INFORMED SEARCH (HEURISTICS), EXPLORATION

In which we see how information about the state space can prevent algorithms from blundering about in the dark.

Outline

- Best-first search
  - Greedy best-first search
  - A^* search
- Heuristics
  - Admissibility
  - Consistency/ Monotonicity
  - Quality and Dominance
  - Invention
    - Relaxed Problem
    - Cost of Subproblem
- Memory-bounded search
  - Iterative-deepening A* (IDA*)
  - Recursive best-first search(RBFS)
- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Genetic Algorithms

Review: Tree search and Graph search

```
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](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    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](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

A strategy is defined by picking *the order of node expansion*

Best-first search

- Idea: use an evaluation function *f(n)* for each node
  - estimate of "desirability", i.e. measures distance to the goal

  → Expand most desirable unexpanded node

- Implementation:

  Order the nodes in fringe in decreasing order of desirability

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

A heuristic function

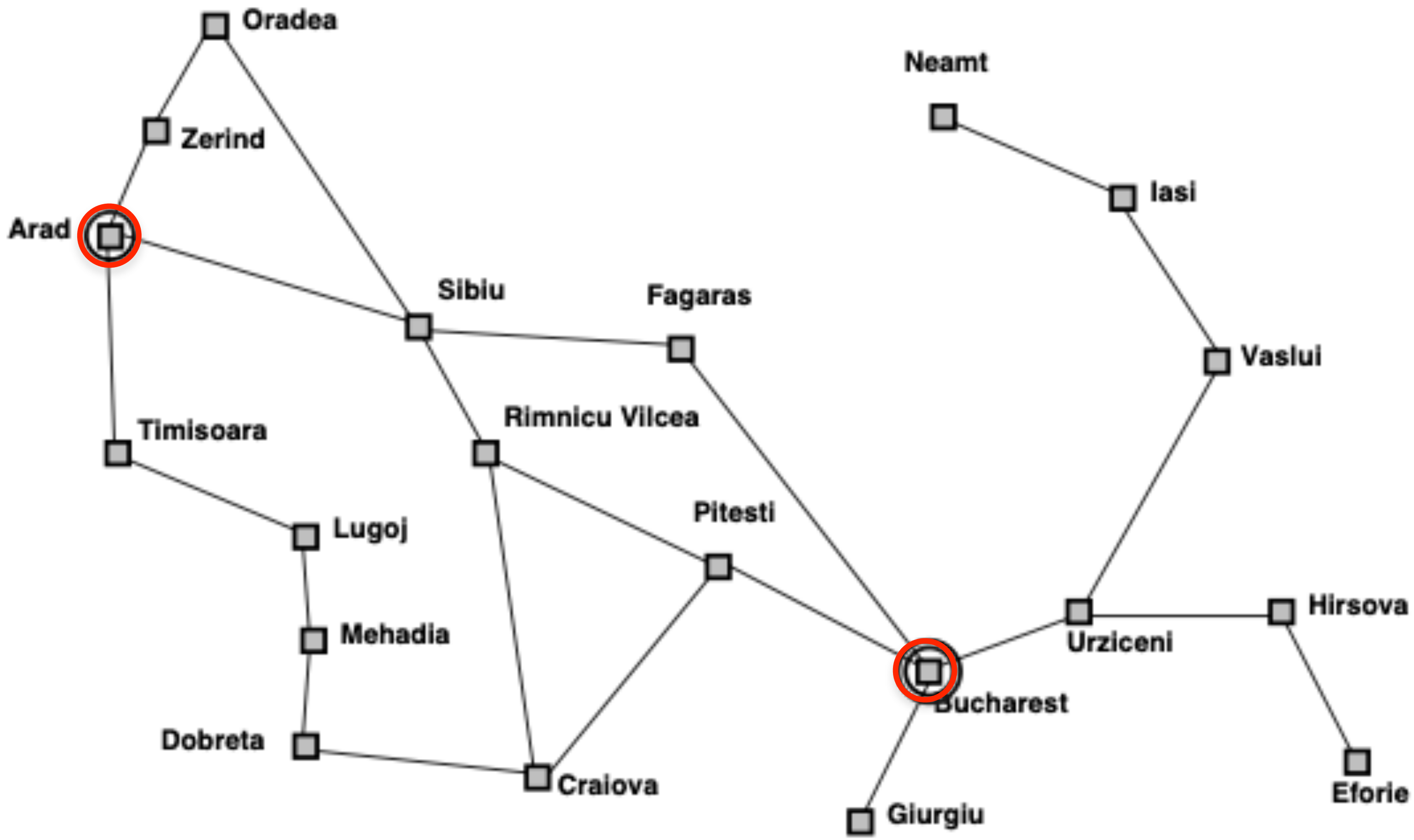
- [dictionary] *"A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood."*
  - $h(n)$ = estimated cost of the cheapest path from node *n* to goal node.
  - If *n* is goal then $h(n)=0$
  - Its value is **independent of the current search tree**; it depends *only on the state(n)* and the *goal test.*

More information later…

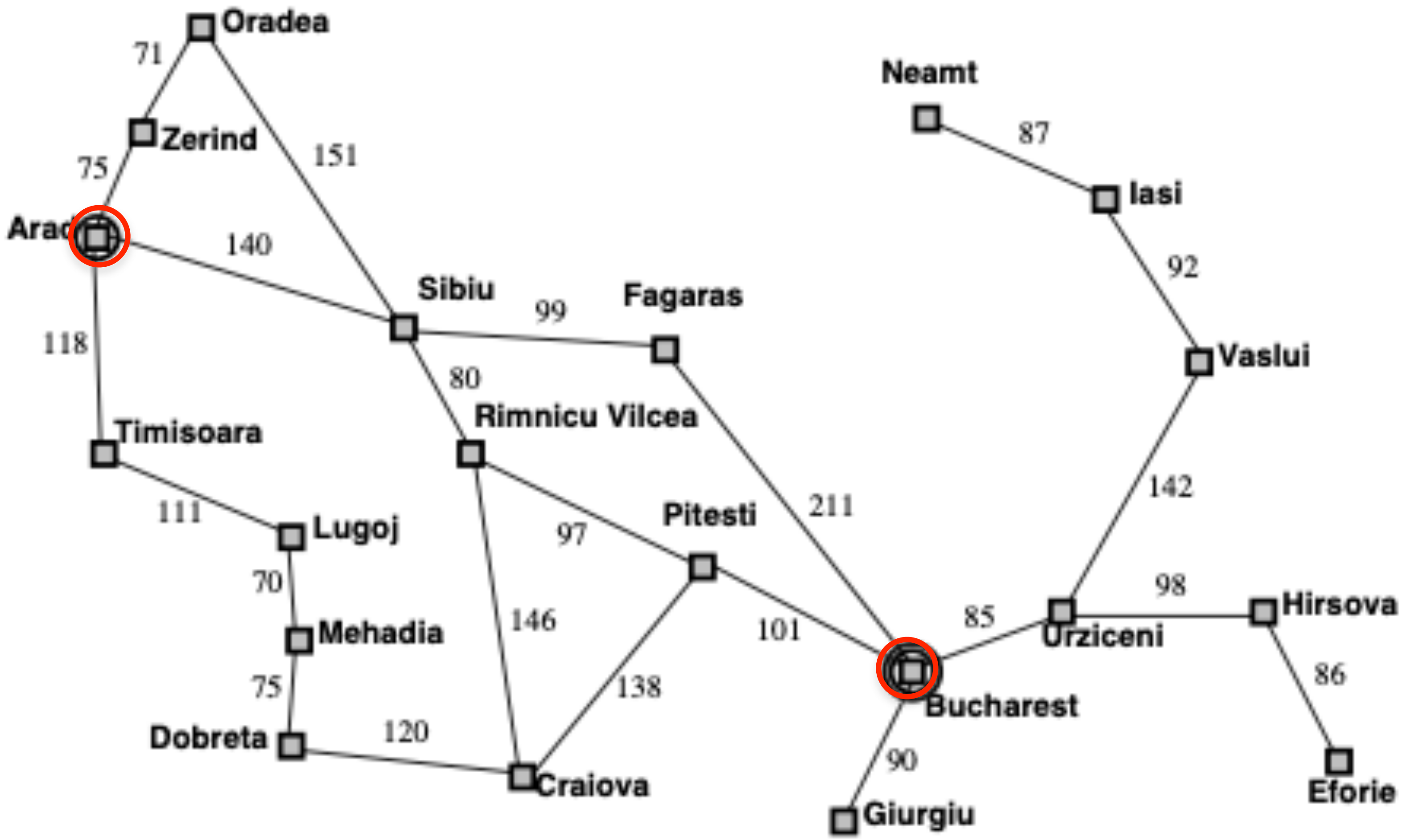
Greedy best-first search

- Evaluation function $f(n) = h(n)$ (heuristic) = estimate of cost from *n* to *goal*
- e.g., $h_{SLD}(n)$ = straight-line distance from *n* to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal

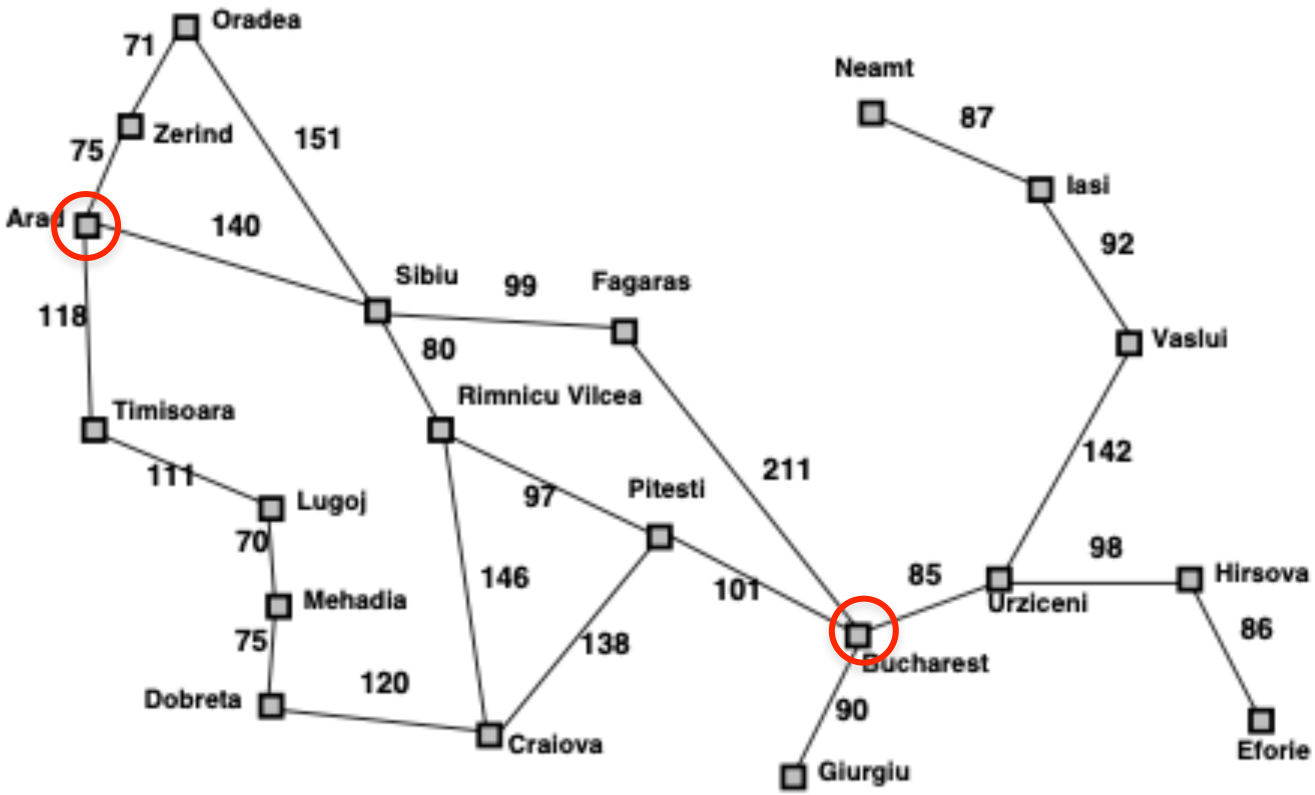
Routing Problem:



Routing Problem: Romania with step costs in km



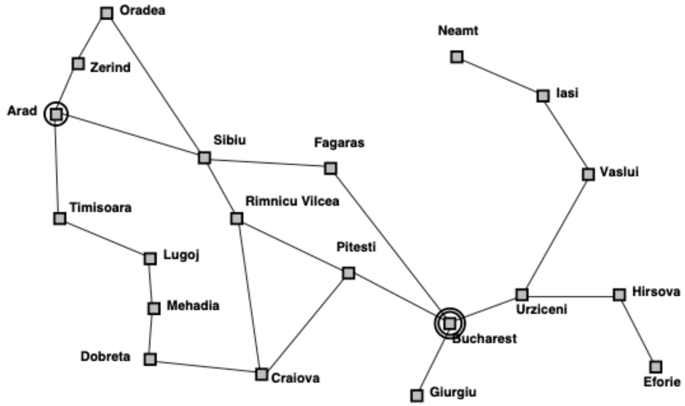
Routing Problem: Romania with step costs in km



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

h_{SLD}=straight-line distance heuristic.
