessary
     insert s on Queue
  end
```

Properties of SMA*

- Is complete if $M \geq d$
- Is optimal if M $\geq$ d^*
- Space complexity: $O(M)$
- Time complexity : worse case $O(b^d)$

Admissible Heuristics

- Take the 8-puzzle problem for example
  - Two heuristic functions considered here
    - $h_1(n)$: number of misplaced tiles
    - $h_2(n)$: the sum of the distances of the tiles from their goal positions (tiles can move vertically, horizontally), also called Manhattan distance or city block distance



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

Admissible Heuristics

- Take the 8-puzzle problem for example
  - Comparison of IDS and A*

branching factor for 8-puzzle: 2~4

solution length

	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	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	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 4.8 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 various solution lengths.

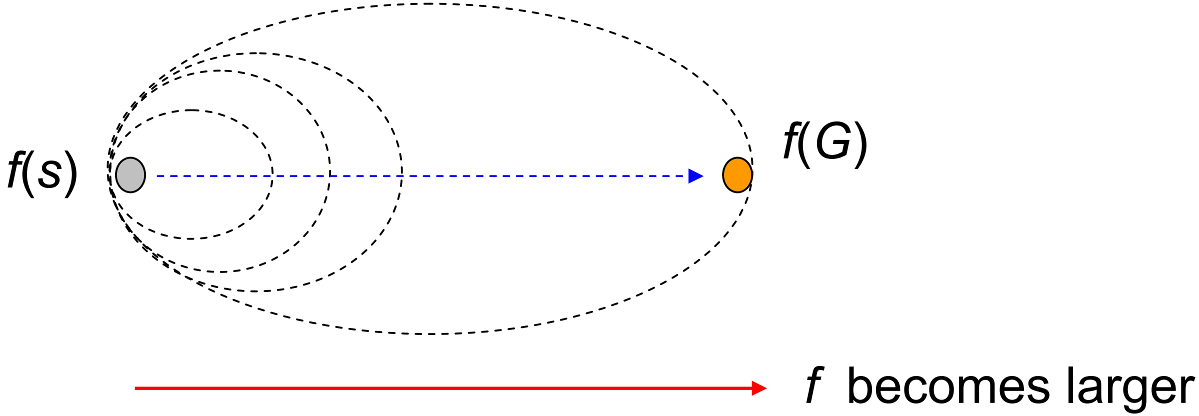
100 random problems for each number

$$\underbrace{N+1}_{\text{Nodes generated by A*}}=1+b^*+(b^*)^2+(b^*)^3+\ldots+(b^*)^d$$

b^*: effective branching factor

Dominance

- For two heuristic functions h_1 and h_2 (both are admissible), if $h_2(n) \geq h_1(n)$ for all nodes n
  - Then h_2 **dominates** h_1 and is better for search
  - A* using h_2 will not expand more node than A* using h_1



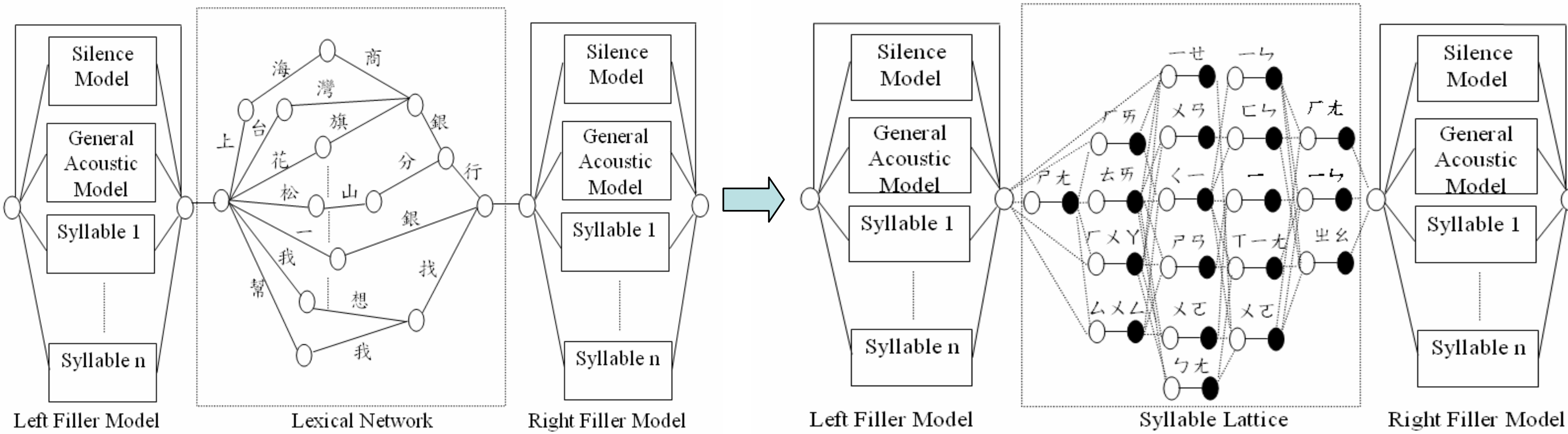
Inventing Admissible Heuristics

- Relaxed Problems
