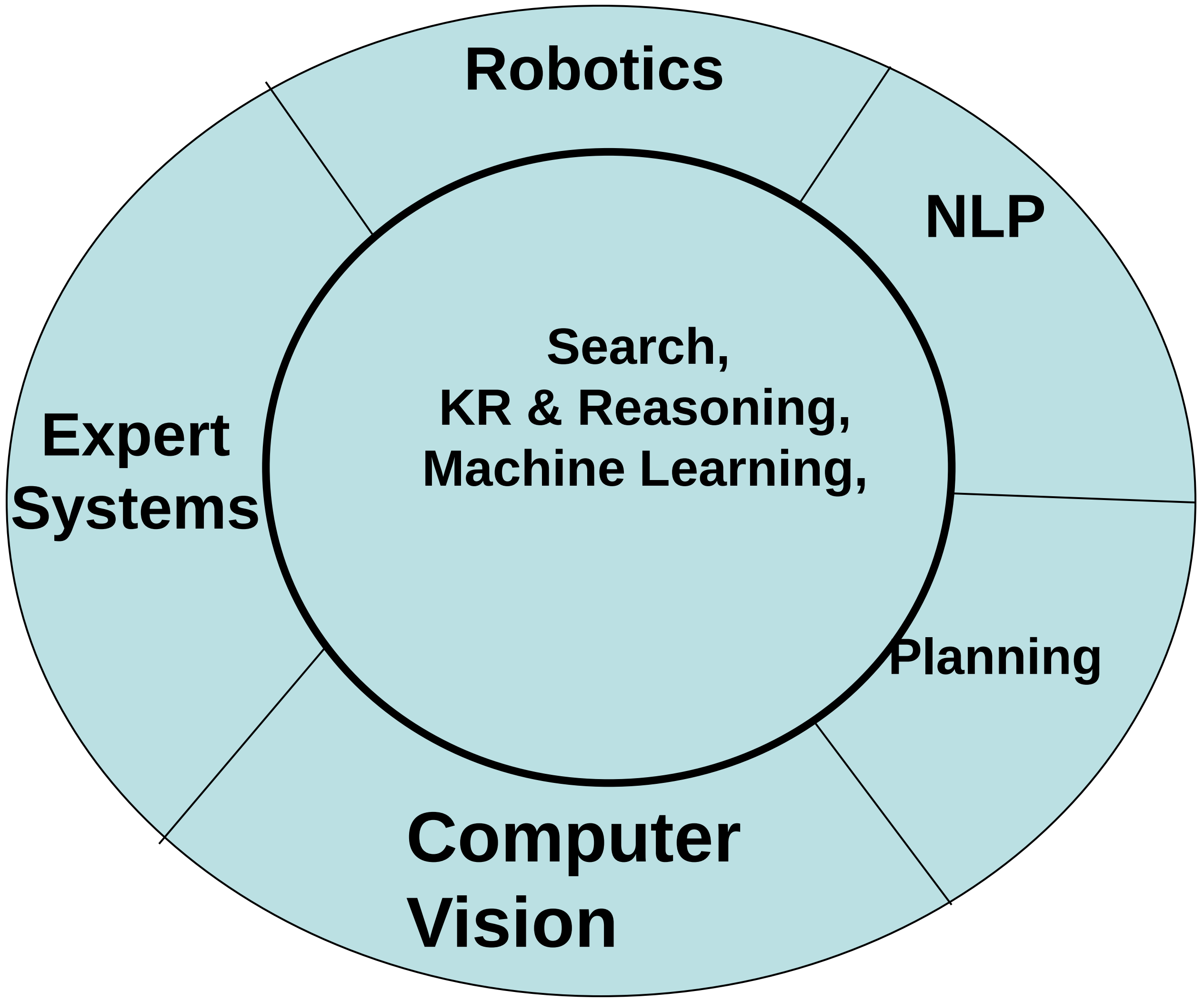
CS 561/571: Artificial Intelligence

Search

Outline

- Search Problems
- Search trees and state space graphs
- Uninformed search
  - Depth-first, Breadth-first, Uniform cost
  - Search graphs
- Informed search
  - Greedy search, A* search
  - Heuristics, admissibility
- Local search and optimization
  - Hill-climbing
  - Simulated annealing
  - Genetic algorithms

Disciplines which form the core of AI- inner circle
Fields which draw from these disciplines- outer circle.



Uninformed vs. informed search

- Uninformed search (or, blind search)
  - Uses only the information available in the problem definition
  - Searches through the space of possible solutions
  - Uses no knowledge about which path is likely to be best
  - E.g. BFS, DFS, uniform cost etc.
- Informed search (or, heuristic search)
  - Knows whether a non-goal state is more promising
  - E.g. Greedy search, A* Search etc.

What are heuristics?

Problem-specific knowledge that reduces expected search effort

Heuristic

Webster's Revised Unabridged Dictionary (1913)

Heuristic \Heu*ris"tic\, a. [Greek. to discover.] Serving to discover or find out.

The Free On-line Dictionary of Computing

heuristic 1. <programming> A rule of thumb, simplification or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. Unlike algorithms, heuristics do not guarantee feasible solutions and are often used with no theoretical guarantee. 2. <algorithm> approximation algorithm.

From WordNet (r) 1.6

heuristic adj 1: (computer science) relating to or using a heuristic rule 2: of or relating to a general formulation that serves to guide investigation [ant: algorithmic] n : a commonsense rule (or set of rules) intended to increase the probability of solving some problem [syn: heuristic rule, heuristic program]

Heuristics (some examples)

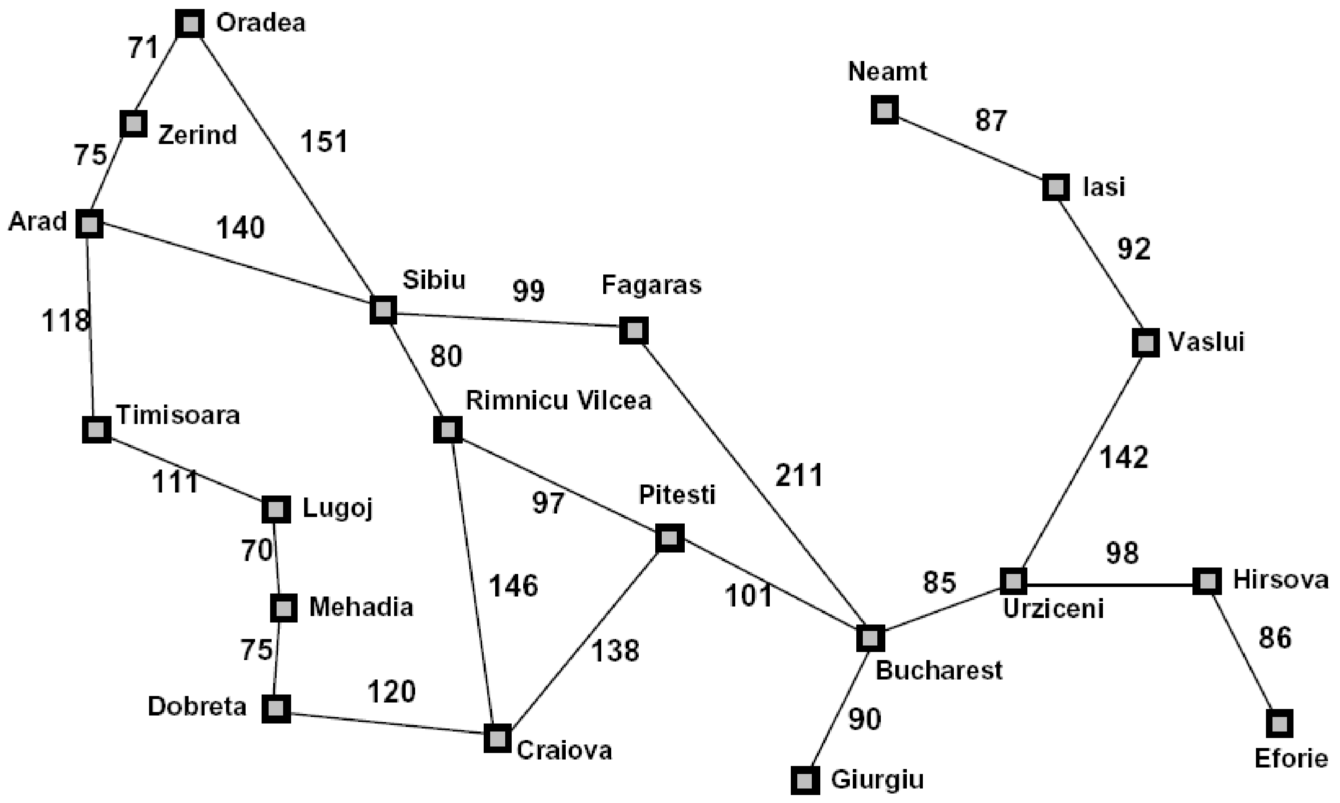
- Travel planning
  - Euclidean distance
- 8-puzzle
  - Manhattan distance (how far the goal is?)
  - No. of misplaced tiles
- Travelling salesman problem
  - Minimum spanning tree

Best-first search

- An instance of the general tree search
- Idea: use an evaluation function *f(n)* for each node
  - Estimate of "desirability“
  - Expand most desirable unexpanded node, i.e. node with lowest f(n)
- Implementation:
  - Priority queue: Order the nodes in fringe in ascending order of *f-values*

- Key component: heuristic function (*h(n)*)
  - h(n)= estimated cost of the cheapest path from node n to the goal
  - Most common form in which additional knowledge of the problem is imparted
- Special cases:
  - greedy best-first search
  - A^* search

Heuristics



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

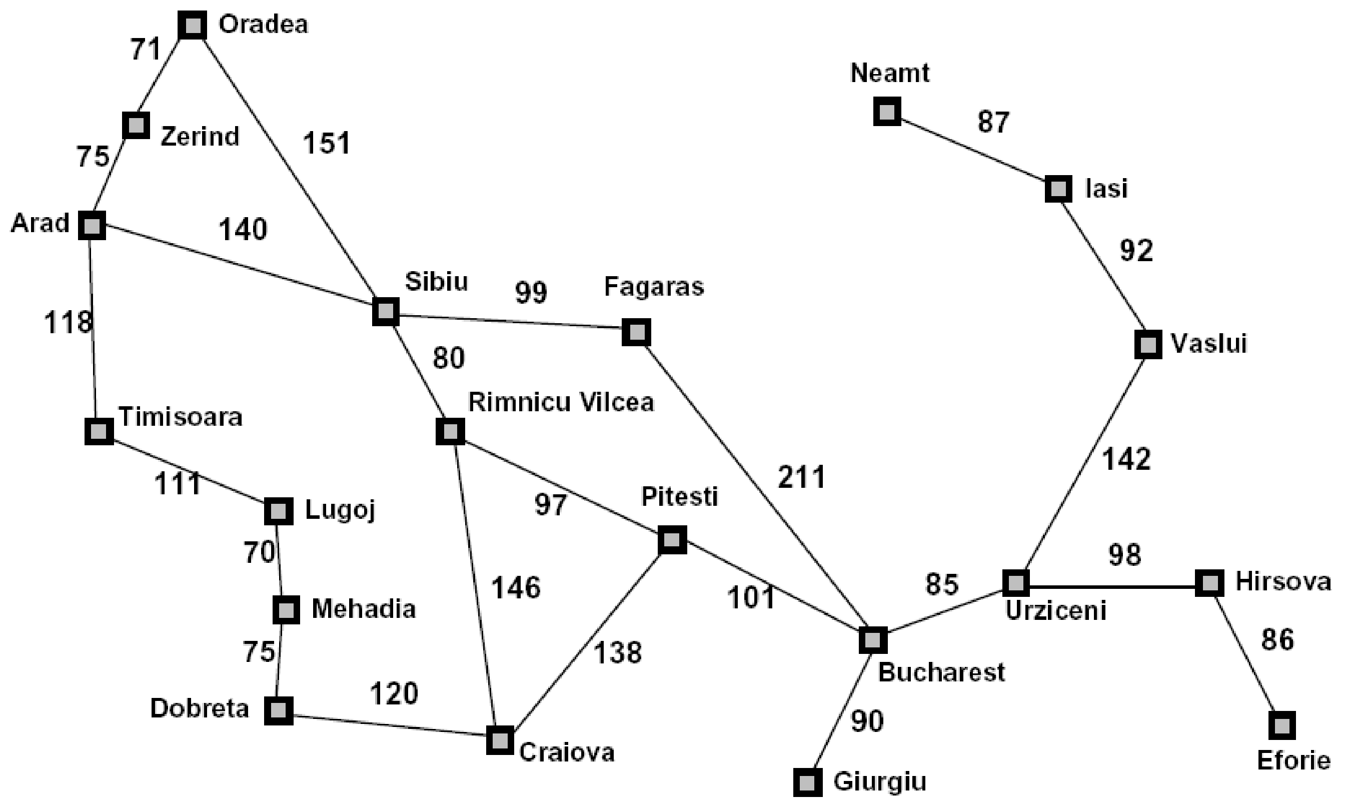
- Greedy best-first search expands the node that appears to be closest to goal
- Finds the solution quickly