h_{SLD} can **NOT** be computed from the problem description itself

Greedy best-first search example



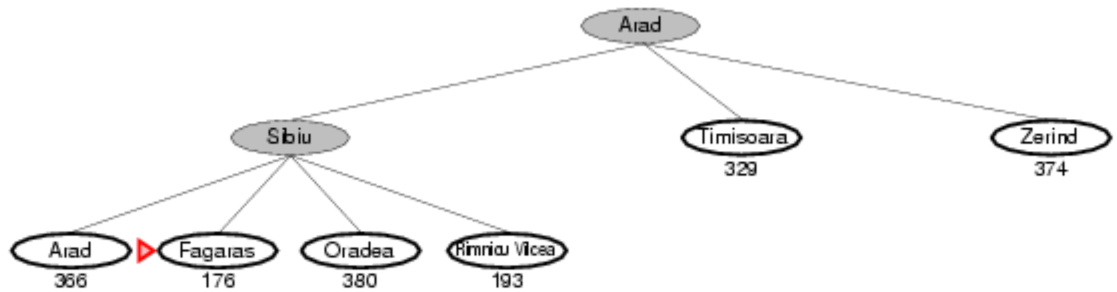
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example

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	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

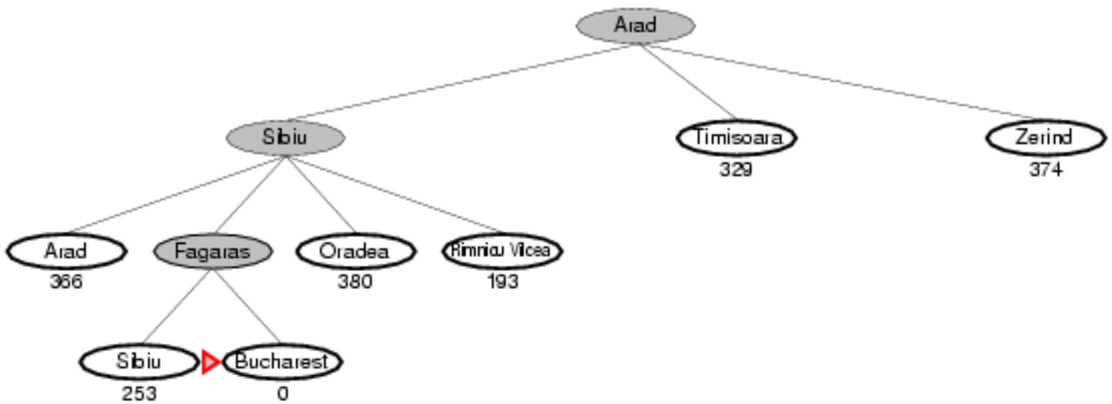
Greedy best-first search example



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	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

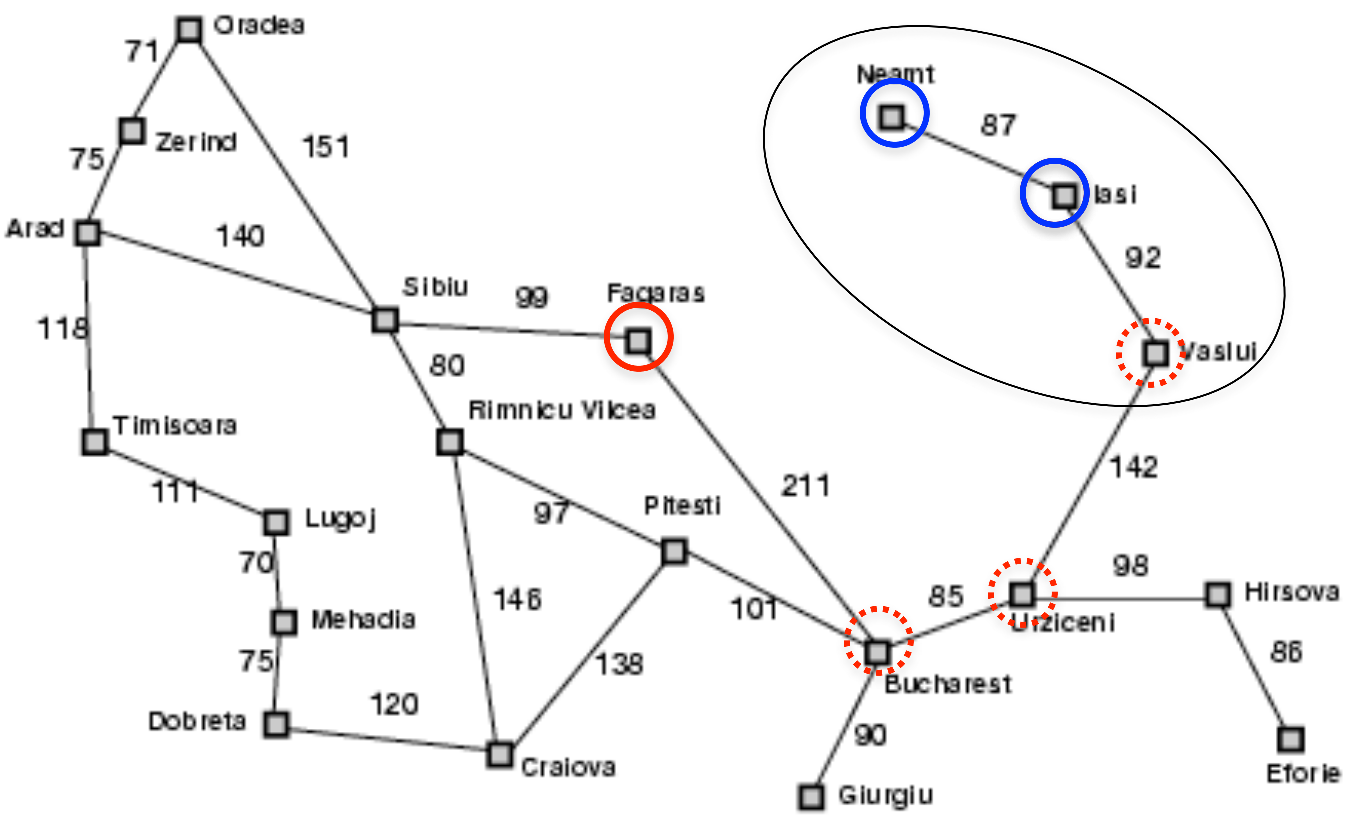
Greedy best-first search example



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	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Properties of greedy best-first search

Complete?



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	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Properties of greedy best-first search

- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- Time?

Properties of greedy best-first search

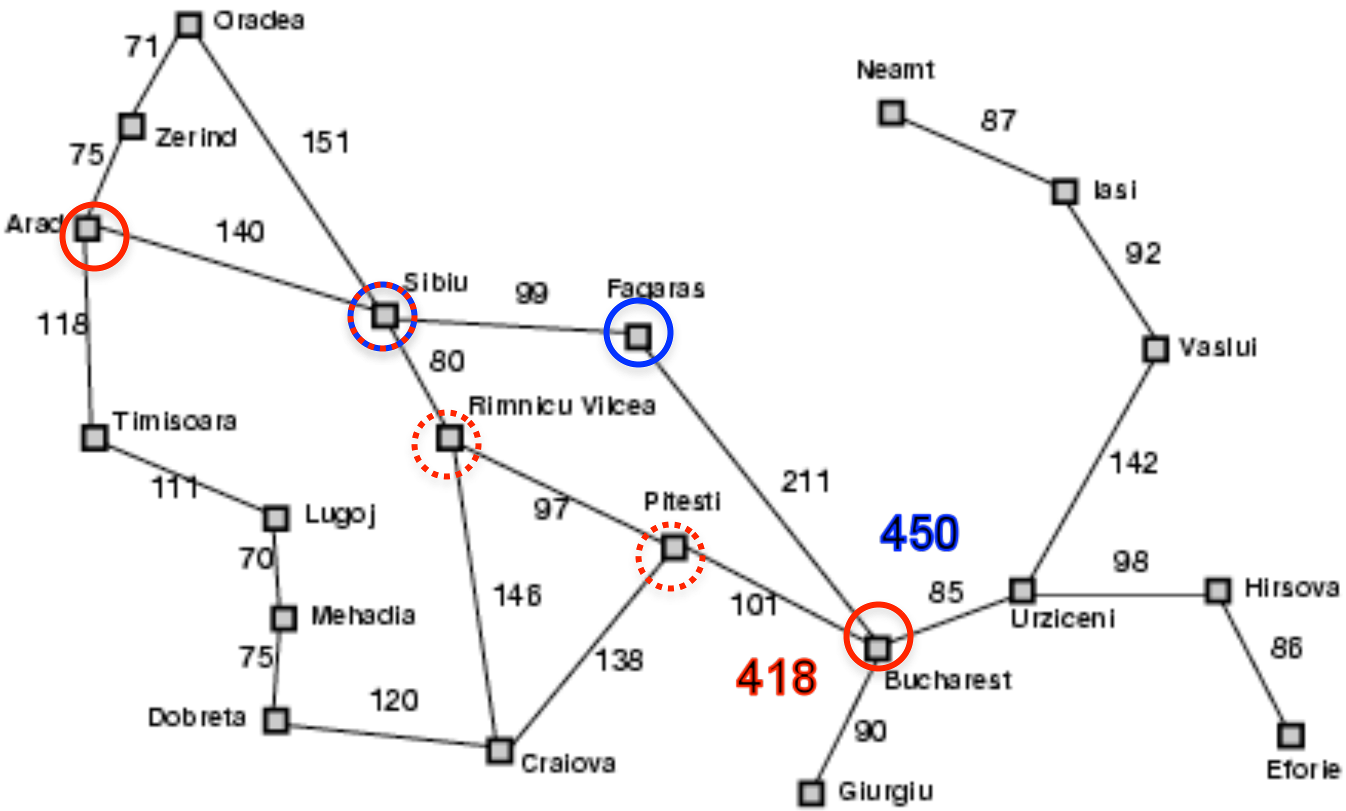
- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space?