  - The search heuristics can be achieved from the relaxed versions the original problem
    - Key point: the optimal solution cost to a relaxed problem is an admissible heuristic for the original problem
      (not greater than the optimal solution cost of the original problem)

  - Example 1: the 8-puzzle problem
    - If the rules 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 any adjacent square then $h_2(n)$ gives the shortest solution

Inventing Admissible Heuristics

– Example 2: the speech recognition problem



Original Problem
(keyword spotting)

Relaxed Problem
(used for heuristic calculation)

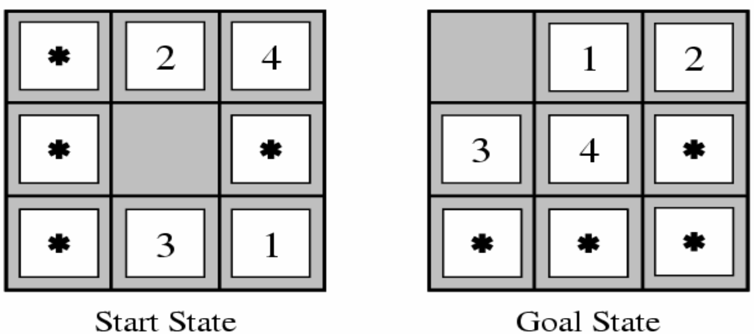
Note: if the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expansive to obtain

Inventing Admissible Heuristics

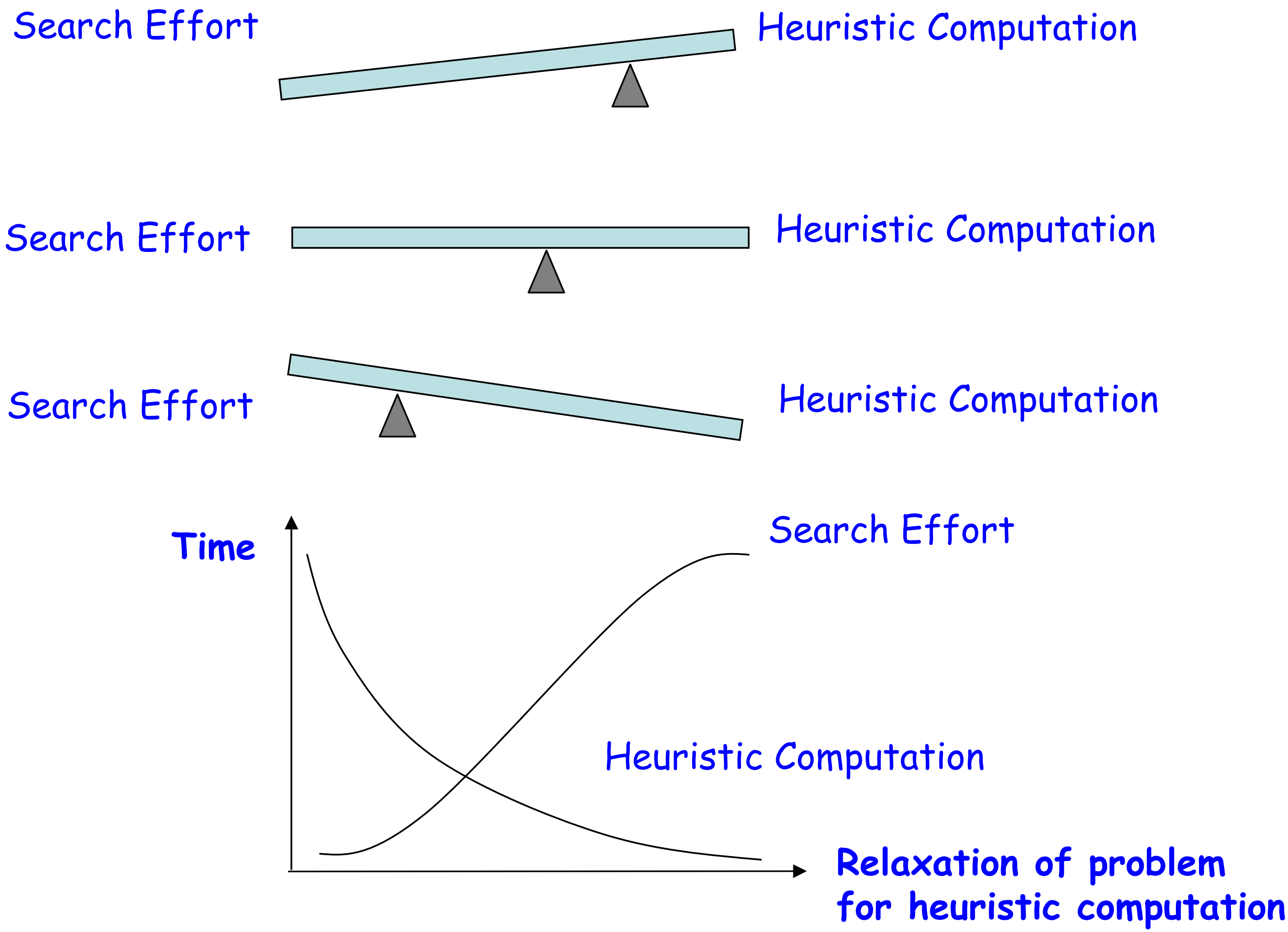
- Composite Heuristics
  - Given a collection of admissible heuristics $h_1, h_2, \ldots, h_m$, none of them dominates any of other

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

- Subproblem Heuristics
  - The cost of the optimal solution of the subproblem is a lower bound on the cost of the complete problem



Tradeoffs



Search Effort
Heuristic Computation
Search Effort
Heuristic Computation
Search Effort
Heuristic Computation
Time
Search Effort
Heuristic Computation
Relaxation of problem
for heuristic computation

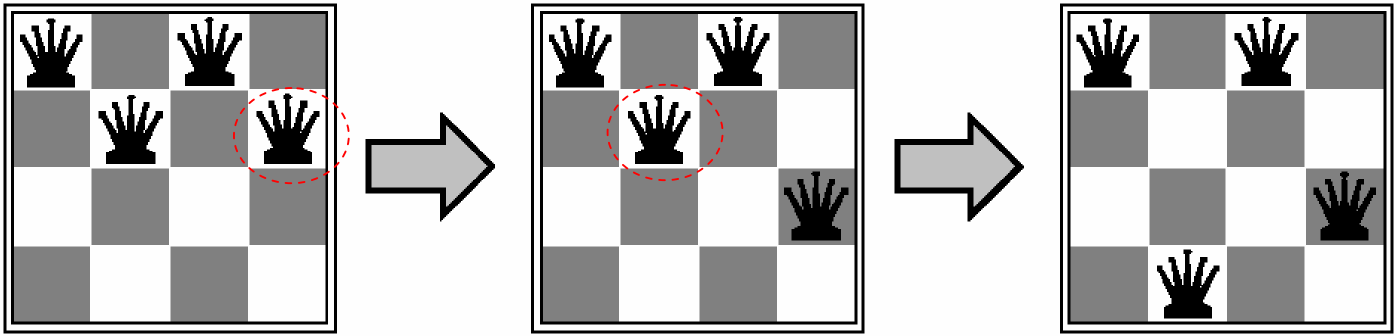
Iterative Improvement Algorithms

- In many optimization, path to solution is irrelevant
  - E.g., 8-queen, VLSI layout, TSP etc., for finding optimal configuration