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

= estimate of cost from n to *goal*

e.g., $h_{SLD}(n)$ = straight-line distance in the route-finding problem

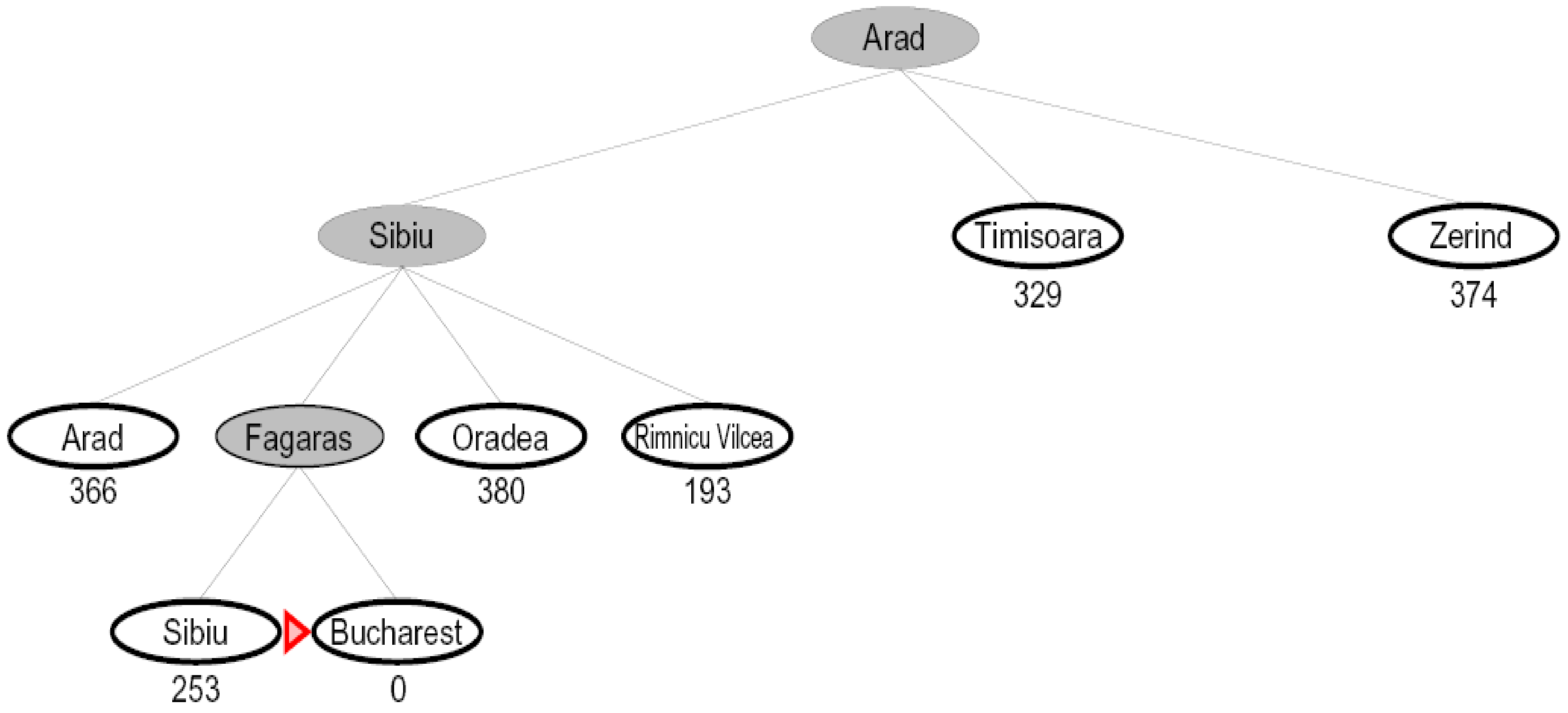
Heuristics



Straight−line distance to Bucharest

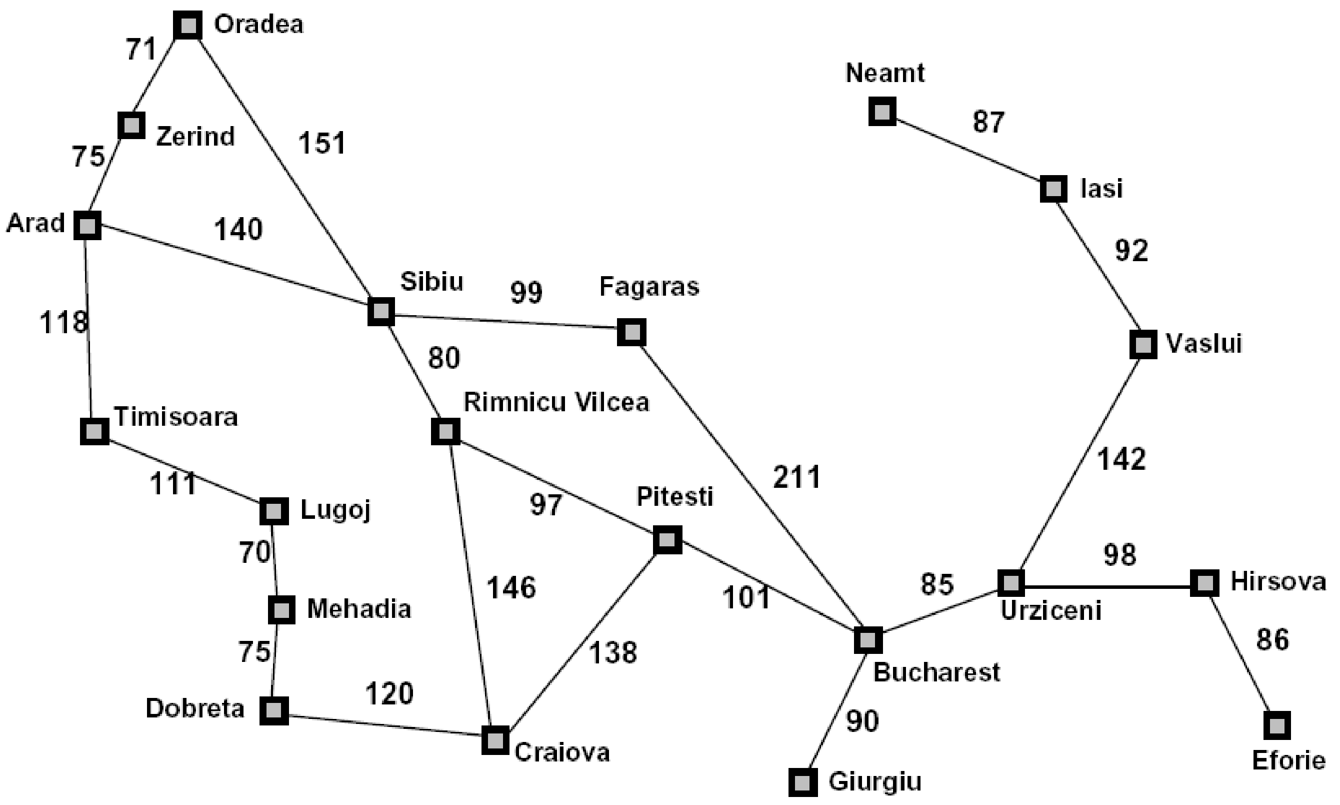
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy Best First Search



- What can go wrong?

Heuristics



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

- A common case:
  - Can guide to non-optimal solution
  - Path via *Sibiu* → *Fagars* →*Bucharest* is 32 km longer than through *Rimnicu Vilcea*→*Pitesi*

    (99+211)-(80+97+101)=32
- Minimizing h(n) is susceptible to false starts (*unnecessary nodes are expanded*)

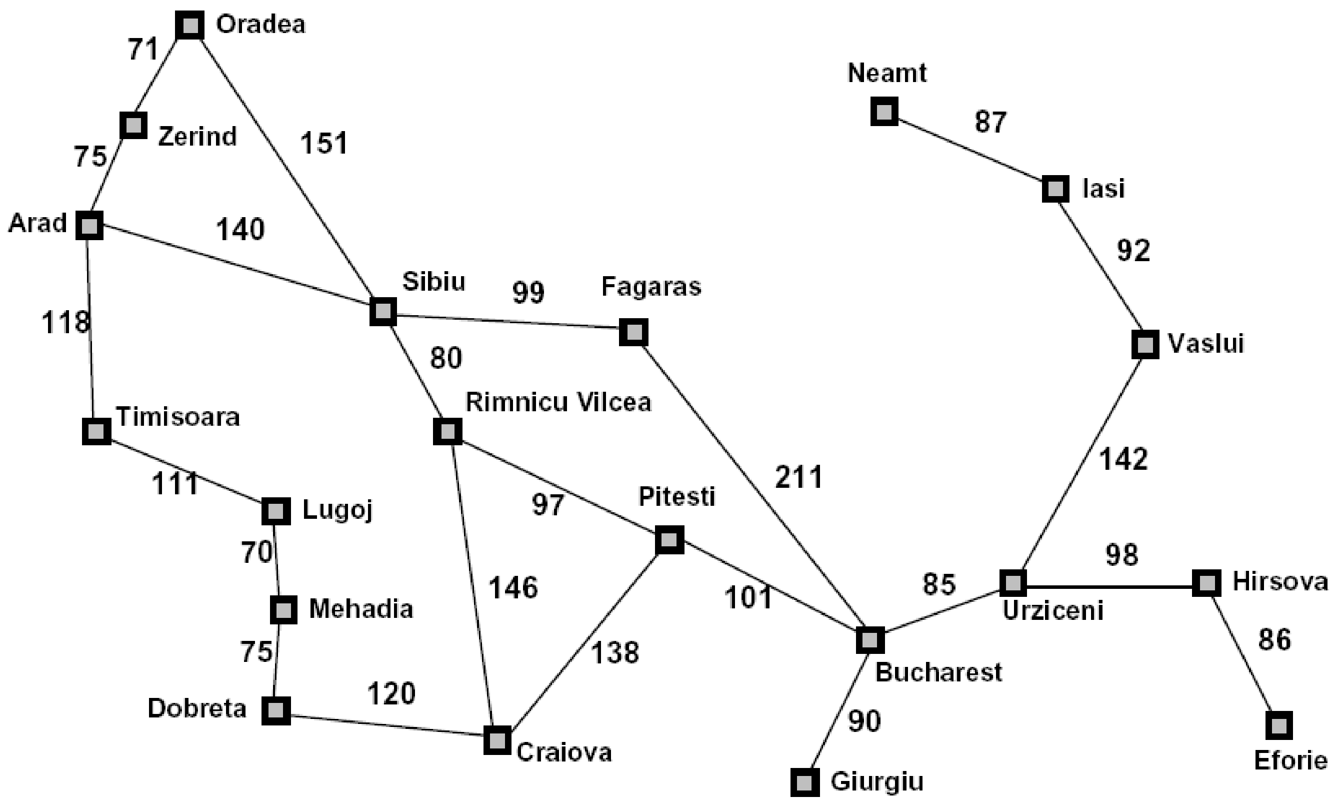
  E.g. Iasi→Fagras

  According to heuristic, Neamt is expanded but it is dead end!

  Solution: Vaslui (farther from goal)→Urziceni→Bucharest→Fagras

- Worst-case: like a badly-guided DFS
  - Can explore everything
  - Can get stuck in loops if no cycle checking
    - e.g., Iasi → Neamt → Iasi → Neamt
  - Can go infinite path and never try other possibilities

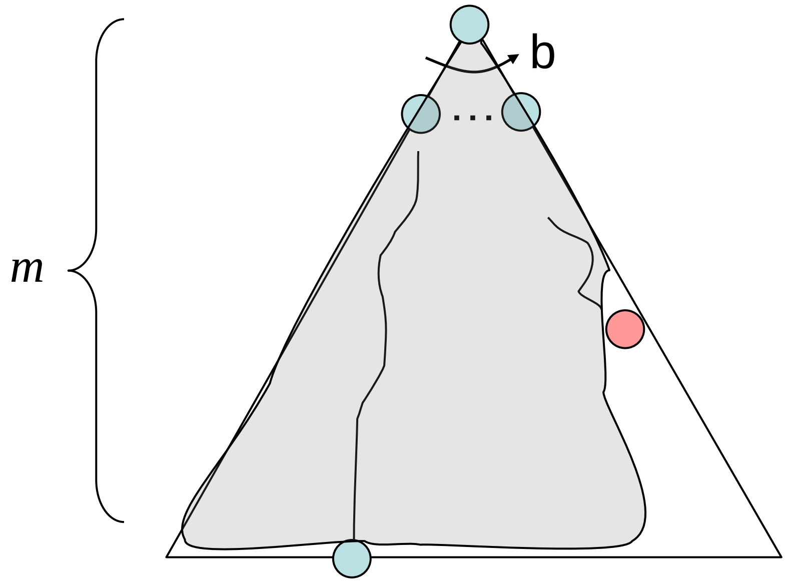
Heuristics



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

Best First Greedy Search

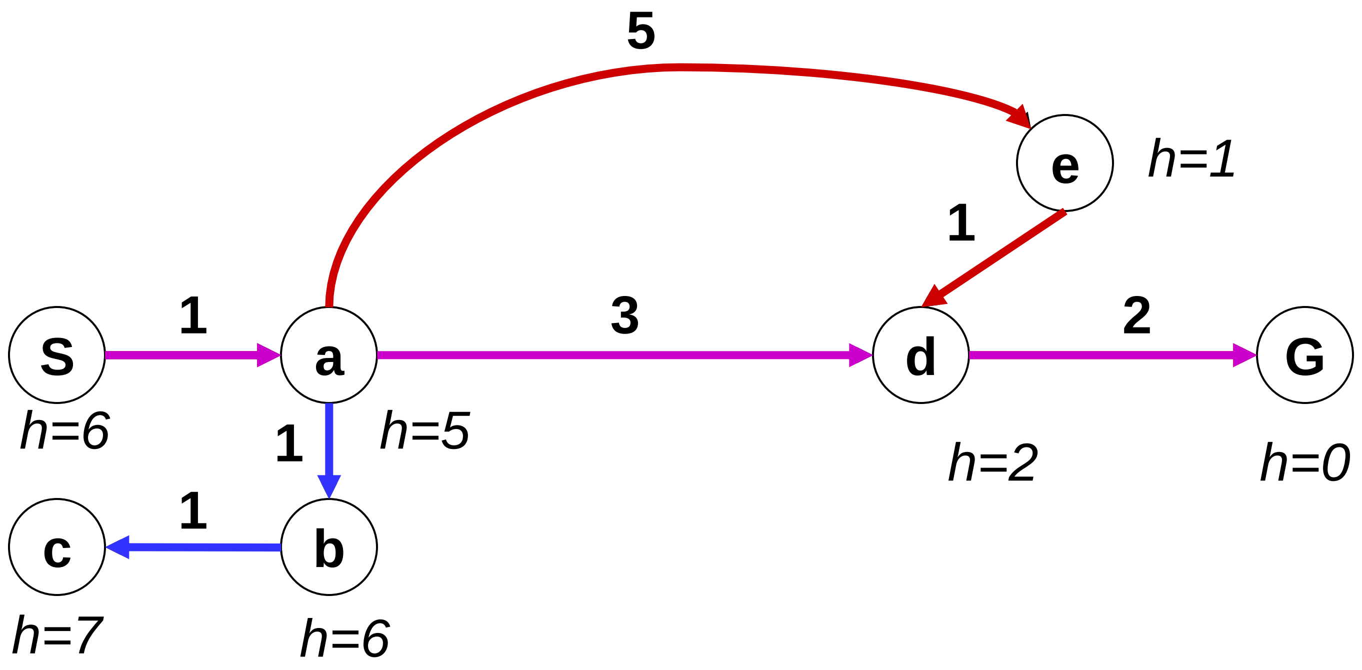
Algorithm	Complete	Optimal	Time	Space
Greedy Best-First Search	Y*	N	$O(b^m)$	$O(b^m)$



- What do we need to do to make it complete? (*repeated cycle check*)
- Can we make it optimal?

Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* **g(n)**
- Best-first orders by distance to goal, or *forward cost* **h(n)**

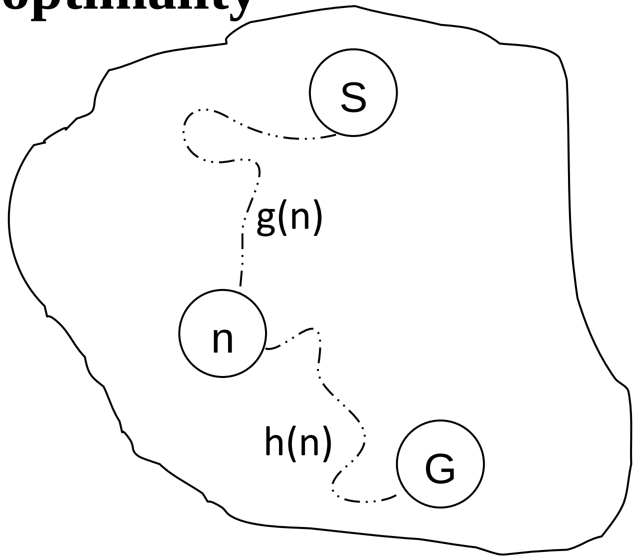


- A* Search orders by the sum: **f(n) = g(n) + h(n)**

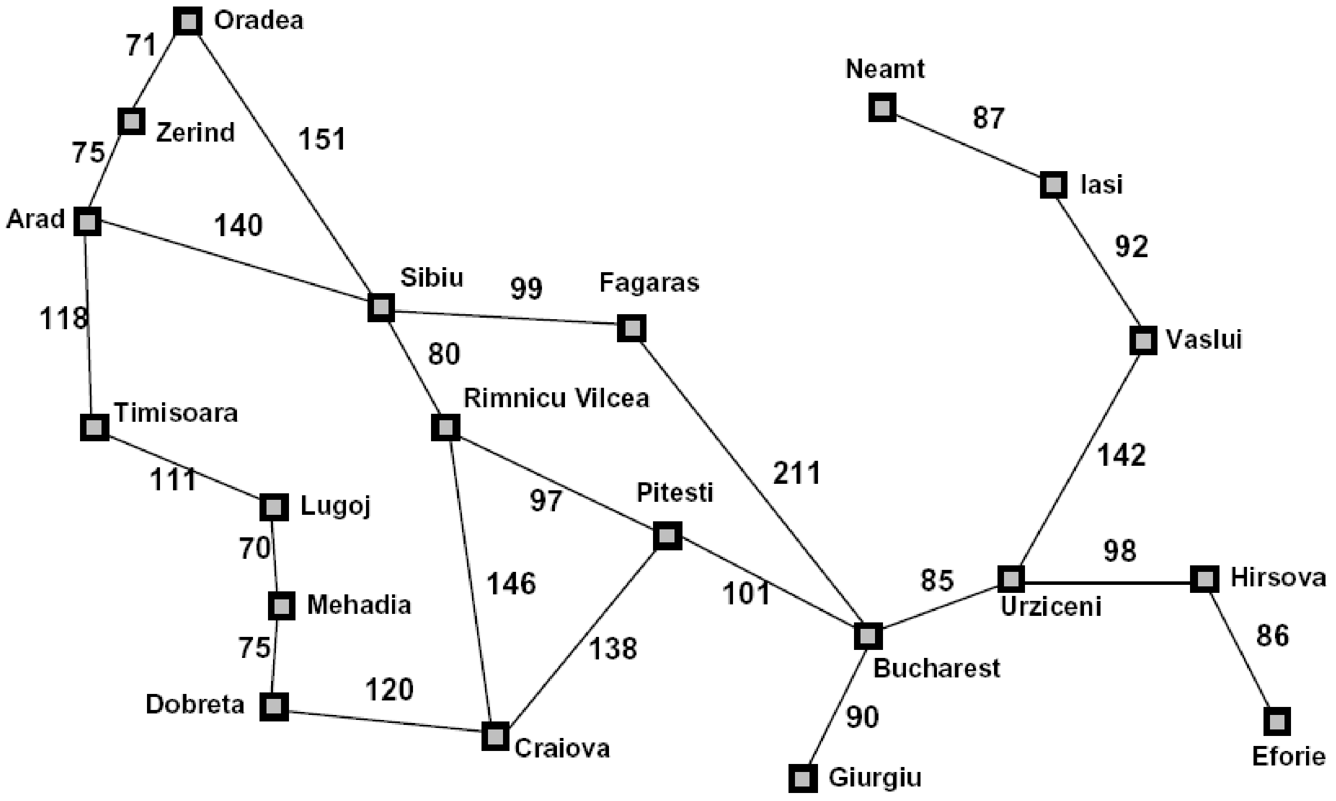
Algorithm A*

- One of the most important advances in AI
  - Idea: *avoid expanding paths that are already expensive*
- *g(n)* = least cost path to n from S found so far
- *h(n)* <= *h*(n)* where *h*(n)* is the actual cost of optimal path to G(node to be found) from *n*

"Optimism leads to optimality"