Properties of greedy best-first search

- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$, keeps all nodes in memory
- Optimal?

Properties of greedy best-first search

Optimal?



Properties of greedy best-first search

- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$, keeps all nodes in memory
- Optimal? No

Minimizing total path cost: A* search

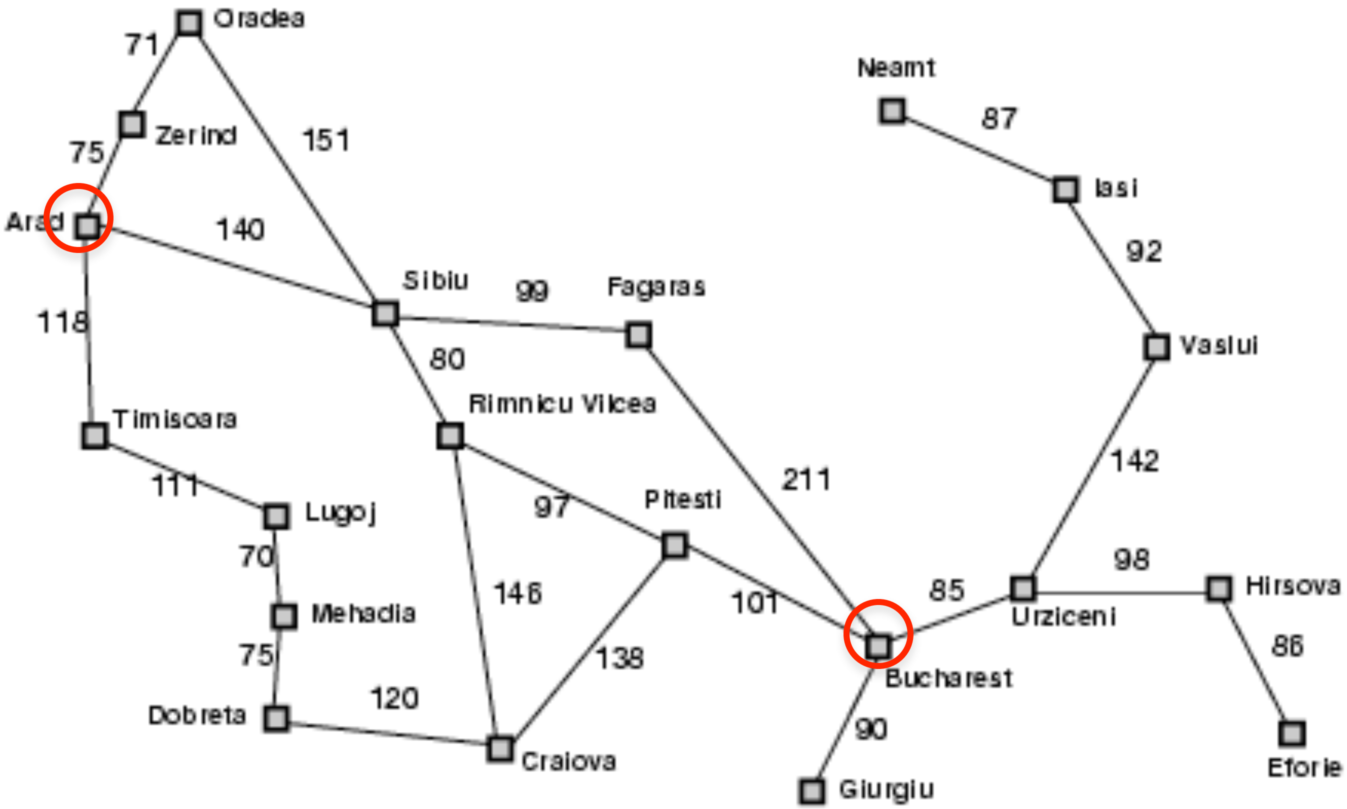
- **Greedy search** minimizes the estimated cost to the goal *h(n),* and thereby cuts the search cost considerably.
  - But neither optimal nor complete
- **Uniform-cost** search minimizes the cost of the path so far *g(n)*
  - It is optimal and complete
  - But can be very inefficient
- How about **combining** these two strategies to get advantages of both?

  → A* algorithm (due to Nils Nilsson for *Shaky* the robot)

A* search

- *Best-known form of best-first search*.
- Idea: avoid expanding paths that are already expensive
- Combines the two evaluation functions (of UCS and GBFS) by summing them up
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = cost (so far) from start node to reach n
  - $h(n)$ = estimated cost to get from n to goal
  - $f(n)$ = estimated total cost of cheapest path solution through n to goal

Routing Problem: Romania with step costs in km

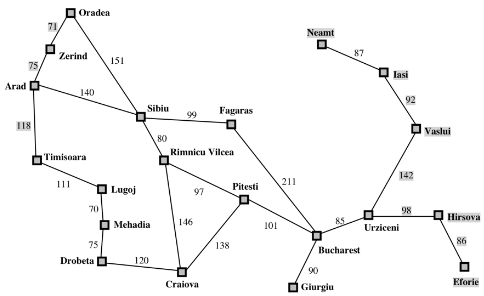


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h_{SLD}=straight-line distance heuristic.

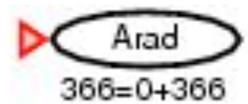
h_{SLD} can **NOT** be computed from the problem description itself

A* search example

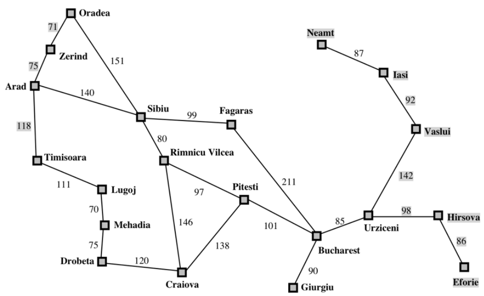


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

(a) The initial state

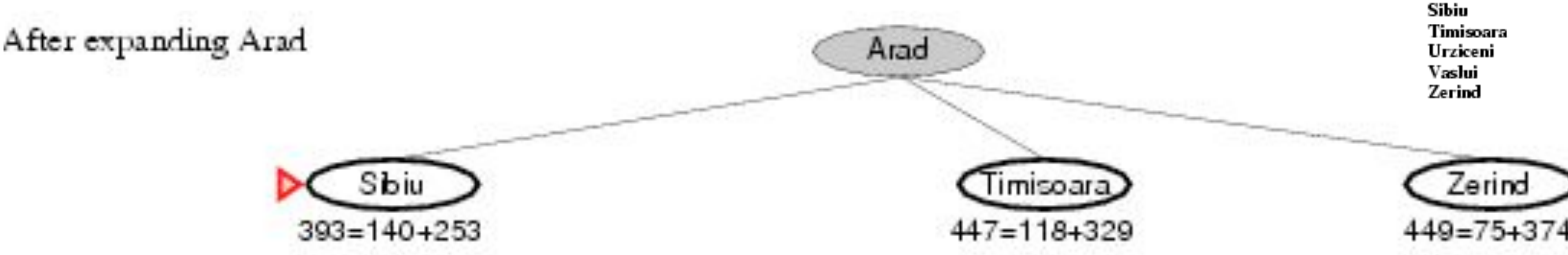


- Find Bucharest starting at Arad
  - f(Arad) = g(Arad, Arad)+h(Arad)=0+366=366

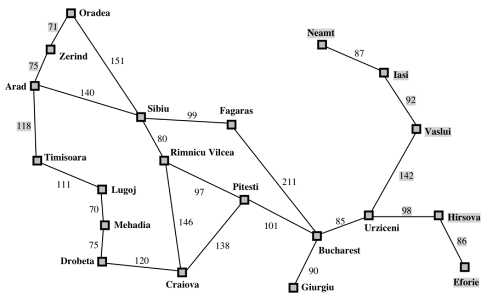


A* search example

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	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



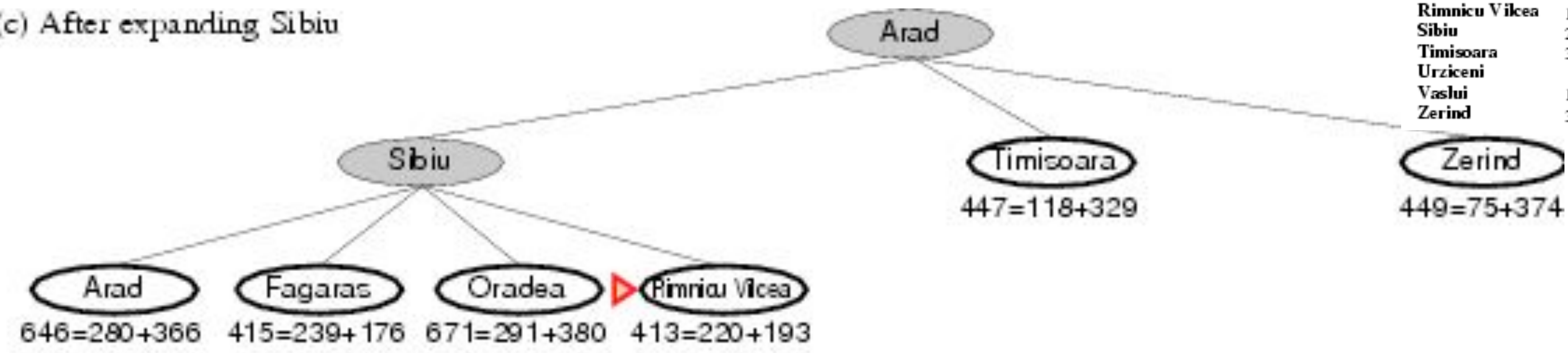
- Expand Arrad and determine *f(n)* for each node
  - f(Sibiu)=g(Arad,Sibiu)+h(Sibiu)=140+253=393
  - f(Timisoara)=g(Arad,Timisoara)+h(Timisoara)=118+329=447
  - f(Zerind)=g(Arad,Zerind)+h(Zerind)=75+374=449
- Best choice is Sibiu



A* search example

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	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

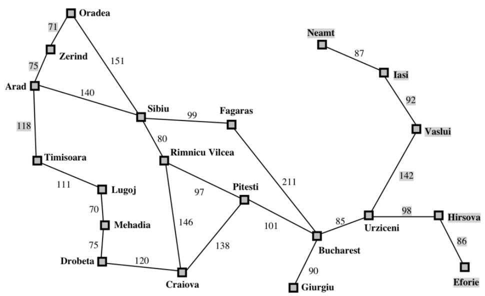