  - The goal state itself is the solution
  - The state space is a complete configuration

- In such case, iterative improvement algorithms can be used
  - Start with a complete configuration (represented by a single "current" state)
  - Make modifications to improve the quality

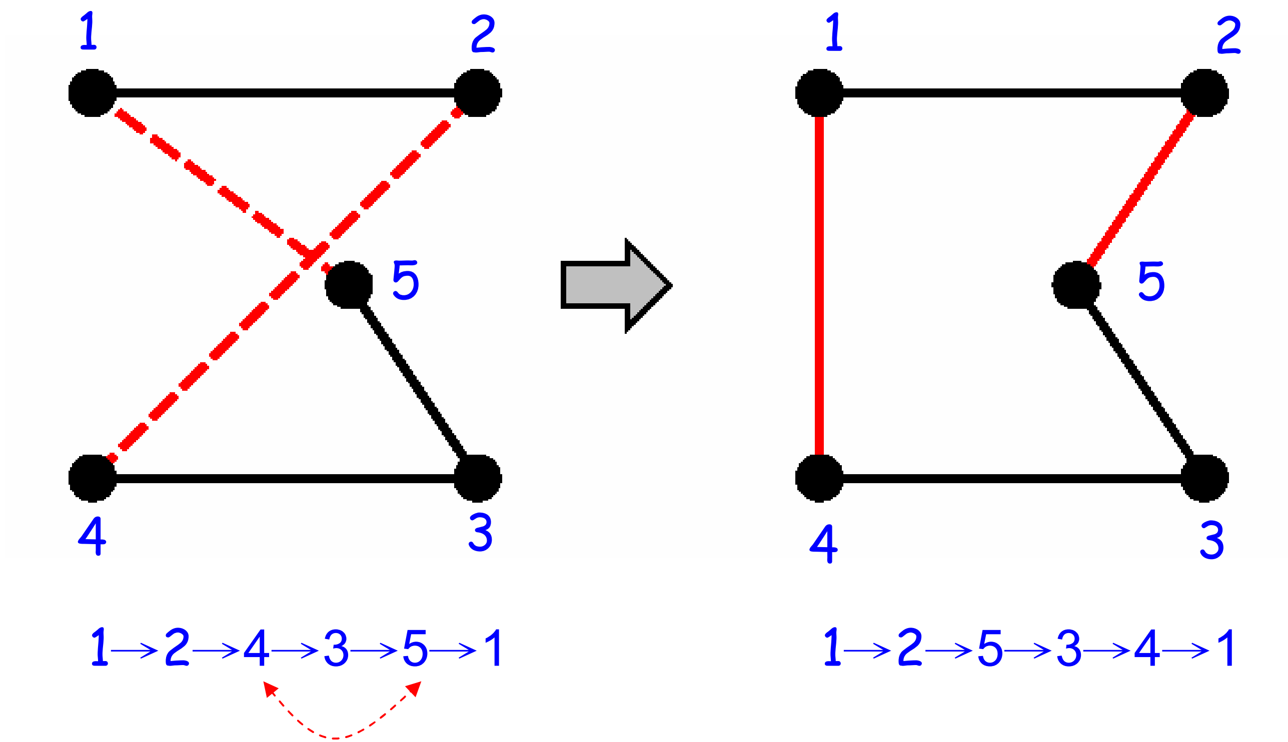
Iterative Improvement Algorithms

- Example: the *n*-queens problem
  - Put *n* queens on an *nxn* board with no queens on the same row, column, or diagonal
  - Move a queen to reduce number of conflicts



Iterative Improvement Algorithms

- Example: the traveling salesperson problem (TSP)
  - Find the shortest tour visiting all cities exactly one
  - Start with any complete tour, perform pairwise exchanges

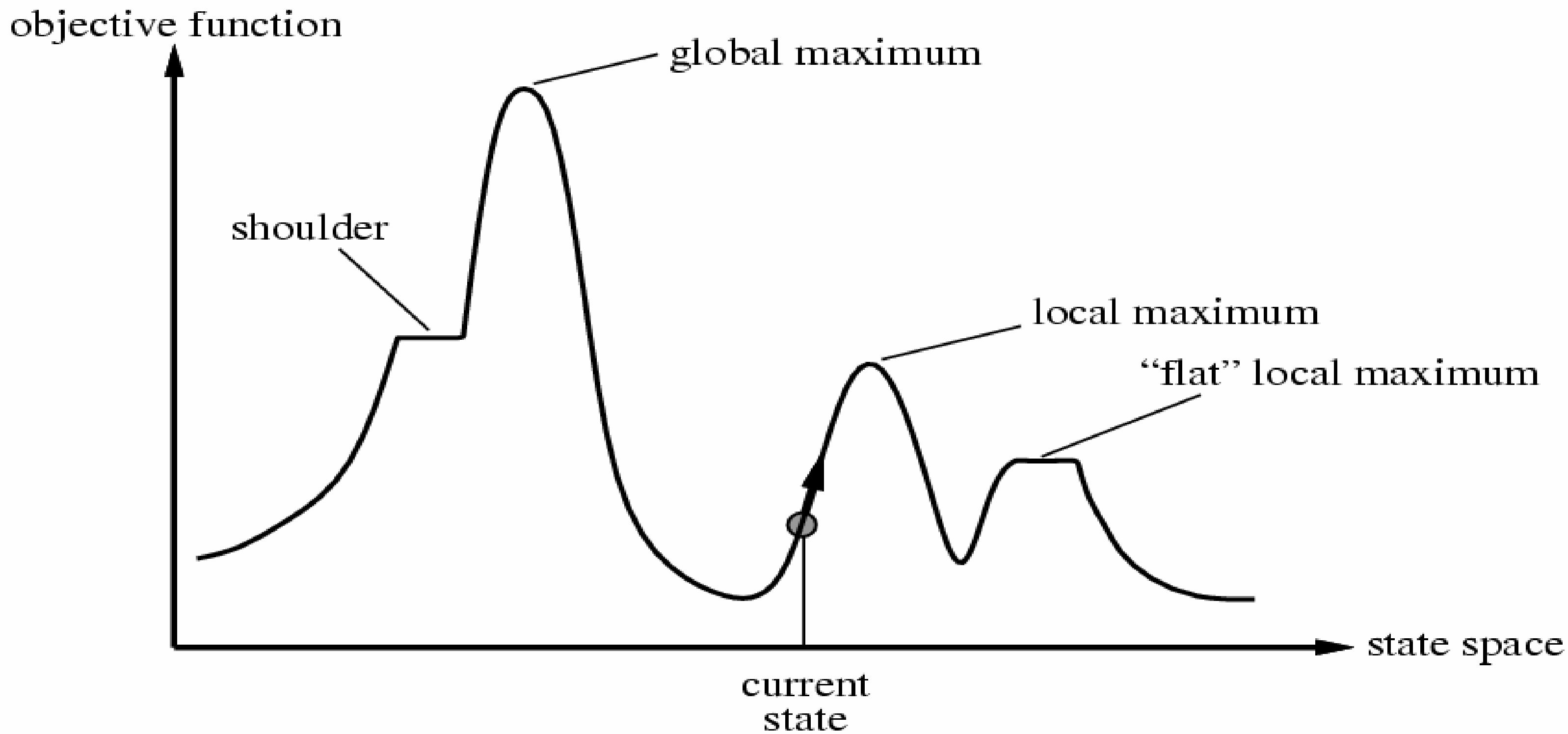


Iterative Improvement Algorithms

- *Local search algorithms* belongs to iterative improvement algorithms
  - Use a current state and generally move only to the neighbors of that state
  - Properties
    - Use very little memory
    - Applicable to problems with large or infinite state space

- Local search algorithms to be considered
  - Hill-climbing search
  - Simulated annealing
  - Local beam search
  - Genetic algorithms

Iterative Improvement Algorithms



- Completeness or optimality of the local search algorithms should be considered

Hill-Climbing Search

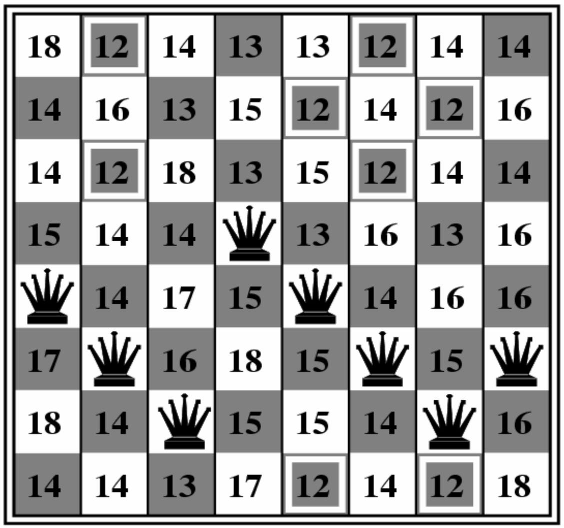
- "Like climbing Everest in the thick fog with amnesia"
- Choose any successor with a higher value (of objective or heuristic functions) than current state
