Part 3:
Informed (Heuristic) Search

Part 3A: Best-First Search Algorithms

(NOTE: Part 3B – "Local Search and Iterative Improvement" – will not be covered in this class, because of time constraints)

Univ.-Prof. Dr. Gerhard Widmer
Department of Computational Perception
Johannes Kepler University Linz

gerhard.widmer@jku.at
http://www.cp.jku.at/people/widmer

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Overview

PART A: Best-First Search Algorithms

- Greedy Best-First Search
- A* Search
- Optimality of A* Search
- Designing heuristic functions

PART B: Local Search and Iterative Improvement Algorithms
(not covered in this class)

- Hill-climbing (gradient descent) search
- Variants of hill-climbing
- Stochastic search: Simulated Annealing
- Local beam search
- Genetic / evolutionary algorithms

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Remember: The General Tree Search Algorithm

Uninformed Search Strategies:

- Can find solutions to problems by systematically generating new states and testing for the goal
- Different search strategies differ in how they select the next node for expansion
- Problem: States are just "black boxes", selection/expansion decisions are made independently of the actual contents of a state ➔ search is rather "blind"

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST[problem] applied to STATE(node) succeeds return node
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Informed ("Heuristic") Search: The Best-First Search Schema

Basic idea of informed search:

- Use problem-specific knowledge (about "desirability" / "promise" of states)
- Try to expand "best" (most promising) node next
- Knowledge usually encapsulated in *heuristic evaluation function*

Best-First Search:

- General schema based on TREE-SEARCH (or GRAPH-SEARCH)
- Basic characteristic: next node n to be expanded is the one that minimises some **evaluation function** $f(n)$
- Evaluation function encapsulates the agent's "knowledge" about the problem domain (i.e., what kinds of states seem preferable)
- Usually: f estimates some kind of **distance** between n and a goal
  ➔ expand node with minimal $f(n)$

Implementation:

- Fringe in TREE-SEARCH is a **priority queue**
  ➔ maintains successors sorted according to ascending f-values

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Key Component: The Heuristic Function

"**Heuristic** is the art and science of discovery and invention. The word comes from the same Greek root as "eureka": εύρισκω, which means "I find". **A heuristic is a way of directing your attention fruitfully**. The term was introduced by Pappus of Alexandria in the 4th century." *(www.wikipedia.org)*

"Heuristic" in AI =

simple, efficient "rule of thumb" ("Faustregel") that works "reasonably well" "most of the time", but gives no guarantee of always being correct

Key Ingredient: The Heuristic Function $h(n)$

$h(n)$ = estimated cost of the cheapest path from node n to a goal node;
if n is a goal node ➔ $h(n) = 0$

Points to note:

- Node with the minimal cost path to a goal would be the optimal choice for expansion in TREE-SEARCH, from the start
- If heuristic $h(n)$ were accurate (estimated cost = true cost), search would be reduced to 1 path
- But: heuristics usually encode only partial, imperfect information
- Heuristics are the most common form in which knowledge about the problem can be given to a search algorithm (apart from the representation of the problem itself, which is also important)

In the following: 2 strategies for using heuristics to guide the search:
Greedy Best-First Search + A* Search

Greedy Best-First Search (Greedy Search)

Idea:

- Expand node n that appears to be closest to the goal, i.e., that has the **lowest estimated cost** from n to a goal

➔ Evaluation function $f(n) = h(n)$

Assumption:

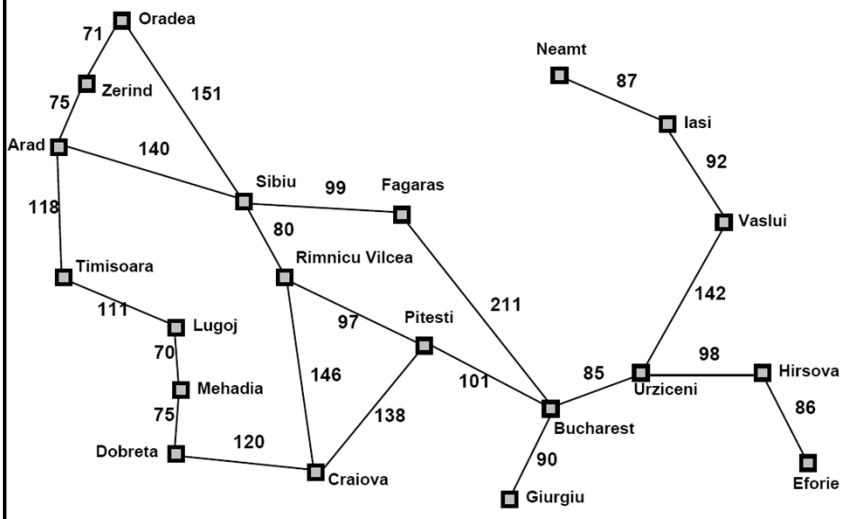
- This will lead to a goal quickly, most of the time (if $h(n)$ is a "reasonably" accurate estimate)

Example:

- Find shortest path from Arad to Bucharest
- Consider the following heuristic function (additional information): $h_{SLD}(n)$ = straight-line distance ("Luftlinie") from n to Bucharest
- h_{SLD} may be correlated with actual road distances, but not necessarily …

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Greedy Best-First Search in Romania



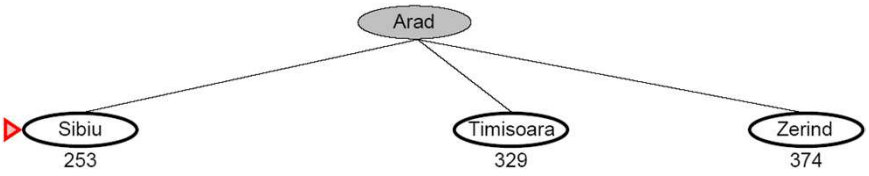
$h_{SLD}(n)$:

Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

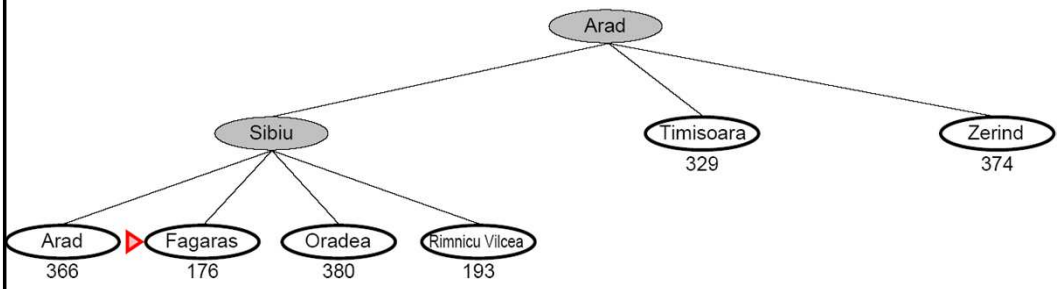
© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Greedy Best-First Search in Romania

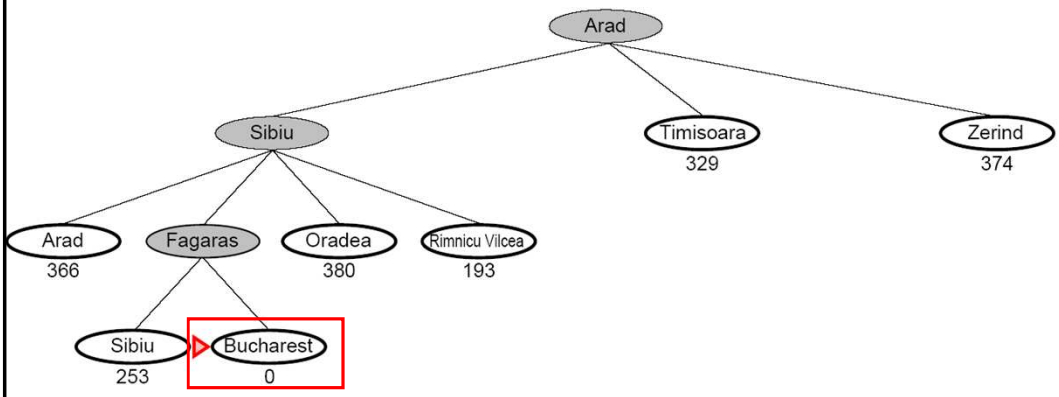
Greedy Best-First Search in Romania



Greedy Best-First Search in Romania



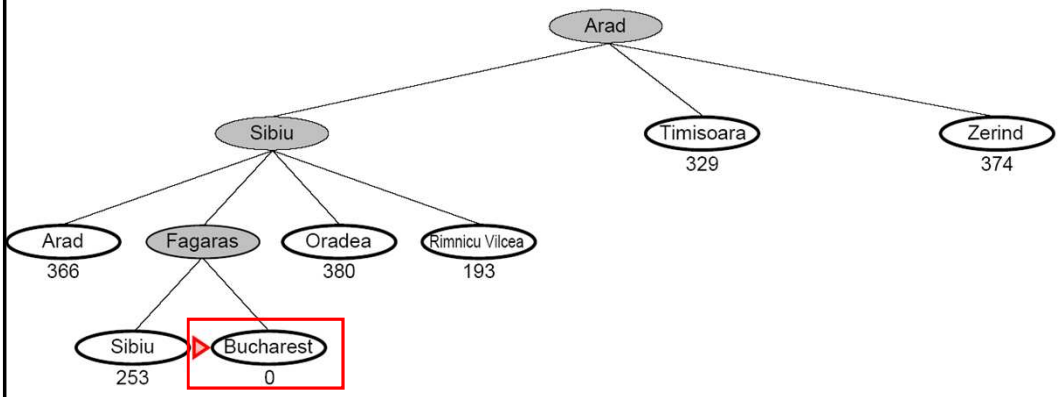
Greedy Best-First Search in Romania



Actual cost of final solution:

C = Arad→Sibiu + Sibiu→Fagaras + Fagaras→Bucharest
= 140 + 99 + 211
= 450

Greedy Best-First Search in Romania



Notes:

- Greedy Best-First + h_{SLD} found a solution without ever expanding a node that is not on the solution path (in this case) ➔ no backtracking etc.
  ➔ *very focused and efficient search*
- But: solution is not optimal …

Properties of Greedy Best-First Search

"Greedy" algorithm:

- Always looks for the next point to continue that *seems to be* closest to the goal (lowest $h(n)$)
- Greedy search is short-sighted, relies only on local information / guesses: nodes that seem to be good may turn out to be dead-ends in the longer term
- Similar to Breadth-First Search with a "focus towards the goal"

  ➔ same weaknesses
  (though may be much more efficient on average, given a good heuristic $h(n)$)

Properties of Greedy Best-First Search

Completeness:
- Not complete in general: can get stuck in loops;
  complete in finite spaces with repeated-state checking

Optimality:
- Not optimal; heuristic may lead search to suboptimal solution first

Time and space complexity (worst case):
- $O(b^m)$ (search may follow the wrong paths all the way)
- ***but*** a good heuristic may reduce that dramatically (at least on average)

> ***b*** ... branching factor (max. number of successors of any node)
> ***d*** ... depth of the shallowest goal node (i.e., goal with minimum length path)
> ***m*** ... maximum length of any path in the state space (may be infinite)

A* Search:
Minimising the Total Estimated Solution Cost

Idea:

- Avoid expanding paths that are already expensive
- Use more complex evaluation function $f(n)$ than greedy search:

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

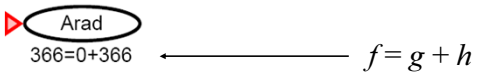
where
$g(n)$ = (actual) path cost from the start node to node n
$h(n)$ = estimated cost of the cheapest path from n to a goal

➔ $f(n)$ = estimated cost of the cheapest solution that goes through n

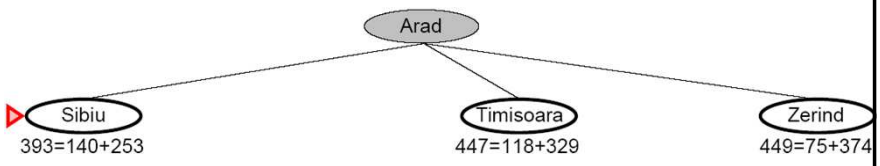
➔ Effect: a kind of combination of uniform-cost search and greedy search

We will prove that if the heuristic function $h(n)$ satisfies certain conditions (it is "admissible"), then A* search is both complete and optimal!