(c) After expanding Sibiu



- Expand Sibiu and determine *f(n)* for each node
  - f(Arad)=g(Sibiu,Arad)+h(Arad)=280+366=646
  - f(Fagaras)=g(Sibiu,Fagaras)+h(Fagaras)=239+179=415
  - f(Oradea)=g(Sibiu,Oradea)+h(Oradea)=291+380=671
  - f(Rimnicu Vilcea)=g(Sibiu,Rimnicu Vilcea)+ h(Rimnicu Vilcea)=220+192=413

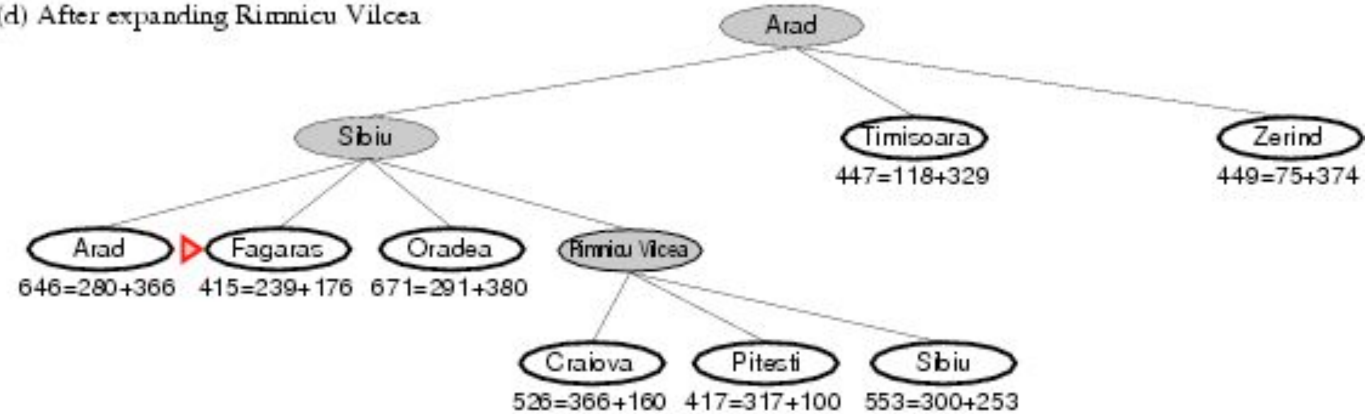
- Best choice is Rimnicu Vilcea

A* search example

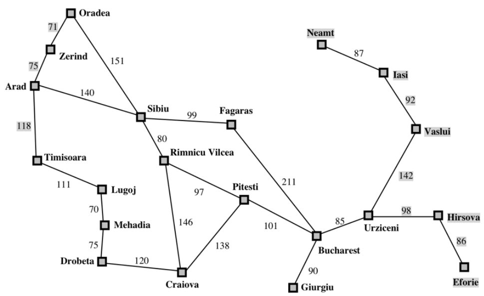


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

(d) After expanding Rimnicu Vilcea



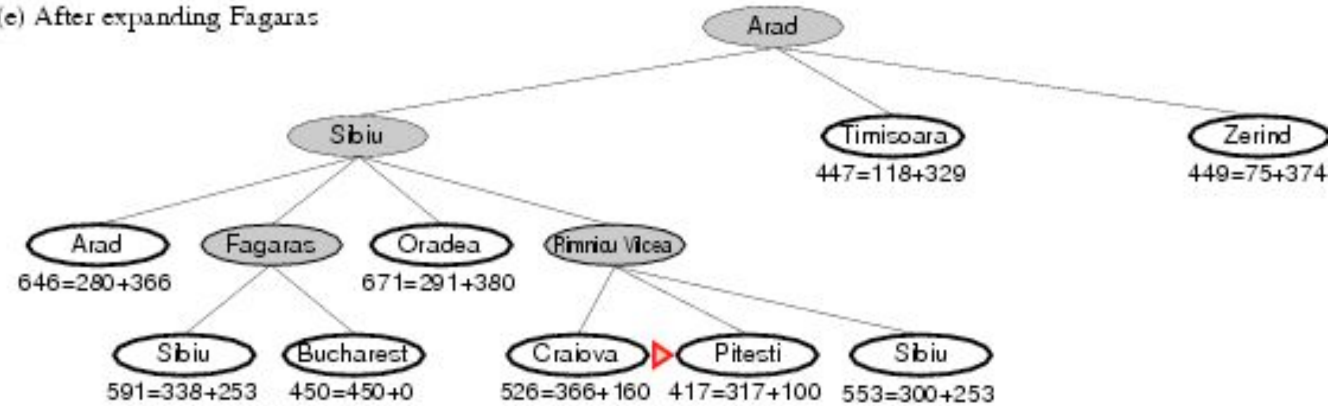
- Expand Rimnicu Vilcea and determine *f(n)* for each node
  - f(Craiova)=g(Rimnicu Vilcea, Craiova)+h(Craiova)=360+160=526
  - f(Pitesti)=g(Rimnicu Vilcea, Pitesti)+h(Pitesti)=317+100=417
  - f(Sibiu)=g(Rimnicu Vilcea,Sibiu)+h(Sibiu)=300+253=553
- Best choice is Fagaras



A* search example

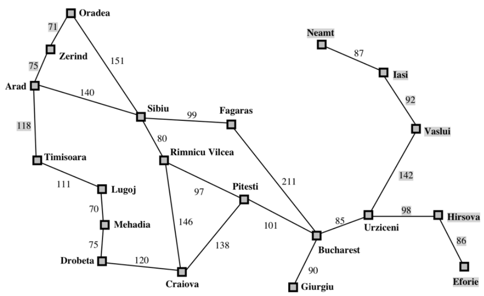
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

(e) After expanding Fagaras



- Expand Fagaras and determine *f(n)* for each node
  - f(Sibiu)=g(Fagaras, Sibiu)+h(Sibiu)=338+253=591
  - *f(Bucharest)=g(Fagaras,Bucharest)+h(Bucharest)=450+0=450*
- Best choice is Pitesti !!!

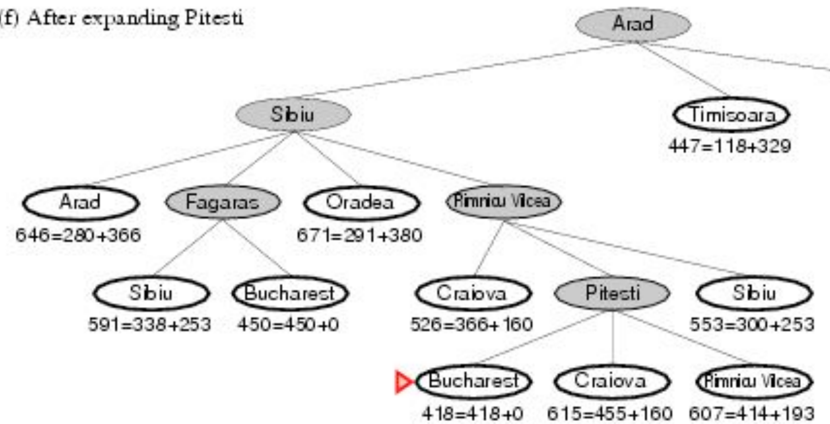
A* search example



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	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

(f) After expanding Pitesti



- Expand Pitesti and determine *f(n)* for each node
  - *f(Bucharest)=g(Pitesti,Bucharest)+h(Bucharest)=418+0=418*
- Best choice is Bucharest !!!
  - Optimal solution (only if *h(n)* is admissible)
- Note values along optimal path !!

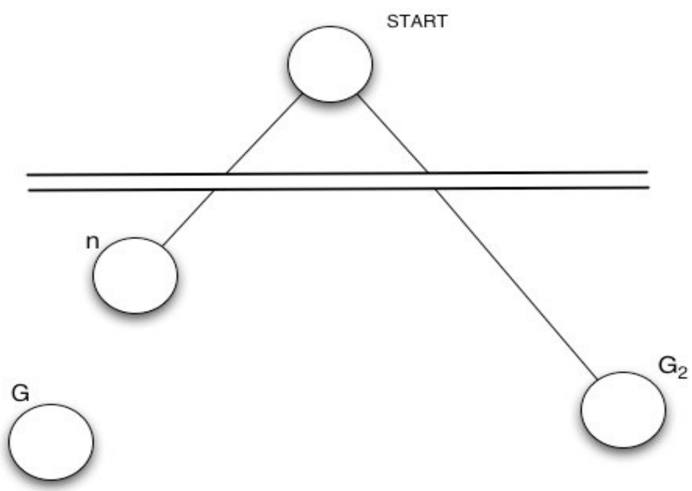
Admissible heuristics

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
  - Formally, a heuristic $h(n)$ is admissible if for every node n:
    - $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n.
    - $h(G) = 0$ for any goal G.
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- This optimism transfers to the f function:

  If h is admissible, since $g(n)$ is the exact cost to reach n, $f(n)$ never overestimates the actual cost of the best solution through n.

- Theorem: If $h(n)$ is admissible, A^* using `TREE-SEARCH` is optimal

Optimality of A*(standard proof)



- Suppose suboptimal goal G_2 in the queue.
- Let *n* be an unexpanded node on a shortest path to optimal goal *G*.

