Informed Search

Searches

Uninformed searches
- Easy to implement
- Very inefficient in many situations
  - Huge search tree, time & space complexities

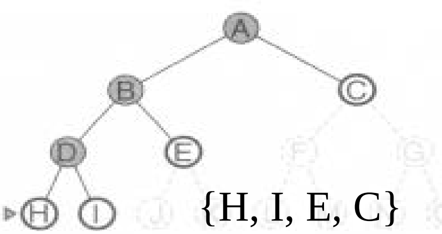
Informed searches (Heuristic)
- Use problem-specific information
- Reduce size of search tree
  - Resolve time & space complexities

Both give similar results
- Completeness & Optimality

Informed Search

Best-First Search

- Arrange nodes in the queue
- In the order of their goodness

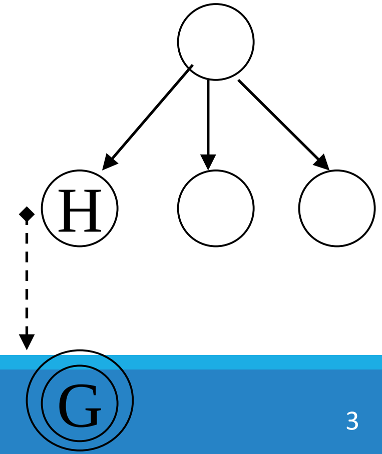


Use evaluation function, f(n)

- Calculate goodness towards the goal

The order of expanding nodes is essential

- Affect size of the search tree
- Smaller size → less space, faster



Best-First Search

Applying evaluation function *f(n)*

- Every node is with a value stating its goodness

Based on values of *f(n)*

- Nodes in the queue are arranged
  - The best one is placed/expanded firstly

Do not guarantee

- The node to expand is really the best

The node only appears to be the best

- *f(n)* is not omniscient

Evaluation Function *f(n)*

Path cost *g(n)* is an example

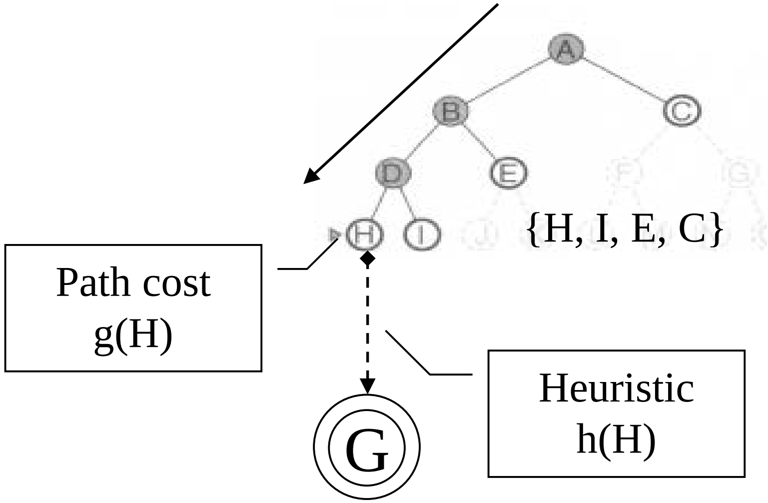
- *f(n)* = *g(n)* → uniform-cost search
- Do not direct search towards the goal

Heuristic function *h(n)* is required

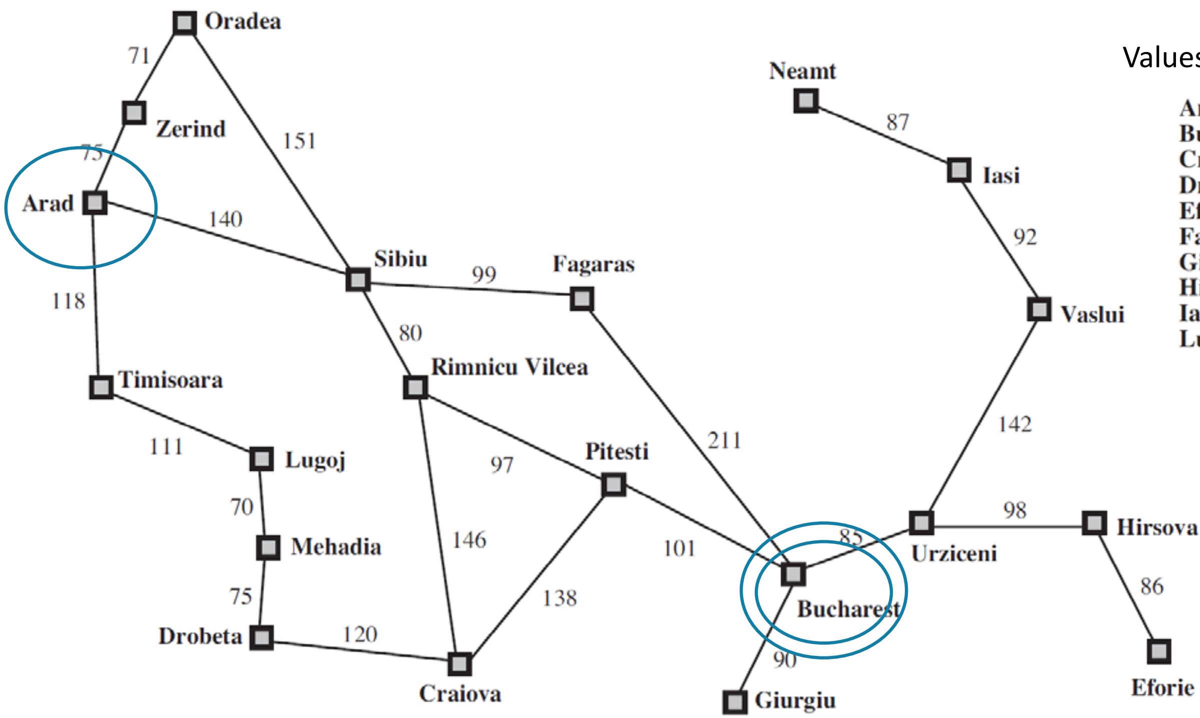
- Estimated cost of the cheapest path
  - From node *n* to a goal state
- Expand node closest to the goal
  - Node with least cost to goal
- If *n* is a goal state, *h(n)* = 0

f(n) = *h(n)* → greedy search

- Only try to expand the node closest to the goal
  - Likely lead to solution quickly



Example of $h(n)$ – h_{SLD}



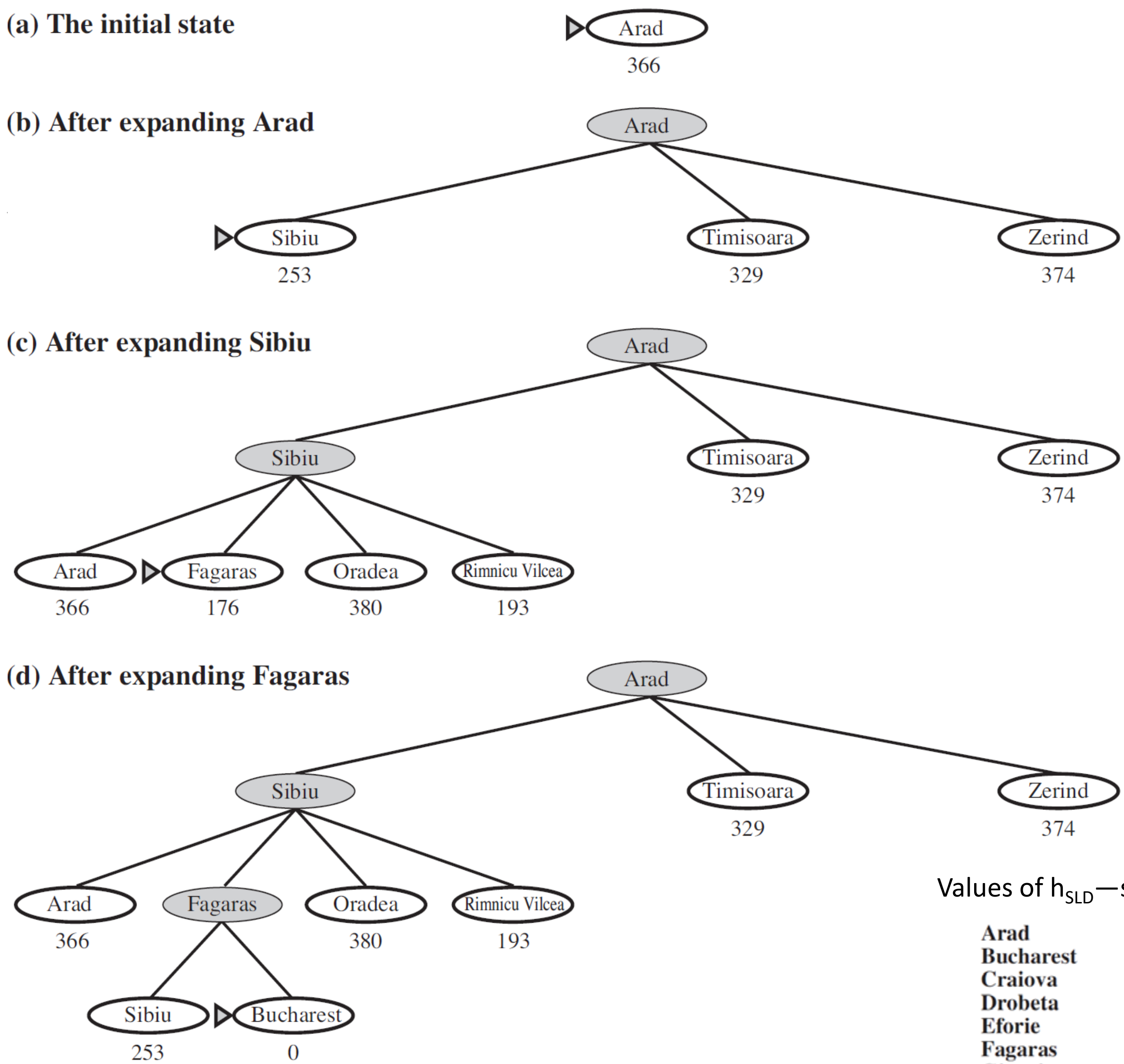
Values of h_{SLD}—straight-line distances to Bucharest.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	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