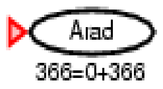
Heuristics



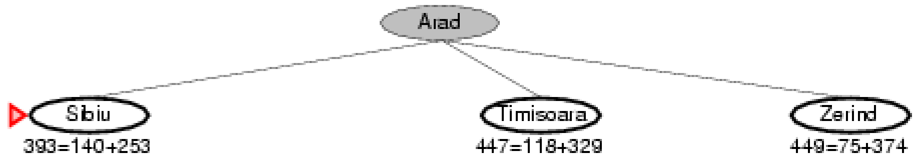
Straight−line distance
to Bucharest

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

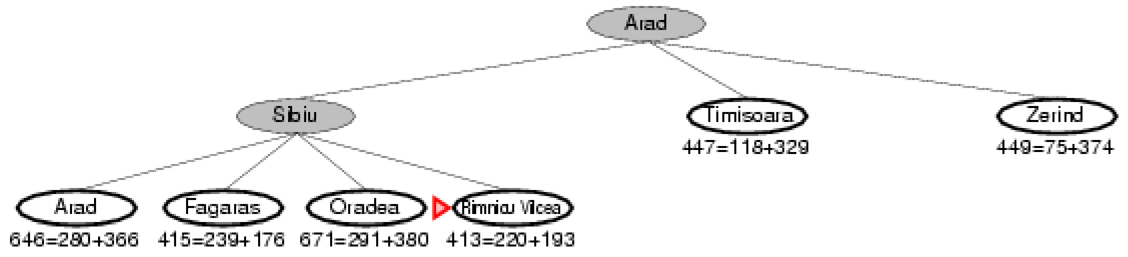
A* search example



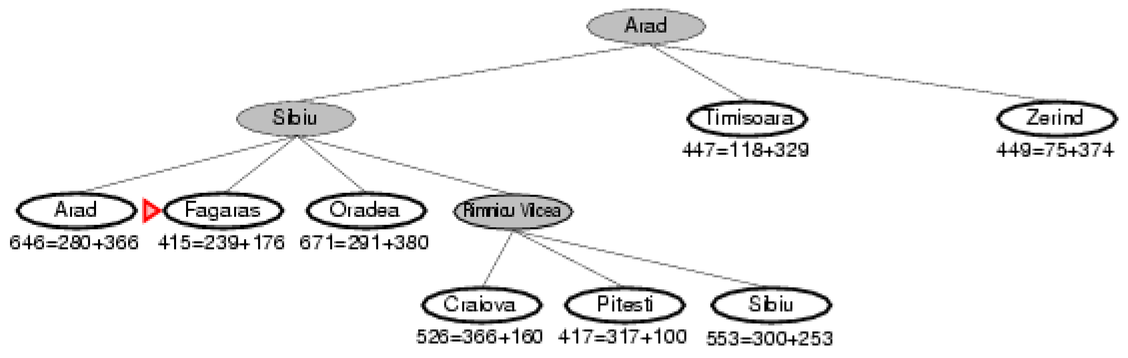
A* search example



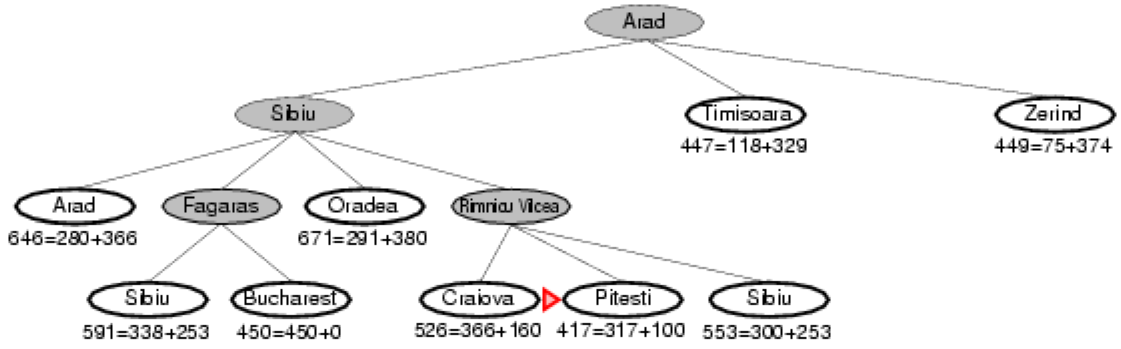
A* search example



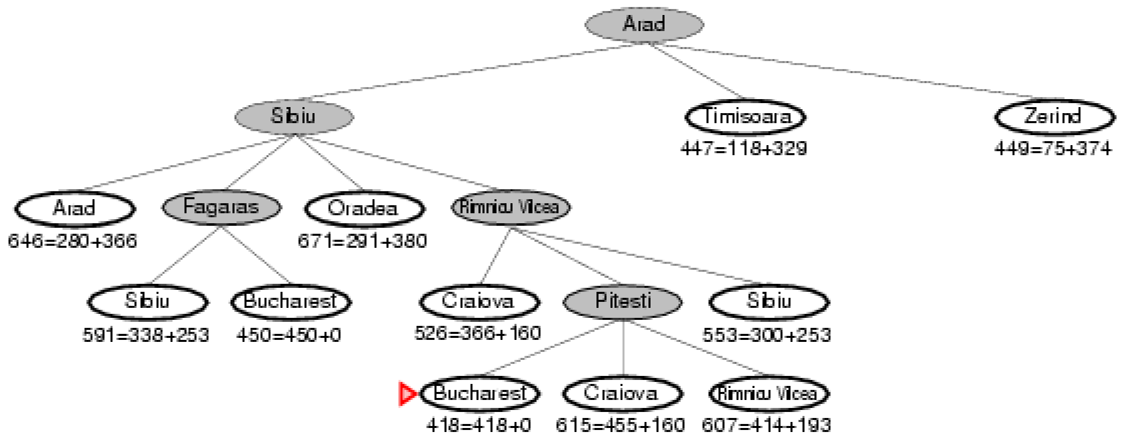
A* search example



A^* search example

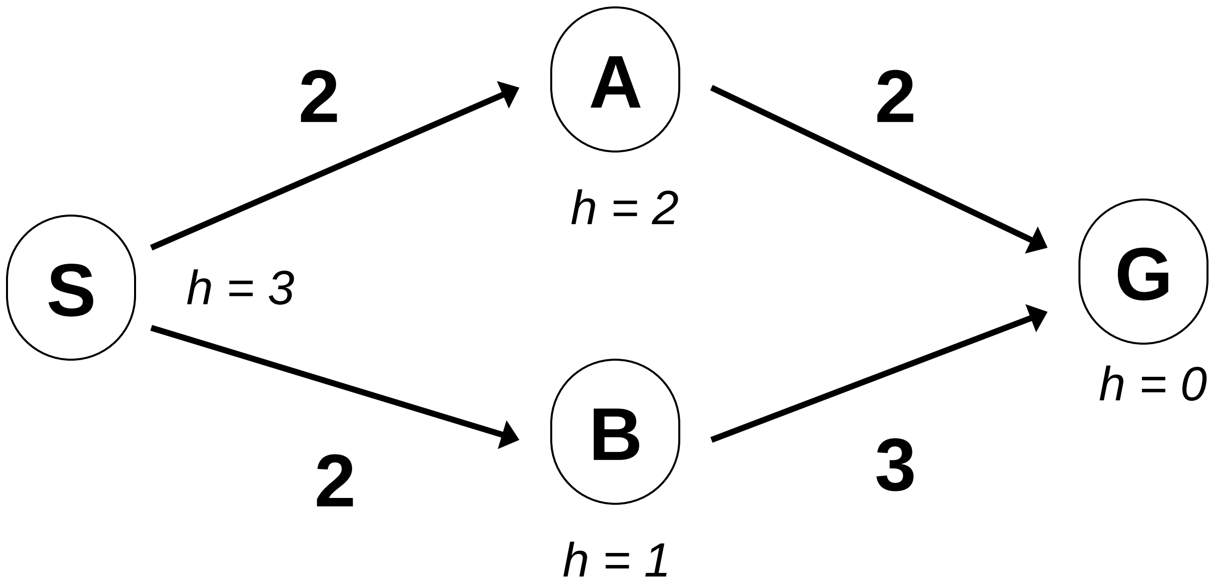


A* search example



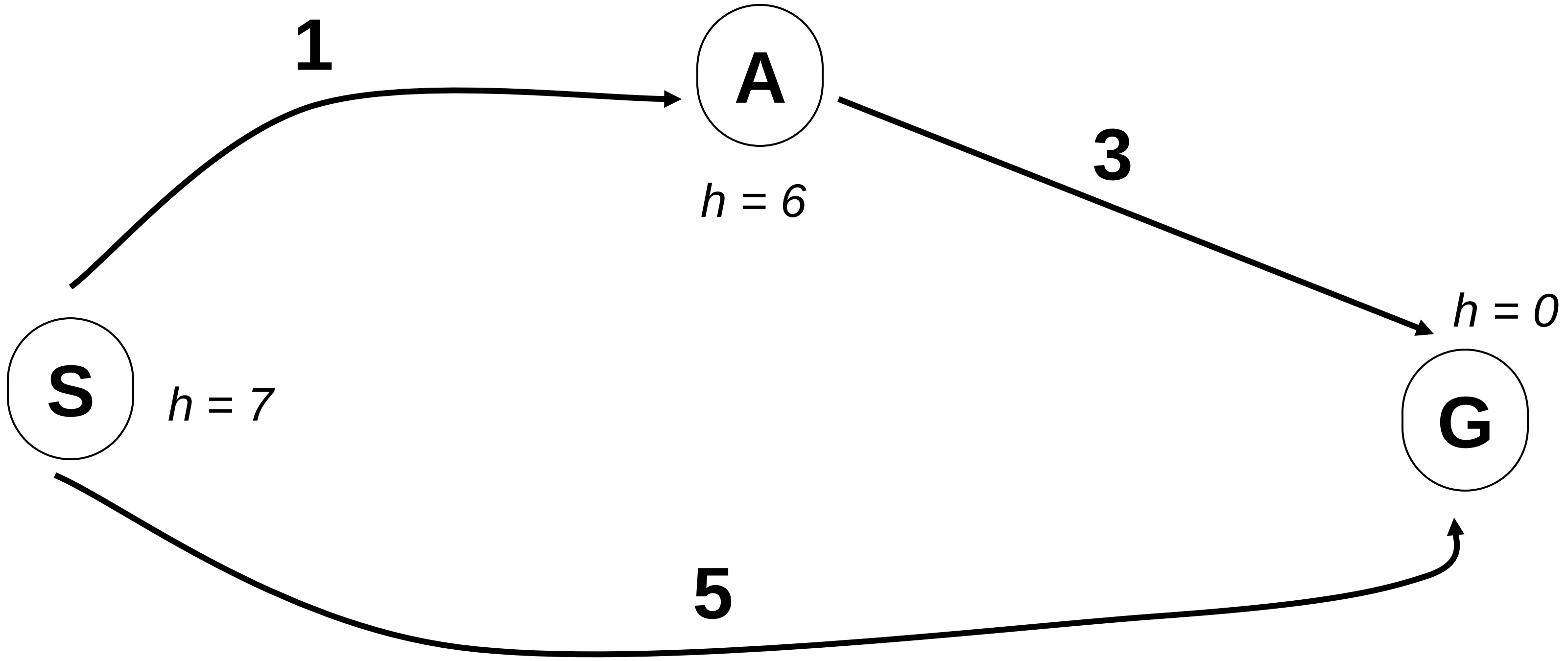
When should A* terminate?

- Should we stop when we enqueue a goal?



- No: only stop when we dequeue a goal

Is A* Optimal?



- What went wrong?
- Actual bad path cost (5) < estimated good path cost (1+6)
- We need estimates (h=7) to be less than actual (5) costs!

Admissible Heuristics

- A heuristic *h* is *admissible* (optimistic) if:

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

where $h^*(n)$ is the true cost to a nearest goal

Never overestimate!

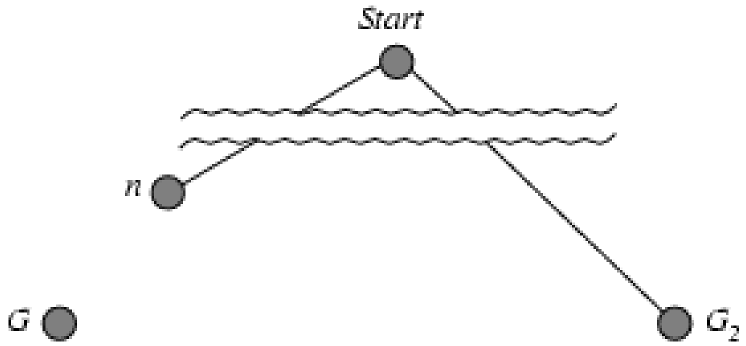
Optimistic: *cost of solution is less than the actual cost!*

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)

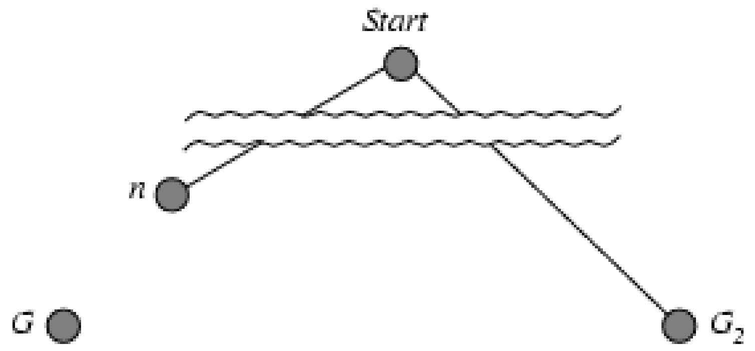
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*



$f(G_2) = g(G_2)$	since $h(G_2) = 0$
$g(G_2) > g(G)$	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(n)=g(n)+h(n) ≤ f(G) since h is admissible

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

BUT … graph search

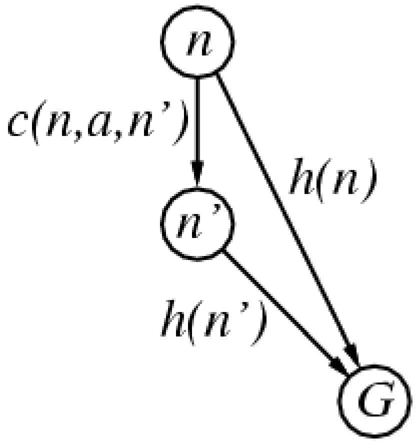
- Discards new paths to repeated state
  - Previous proof breaks down
  - First one may not lead to the optimal one and so can be a problem
- Solution:
  - Add extra bookkeeping i.e. remove more expensive of two paths
  - Ensure that optimal path to any repeated state is always first followed
    - Extra requirement on *h(n)*: consistency (*monotonicity*)

Consistent heuristics

- A heuristic is consistent (or, monotonic) if for every node *n*, every successor *n'* of *n* generated by any action *a*,

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

  -general form of triangle inequality
  -straight line distance



If *h* is consistent, we have

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

i.e., *f(n)* is non-decreasing along any path (*so the first goal node expanded is always least expensive*)

- Theorem: If *h(n)* is consistent, A* using `GRAPH-SEARCH` is optimal

A* search (properties)

- Completeness: YES
  - Since bands of increasing *f* are added
  - Unless there are infinitely many nodes with $f<f(G)$

A* search (properties)

- Completeness: YES
- Time complexity:
  - Number of nodes expanded is still exponential in the length of the solution

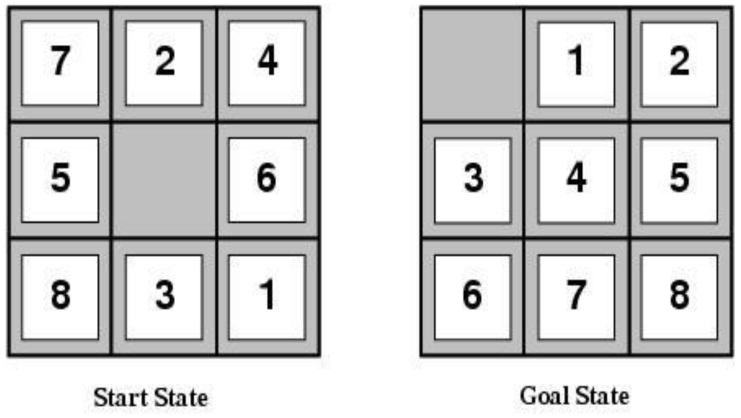
A* search (properties)

- Completeness: YES
- Time complexity: (exponential with path length)
- Space complexity:
  - It keeps all generated nodes in memory
  - Hence space is the major problem not time

A* search (properties)

- Completeness: YES
- Time complexity: exponential with path length
- Space complexity: all nodes are stored
- Optimality: YES
  - Cannot expand f_{i+1} until f_i is finished.
  - A* expands all nodes with $f(n)< C^*$
  - A* expands some nodes with $f(n)=C^*$
  - A* expands no nodes with $f(n)>C^*$
- *Optimally efficient*
  - *No other algorithm guaranteed to expand fewer nodes than A**

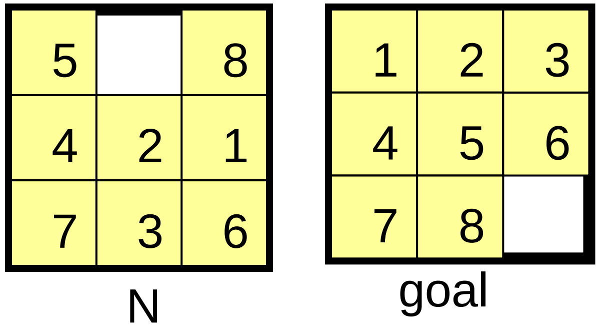
Heuristic functions



- E.g for the 8-puzzle
  - Avg. solution cost is about 22 steps (branching factor +/- 3)
  - Exhaustive search to depth 22: 3^{22} (3.1×10^{10}) states (*for tree*)
  - 9!/2 (=181,440) states are actually reachable (*for graph*)
  - A good heuristic function can reduce the search process

Heuristic Function (Admissible heuristic)

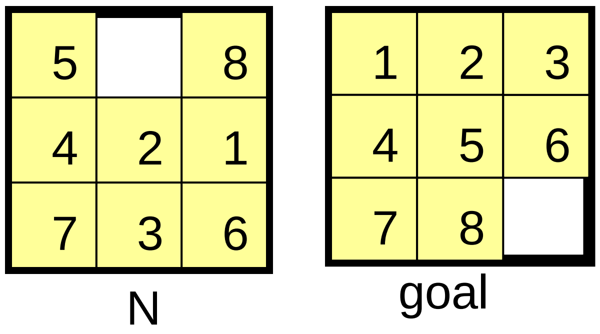
- Function h(N) that estimates the cost of the cheapest path from node N to goal node.
- Example: 8-puzzle



$h_1(N)$ = number of misplaced tiles
= 6

Heuristic Function (Admissible heuristic)

- Function h(N) that estimate the cost of the cheapest path from node N to goal node.
- Example: 8-puzzle



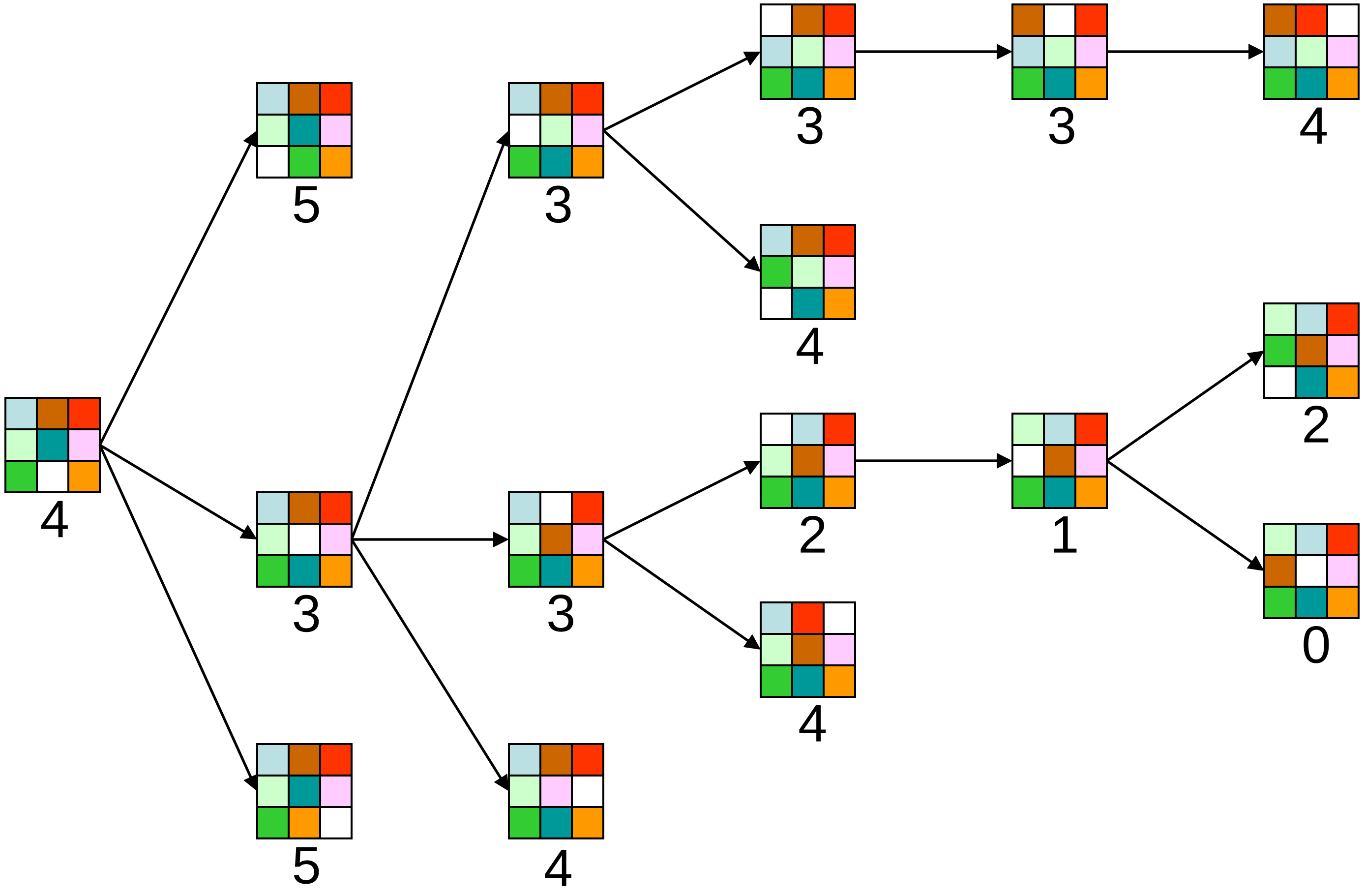
total Manhattan distance =$h_2(N)$ = sum of the distances of every tile to its goal position