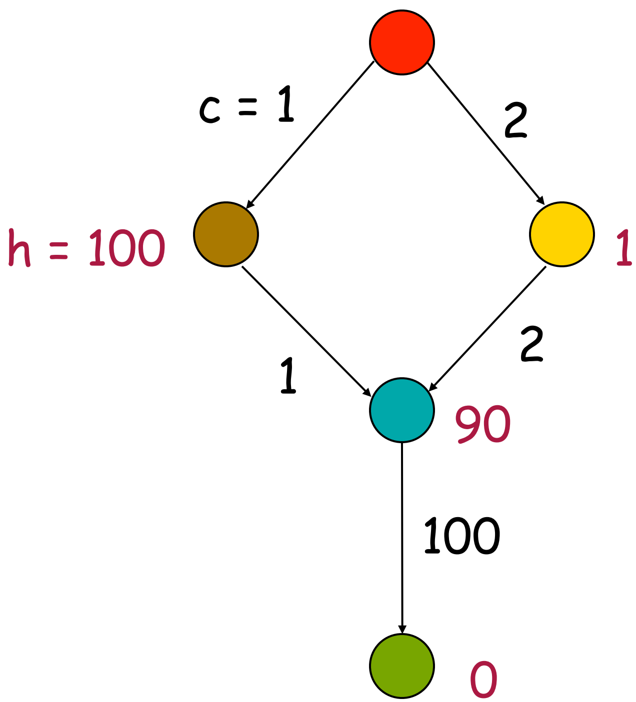
$f(G_2)$	$= g(G_2)$	since $h(G_2)=0$
	$> g(G)$	since G_2 is suboptimal
	$>= f(n)$	since *h* is admissible (i.e. g(G) >= f(n) = g(n) + h(n))

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

BUT … with `GRAPH-SEARCH`

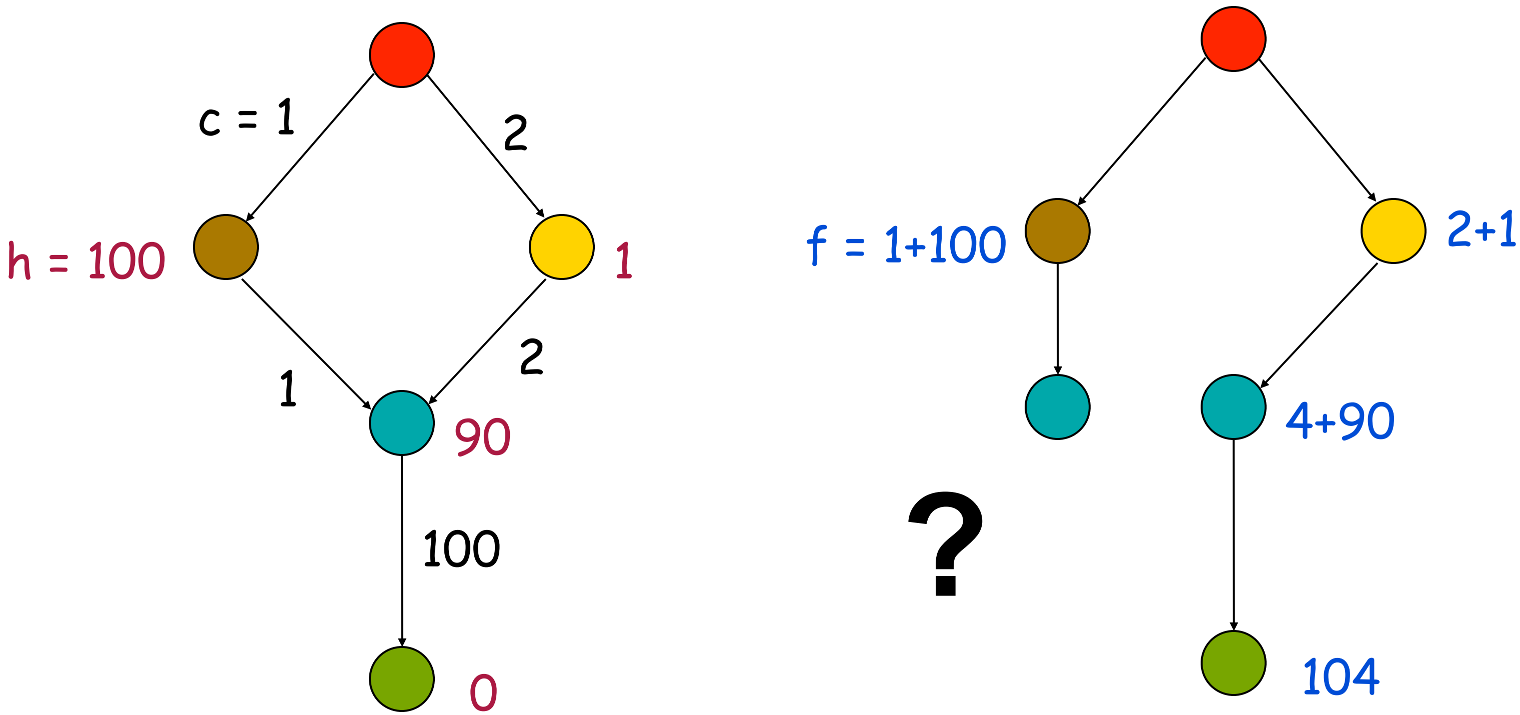
- Previous proof breaks down:
  - because `GRAPH-SEARCH` can discard the optimal path to a repeated state if it is not the first one generated.

What to do with revisited states?



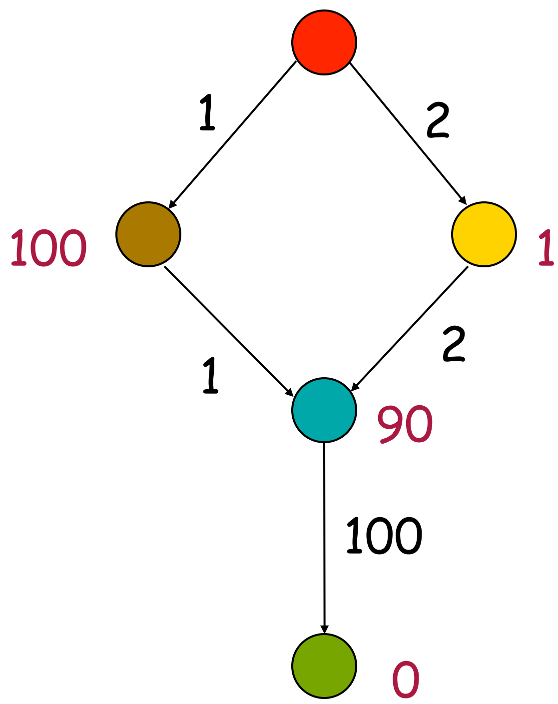
The heuristic h is clearly admissible

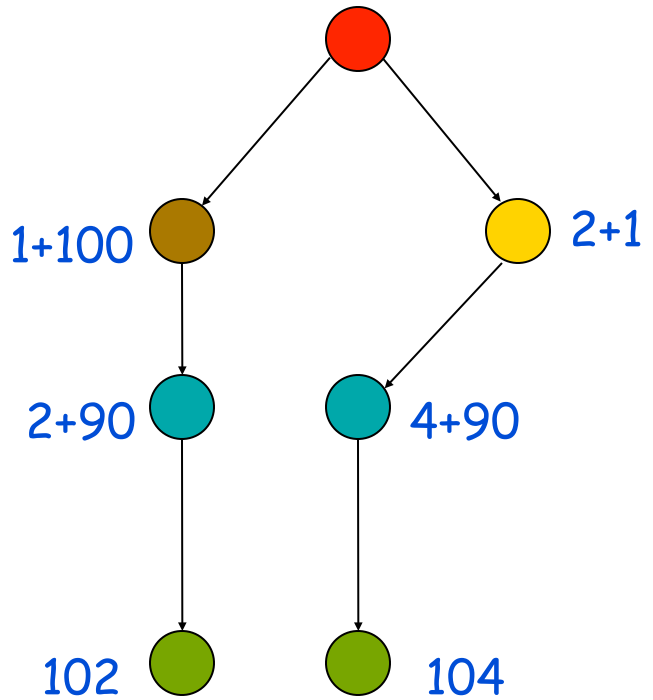
What to do with revisited states?



If we discard this new node, then the search algorithm expands the goal node next and returns a non-optimal solution

What to do with revisited states?

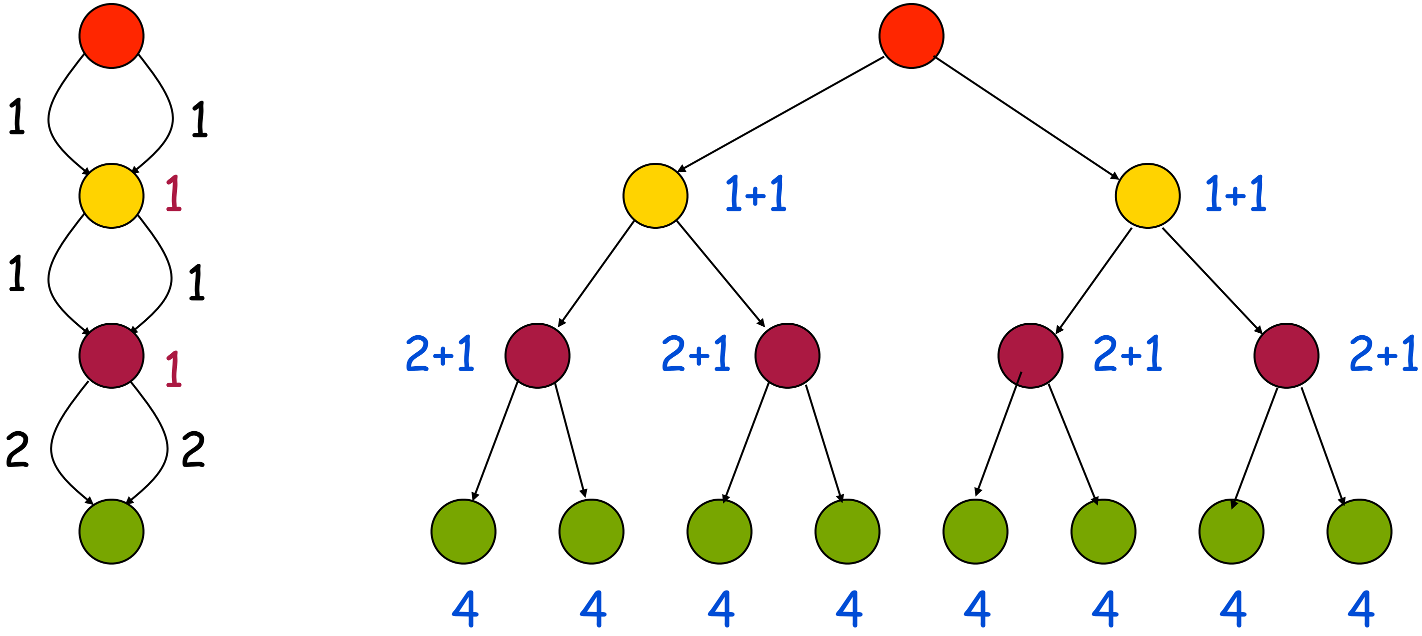




Instead, if we do not discard nodes of revisiting states, the search terminates with an optimal solution

But ...

If we do not discard nodes of revisiting states, the size of the search tree can be exponential in the number of visited states



- It is not harmful to discard a node revisiting a state if the cost of the new path to this state is ≥ cost of the previous path
  [so, in particular, one can discard a node if it re-visits a state already visited by one of its ancestors]

- A* remains optimal, but states can still be re-visited multiple times
  [the size of the search tree can still be exponential in the number of visited states]

- Fortunately, for a large family of admissible heuristics – consistent heuristics – there is a much more efficient way to handle revisited states

Consistency for Optimality of with `GRAPH-SEARCH`

- Proof of Optimality of A* breaks down with `GRAPH-SEARCH` because it can discard the optimal path to a repeated state if it is not the first one generated.