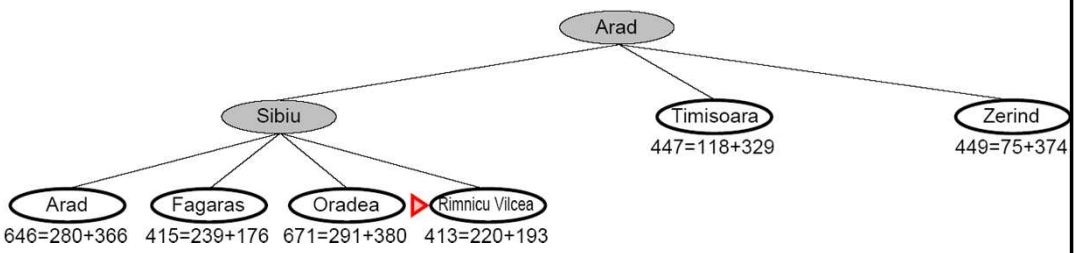
A* Search in Romania



A* Search in Romania

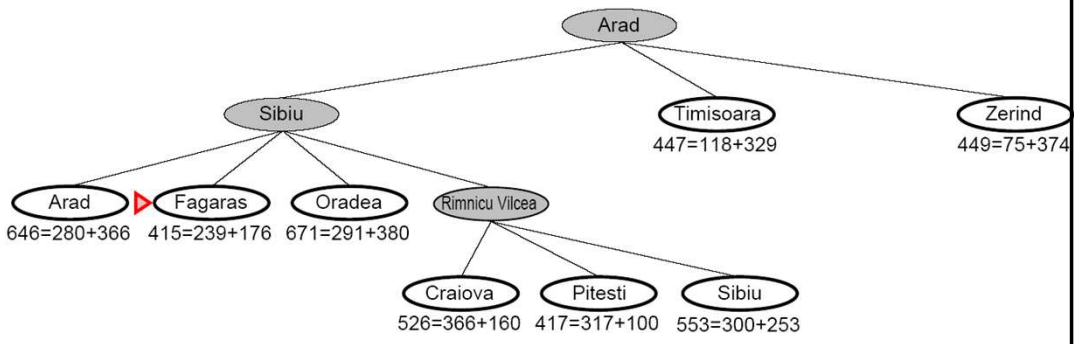


A* Search in Romania



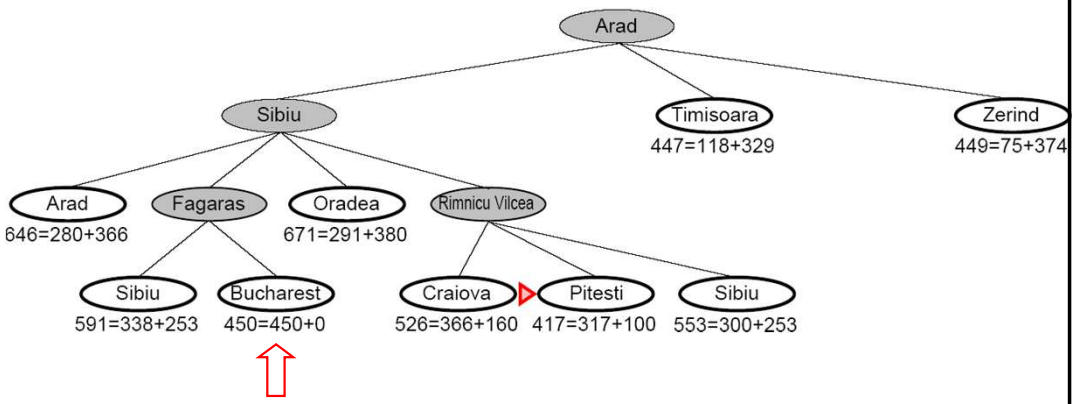
© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

A* Search in Romania

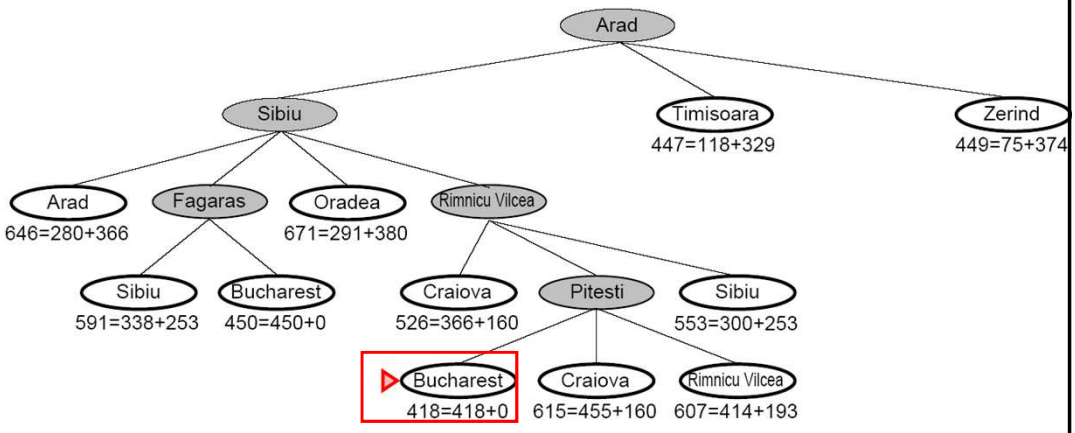


© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

A* Search in Romania



A* Search in Romania



Actual cost of final solution:

C = Arad→Sibiu + Sibiu→Rimnicu Vilcea + Rimnicu Vilcea→Pitesti + Pitesti→Bucharest
= 140 + 80 + 97 + 101
= 418

Compare this search tree to the tree produced by simple uniform-cost search ...