h_{SLD} - Straight Line Distance

Cannot be computed from problem itself

Only obtainable from experience



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

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	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

Greedy Best-First Search

$f(n) = h(n)$

Good ideally

- Poor practically
- Cannot make sure
  - Heuristic $h(n)$ give correct estimation or not

Dangerous

- Just depending on the estimates $h(n)$

Analysis of Greedy Search

Similar to depth-first search

- Not optimal
  - Find the closest goal
- Incomplete
  - Repeated states may happen
  - Solution may never be found
- Time and space complexities
  - Depend on quality of heuristic function *h*

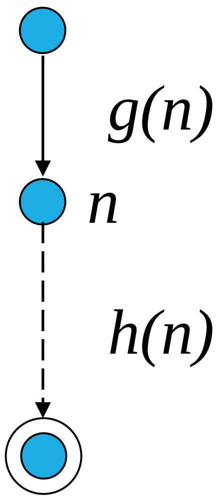
POOR

A* Search

A* Search

Most well-known best-first search

- Evaluate nodes by combining
  - Path cost *g(n)* and heuristic *h(n)*
  - $f(n) = g(n) + h(n)$
  - *g(n)* – cheapest known path
  - *h(n)* – cheapest estimated path



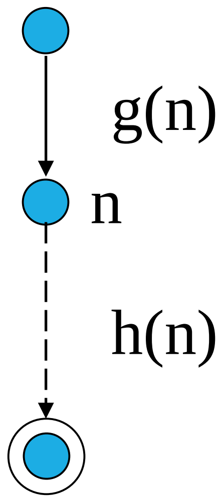
Minimize total path cost

- Combine
  - Uniform-cost search
  - Greedy search

A* Search

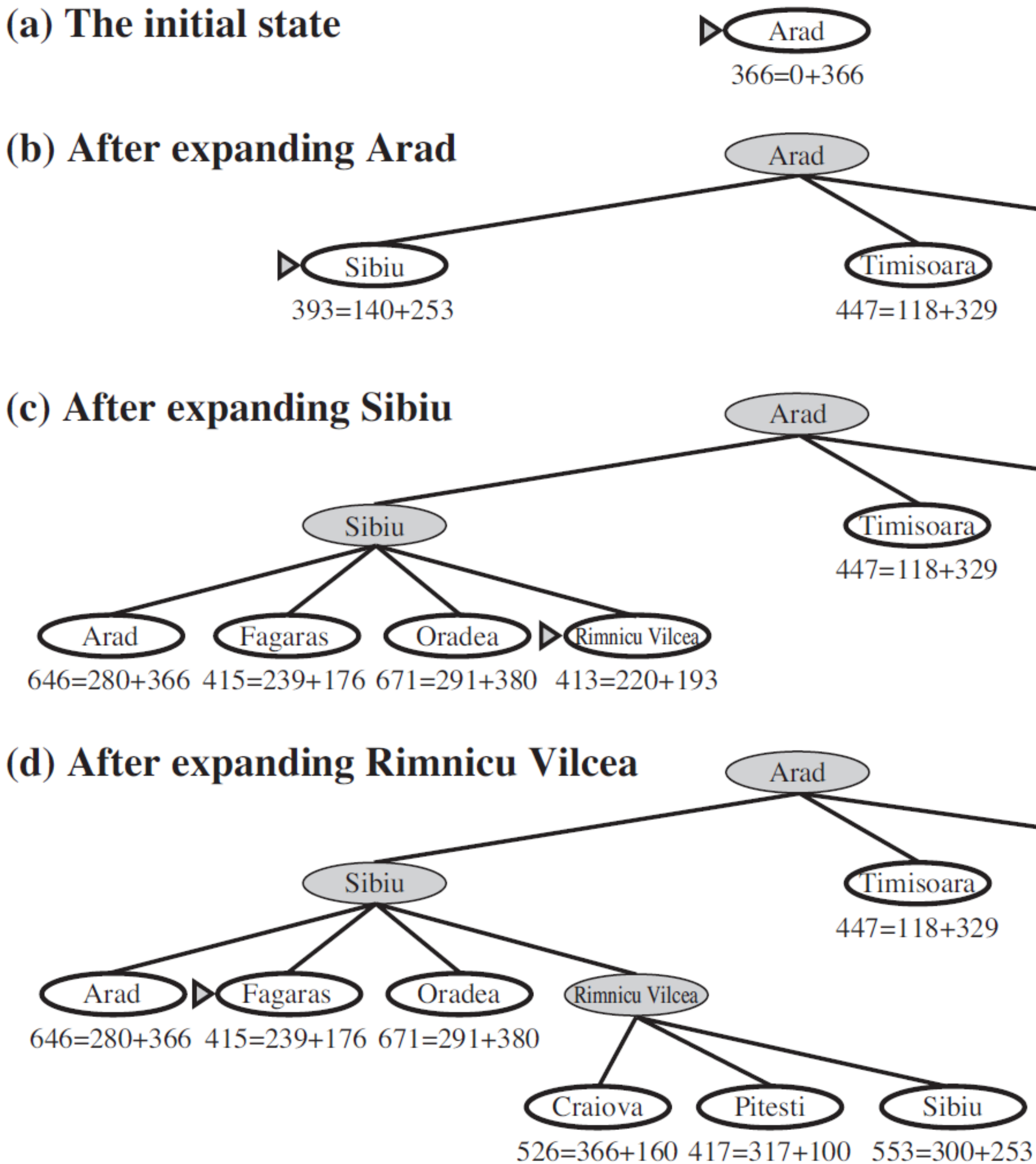
Uniform-cost search

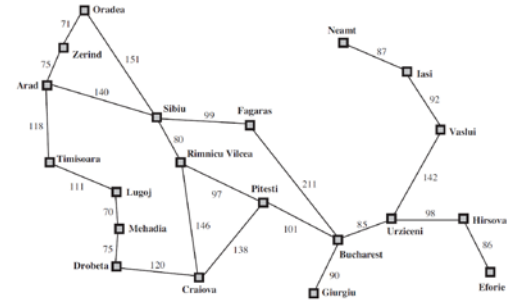
- Optimal and complete
- Minimize cost of the path so far, *g(n)*
- Can be very inefficient



Greedy search + Uniform-cost search

- Evaluation function *f(n) = g(n) + h(n)*
- [Evaluated so far + Estimated future]
- *f(n)* = Estimated cost of the cheapest solution through n