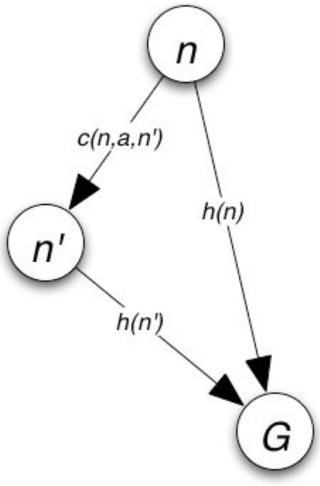
- Two solutions:
  – Extend GraphSearch with an extra bookkeeping i.e. remove more expensive of two paths
  – Ensure that optimal path to any repeated state is always followed first (as with uniform-cost search)
  ➔ Extra requirement on *h(n)*: consistency (monotonicity)

Consistent Heuristic

A heuristic *h* is consistent (or monotone) if

1) for each node *n* and each child *n'* of *n* generated by any action *a*:

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



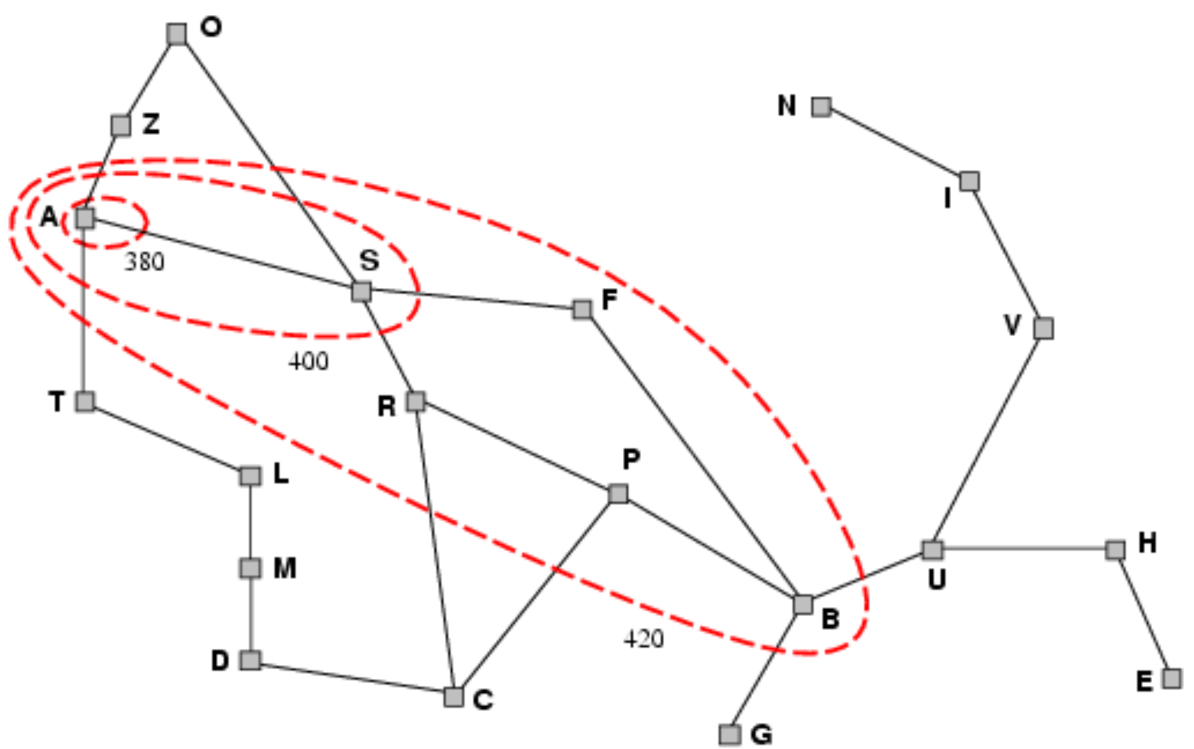
(triangle inequality)

A consistent heuristic is also admissible

→ Intuition: a consistent heuristics becomes more precise as we get deeper in the search tree

Optimality of A*

- A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Admissibility and Consistency

- A consistent heuristic is also admissible
- An admissible heuristic may not be consistent, but many admissible heuristics are consistent

Properties of A*

- Completeness?

Properties of A*

- Completeness? Yes
  - Since bands of increasing *f* are added
  - Unless there are infinitely many nodes with $f \leq f(G)$
- Time complexity?

Properties of A*

- Completeness: Yes
- Time complexity:
  - Number of nodes expanded is still exponential in the length of the solution.
- Space complexity?

Properties of A*

- 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
- Optimality?

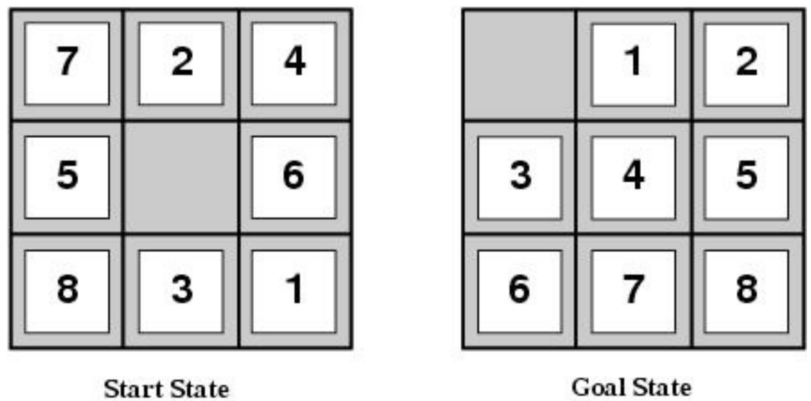
Properties of A*

- 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 $f(n)>C^*$

On Completeness and Optimality

- A* with a consistent heuristic function has nice properties: completeness, optimality, no need to revisit states
- Theoretical completeness does not mean "practical" completeness if you must wait too long to get a solution (remember the time limit issue)
- So, if one can't design an accurate consistent heuristic, it may be better to settle for a non-admissible heuristic that "works well in practice", even through completeness and optimality are no longer guaranteed

Heuristic functions

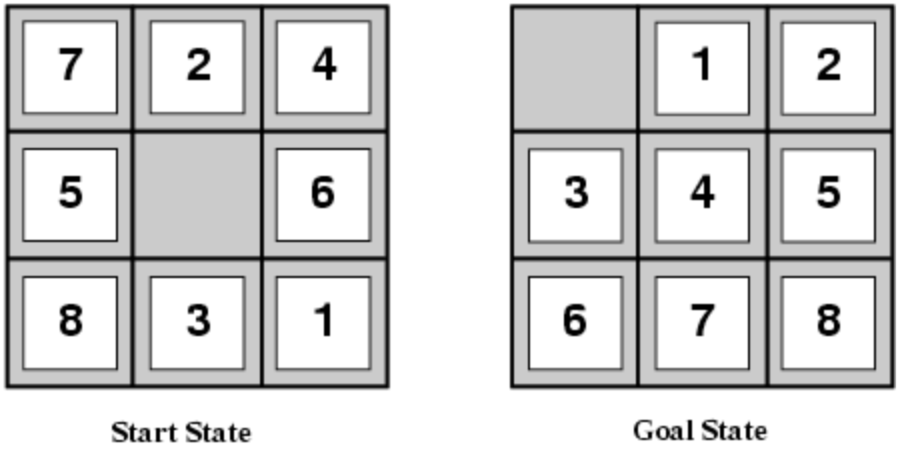


- E.g for the 8-puzzle
  - Avg. solution cost is about 22 steps (branching factor +/- 3)
  - Exhaustive search to depth 22 looks at $3^{22} \approx 3.1 \times 10^{10}$ states.
  - A good heuristic function can reduce the search process.
  - With repeated states only 9!/2 = 181,440.

Heuristic Function Example

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = the sum of the distances of the tiles from their goal positions, i.e. no. of squares from desired location of each tile, (total Manhattan distance)

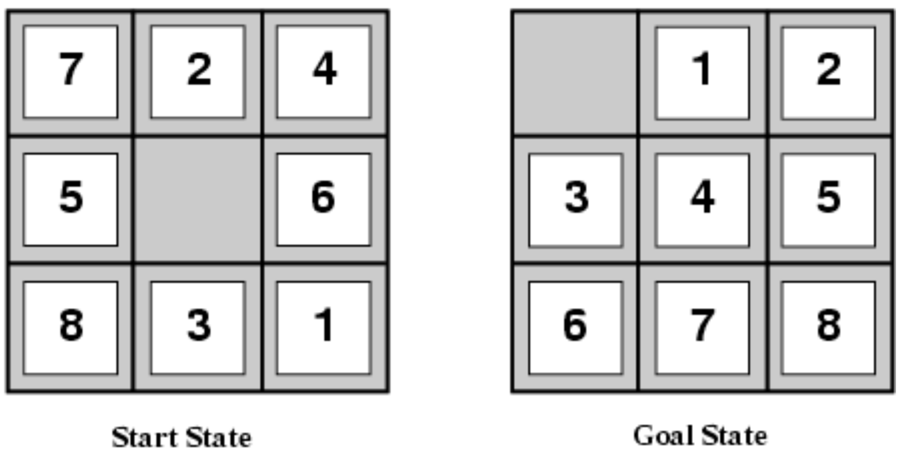


- $h_1(S) = ?$
- $h_2(S) = ?$

Heuristic Function Example

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = the sum of the distances of the tiles from their goal positions, i.e. no. of squares from desired location of each tile, (total Manhattan distance)

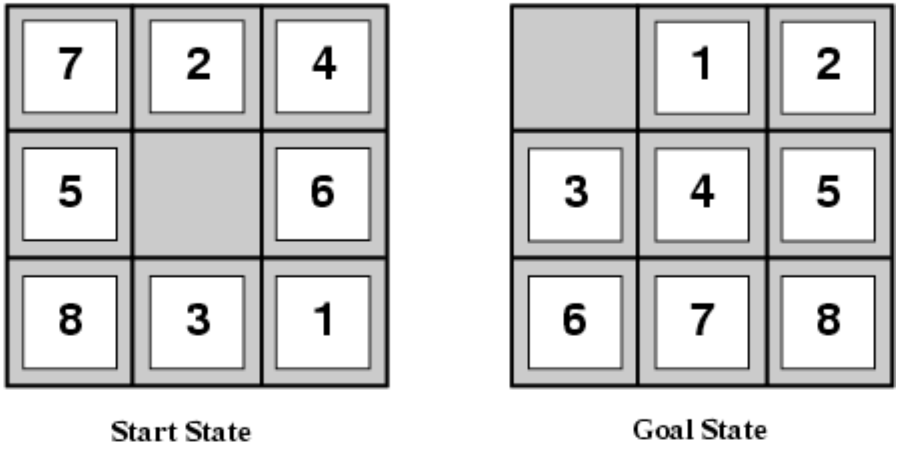


- $h_1(S)$ = ? 8
- $h_2(S)$ = ?

Heuristic Function Example

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = the sum of the distances of the tiles from their goal positions, i.e. no. of squares from desired location of each tile, (total Manhattan distance)



- $h_1(S)$ = ? 8
- $h_2(S)$ = ? 3+1+2+2+2+3+3+2 = 18

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.

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

  - 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.