Admissible Heuristics and Optimality of A*

Definition:
A heuristic function $h(n)$ is **admissible** if, for any node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the **true** cost of the optimal path from n to a goal

Interpretation:

- An admissible heuristic function $h(n)$ never overestimates the actual cost from n to goal – it is "optimistic"
- If $h(n)$ is admissible, then $f(n)=g(n)+h(n)$ never overestimates the *true cost* of a solution through n
  (because $g(n)$ is the exact (known) cost of reaching n from the root)

Example:

- $h_{SLD}(n)$ is admissible (road connection cannot be shorter than straight line)

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Admissible Heuristics and Optimality of A*

Theorem:
If A* search is given a heuristic function $h(n)$ that is admissible, then A* search is **optimal**.

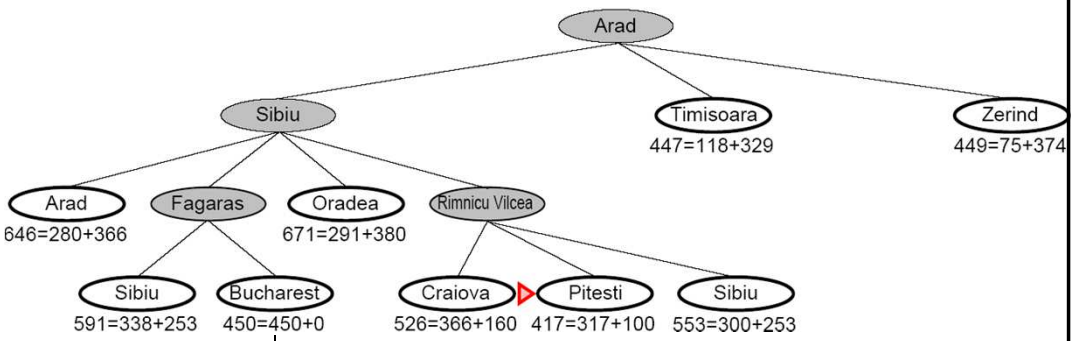
Intuitive reason: same as with Uniform-Cost Search (UCS):

If a non-optimal solution n_{sub} is already in fringe, any node n^* with the potential of leading to a better solution will have $f(n^*) < f(n_{sub}) = g(n_{sub})$ and will thus be sorted into the fringe *before* n_{sub} ➔ will eventually be expanded before n_{sub}.

Specifically, that also holds if n^* is a solution node.

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Idea of Proof: Consider Romania example



Bucharest first appears on fringe with f-cost 450

… but is not selected for expansion because its f-cost is higher than that of ▷Pitesti (and Timisoara, and …)

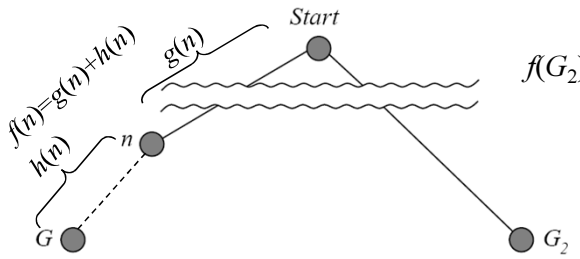
➔ Even though Bucharest (450) is a solution, there might be a solution through Pitesti whose cost is as low as 417!

➔ A* considers this alternative first

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Admissible Heuristics and Optimality of A*: Formal Proof

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



$$f(G_2) = g(G_2) + 0 \quad \text{(because } h(G_2) = 0\text{)}$$
$$> g(G) \quad \text{(because } G_2 \text{ is suboptimal)}$$
$$\geq f(n) \quad \text{(because } h \text{ is admissible)}$$

This holds for all nodes n on the path to G (including G itself)

➔ Since $f(n) < f(G_2)$, A* will expand **all** n on the path to G before it selects G_2

➔ That also holds for the goal node G

➔ The first goal node that A* will select will be an optimal one (q.e.d.)

Another consequence of this:

A* expands all n with $f(n) \leq f(G)$ before it expands/recognises the (optimal) goal G

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Properties of A* Search

Completeness:
- Complete if there are only finitely many nodes with $f(n) \leq f(G)$
- That is, complete if all step costs are $\geq \varepsilon$ and branching factor b is finite

Optimality:
- Optimal if $h(n)$ is admissible (see above)

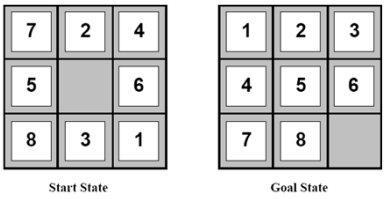
Time complexity (worst case):
- In the *worst case*, the number of nodes with $f(n) \leq f(G)$ can still be exponential
  ➔ time complexity is $O(b^m)$

Space complexity: Fringe can be exponential in tree depth ➔ $O(b^m)$

But again: a good heuristic can dramatically reduce *average* complexity!