  - Choose Value[next] $\geq$ Value[current]

```
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
```

- Also called *greedy local search*

Hill-Climbing Search

- Example: the 8-queens problem
  - The heuristic cost function is the number of pairs of queens that are attacking each other



  - h=3+4+2+3+2+2+1=17 (calculated from left to right)
  - Best successors have h=12
    (when one of queens in Column 2,5,6, and 7 is moved)

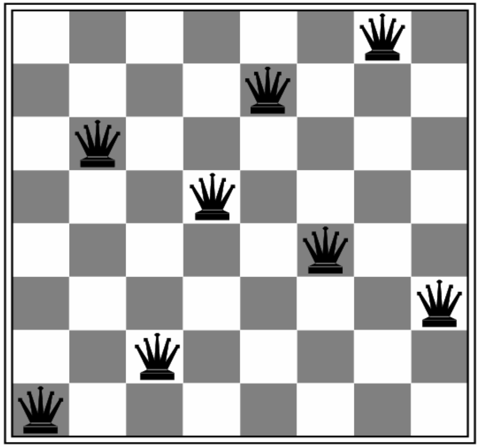
Hill-Climbing Search

- Problems:
  - Local maxima: search halts prematurely
  - Plateaux: search conducts a random walk
  - Ridges: search oscillates with slow progress

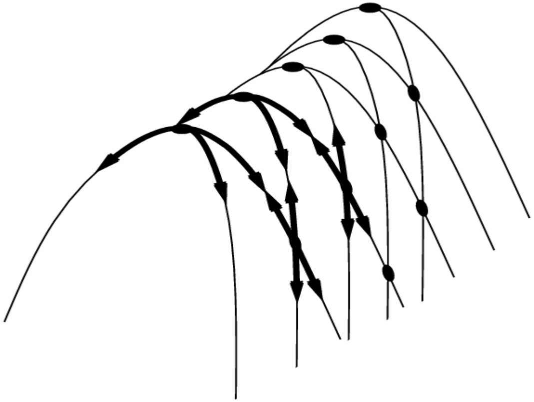
Neither complete nor optimal

- Solution
  - Start from randomly generated initial states
    - Random-restart hill climbing

8-queens stuck in a local minimum



Ridges cause oscillation



Simulated Annealing Search

- Combine hill climbing with a random walk to yield both efficiency and completeness
  - Pick a random move at each iteration instead of picking the best move
  - If the move improve the situation $\rightarrow$ accept!

$$\Delta E = \text{VALUE}[next] - \text{VALUE}[current]$$

  - Otherwise($\Delta E < 0$) , have a probability ($e^{\Delta E / T}$) to move to a worse state
    - The probability decreases exponentially as ΔE decreases
    - The probability decreases exponentially as T (temperature) goes down (as time goes by)

Simulated Annealing Search