Heuristic quality and dominance

- To test h_1 *and* h_2, generated 1,200 random problems with solution lengths from 2 to 24.

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- If $h_2(n)$ >= $h_1(n)$ for all n (both admissible)* note: >=, not <=

  then h_2 *dominates* h_1 and is better for search

- Given any collection of admissible heuristics, their maximum value is also admissible and dominates

Learning to search better

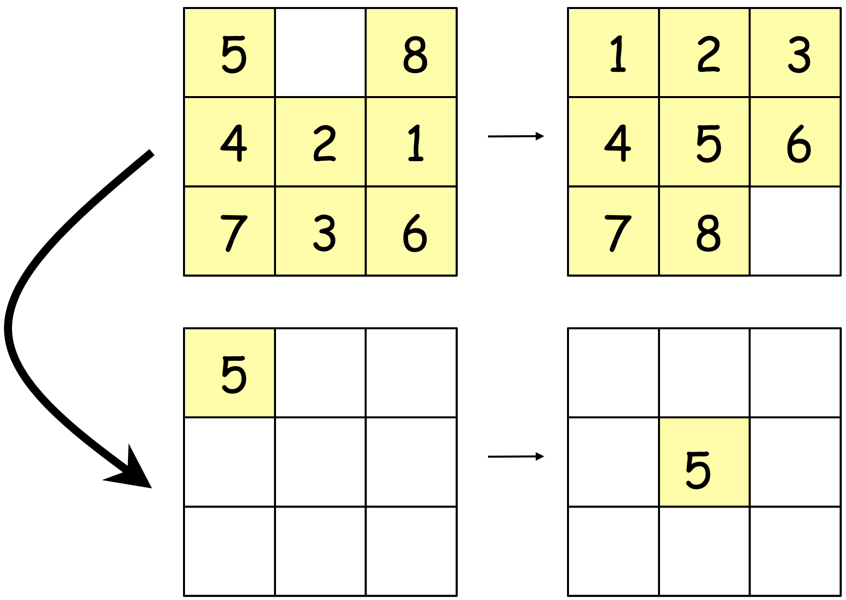
- All previous algorithms use *fixed strategies*.
- Agents can learn to improve their search by exploiting the *meta-level state space*.
  - Each meta-level state is an internal (computational) state of a program that is searching in *the object-level state space (e.g. Romania)*
  - In A* such a state consists of the current search tree
- A meta-level learning algorithm from experiences at the meta-level to avoid exploring unpromising subtrees:
  - Can be done using *reinforcement learning*: the goal of learning is to minimize the **total cost** of problem solving, trading off computational expense and path cost (e.g. path to Fagaras not useful to expand)

Inventing admissible heuristics: Relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem
  - Relaxed 8-puzzle for h_1 : a tile can move anywhere (vs. just to adjacent empty square):
    - As a result, $h_1(n)$ gives the shortest solution
  - Relaxed 8-puzzle for h_2 : a tile can move one square in any direction, (even onto occupied square):
    - As a result, $h_2(n)$ gives the shortest solution.
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If problem definition is written down in a formal language, it's possible to construct relaxed problems automatically (see Logical Agents and First-Order Logic)

Inventing admissible heuristics: Relaxed problem example

- By solving relaxed problems at each node
- In the 8-puzzle, the sum of the distances of each tile to its goal position (h_2) corresponds to solving 8 simple problems:



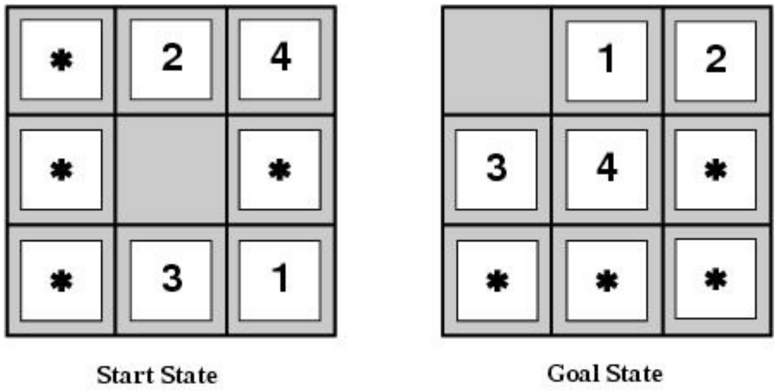
d_i is the length of the shortest path to move tile i to its goal position, ignoring the other tiles, e.g., $d_5 = 2$

$h_2 = \Sigma_{i=1,\ldots 8}\, d_i$

- It ignores negative interactions among tiles

Inventing admissible heuristics: Solution Cost of Subproblem

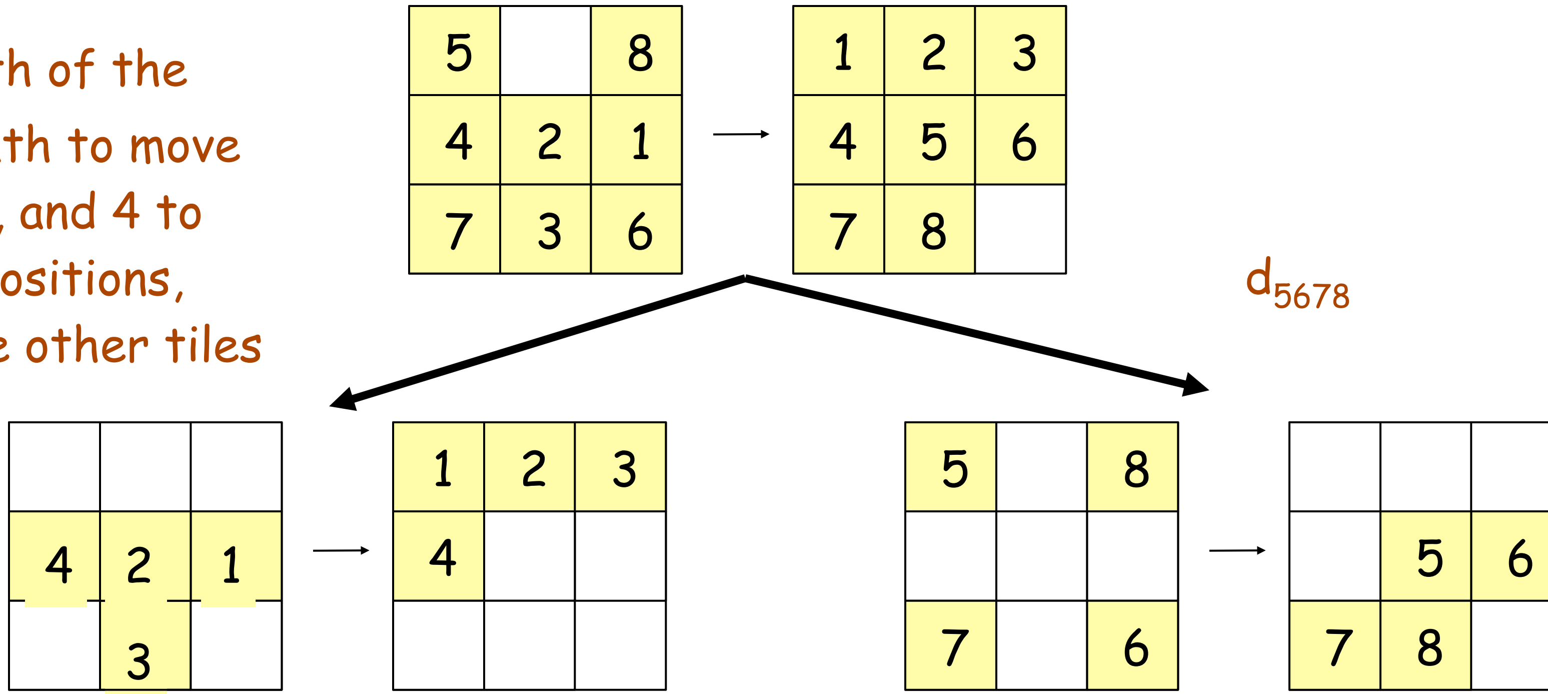
- Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem.
- This cost is a lower bound on the cost of the real problem.
- Pattern databases store the exact solution for every possible subproblem instance.