> ***b*** ... branching factor (max. number of successors of any node)
> ***d*** ... depth of the shallowest goal node (i.e., goal with minimum length path)
> ***m*** ... maximum length of any path in the state space (may be infinite)

Designing Heuristic Functions

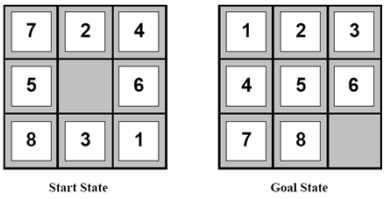


Example: 8-Puzzle

- One of the earliest heuristic search problems studied
- Solution cost = number of steps
- Average branching factor ≈ 3
- Average solution cost (length) for a randomly generated puzzle is ≈ 22

Problem Complexity:
- Breadth-first search to depth 22 would have to generate $3^{22} \approx 3.1 \times 10^{10}$ states
- By keeping track of repeated states, this can be reduced by a factor of 170,000: only 9!/2 = 181,440 *different* states are reachable (exercise ☺)
  ➔ 8-Puzzle is o.k.
- BUT: 15-Puzzle has 10^{13} different *reachable* states …
  ➔ need a good heuristic function

Example: Heuristic Functions for the 8/15-Puzzle



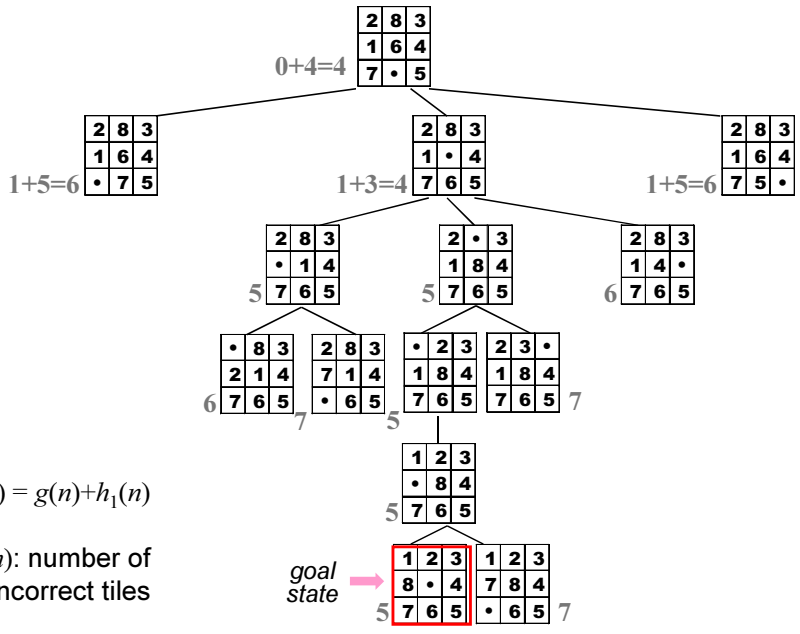
Two simple heuristics:

$h_1(n)$ = Number of tiles not in the "correct" place (according to desired goal config.)

- Example: $h_1(StartState) = 6$
- h_1 is admissible (any tile that is out of place must be moved at least once)

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Solving the 8-Puzzle with A* and Heuristic $h_1(n)$



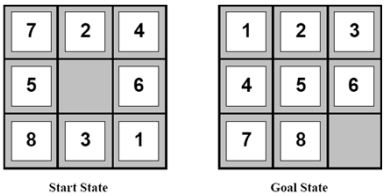
$f(n) = g(n)+h_1(n)$

$h_1(n)$: number of incorrect tiles

goal state

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Example: Heuristic Functions for the 8/15-Puzzle



Two simple heuristics:

$h_1(n)$ = Number of tiles not in the "correct" place (according to desired goal config.)

- Example: $h_1(StartState) = 6$
- h_1 is admissible (any tile that is out of place must be moved at least once)

$h_2(n)$ = Sum of the distances of the tiles from their goal positions; distance counted as sum of horizontal and vertical distance ("city block distance", "Manhattan distance")

- Example: $h_2(StartState) = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$
- h_2 is admissible (all a move can do is move one tile one step closer to goal)

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

The Effect of Heuristic Accuracy on Search Performance

Definition: Effective Branching Factor b^* :

If the total number of nodes generated by A* for a particular problem is N, and the solution depth is d, then b^* is the branching factor that a balanced, uniform tree of depth d would have to have in order to contain $N+1$ nodes; that is:

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

- Example: if A* finds a solution at depth 5 using 52 nodes, $b^* = 1.92$
- Effective branching factor is a measure of the "effectiveness" of a heuristic (i.e., how well it focuses the search)
- Good heuristics should have a b^* close to 1 (so that they can solve large problems)
- b^* cannot be computed analytically for a given heuristic, but we can measure it in experiments (see next slide), and in this way characterise the "power" of different heuristics

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Comparing h_1 and h_2 experimentally

- Generated 1,200 random 8-puzzle problems with solution lengths from 2 to 24 (100 problems with $d = 2$, 100 with $d = 4$, ...)
- Solved them with Iterative Deepening Search (IDS) and A* search with h_1 and h_2

	Search cost: nodes generated (avg.)			*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	-	39139	1641	-	1.48	1.26

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

The Effect of Heuristic Accuracy on Search Performance

Question: Is h_2 always better than h_1? and if so, why?

Definition: Dominance

A heuristic h_2 **dominates** a heuristic h_1 if, for any node n, $h_2(n) \geq h_1(n)$

Example: in 8/15-puzzle, h_2 dominates h_1

Theorem:

If h_1 and h_2 are both **admissible** heuristics, and if h_2 **dominates** h_1, then A* using h_2 will never expand more nodes than A* using h_1 (and every node expanded by h_2 is also expanded by h_1)

Intuition: Dominating heuristic is closer to true costs C^* (if both heuristics are admissible) ➔ more reliable guide in the search

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

The Effect of Heuristic Accuracy on Search Performance

Theorem:
If h_1 and h_2 are both admissible heuristics, and if h_2 dominates h_1, then A* using h_2 will never expand more nodes than A* using h_1 (and every node expanded by h_2 is also expanded by h_1)

Proof:
- A* expands every node with $f(n) < C^*$
- $f(n) = g(n) + h(n)$ => A* expands every node with $h(n) < C^* - g(n)$
- $h_1(n) \leq h_2(n)$
  ➔ Every node expanded with $h_2(n)$ will also be expanded by $h_1(n)$ (and h_1 might cause other nodes to be expanded too) *q.e.d*

Consequence:
- It is always better to use a heuristic with higher values, as long as it is admissible (i.e., does not overestimate the cost)
- In other words: it pays trying to invent heuristics that get as close as possible to the real cost (even if they are more expensive to compute)

Note: Given any admissible heuristics h_a and h_b, the function $h(n)=\max(h_a(n),h_b(n))$ is also admissible and dominates both h_a and h_b!