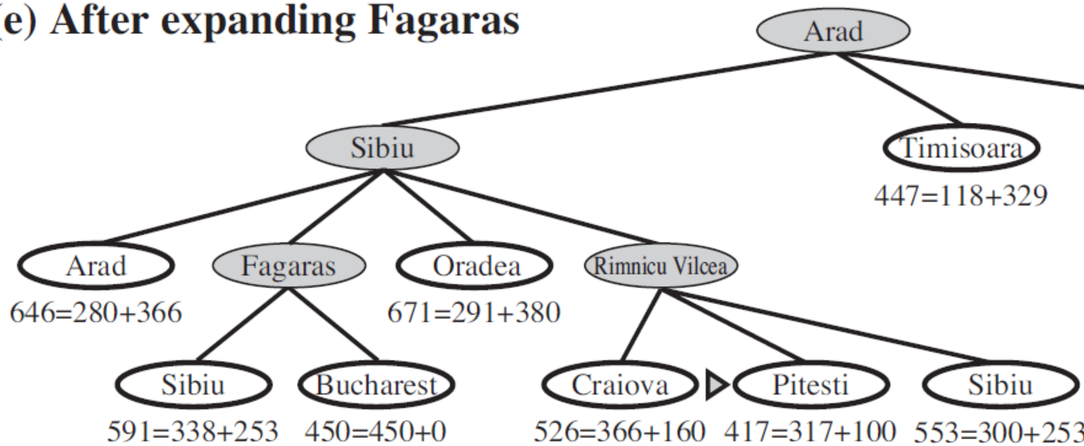


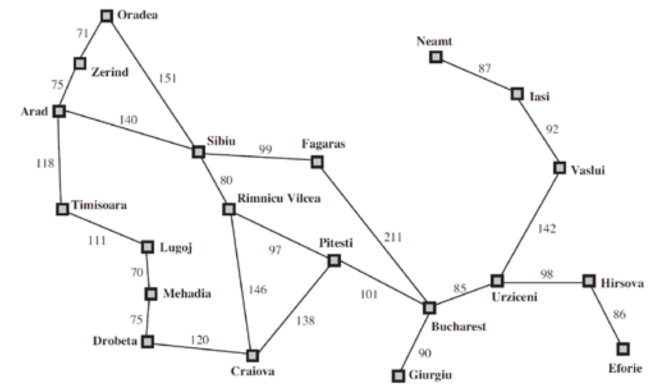


Values of hSLD—straight-line distances to Bucharest.

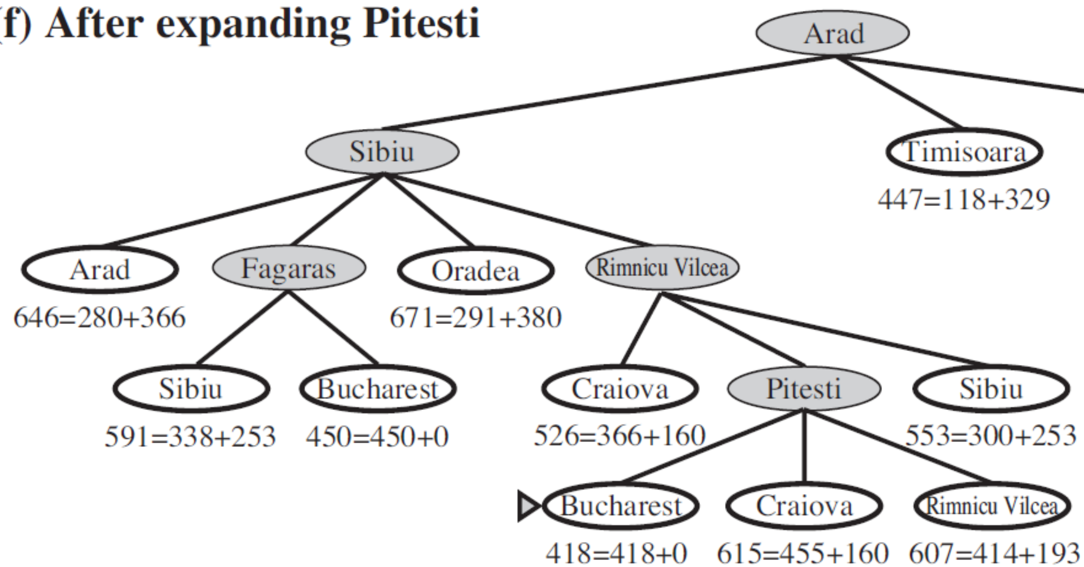
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

(e) After expanding Fagaras





(f) After expanding Pitesti



Values of hSLD—straight-line distances to Bucharest.

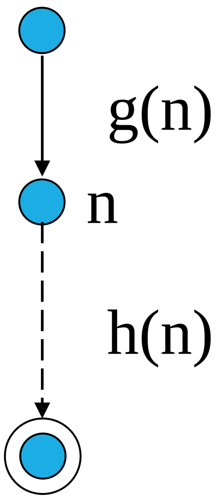
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Analysis of A* Search

Complete and optimal
- Uniform-cost search

Reasonable time & space complexities
- Greedy best-first search

Optimality can be assured
- Only *h(n)* is admissible
  - Never overestimates cost from *n* to the goal
  - Can underestimate
- h_{SLD}, never overestimate
  - Straight Line Distance = Shortest distance between 2 places



Memory Bounded Search

Memory

- Another issue besides time constraint
- More important than time
  - Solution cannot be found
    - Not enough memory
- Solution can still be found
  - Even need a long time

Iterative Deepening A* Search

Iterative deepening (ID) + A*

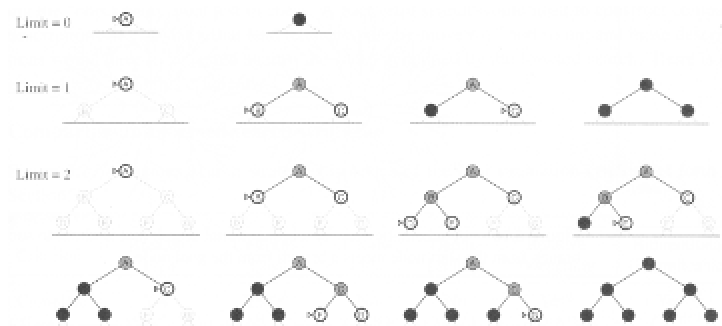
- Similar to ID
- Reduces memory constraints effectively

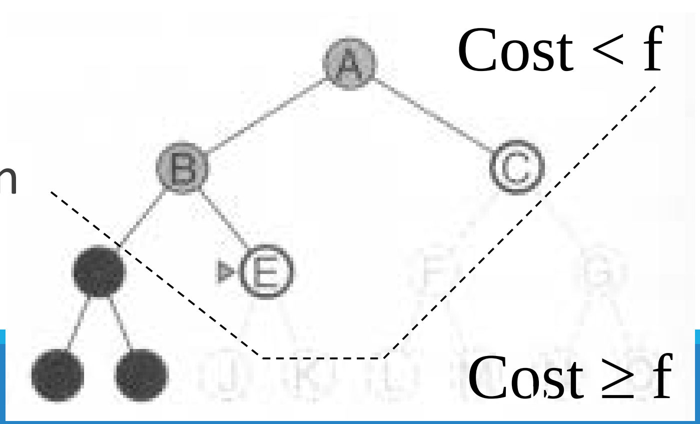
Complete and optimal

- Similar to A*

Use f = g + h for cutoff

- Instead of depth *d* or cost *p*
- Smallest f-cost of any nodes
  - Exceeded cutoff value in previous iteration



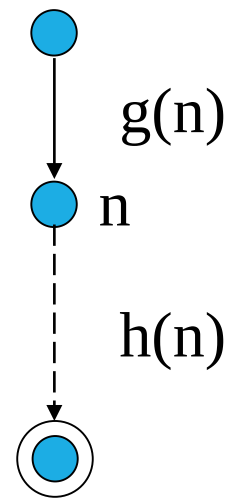


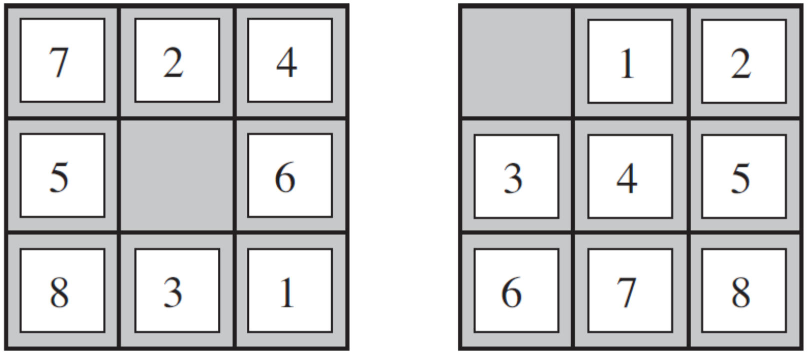
Heuristic Function

Heuristic Functions

Problem of 8-puzzle

- Two heuristic functions
  - Cut down search tree
- h_1 = number of misplaced tiles
- h_1 is admissible, never overestimates
  - At least h_1 steps to reach the goal