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
      schedule, a mapping from time to "temperature"
  local variables: *current*, a node
      next, a node
      T, a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

Local Beam Search

- Keep track of *k* states rather than just one
  - Begin with *k* randomly generated states
  - All successors of the *k* states are generated at each iteration
    - If any one is a goal → halt!
    - Otherwise, select *k* best successors from them and continue the iteration
  - Information is passed/exchanged among these *k* search threads
    - Compared to the random-restart search

Local Beam Search

- Problem
  - The k states may quickly become concentrated in a small region of the state space
  - Like an expensive version of hill climbing
- Solution
  - A variant version called stochastic beam search
    - Choose a given successor at random with a probability in increasing function of its value

Genetic Algorithms (GAs)

- Developed and patterned after biological evolution

- Also regarded as a variant of stochastic beam search
  - Successors are generated from multiple current states
    - A population of potential solutions are maintained
  - States are often described by bit strings (like chromosomes) whose interpretation depends on the applications
    - Binary-coded or alphabet

      (11, 6, 9) $\rightarrow$ (101101101001)
    - Encoding: translate problem-specific knowledge to GA framework
  - Search begins with a population of randomly generated initial states

Genetic Algorithms (GAs)

- The successor states are generated by combining two parent states, rather then by modifying a single state
  - Current population/states are evaluated with a *fitness function* and selected probabilistically as seeds for producing the next generation
    - Fitness function: the criteria for ranking
    - Recombine parts of the best (most fit) currently known states
    - Generate-and-test beam search

- Three phases of GAs
  - Selection → Crossover → Mutation

Genetic Algorithms (GAs)

- Selection
  - Determine which parent strings (chromosomes) participate in producing offspring for the next generation
  - The selection probability is proportional to the fitness values

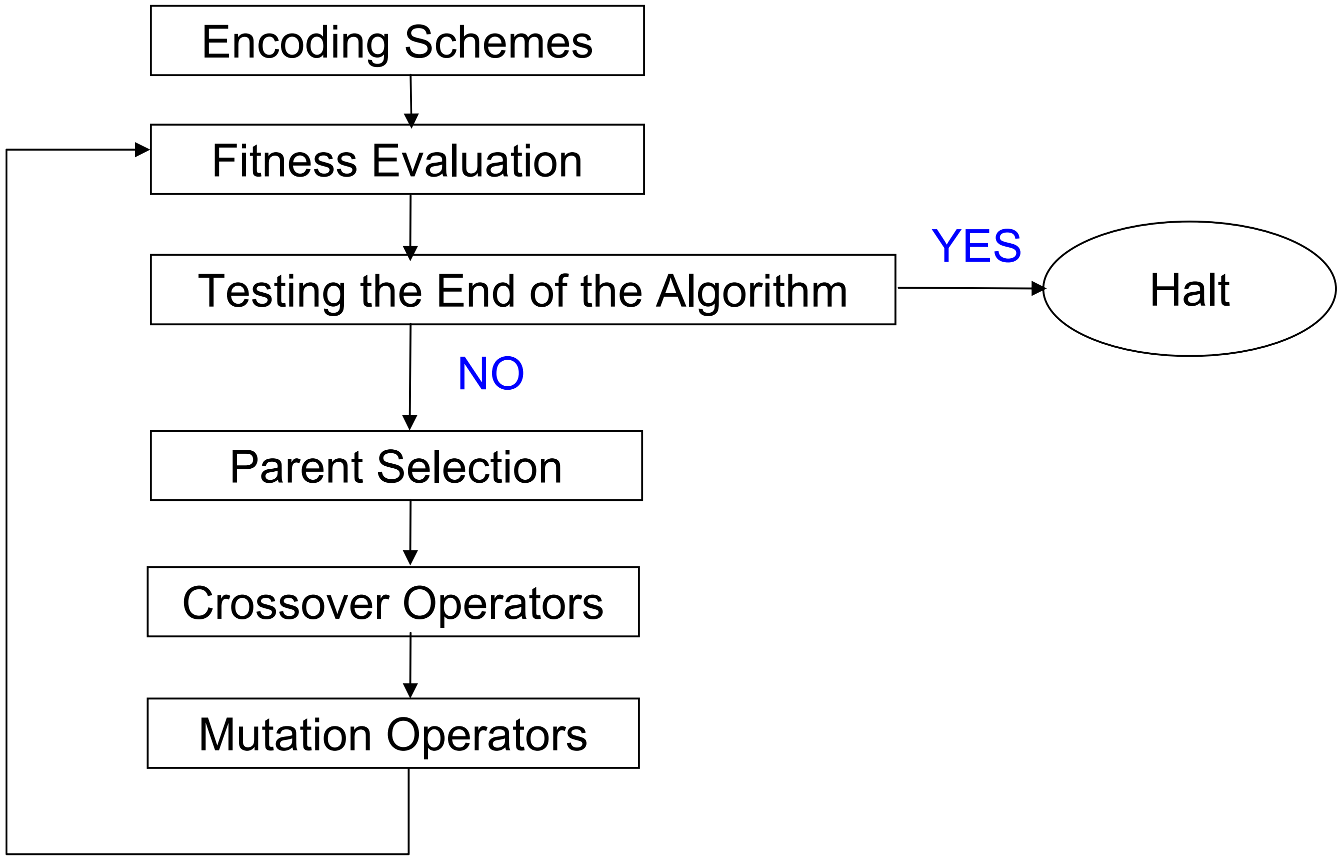
$$\Pr(h_i) = \frac{Fitness\,(h_i)}{\sum_{j=1}^{P} Fitness\,(h_j)}$$

  - Some strings (chromosomes) would be selected more than once

Genetic Algorithms

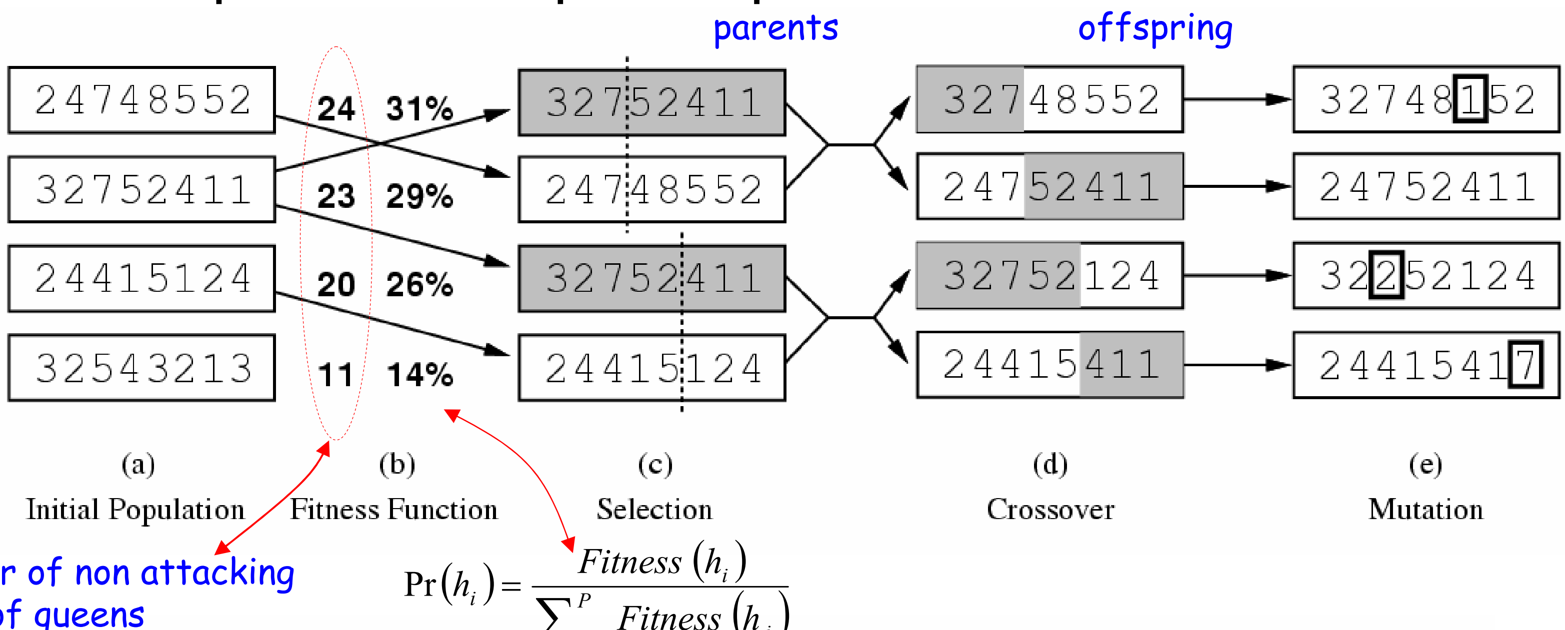
- Two most common (genetic) operators which try to mimic biological evaluation are performed at each iteration
  - Crossover
    - Produce new offspring by crossing over the two mated parent strings at randomly (a) chosen crossover point(s) (bit position(s))
    - Selected bits copied from each parent
  - Mutation
    - Often performed after crossover
    - Each (bit) location of the randomly selected offspring is subject to random mutation with a small independent probability
- Applicable problems
  - Function approximation & optimization, circuit layout etc.