Improving on A* Search

Main problem:
- Exponential space complexity (as in breadth-first search or UCS)

Ways to alleviate the problem:
- Develop variants of A* that find *suboptimal* solutions *quickly*
  or
- Design heuristics that are more accurate, but not strictly admissible
  or
- Use more complex algorithms that sacrifice optimality / completeness: *memory-bounded heuristic search*
  (e.g., Iterative Deepening A* (IDA*),
  Recursive Best-First Search (RBFS),
  Memory-bounded A* (MA*),
  etc.)

Not treated in this class …

Summary (Part A)

- **Best-First Search =** TREE-SEARCH / GRAPH-SEARCH with heuristic strategy for selecting nodes for expansion that appear to be promising
- **Heuristic function** $h(n)$ estimates costs of a solution from n to goal; encodes knowledge about the problem
- **Greedy best-first search** expands node with lowest $h(n)$ – not optimal, but often efficient
- **A* search** expands nodes with lowest $f(n) = g(n) + h(n)$
- A* **is complete and optimal** (if $h(n)$ is **admissible**), but space complexity is often prohibitive
- Performance of heuristic search algorithms depends of **quality of heuristic function**; good heuristics can dramatically reduce search cost

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Part 3: Informed (Heuristic) Search

Note: This material is here for your information only.
It will not be part of the exam.

Part 3B: Local Search and Iterative Improvement

Univ.-Prof. Dr. Gerhard Widmer
Department of Computational Perception
Johannes Kepler University Linz

gerhard.widmer@jku.at
http://www.cp.jku.at/people/widmer

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

PART B: Local Search and Iterative Improvement Algorithms

Search algorithms studied so far …

- explore search spaces systematically
- keep one or more paths in memory
- record which solutions have been tried at each point along the path
- when goal is found: path to goal constitutes solution

But:

- in many problems the path to a solution is irrelevant
- the goal state itself is the solution
- examples: 8-queens problem; integrated circuit design; job-shop scheduling; telecom networks optimisation; transport routing; etc.

Local Search and Iterative Improvement Algorithms

Idea of "local search":

- reduce state space to set of "complete" configurations
- iterative improvement algorithm: keep a single "current" state and try to improve it step-wise
  (perform local search around some complete state in state space)

Advantages:

- need very little memory (usually only constant amount)
- often find reasonable (though not necessarily optimal) solutions in large state spaces for which systematic search is impossible
- local search algorithms are also useful for solving optimisation problems

Local search algorithms include methods inspired by …

- statistical physics (=> simulated annealing)
- evolutionary biology (=> genetic algorithms)

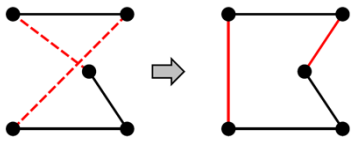
Example 1: The Travelling Salesman Problem

Problem:

- given a set of n cities connected by roads of given lengths, find a tour that visits each city exactly once and has minimum total length
- extremely hard problem: known to be NP-complete …

Iterative, local search algorithm:

- start with any complete tour
- in each step, perform pairwise exchanges between edges (roads)
- variants of this approach get within 1% of optimal very quickly with thousands of cities



Question:

- which pair of edges to exchange next?
  => again: need selection (search) strategy

Example 2: The n-Queens Problem

Problem:

- put n queens on an n x n board with no two queens attacking each other (i.e., with no two queens on the same row, columns, or diagonal)
- exponentially large search space

Iterative, local search algorithm:

- start with some random placement of the queens
  (or better: with one queen in each column, randomly placed in a row)
- in each step, move one queen to another place (in its column) so that the number of conflicts is reduced
  => "number of conflicts" used as a heuristic measure
  => number of conflicts = 0 => solution found
- almost always solves n-queens very quickly for very large n
  (e.g., n = 1 million!)

A Simple Search Strategy: Hill-Climbing Search (or Gradient Descent Search)

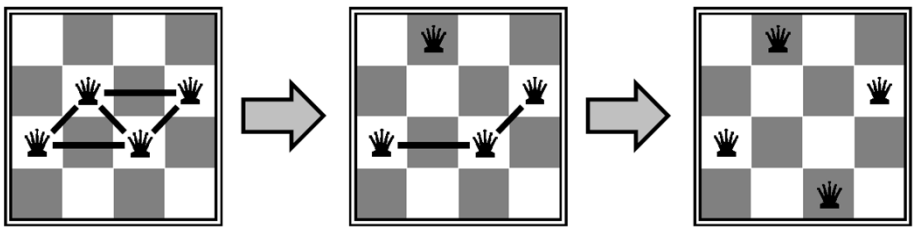
- "Like climbing Mount Everest in thick fog with amnesia" (Russell & Norvig)
- greedy algorithm
- again: central: evaluation function ("objective function") VALUE

```
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
  end
```


Example: The 4-Queens Problem

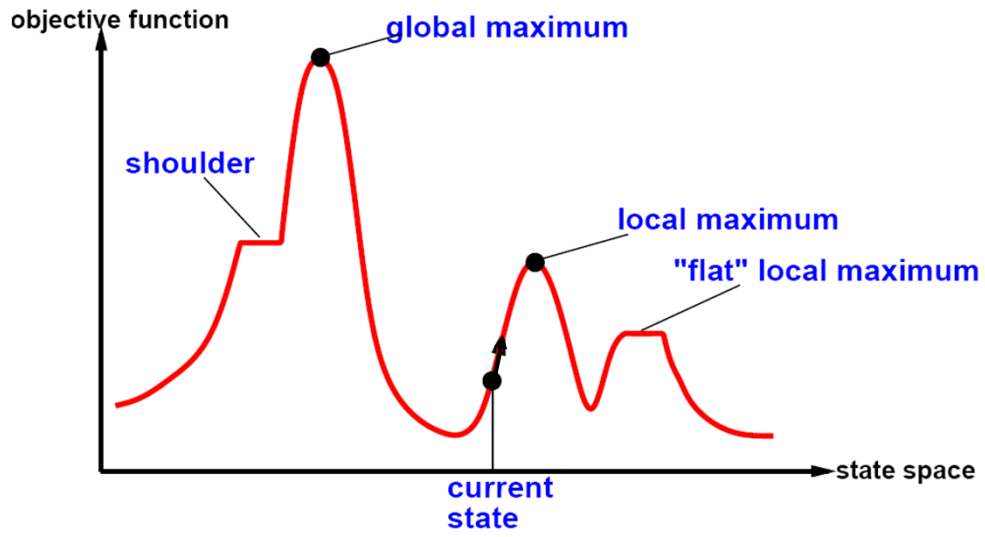
Formulation as a local search problem:

- **successor function:** given a board state, returns all possible board states that can be produced by moving one queen to a different place in its column
- **VALUE** function (to be minimised in this case): the number of conflicts between pairs of queens



h = 5 h = 2 h = 0

The State-Space Landscape

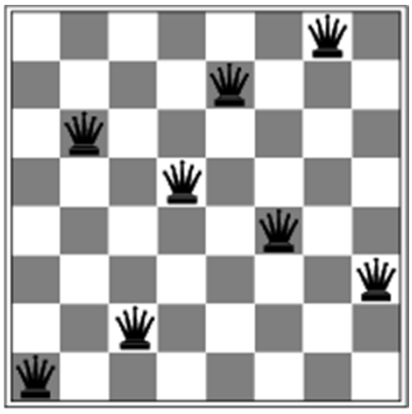


© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

The Incompleteness of Hill-Climbing: An Example

(a) An 8-queens state with heuristic cost estimate h = 17

(b) A local minimum in 8-queens state space; h = 1; all successors have higher cost



boxes: best moves

h of successor state if queen in this column were moved to this place

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Properties of Hill-Climbing

Completeness:
- not complete (may not be able to reach a solution; may get stuck in local optimum)

Optimality:
- not optimal (may get stuck in local optima, miss global ones)

Time complexity:
- hard to analyse analytically
- usually extremely fast (often reaches local optimum in a small number of steps)

Space complexity: $O(1)$ (constant)

Variants of Hill-Climbing

Goal:
- alleviate Hill-climbing's problems with incompleteness and non-optimality

Stochastic Hill-climbing:
- choose a random one of the possible uphill moves
- probability of selection may be made to depend on steepness of uphill move (i.e., the immediate improvement in VALUE)
- usually converges more slowly than steepest ascent (deterministic hill-climbing), but sometimes finds better solutions

Random-restart Hill-climbing:
- conduct a series of hill-climbing searches from randomly generated initial states
- asymptotically complete – if restarted a sufficient number of times, it will at some point find an optimal solution (e.g., because it generated a goal state as initial state …)
- *But:* will it know that this is a global optimum … ??

Hill-Climbing: Summary

- simple, space-efficient local search method that often finds relatively good solutions quickly
- success depends very much on shape of state-space landscape (e.g., NP-hard problems typically have an exponential number of local maxima!)
- hill-climbing is one of the most often used search algorithms in thousands of applications

Simulated Annealing Search

Idea:

- a hill-climbing algorithm that never makes “downhill” moves towards states with lower values is guaranteed to be incomplete
- in contrast: a “random walk” (choosing successor purely randomly) is complete, but extremely inefficient

Goal:

- try to combine aspects of the two
- achieve some level of efficiency and completeness

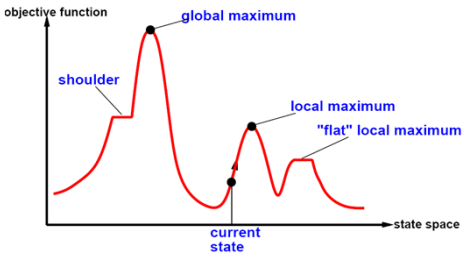
An Excursion to Metallurgy and Statistical Physics

"The name and inspiration come from annealing in metallurgy, a technique involving heating and controlled cooling of a material to increase the size of its crystals and reduce their defects. The heat causes the atoms to become unstuck from their initial positions (a local minimum of the internal energy) and wander randomly through states of higher energy; the slow cooling gives them more chances of finding configurations with lower internal energy than the initial one." (wikipedia.org)

Simulated annealing in search and optimisation problems:

"In the simulated annealing (SA) method, each point s of the search space is compared to a state of some physical system, and the function E(s) to be minimized is interpreted as the internal energy of the system in that state. Therefore the goal is to bring the system, from an arbitrary initial state, to a state with the minimum possible energy." (wikipedia.org)

Simulated Annealing: The Intuition



Assume a search problem with an evaluation function to be *minimised*:

- **hill-climbing / gradient-descent** = dropping a ball at a random place and letting it roll => ball will end up in local minimum
- **Simulated Annealing:**
  - drop ball at a random place
  - while it is rolling: shake the landscape randomly
    => ball may be bounced out of local minima
  - problem: how hard to shake? (don't want to bounce out of global minimum)
  - basic idea: shake hard first, gradually reduce intensity of shaking
- in simulated annealing: "temperature" ≡ degree of randomness in selection