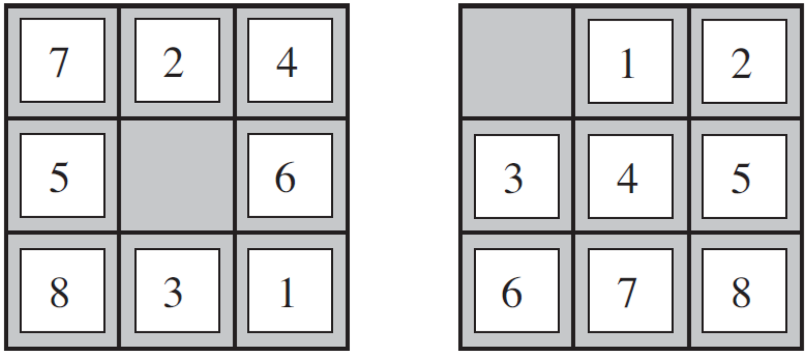




Start State Goal State

Heuristic Functions

- h_2 = sum of distances of tiles from their goal positions
- City block distance or Manhattan distance
  - Count horizontally and vertically
- h_2 is admissible
  - $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$
  - Actual cost = 26
- *Choose h_1 or h_2*
  - Quality of heuristic



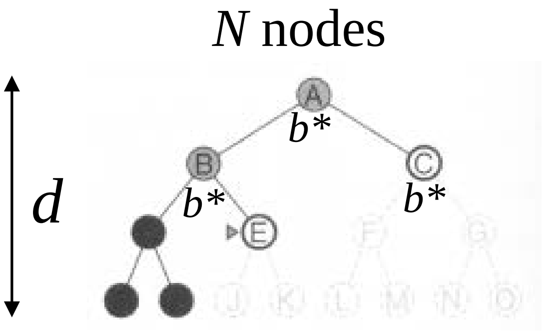
Start State Goal State

Branching Factor

Represent quality of a heuristic

Assume

- N = Total number of nodes expanded by A*
- d = Solution depth
- b^* = Average branching factor of the tree
  - $N = 1 + b^* + (b^*)^2 + \ldots. + (b^*)^d$
- If b^* tends to 1
  - N tends to be smallest
  - Tree becomes minimized



Branching Factor

h_1 = number of misplaced tiles

h_2 = sum of distances of tiles from their goal positions

	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

Tree size N

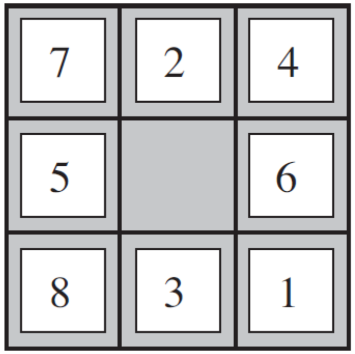
Branching Factor

h_2 is better than h_1

- h_1 = number of misplaced tiles
- h_2 = sum of distances of tiles from their goal positions

h_2 dominates h_1

- Any node *n*
- $h_2(n) \geq h_1(n)$
- Do not overestimate



Conclusion

- Better use heuristic function with higher values
- As long as it does not overestimate

Admissible Heuristic Functions

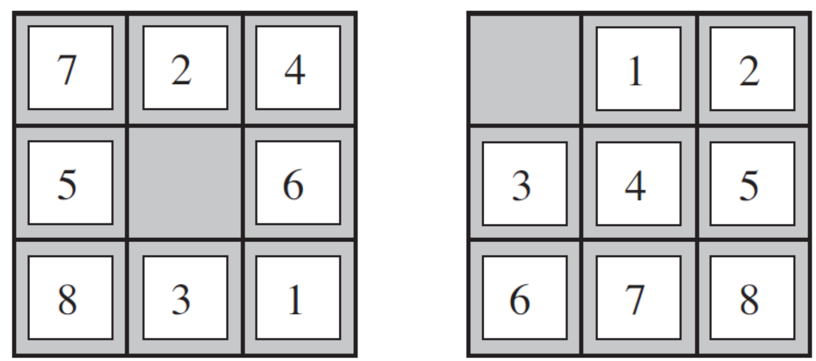
Relaxed problem

- Problem with less restriction on operators

Cost of exact solution to relaxed problem

- Often good heuristic for original problem

h_2 = sum of distances of tiles from their goal positions



Start State   Goal State

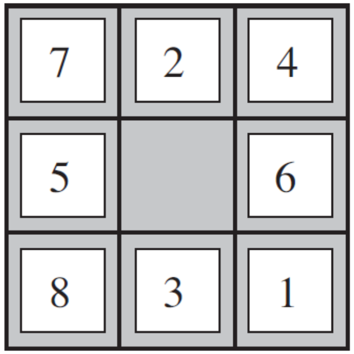
Admissible Heuristic Functions

Original problem

- Tile can move from square A to square B
  - A is horizontally or vertically adjacent to B
  - B is blank

Relaxed problem

1. Tile can move from square A to square B
   - A is horizontally or vertically adjacent to B
2. Tile can move from square A to square B
   - B is blank
3. Tile can move from square A to square B



Admissible Heuristic Functions

Given 3 heuristics

- Choose the easiest one
- Do not waste too much time on *h(n)*

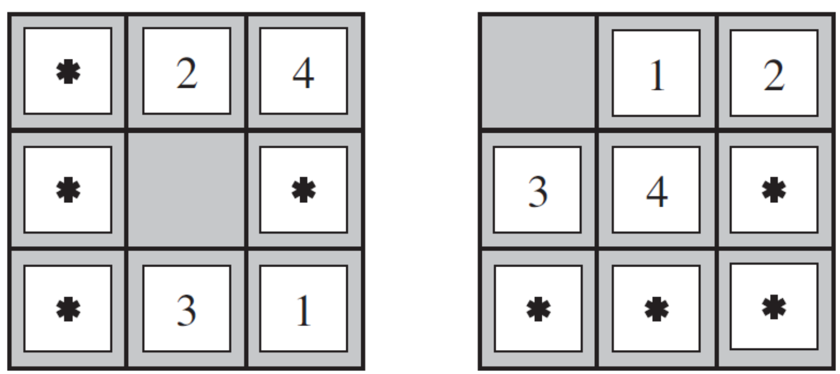
Do not know best heuristic

- Among $h_1, \ldots, h_m$ heuristics
- Set $h(n) = max(h_1(n), \ldots, h_m(n))$
- Let computer run it
  - Determine at run time

Admissible Heuristic Functions

Can also be derived

- From solution cost of a subproblem of given problem
- Get only 4 tiles into their positions
- Cost of optimal solution of this subproblem
  - Used as a lower bound, as a heuristic value



Start State   Goal State

Local Search

Local Search

Previously, find solution by searching paths

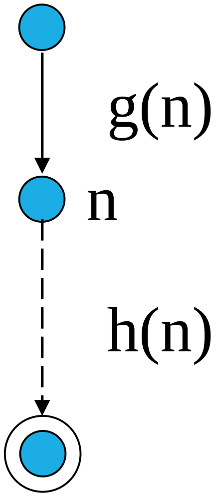
Initial state → goal state

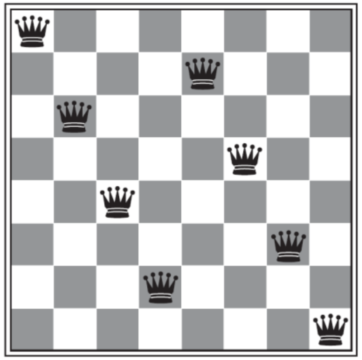
Many problems

- Solution is not a path to goal
  - 8-queens problem
- Solution
  - Final configuration
  - Not the order they are added or modified