Genetic Algorithms

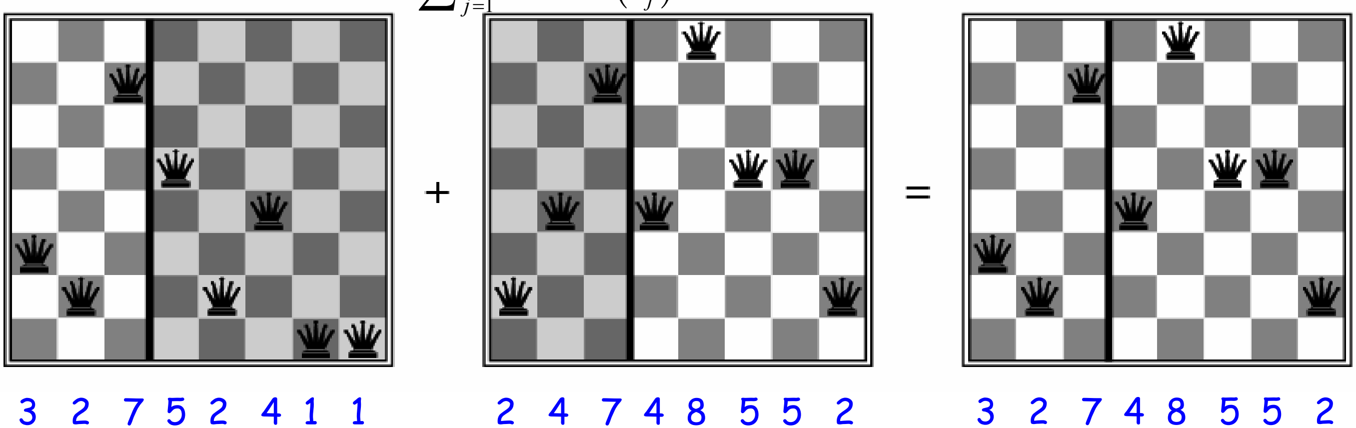


Genetic Algorithms

- Example 1: the 8-queens problem

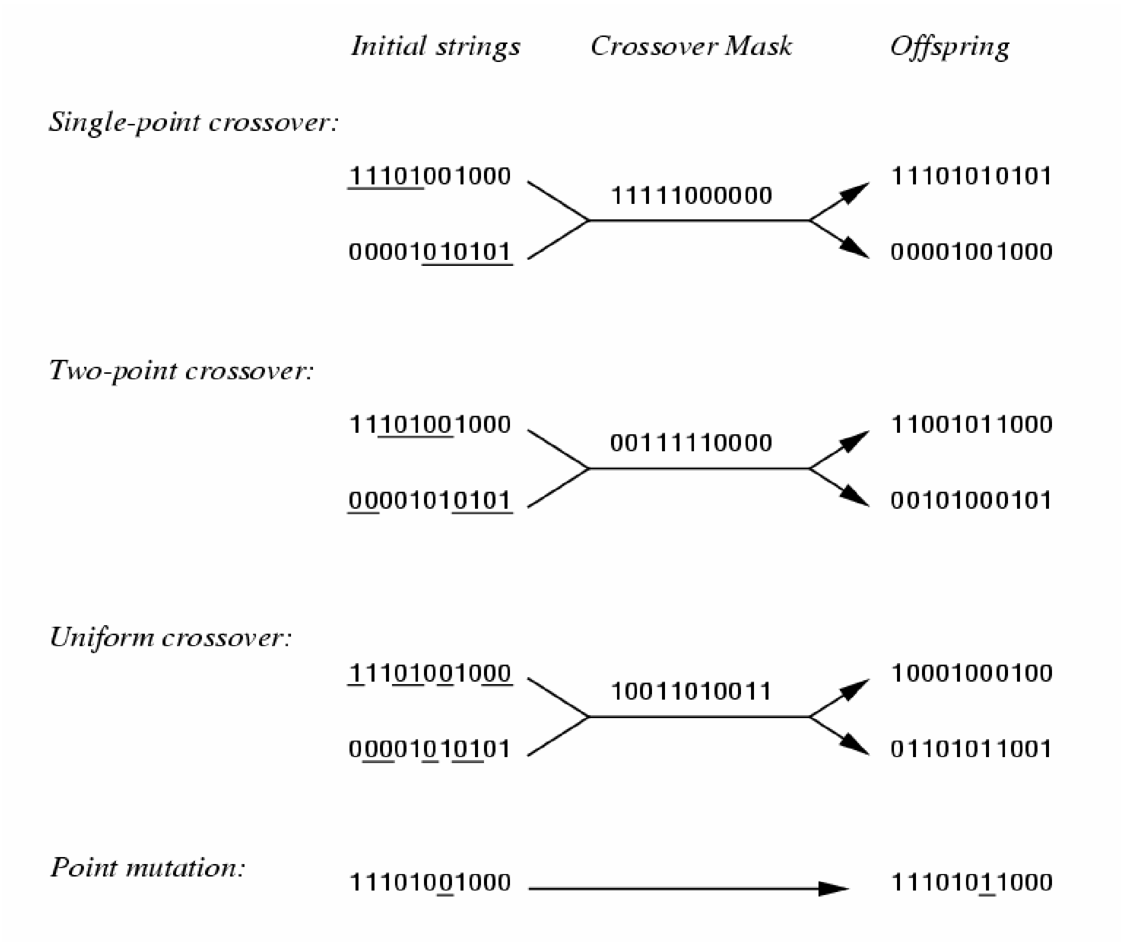


$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{P} Fitness(h_j)}$$



Genetic Algorithms

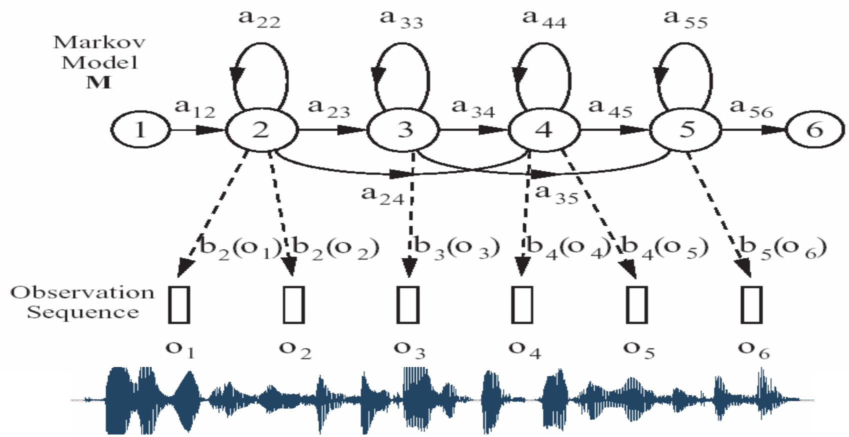
- Example 2: common crossover operators



Genetic Algorithms

- Example 3: HMM adaptation in Speech Recognition

$$\Pr(\boldsymbol{h}_i) = \frac{\exp\left(\dfrac{P(\boldsymbol{O}|\boldsymbol{h}_i)}{T}\right)}{\sum_{j=1}^{P}\exp\left(\dfrac{P(\boldsymbol{O}|\boldsymbol{h}_j)}{T}\right)}$$



sequences of HMM mean vectors

$$\boldsymbol{h}_1 = (k_1, k_2, k_3, \ldots, k_D) \qquad \boldsymbol{s}_1 = (k_1 \cdot i_f + m_1 \cdot (1-i_f), k_2 \cdot i_f + m_2(1-i_f), m_3 \cdot i_f + k_3(1-i_f), \ldots, m_3 \cdot i_f + k_D(1-i_f))$$

$$\boldsymbol{h}_2 = (m_1, m_2, m_3, \ldots, m_D) \qquad \boldsymbol{s}_2 = (\underbrace{m_1 \cdot i_f + k_1 \cdot (1-i_f)}, m_2 \cdot i_f + k_2(1-i_f), k_3 \cdot i_f + m_3(1-i_f), \ldots, k_3 \cdot i_f + m_D(1-i_f))$$

crossover
(reproduction)

$$g_d \Rightarrow \hat{g}_d = g_d + \varepsilon \cdot \sigma_d$$

mutation

Genetic Algorithms

```
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