= 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1

= 13

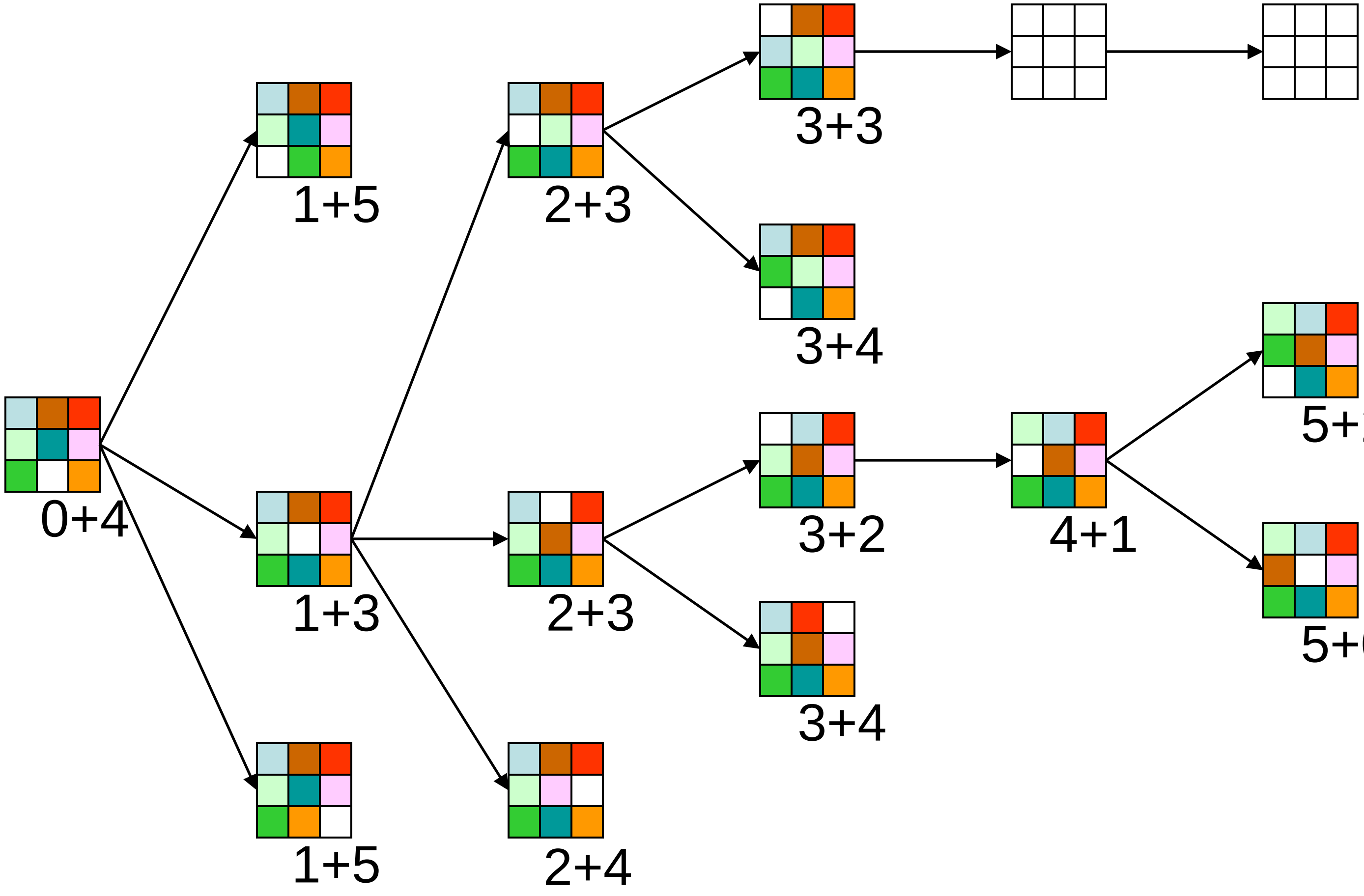
8-Puzzle

f(N) = h_1(N) = number of misplaced tiles



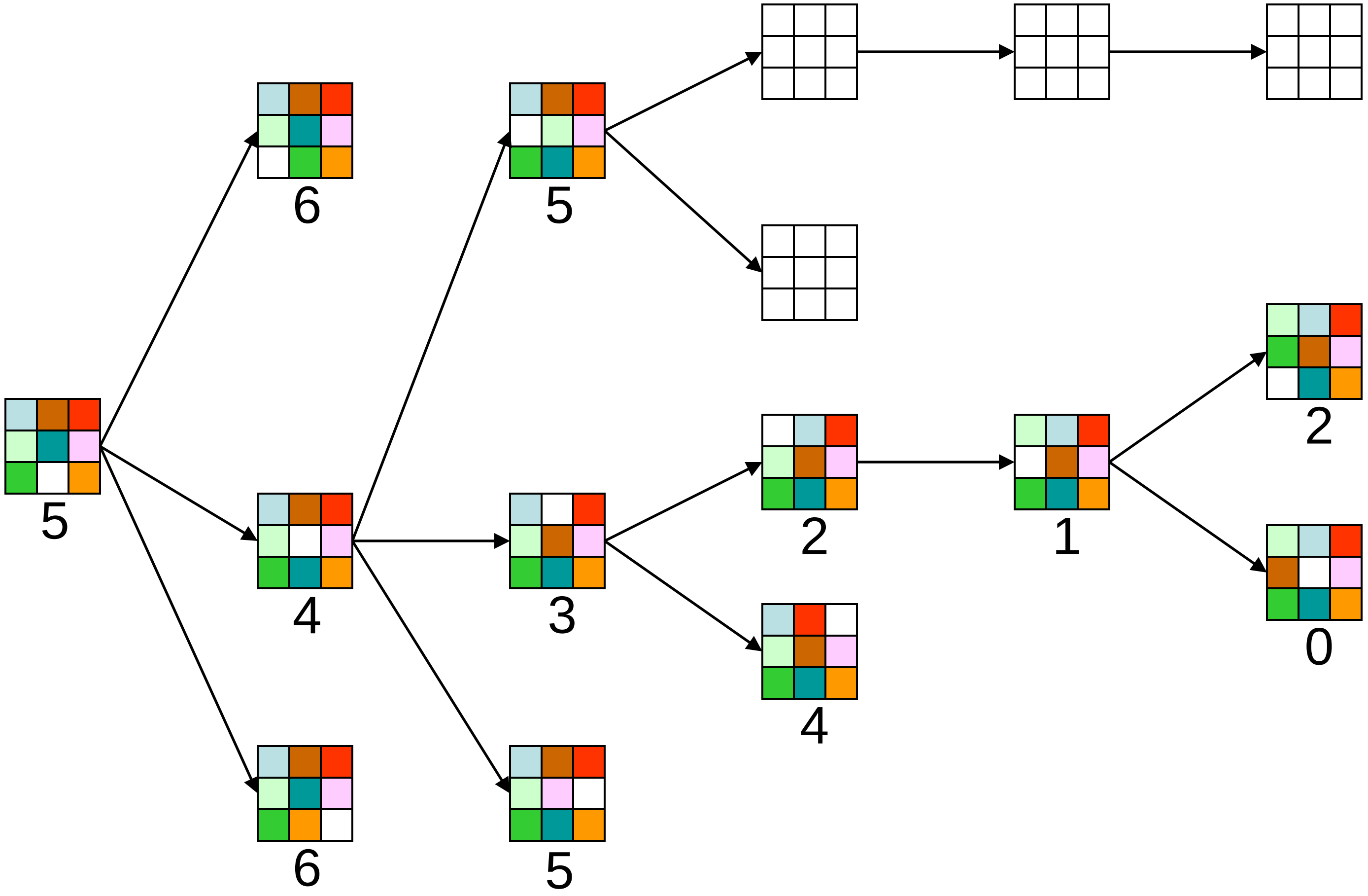
8-Puzzle

f(N) = g(N) + h(N)
with $h_1(N)$ = number of misplaced tiles



8-Puzzle

$f(N) = h_2(N) = \Sigma$ distances of tiles to goal



Heuristic quality

- Effective branching factor b*
  - Is the branching factor that a uniform tree of depth *d* would have in order to contain *N+1* nodes

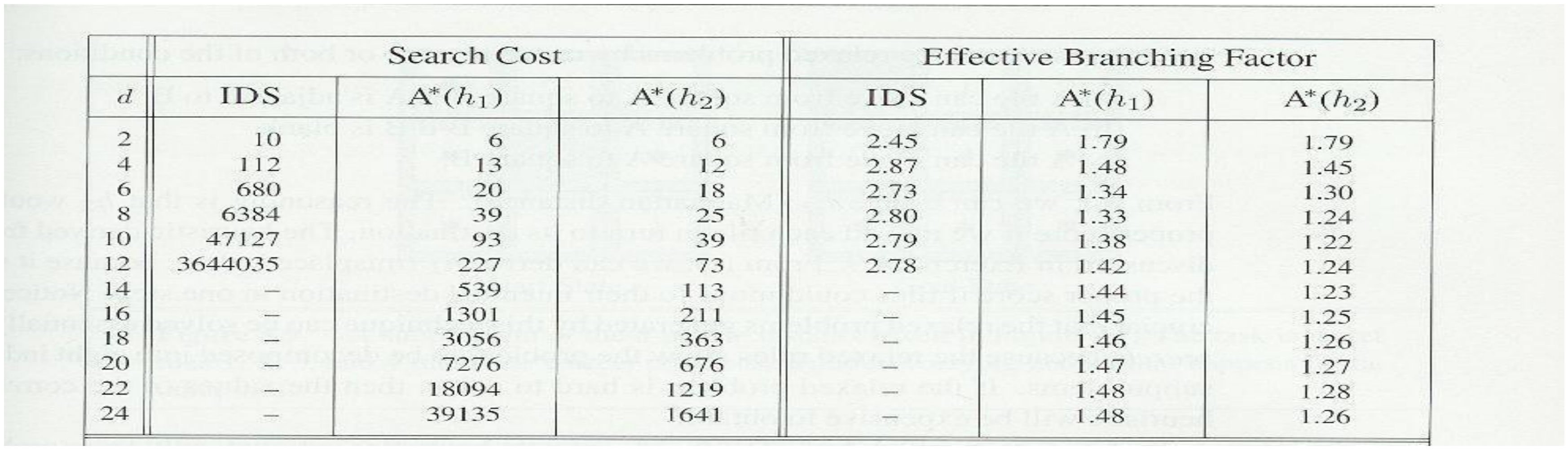
$$N+1=1+b^*+(b^*)^2+\ldots+(b^*)^d$$

  - Measure is fairly constant for sufficiently hard problems
    - Can thus provide a good guide to the heuristic's overall usefulness
    - A good value of b* is 1 (*denoting fairly complex problems to be solved at reasonable computational cost*)

Heuristic quality and 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):



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

$h_2 > h_1$

How to come up with h2?

Can a computer invent mechanically?

Solution: convert the problem to be a relaxed problem with fewer restrictions!