Other kinds of method

- Local search



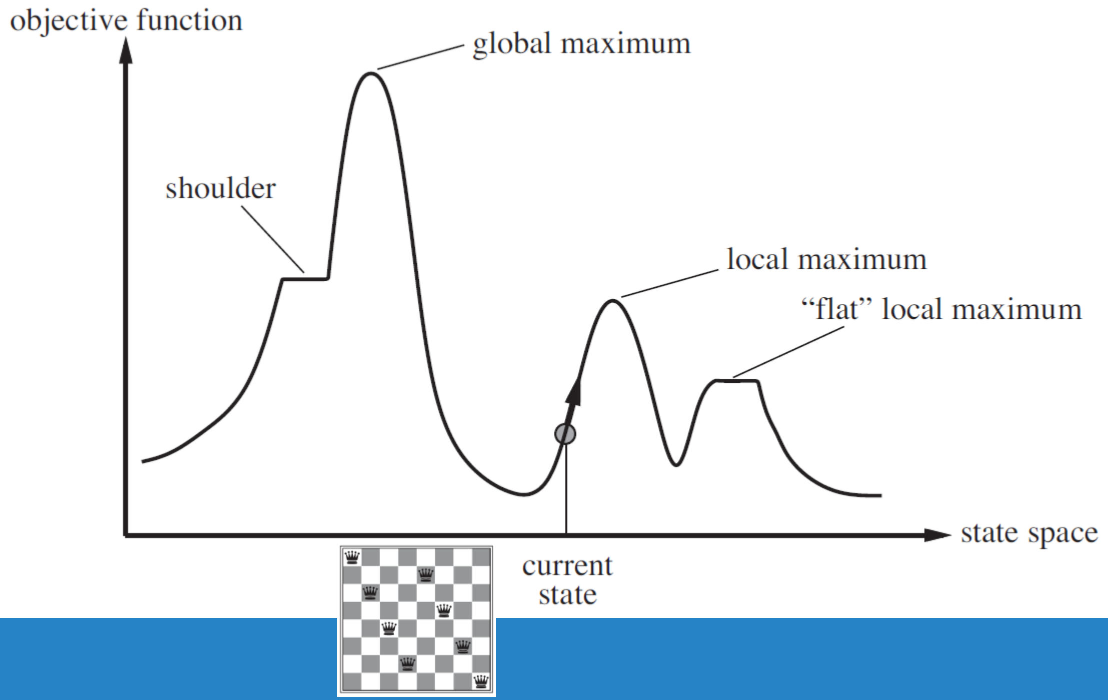


Local Search

A group of searches work in search space

Landscape has two axes

- Location (defined by states, a vector)
- Elevation (defined by objective function)

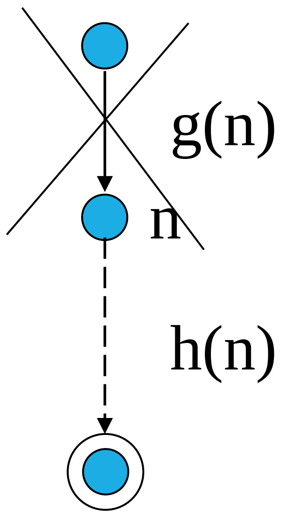


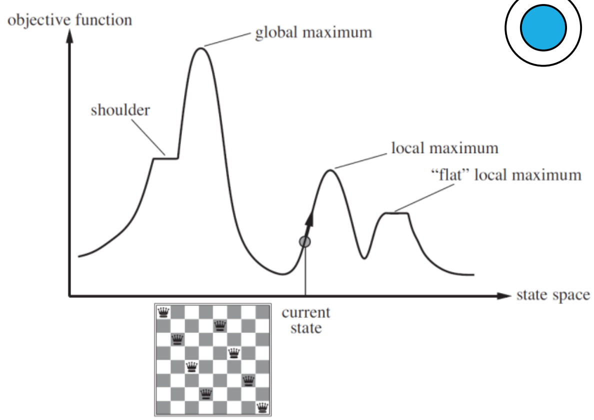
Local Search

Operate on a single current state
- Instead of a tree of multiple paths

Only move to
- Neighbors of current state

Paths in the search
- Not retained
  - Only retain current state
- Method is not systematic

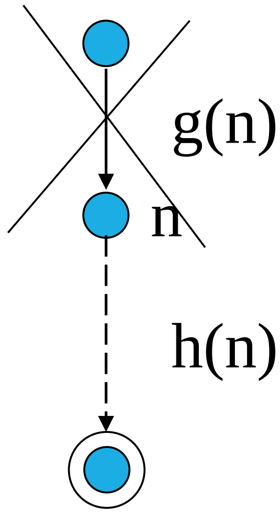




Local Search

Two advantages

- Little memory – constant amount
  - Current state and some information
- Find reasonable solutions
  - Large or infinite (continuous) state spaces
  - Systematic algorithms are unsuitable



Suitable

- Optimization problems
- Find the best state with
  - Objective function

Analysis of Local Search

Completeness
- Complete local search algorithm
  - Find a goal if exists
- Most local search methods
  - Incomplete

Optimality
- Optimal algorithm
  - Find a global maximum or minimum (optimum)
- No local search methods are optimal

Theoretically, POOR

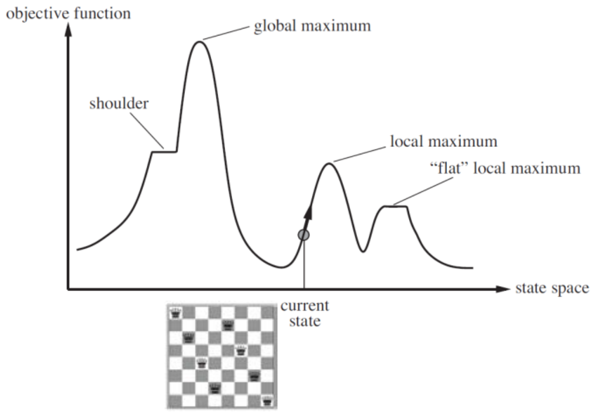
Practically, work well
- At least reasonable

Hill-climbing Search

One type of local search

- Loop
  - Current state → multiple successors (neighbors)
  - Evaluate each neighbors → evaluation value
- Continually move in direction of increasing value
  - Current state ← best neighbor
  - Uphill (Climbing Hill)

No search tree is maintained



Node

- State
- Its evaluation
  - Objective value, real number

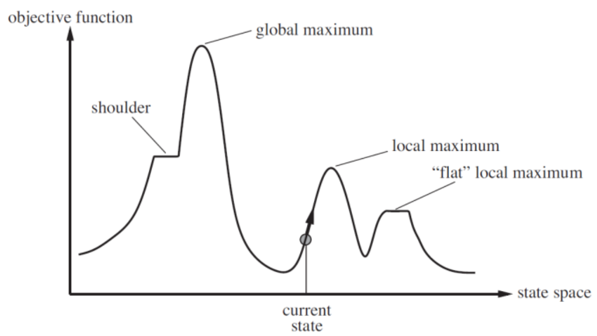
Hill-climbing Search

Multiple best neighbors

- Select among them randomly

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

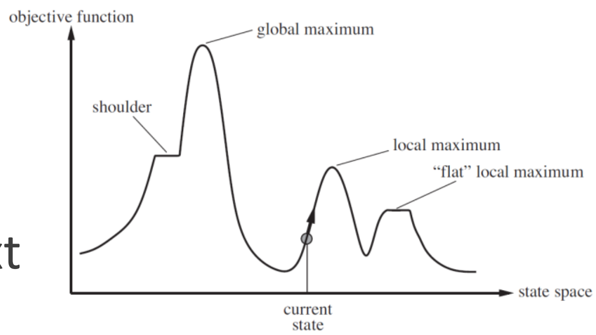
$current \leftarrow$ MAKE-NODE(*problem*.INITIAL-STATE)
loop do
    $neighbor \leftarrow$ a highest-valued successor of *current*
    if neighbor.VALUE $\leq$ current.VALUE **then return** *current*.STATE
    $current \leftarrow neighbor$



Drawbacks of Hill-climbing Search

Hill-climbing is also called

- Greedy local search
- Grab a good neighbor state
  - Without thinking about where to go next



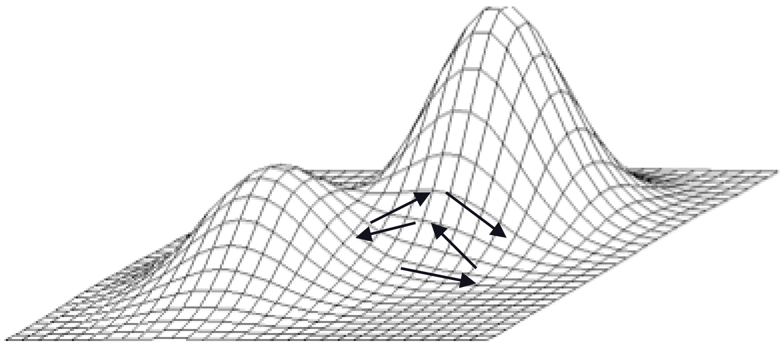
Local maxima

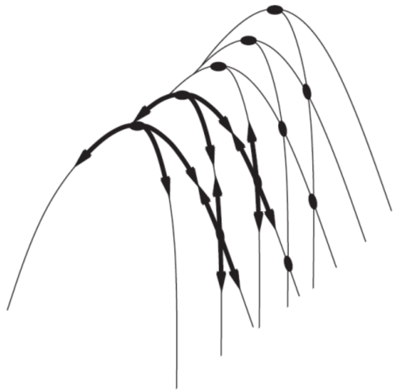
- Peak lower than the highest peak in state space
- Algorithm stops
  - Even though solution is far from satisfactory