     loop for i from 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))
```

Genetic Algorithms

- Main issues
  - Encoding schemes
    - Representation of problem states
  - Size of population
    - Too small $\rightarrow$ converging too quickly, and vice versa
  - Fitness function
    - The objective function for optimization/maximization
    - Ranking members in a population

Properties of GAs

- GAs conduct a randomized, parallel, hill-climbing search for states that optimize a predefined fitness function
- GAs are based an analogy to biological evolution
- It is not clear whether the appeal of GAs arises from their performance or from their aesthetically pleasing origins in the theory of evolution

Local Search in Continuous Spaces

- Gradient Search
  - A hill climbing method
  - Search in the space defined by the real numbers
  - Guaranteed to find local maximum
  - Not Guaranteed to find global maximum

maximization

$$\hat{\boldsymbol{x}} = \boldsymbol{x} + \alpha \nabla f\left(\boldsymbol{x}\right) = \boldsymbol{x} + \alpha \frac{df\left(\boldsymbol{x}\right)}{d\boldsymbol{x}}$$

minimization

$$\hat{\boldsymbol{x}} = \boldsymbol{x} - \alpha \nabla f\left(\boldsymbol{x}\right) = \boldsymbol{x} - \alpha \frac{df\left(\boldsymbol{x}\right)}{d\boldsymbol{x}}$$