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

Inventing admissible heuristics

- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem:

  - Relaxed 8-puzzle for h_1 : a tile can move *anywhere*

    As a result, $h_1(n)$ gives the shortest solution

    Tiles move their intended destination in one step

  - Relaxed 8-puzzle for h_2 : a tile can move to *any square even on the occupied one*

    As a result, $h_2(n)$ gives the shortest solution

    Each tile is moved in turn to its destination

Inventing admissible heuristics

- *Heuristic is admissible*
  - Optimal solution in the original problem is also a solution in the relaxed problem
  - Must be at least as expensive as that of the optimal solution in the relaxed problem
- *Heuristic is consistent*: derived heuristic is an exact cost for the relaxed problem
- Relaxed problems can be solved essentially without search
  - Relaxed rules allow the problem to be decomposed into 8 independent sub-problems (for 8-puzzle)
  - Heuristics will be expensive if relaxed problem is hard to solve

Inventing admissible heuristics

- ABSOLVER
  - Can generate heuristics automatically from problem definition
  - Uses relaxed problem method and some other techniques (Prieditis, 1993)
- Getting clearly best heuristic is difficult

Let h1, h2, …, hn be admissible heuristics and none of them dominates each other

Which to choose? (*none!*)

Solution: define a composite heuristic h(n) for any node n

h(n)= max{h1(n), h2(n), …, hn(n)}

Choose the most accurate function for n

h is admissible as all component heuristics are admissible

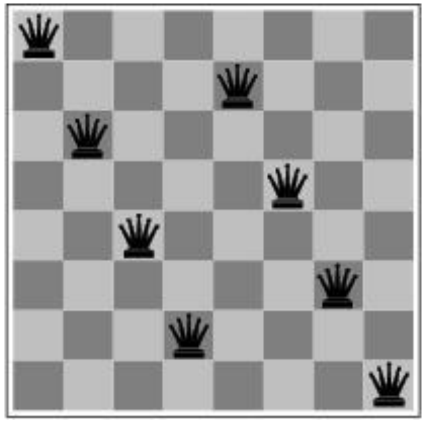
h dominates all components

Another approach… Local Search

Local Search

- Previously: systematic exploration of search space
  - Path to goal is solution to problem
- YET, for some problems path is irrelevant
  - E.g. 8-queens (only final configuration matters and not the path)

- Other applications
  - Integrated circuit design
  - Factory-floor layout
  - Job-shop scheduling
  - Automatic programming
  - Network optimizations
  - Vehicle routing
  - Protfolio management



Local Search

- Different algorithms can be used: Local Search
    - Hill-climbing or Gradient ascent
    - Simulated Annealing
    - Genetic Algorithms, others…

- Useful for solving pure optimization problems
    - Systematic search doesn't work
    - Aim is to find best state according to an *objective function*
    - However, can start with a suboptimal solution and improve it

- Many optimization problems don't fit the *standard search technique*
    - Nature provides an *objective function-reproductive fitness* that *Darwinian* evolution could be seen as attempting to *optimize*, but there is no *goal test* and *path cost* for the problem

Local search and optimization

- Local search: *use single current state and move to neighboring states*
- Advantages:
  - Uses very little memory
  - Finds often reasonable solutions in large or infinite state spaces
- Are also useful for pure optimization problems
  - Find best state according to some *objective function*

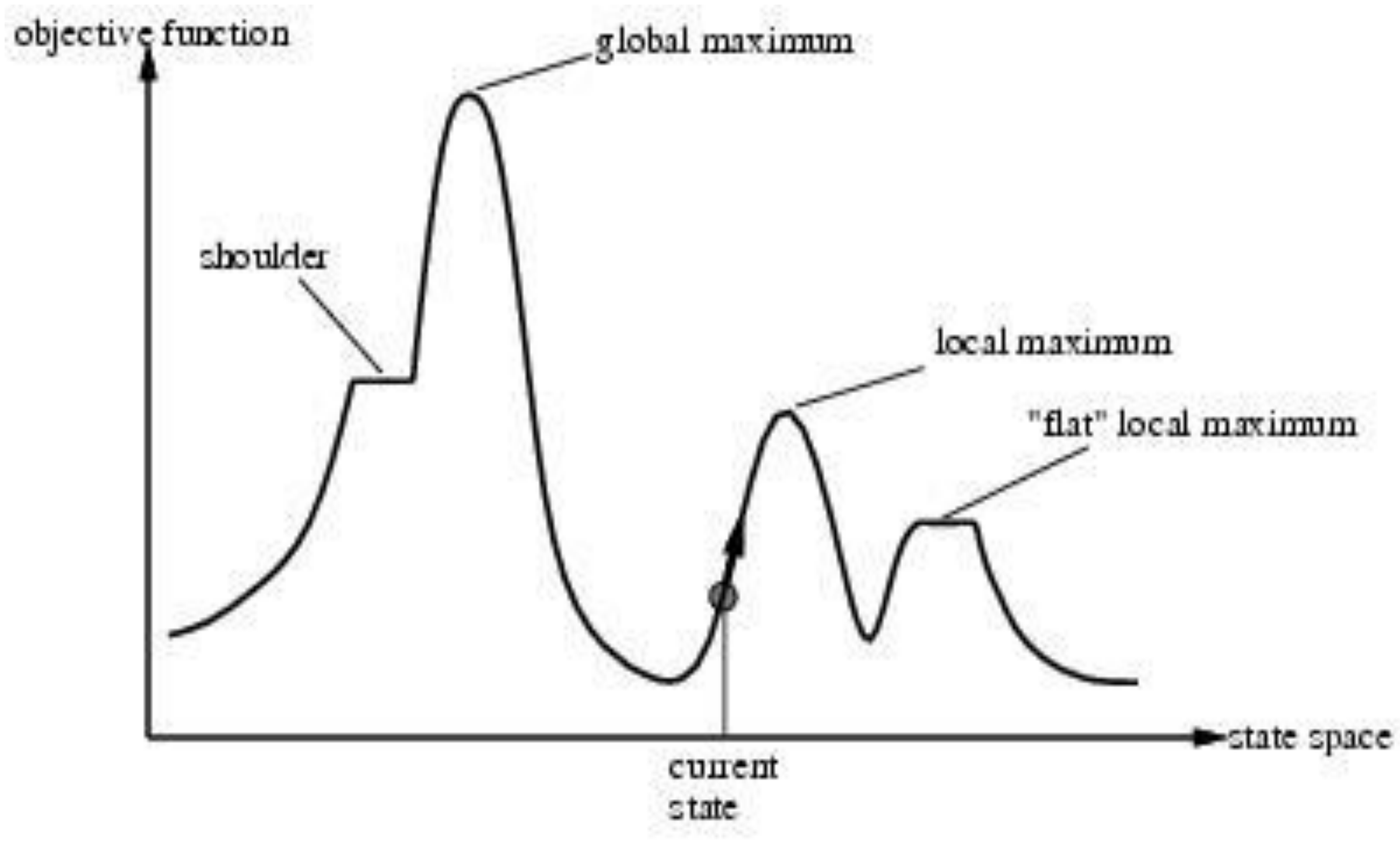
Local Search and Optimization

- State space landscape
  - Location (defined by state)
- Elevation
  - Defined by the value of the *heuristic function* or *objective function*
- Elevation corresponds to *cost*
  - *Global minimum* (aim is to find the lowest valley)
- Elevation corresponds to an *objective function*
  - *Global maximum* (aim is to find the highest peak)

- Complete algorithm always finds a goal if one exists

- Optimal algorithm always finds global minimum/maximum

Local search and optimization



Hill-climbing search

- "is a loop that continuously moves in the direction of increasing value"
  - It terminates when a peak is reached
  - No neighbor has higher value
- does not maintain any search tree
  - Current node data structure records *state* and *objective function value*
- does not look ahead beyond the immediate neighbors of the current state
- chooses randomly among the set of best successors, if there is more than one

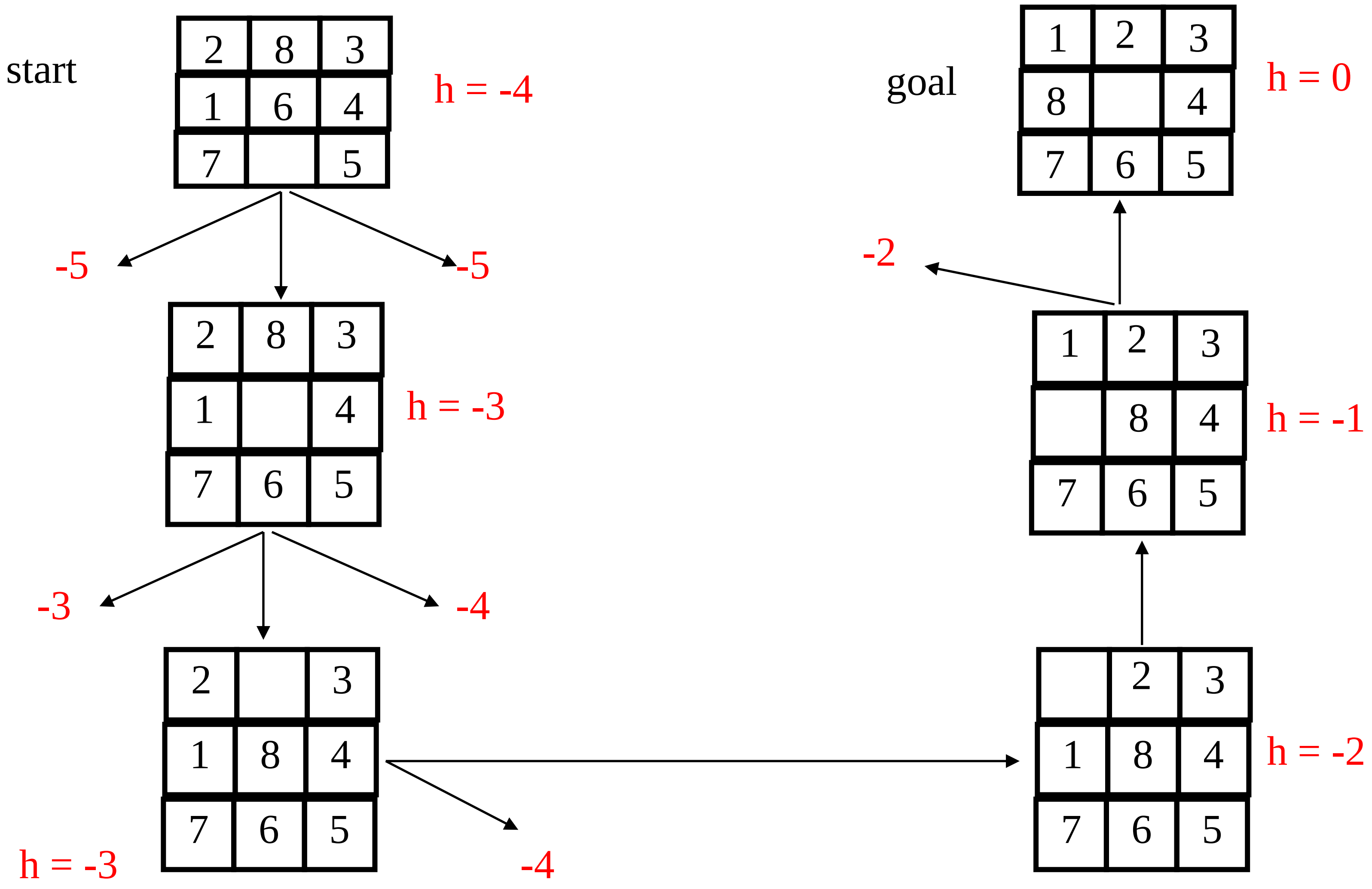
Hill-climbing search

- Hill-climbing a.k.a. greedy local search
  - Grabs a good neighbor state without thinking ahead about where to go next
- Makes very rapid progress towards a solution
  - Quite easy to improve a bad state
- Some problem spaces are great for hill climbing and others are terrible

Hill-climbing search (*steepest ascent*)