Drawbacks of Hill-climbing Search

Ridges

- Grid of states is overlapped on a ridge
- Rise from left to right, right to left
- Unless move directly along top of ridge
- Search may oscillate from side to side
- Make little progress

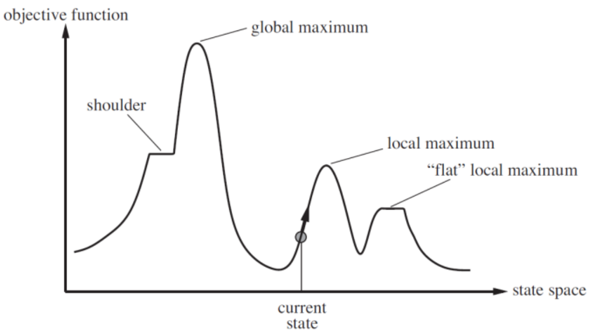




Drawbacks of Hill-climbing Search

Plateaux

- Area of the state space landscape
  - Evaluation function is flat
  - Shoulder
  - Impossible to make progress
- May be unable to find way off the plateau



function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

current ← MAKE-NODE(*problem*.INITIAL-STATE)
loop do
neighbor ← a highest-valued successor of *current*
if neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
current ← *neighbor*

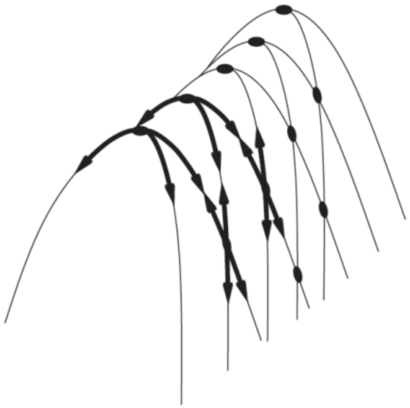
Solution

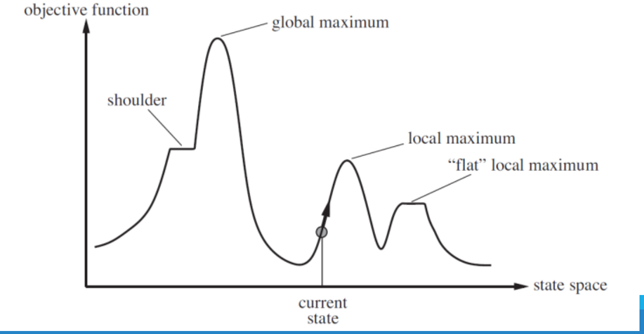
Previous problems happen

- Poor initial states

Random-restart hill-climbing (RRHC)

- Conduct a series of hill-climbing searches
  - Random generated initial states
  - Save best result found
- Fixed number of RRHC
  - Continue until best saved result has not been improved
    - For a certain number of iterations
- Cannot ensure optimality
  - Reasonably good solution





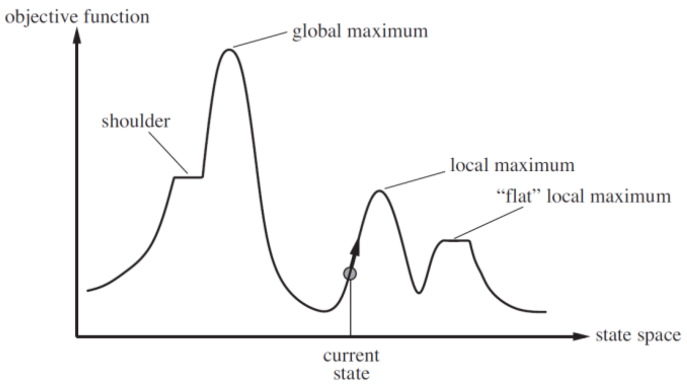
Simulated Annealing

Annealing

- Cooling down liquid until it freezes

Search

- Allow downhill steps
- Leave the local maximum
- Instead of start again randomly



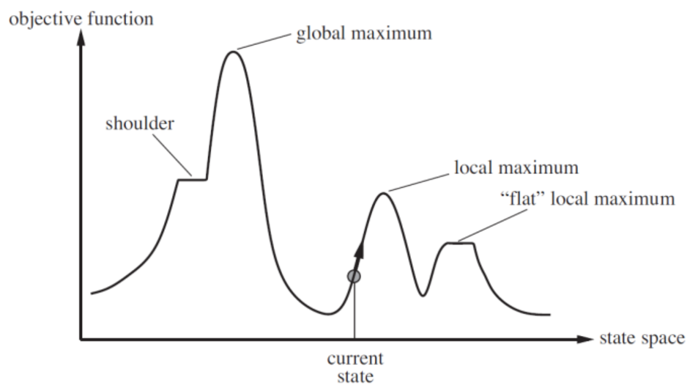
Simulated Annealing

Every iteration

- Do not choose the best move
- Choose a random one

If the move results better

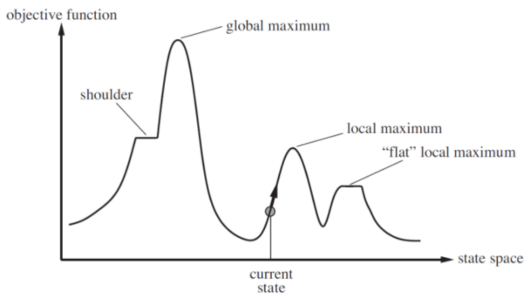
- Always execute
- Otherwise, take the move with a probability less than 1

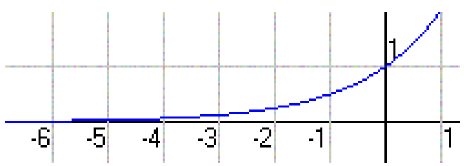


Simulated Annealing

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

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



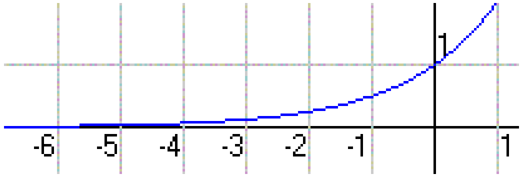


T, temperature also affects the probability
- Since $\Delta E \leq 0$, $T > 0$, $\Delta E/T \leq 0$
- Probability $0 < e^{\Delta E/T} \leq 1$

Probability decreases exponentially
- "Badness" of the move = ΔE

Simulated Annealing

$$0 < e^{\Delta E/T} \leq 1$$



Higher T

- More likely bad move is allowed
- Usually happen in early schedule
- When T is large and |ΔE| is small (≤ 0)
  - $\Delta E/T$ is a negative small value → $e^{\Delta E/T}$ is close to 1

T becomes smaller and smaller

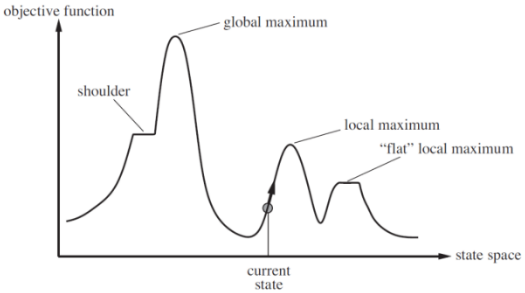
- Until T = 0
  - Normal hill-climbing

Local Beam Search

No good in keeping only one current state

Local beam search

- k current states simultaneously
- All k states are generated randomly initially
- Generate successors
  - All successors of k states are generated
  - m*k successors together
- Halt if any one is a goal
- Select k best successors
  - From complete successor list (m*k) and repeat



Local Beam Search

Different from random-restart hill-climbing

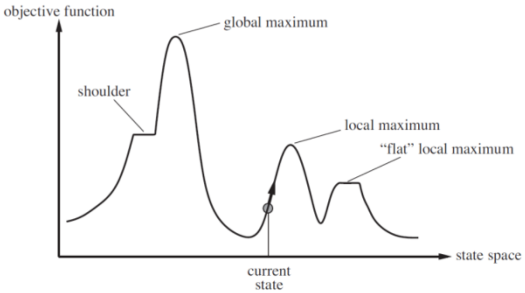
- Do not make k independent searches

Work together

- Collaboration
- Choose the best successors
  - Among those generated together by k states