Simulated Annealing: The General Algorithm

```
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 prob. 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[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```


Simulated Annealing

Properties:

- probability of accepting a successor state that is worse than the current one
  - decreases exponentially with the "badness" of the move (ΔE)
  - also decreases as the temperature goes down

➔ "bad" moves are more probable at the beginning, when temperature high
➔ later, when search has found good part of search space: random moves with low probability

Provable fact (via Boltzmann Distribution):

- if the schedule lowers temperature T slowly enough, the algorithm will find a global optimum with probability approaching 1

Application:

- Simulated Annealing is widely used in large optimisation problems (e.g., VLSI layout design, airline scheduling, etc.)

Local Beam Search

Idea:

- keep k alternative current states in memory instead of just 1

Algorithm:

- begin with k randomly generated complete states
- at each step, generate all successors of all k states
- if any one of these is a goal: stop
- otherwise: select k best successors from complete list and repeat
- terminology: *"beam"* = list of current k states; k = *"beam width"*

Effect:

- searches different parts of state space in parallel
- if one or several states produce particularly promising successors, these will all be kept in the "beam"
  => search continues mainly in that promising part of state space

Stochastic Local Beam Search

Problem with local beam search:

- can suffer from lack of diversity among the k states
- states quickly become concentrated in a small region of state space
  => *"premature convergence"*
- high risk of missing global optimum

Stochastic local beam search:

- analogous to stochastic hill-climbing
- instead of choosing best k from pool of candidate successors, choose successors stochastically (i.e., with probability of a node being chosen being proportional to the evaluation of the node)

  => analogy to natural selection …

Genetic Algorithms (GAs)

Basic idea:

- variant of stochastic beam search
- successor states are generated by *combining two parent states* (rather than modifying a single state), using a *crossover operator*
- again: selection of successors with probability depending on evaluation function

Analogy to natural evolution:

- "sexual reproduction": two states ("parents") combine their ("genetic") information into one or more successor nodes ("children")
- "survival of the fittest": successor states with higher evaluation values ("fitness") have higher probability of being kept ("surviving") (Alternative: states with higher evaluation get higher probability of being selected to produce children)
- GAs often also contain an additional element of chance: random changes to successor states ("random mutation")

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Genetic Algorithms: The Basic Schema

Basic algorithm schema:

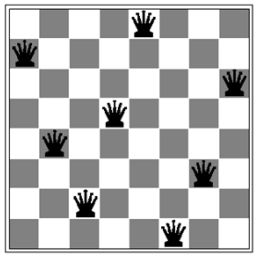
- A successor state is generated by combining two parent states
- Start with k randomly generated states ("population")
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluate all individuals in population via heuristic evaluation function ("fitness function"). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation (where selection is governed by probabilities that are made to be proportional to measured fitness)

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)

Genetic Algorithms on the *n*-Queens Problem

1. Representation Scheme:

- represent n-queens state as a string
- example: state = list of 8 integers $\in$ {1..8}, specifying the row number of each queen (each in one column, from left to right)



25741863

2. Fitness function:

- should assign higher values to better states
- example: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)

Genetic Algorithms on the *n*-Queens Problem

3. Selection strategy for reproduction:

- example: select individuals for reproduction with probability directly proportional to their fitness ("fitness-proportionate selection"):
  24/(24+23+20+11) = 31% (= probability for 1st state to be chosen as parent)
  23/(24+23+20+11) = 29% etc

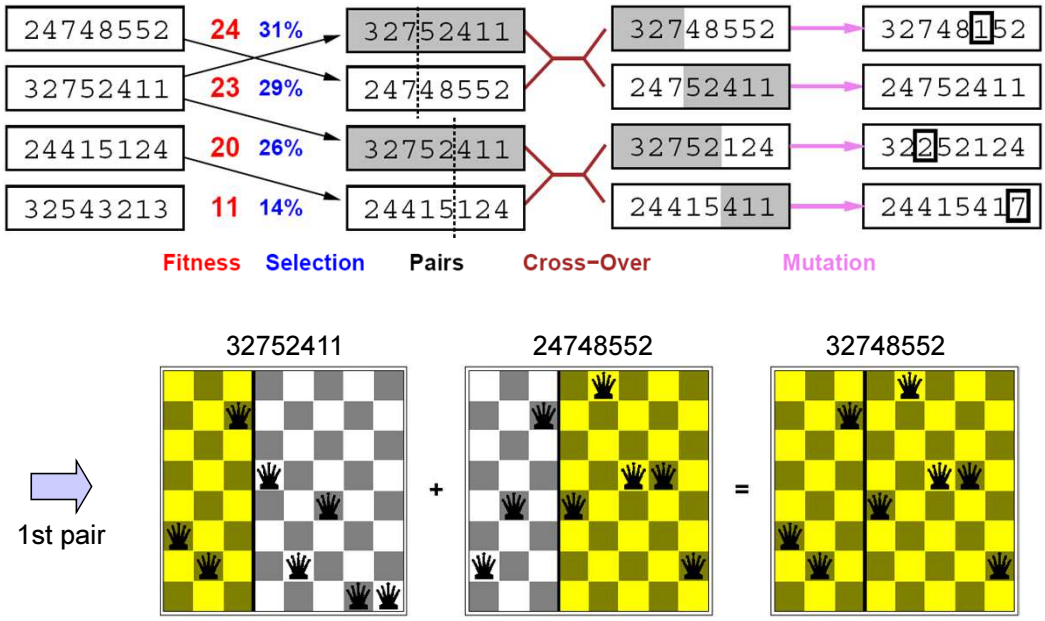
4. Recombination of parent information to produce successor states:

- example: single-point crossover (see below)

5. Random Mutation Operator:

- example: in a successor, replace a digit with a random digit with probability 0.001

Genetic Algorithms on the n-Queens Problem



Genetic Algorithms (GAs)

Properties:

- like stochastic beam search, GAs combine an "uphill tendency" with random exploration of the state space
- crossover can combine useful pieces of information from parents (if substrings are meaningful components)
- GAs are widely used in optimisation problems
- not clear whether there is any systematic advantage over stochastic beam search

Summary (Part B)

- **Local search methods** operate on complete-state problem formulations; return only solutions, but not paths;
- Local search is usually **extremely space-efficient**
- Most common local search method: **hill-climbing**;
  major problems: short-sightedness, inability to escape from local optima
- **More elaborate local search methods:**
  stochastic hill-climbing; simulated annealing; (stochastic) local beam search; genetic algorithms
- **Genetic Algorithms (GAs):** a kind of stochastic local beam search with large population of states and mutation and crossover operators for combining states into successors; weak analogy to natural evolution

© 2015 Gerhard Widmer (based in part on materials by Stuart J. Russell)