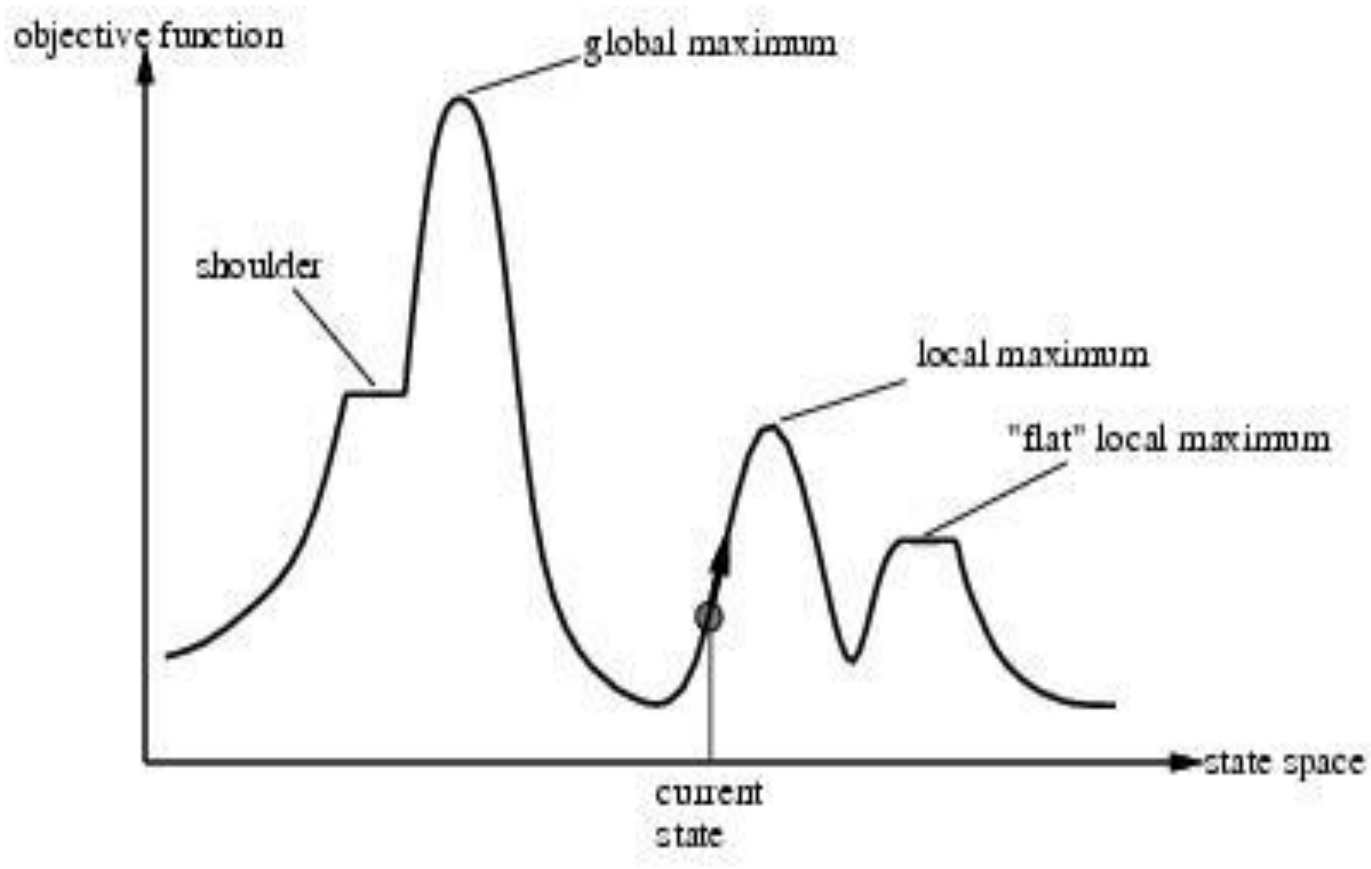
```
function HILL-CLIMBING(problem) return a state that is a
    (global) maximum
    input: problem, a problem
    local variables: current, a node.
                     neighbor, a node.

    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 example



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

Local search and optimization



Drawbacks of hill climbing

- **Local Maxima**
  - peaks that aren't the highest point in the space (*below global maxima*)
  - higher than each of its neighboring states

- **Plateau:** region where evaluation function is flat
  - Flat local maximum: no uphill exit exists
  - Shoulder: possible to make progress
  - Could give the search algorithm no direction (*random walk*) for local maximum

Drawbacks of hill climbing

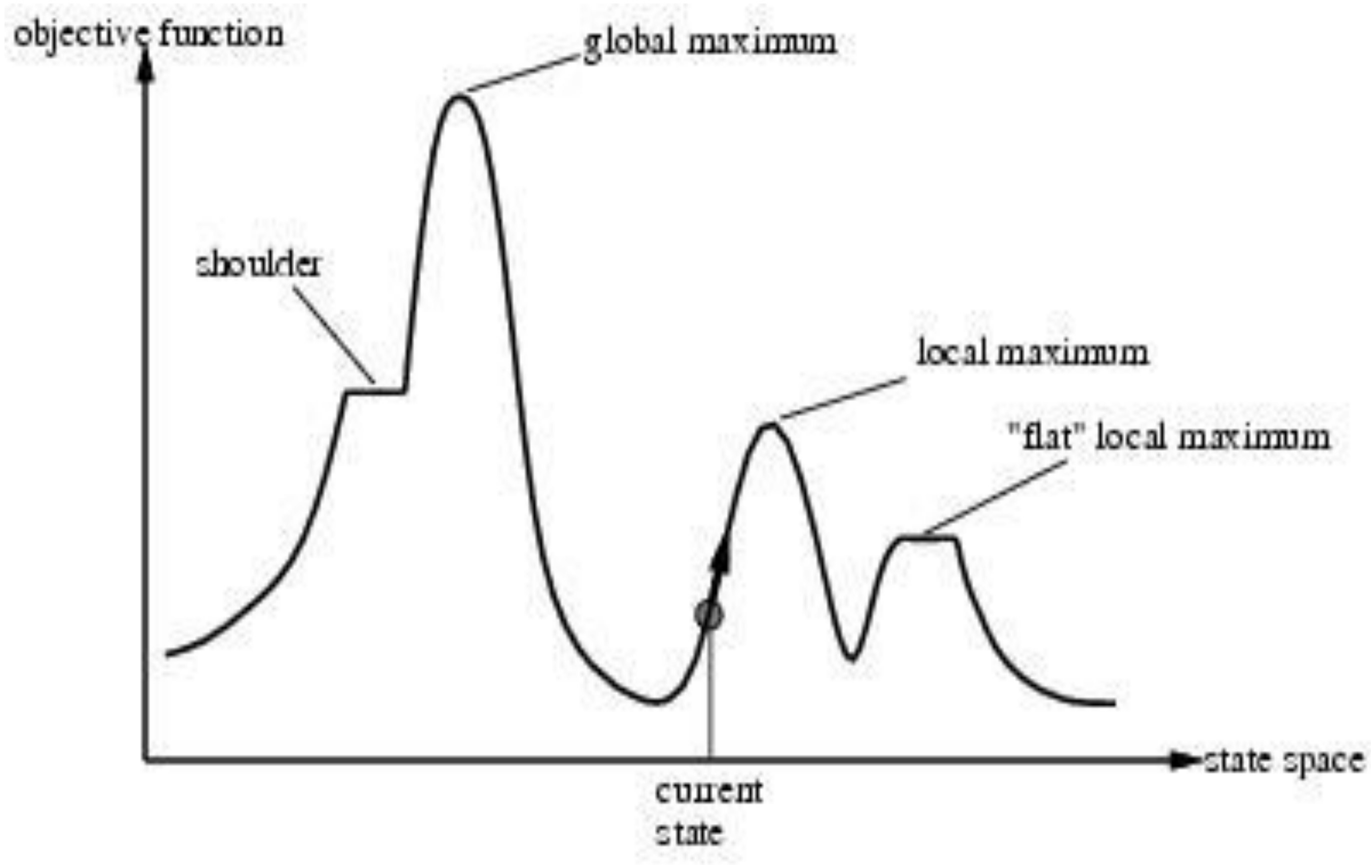
- **Ridges:**
  - flat like a plateau, but with drop-offs to the sides; steps to the North, East, South and West may go down, but a combination of two steps (e.g. N, W) may go up
  - *results in a sequence of local maximum not connected to each other*

- For 8-queens problem
  - Hill climbing gets stuck for the 86% problem instances
  - Solves only 14% instances
  - Average # steps for successful moves: 4
  - Average # steps for unsuccessful moves: 3
  - acceptable solution for a 17-million states

Hill Climbing

- Algorithm halts
  - Reaches a plateau where the best successor has the same value as the current state
- Solution: *sideways move* in the hope that plateau is really a shoulder
  - Infinite loop could occur whenever the algorithm reaches a flat local maxima that is not shoulder
  - Solution: put a limit on the number of consecutive sideways moves allowed
- For the 8-queens problem
  - 100 sideways moves raises the solution level by 80%
  - But with additional cost of roughly 21 steps for each successful instance and 64 for a failure
- Remedy*: Introduce randomness to cope up with the problems*

Local search and optimization



Hill-climbing variations

- Stochastic hill-climbing
  - Random selection among the uphill moves
  - Selection probability can vary with the steepness of the uphill move
  - Converges slowly than steepest ascent (*best first search*) but finds better solution in some cases
- First-choice hill-climbing
  - Stochastic hill climbing by generating successors randomly until a better one is found
  - Good strategy when a state has many successors
- Random-restart hill-climbing
  - Tries to avoid getting stuck in local maxima.
  - If at first you don't succeed, try, try again…

Random-restart hill climbing (*some more details*)

- Conducts a series of hill-climbing searches from randomly generated initial states and stop when goal is found

If there are few local maxima and plateaus, it will find solutions very quickly! (for three million queens solutions can be obtained within a minute)

Random hill climbing (*An Example*)

- Conducts a series of hill-climbing searches from randomly generated initial states and stop when goal is found

Suppose, p: probability of success

# restarts = 1/p

For 8-queens with no sideways, p=0.14

#iterations required =7 for finding a goal (failure:6 and success: 1)

# expected steps = *cost of one successful iteration* + (1-p)/p * *cost of failure*

= 4 + (1-0.14)/0.14 *3 = 4+18=22 [4: average # of steps for a successful move; 3: average # of steps for an unsuccessful move]

- Success of hill climbing depends on the shape of the state-space landscape

If there are few local maxima and plateaus, it will find solutions very quickly!

Random hill climbing (*An Example*)

- With side-ways moves

  1/0.94=1.06 iterations required

  Steps:

  1*21 + (0.06/0.94) * 64 = 25

  Assuming

  21: steps for success

  64: Steps for failure

- Why is hill climbing useful?
  - when the amount of time available to perform a search is limited, such as with real-time systems
  - can return a valid solution even if it's interrupted at any time before it ends

Simulated annealing

- Hill climbing that does *not make downhill move is incomplete*
- Pure random walk: choosing successor from a list is *complete* but *inefficient*

Solution: hill climbing + random walk =simulated annealing

to ensure both efficiency and completeness

Simulated Annealing

Use a more complex Evaluation Function:

- Do sometimes accept candidates with higher cost to escape from local optimum
  - Idea: but gradually decrease their size and frequency
- Adapt the parameters of this evaluation function during execution
- Based upon the analogy with the simulation of the annealing of solids

Simulated annealing

- Origin: metallurgical annealing
  - Temper or harden metals and glass by heating them to a high temperature and then gradually cooling them
  - Allowing material to coalesce into a low-energy crystalline state
- Application: VLSI layout, airline scheduling, TSP etc.

Other Names

- Monte Carlo Annealing
- Statistical Cooling
- Probabilistic Hill Climbing
- Stochastic Relaxation
- Probabilistic Exchange Algorithm

Analogy

- Slowly cool down a heated solid, so that all particles arrange in the ground energy state
- At each temperature wait until the solid reaches its thermal equilibrium
- Probability of being in a state with energy *E* :

$$Pr\{\mathbf{E} = E\} = 1/Z(T) \, . \, \exp(-E / k_B . T)$$

E	Energy
T	Temperature
k_B	Boltzmann constant
$Z(T)$	Normalization factor (temperature dependent)

Simulation of cooling (Metropolis 1953)

At a fixed temperature T :

- Perturb (randomly) the current state to a new state
- ΔE is the difference in energy between new and current state
- If $\Delta E < 0$ (new state is lower), accept new state as current state
- If $\Delta E \geq 0$, accept new state with probability

  $$Pr\ (accepted) = exp\ (- \Delta E / k_B . T)$$
- Eventually the systems evolves into thermal equilibrium at temperature T
- When equilibrium is reached, temperature T can be lowered and the process can be repeated

Simulated Annealing

- Same algorithm can be used for combinatorial optimization problems:

Energy *E* corresponds to the Cost function *C*

Temperature *T* corresponds to control parameter *c*

$$Pr\{\text{configuration} = i\} = 1/Q(c) \,.\, \exp(-C(i)/c)$$

C	Cost
c	Control parameter
Q(c)	Normalization factor (not important)

Simulated annealing

- Unlike hill climbing, it does not always pick the *best move*
- SA picks the move randomly
  - If situation improves then accept the move
  - Otherwise, accept the move with some probability

- Probability decreases as the temperature goes down
- Probability decreases exponentially with the *badness of the move*
- Bad moves are more likely to be allowed at the start when temperature is high, and they are unlike when temperature decreases

Simulated annealing