Stochastic beam search

- Choose k successors at random
- Instead of k best successors



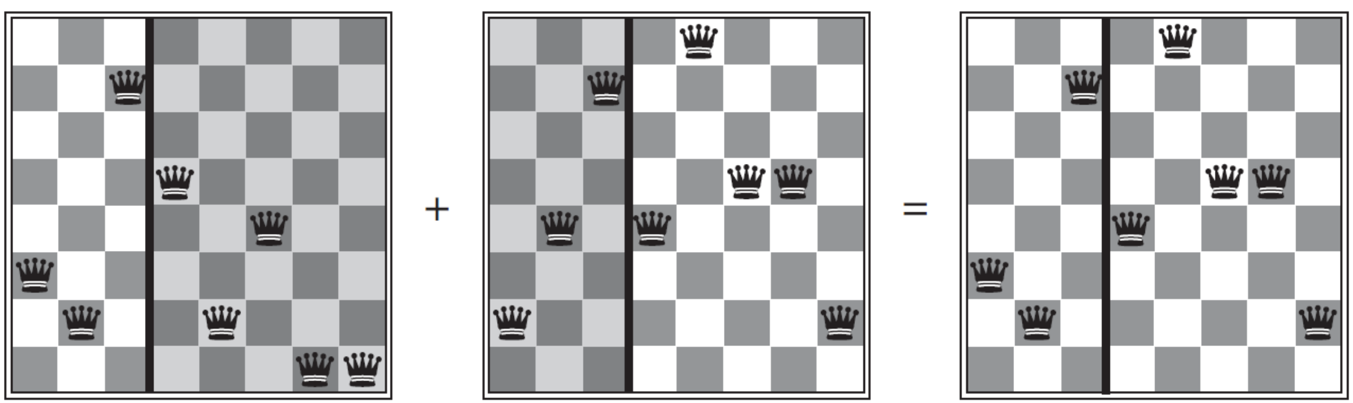
Genetic Algorithms

Variant of beam search

Successor states are generated by

- Combining two parent states
- Instead of modifying a single state

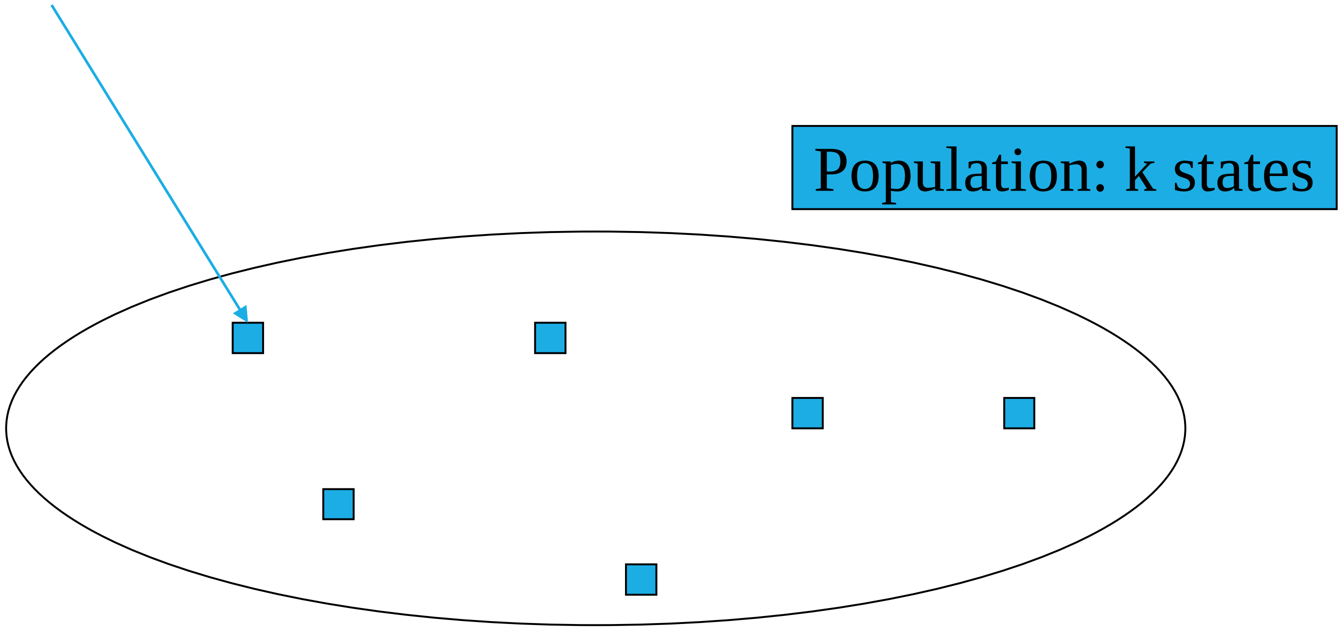
Successor state → offspring



Genetic Algorithms

GA firstly make a population

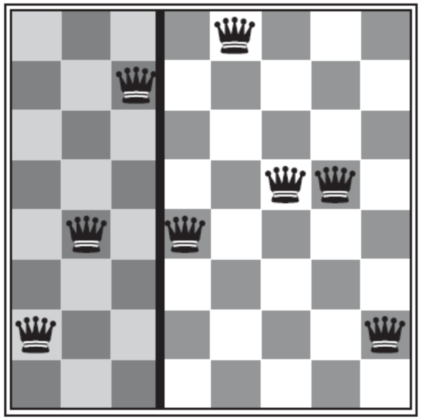
- A set of k randomly generated states
  - Individuals



Genetic Algorithms

Each state or individual

- Represent as a string over a finite alphabet
  - Binary or 1 to 8, etc.



State Representation

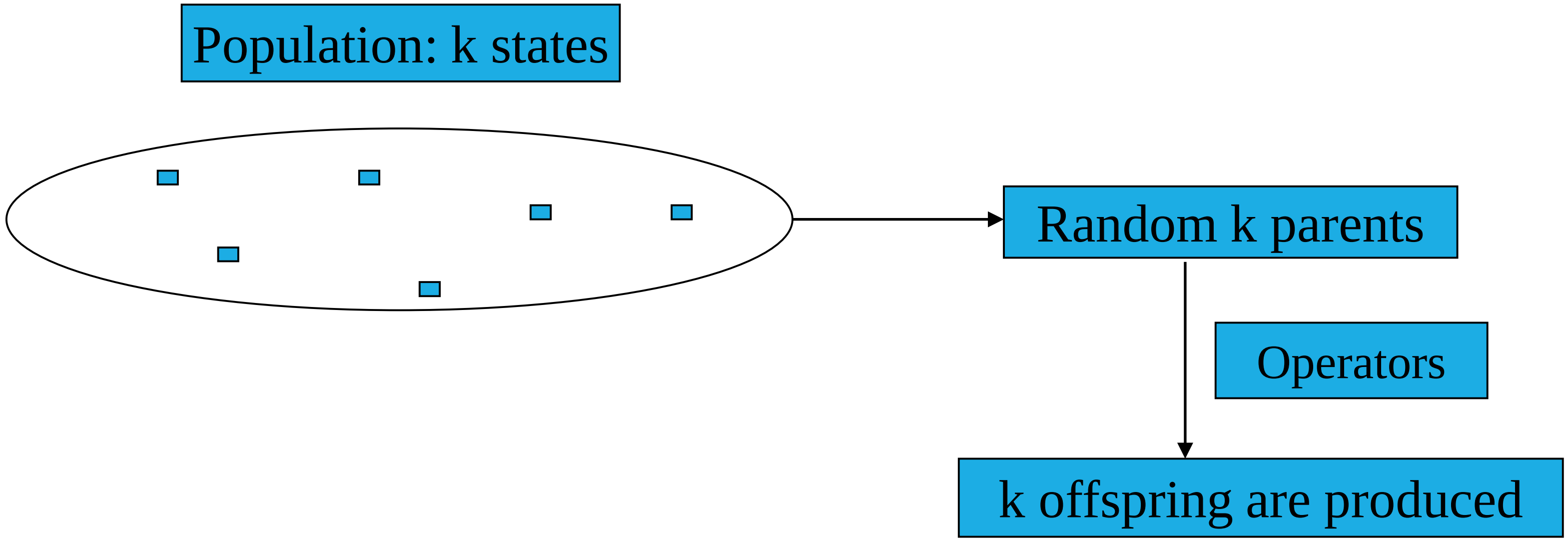
24748552

Genetic Algorithms

Choose k parents randomly

- Parents may occur more than once

Produce next generation of k individuals

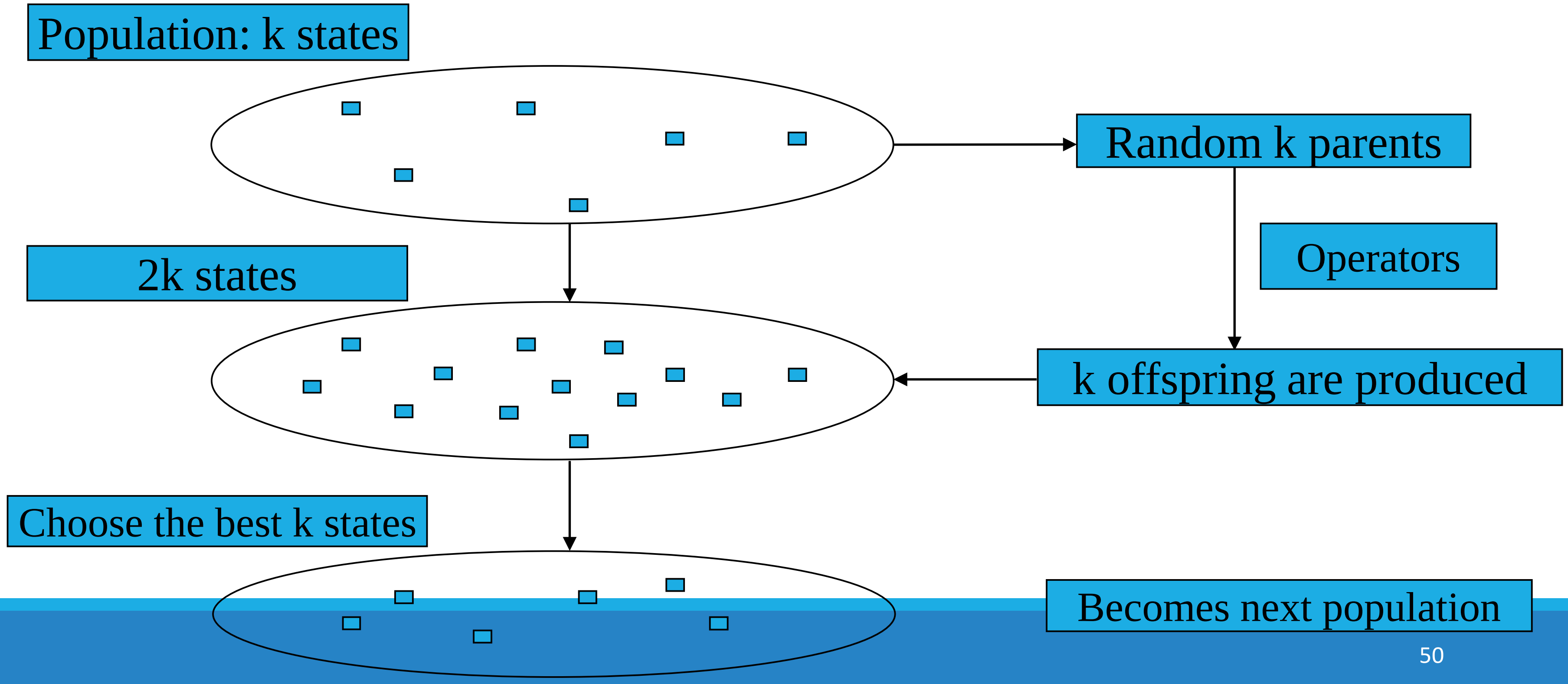


Genetic Algorithms

Population (k) + Next gen (k) = 2k

Choose k best individuals as the next population

- Based on some probabilities over the fitness values



Genetic Algorithms

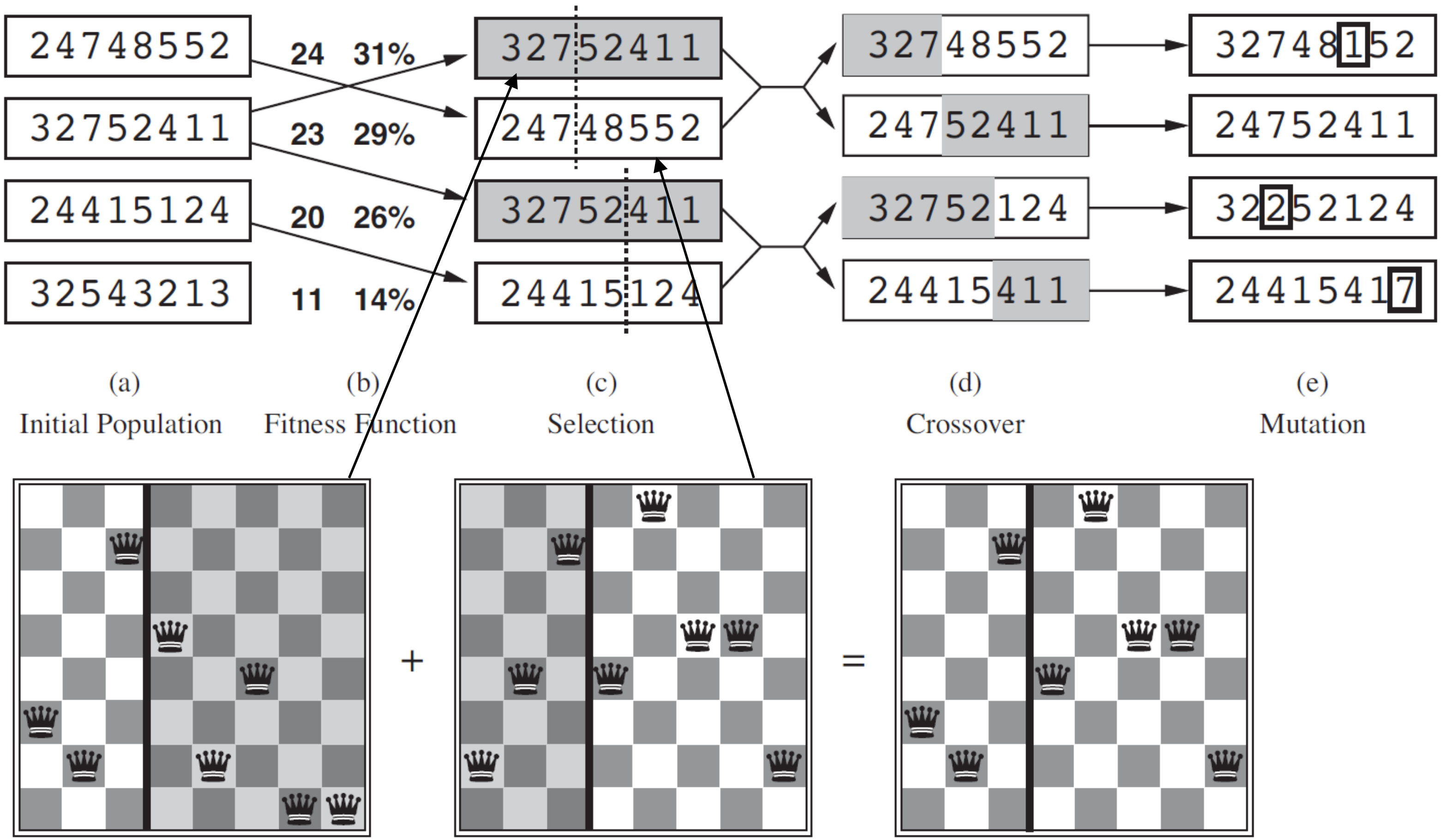
Operations for reproduction

- Crossover
  - Combine two parent states randomly
  - Crossover point is randomly chosen from positions in string
- Mutation
  - Modify the state randomly
  - A small independent probability

Efficiency and effectiveness

- State representation
- Algorithms for operations

Genetic Algorithms



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
    $new\_population \leftarrow$ empty set
    for i = 1 **to** SIZE($population$) **do**
      $x \leftarrow$ RANDOM-SELECTION($population$, FITNESS-FN)
      $y \leftarrow$ RANDOM-SELECTION($population$, FITNESS-FN)
      $child \leftarrow$ REPRODUCE(x, y)
      if (small random probability) **then** $child \leftarrow$ MUTATE($child$)
      add $child$ to $new\_population$
    $population \leftarrow new\_population$
  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

  $n \leftarrow$ LENGTH(x); $c \leftarrow$ random number from 1 to n
  return APPEND(SUBSTRING($x, 1, c$), SUBSTRING($y, c+1, n$))