```
function SIMULATED-ANNEALING( problem, schedule) return a solution state
    input: 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 ← 1 to ∞ do
           T ← schedule[t]
           if T = 0 then return current
           next ← a randomly selected successor of current
           ΔE ←  VALUE[current] - VALUE[next]
           if ΔE > 0 then current ← next
           else current ← next only with probability $e-^{\Delta E / T}$
```

Local beam search

- Keep track of *k* states instead of one
  - Initially: *k* random states
  - Next: determine all successors of *k* states
  - If any of successors is goal $\rightarrow$ finished
  - Else select *k* best from successors and repeat

- *Is it equivalent of running k random restarts in parallel ?*

- Random-restart search vs. Beam search
  - In random-restart, each process runs independently
  - In beam-search, information is shared among *k* search threads

Local beam search

- Kth state generates good successors and all k-1 states generate bad states
  - Abandons k-1 unfruitful searches and move to the kth state
- Can suffer from lack of diversity
  - Quickly become concentrated in a small region of the state space
  - Not very good in comparison to expensive version of hill climbing
- Stochastic variant: choose k successors (*at random and not the best ones*) at proportionally to state success
- Resemblance to the process of natural selection
  - *Successors (offspring)*
  - *State (organism)*
  - *Populate the next generation according to fitness value*

Genetic algorithms

- Variant of stochastic beam search
- *Guided by the principles of natural genetics that follow Darwin evolution*
- 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): *chromosome representation*
- Evaluation function (fitness function): Higher values for better states

- Produce the next generation of states by *selection*, *crossover*, and *mutation*

Genetic Algorithm: Introduction

- Randomized search and optimization technique
- Evolution produces good individuals, similar principles might work for solving complex problems
- Developed: USA in the 1970’s by J. Holland
- Got popular in the late 1980’s
- Early names: J. Holland, K. DeJong, D. Goldberg
- Based on ideas from *Darwinian Evolution*
- Can be used to solve a variety of problems that are not easy to solve using other techniques

GA: Quick Overview

- Typically applied to:
  - discrete optimization
- Attributed features:
  - not too fast
  - good heuristic for combinatorial problems
- Special Features:
  - Traditionally emphasizes combining information from good parents (*crossover*)
  - many variants, e.g., reproduction models, operators

Evolution in the real world

- Each cell of a living being contains ***chromosomes***-strings of *DNA*
- Each chromosome contains a set of ***genes***-blocks of DNA
- Each gene determines some aspects of the organism (like *eye colour*)
- A collection of genes sometimes called a ***genotype***
- A collection of aspects (like *eye colour*) sometimes called a ***phenotype***
- Reproduction involves recombination of genes from parents and then small amount of ***mutation*** (errors) in copying
- The ***fitness*** of an organism is how much it can reproduce before it dies
- Evolution based on "*survival of the fittest*"

Genetic Algorithm: Similarity with Nature

Genetic Algorithms ←→	Nature
A solution (phenotype)	Individual
Representation of a solution (*genotype*)	Chromosome
Components of the solution	Genes
Set of solutions	Population
Survival of the fittest (*Selection*)	Darwins theory
Search operators	Crossover and mutation
Iterative procedure	Generations

Genetic Algorithm: Basic Concepts

- Parameters of the search space encoded in the form of strings - *Chromosomes*
- Collection of chromosomes -*Population*
- *Initial step*: a random population representing different points in the search space
- *Objective or fitness function*: associated with each string
  - represents the degree of *goodness* of the string

Genetic Algorithm: Basic Concepts

- Selection
  - Based on the principle of survival of the fittest, a few of the strings selected
- Biologically inspired operators like *crossover* and *mutation* applied on these strings to yield a new generation of strings
- Processes of *selection*, *crossover* and *mutation* continue for a fixed number of generations or till a termination condition satisfied

Basic Steps of Genetic Algorithm

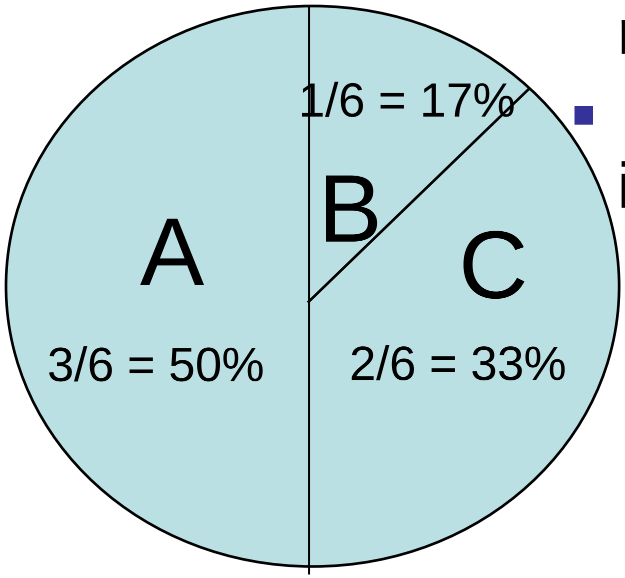
```
1. $t = 0$
2. initialize population $P(t)$  /* $Popsize = |P|$ */
3. for $i = 1$ to $Popsize$
     compute fitness $P(t)$
4. $t = t + 1$
5. if termination criterion achieved go to step 10
6. select $(P)$
7. crossover $(P)$
8. mutate $(P)$
9. go to step 3
10. output best chromosome and stop
End
```

Example population

No.	Chromosome	Fitness
1	1010011010	1
2	1111100001	2
3	1011001100	3
4	1010000000	1
5	0000010000	3
6	1001011111	5
7	0101010101	1
8	1011100111	2

GA operators: Selection

- Main idea: better individuals get higher chance
  - Chances proportional to fitness
  - Implementation: roulette wheel technique
    - Assign to each individual a part of the roulette wheel
    - Spin the wheel n times to select n individuals



$\longleftarrow$

fitness(A) = 3

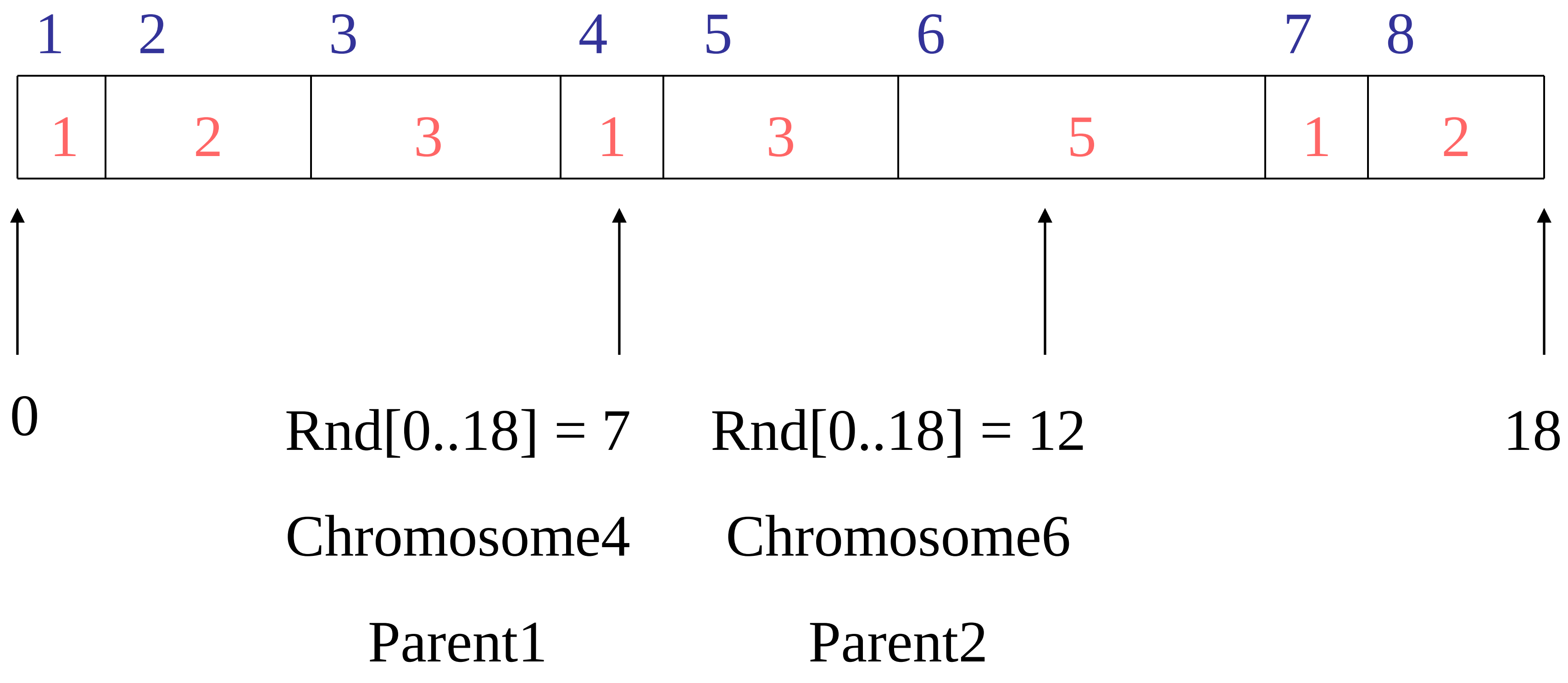
fitness(B) = 1

fitness(C) = 2

GA operator: Selection

- Add up the fitness's of all chromosomes
- Generate a random number R in that range
- Select the first chromosome in the population that -when all previous fitness’s are added - gives you at least the value R

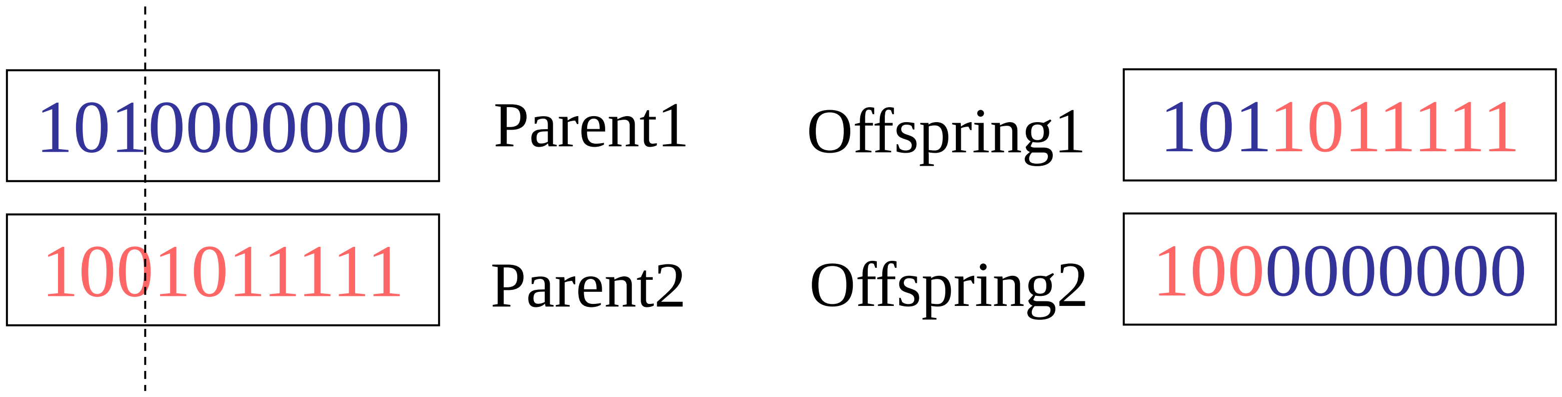
Roulette Wheel Selection



GA operator: Crossover

- Choose a random point on the two parents
- Split parents at this crossover point
- With some high probability (*crossover rate*) apply crossover to the parents
  - P_c typically in range (0.6, 0.9)
- Create children by exchanging tails

Crossover - Recombination

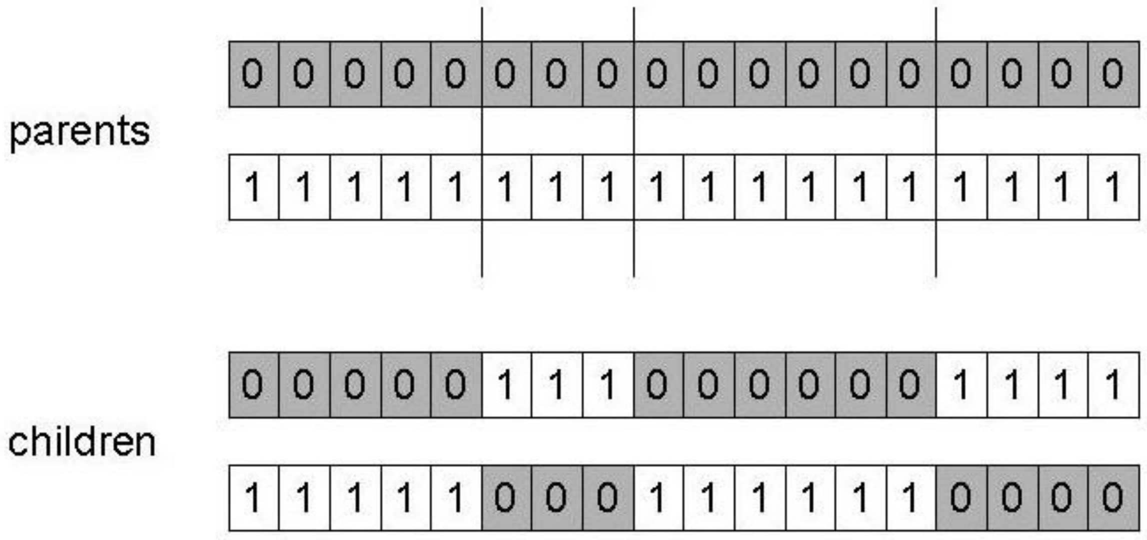


Crossover single point - random

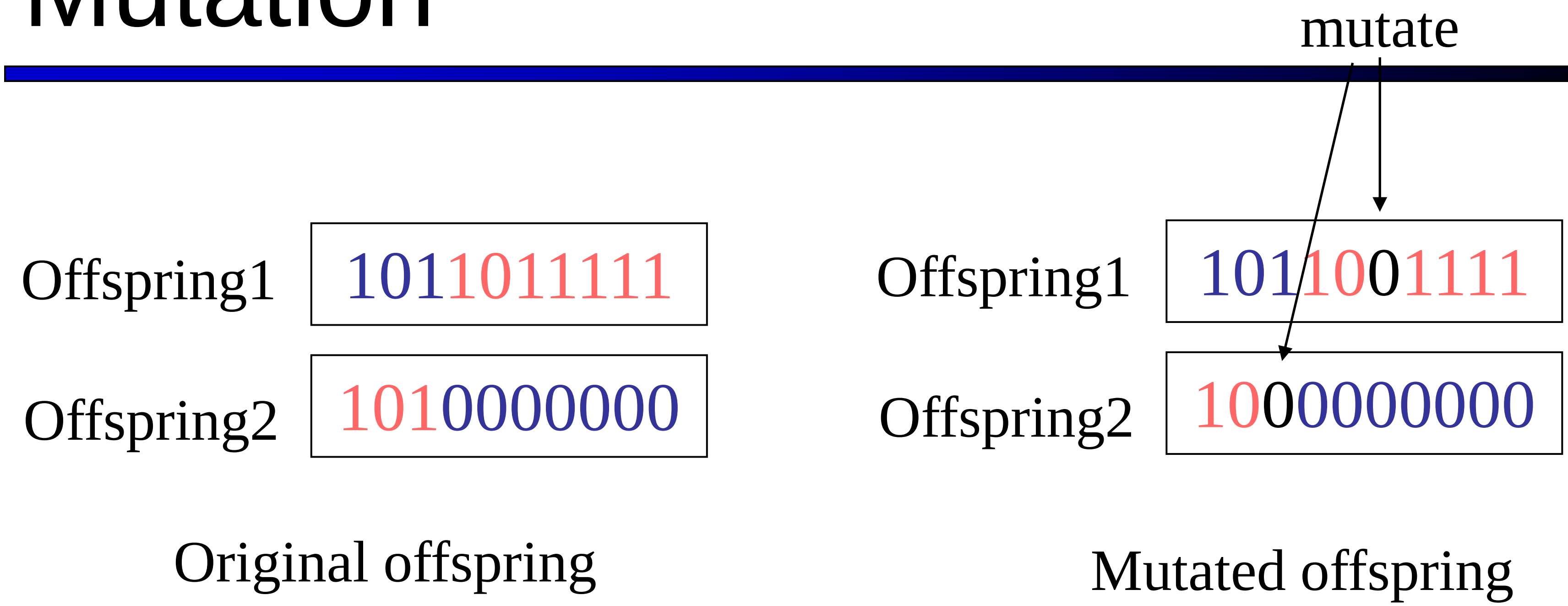
Single Point Crossover

n-point crossover

- Choose n random crossover points
- Split along those points
- Glue parts, alternating between parents
- Generalisation of 1 point (still some positional bias)



Mutation



With some small probability (the *mutation rate*) flip each bit in the offspring (*typical values between 0.1 and 0.001*)

Some points on GA

- Two quite different parents could generate a child that is long away from any of the parents
- More diversity of population at earlier stages
  - Similar to SA
  - Takes large steps early in the search process
  - Smaller steps later on when individuals are quite similar
- Combines uphill tendency with random exploration and exchanges information like stochastic beam search
  - Advantage of GA over stochastic beam search is due to crossover operation
- If random mutation at early stage: no advantage for crossover operation

Some points on GA

- Advantage of crossover comes from the ability of cross-over between large blocks
  - Schema
  - E.g. 246 constitutes a useful block
- Schema
  - 246*****
  - Instances: 24613578 ….
- Number of instances in population grows
  - If average fitness of all instances is greater than the mean
- Effect of schema is insignificant
  - Adjacent bits are completely different

Thus, good chromosome representation is very important