  - The complete heuristic is constructed using the patterns in the DB



Example of Subproblem

- For example, we could consider two more complex relaxed problems:

d_{1234} = length of the shortest path to move tiles 1, 2, 3, and 4 to their goal positions, ignoring the other tiles

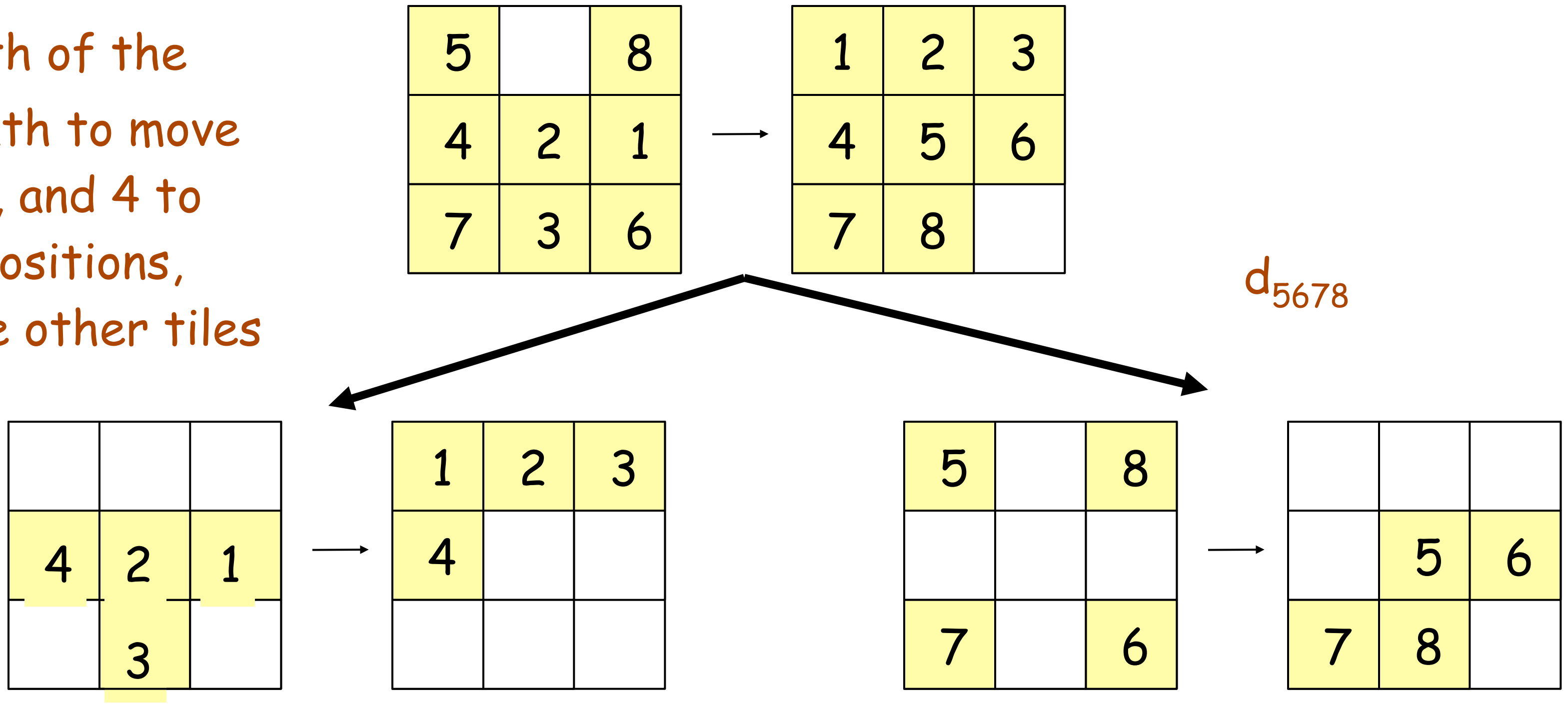


- → $h = d_{1234} + d_{5678}$ [disjoint pattern heuristic]

Example of Subproblem

- For example, we could consider two more complex relaxed problems:

d_{1234} = length of the shortest path to move tiles 1, 2, 3, and 4 to their goal positions, ignoring the other tiles

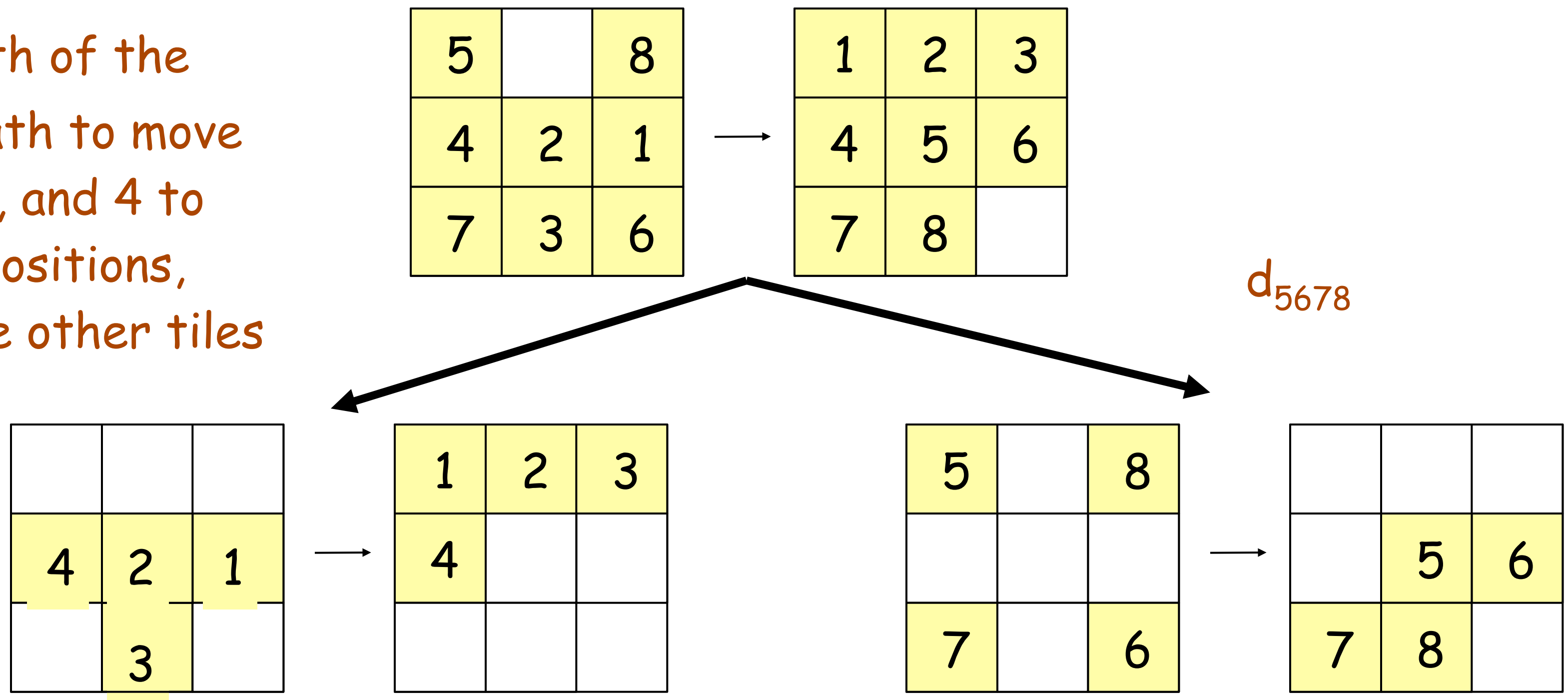


- → $h = d_{1234} + d_{5678}$ [disjoint pattern heuristic]
- How to compute d_{1234} and d_{5678}?

Example of Subproblem

- For example, we could consider two more complex relaxed problems:

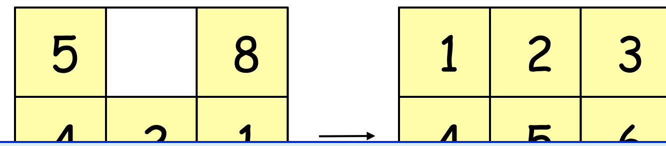
d_{1234} = length of the shortest path to move tiles 1, 2, 3, and 4 to their goal positions, ignoring the other tiles



- → $h = d_{1234} + d_{5678}$ [disjoint pattern heuristic]
- These distances are pre-computed and stored

Example of Subproblem

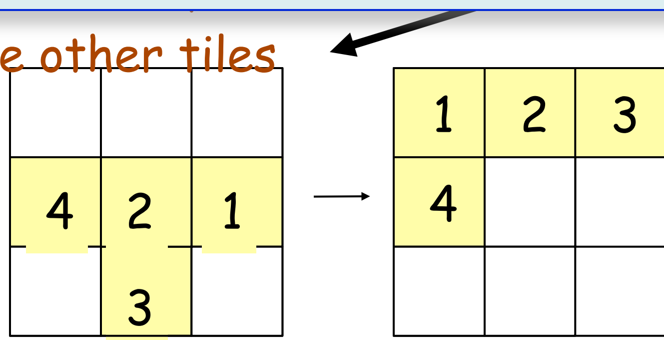
- For example, we could consider two more complex relaxed problems:

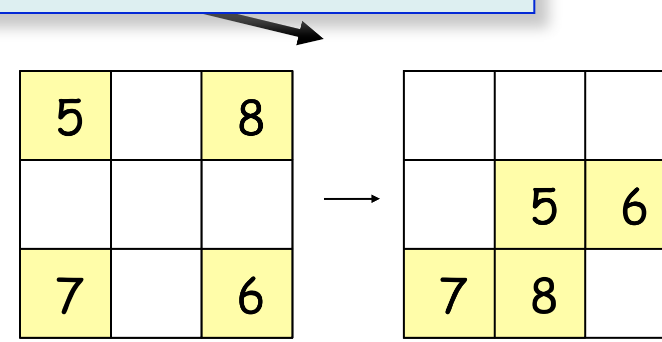


d... = length of the

ignoring the other tiles

> → Several order-of-magnitude speedups for the 15- and 24-puzzle (see R&N)





- → $h = d_{1234} + d_{5678}$ [disjoint pattern heuristic]
- These distances are pre-computed and stored

Inventing admissible heuristics: Learning from Experience

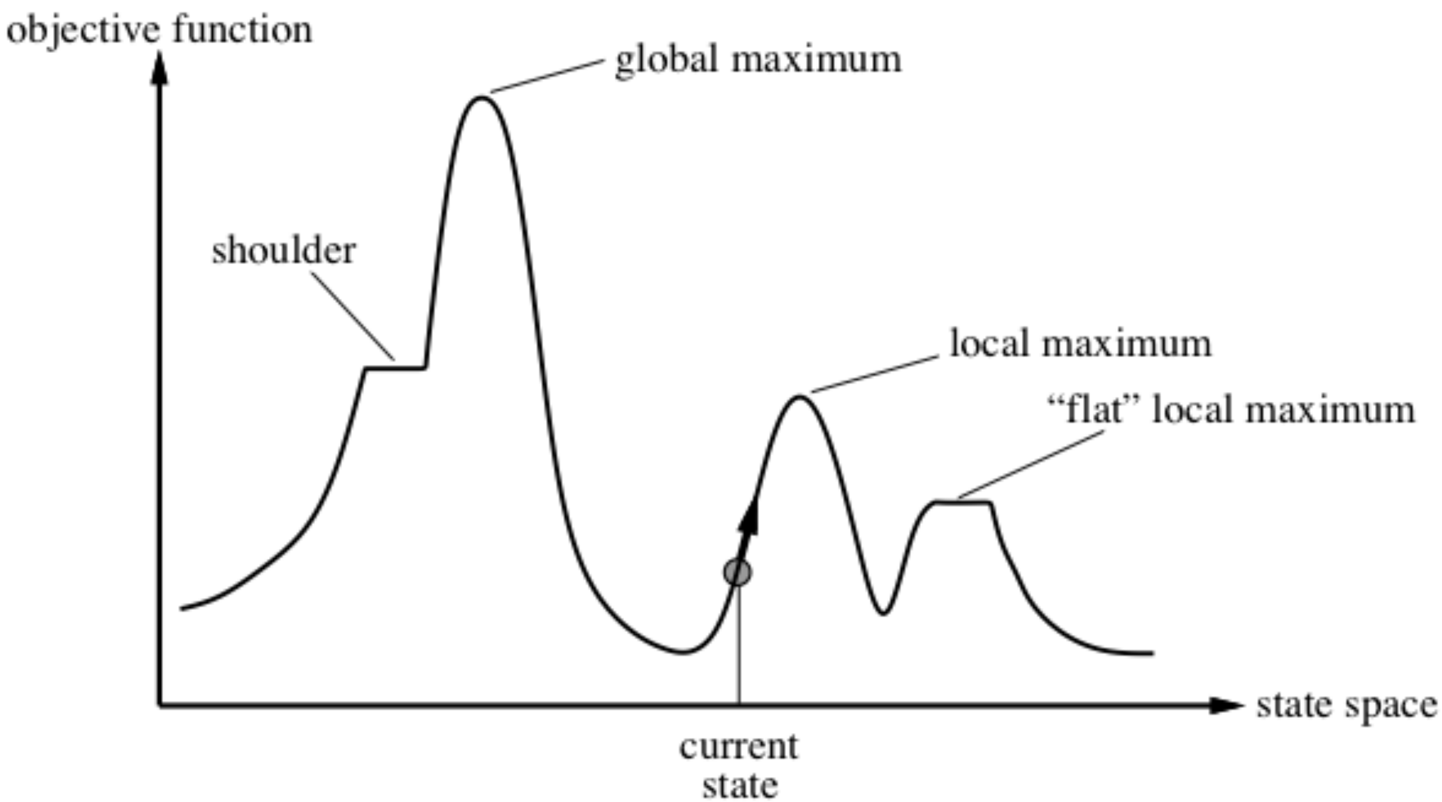
- Another way to find an admissible heuristic is through learning from experience:
  - Experience = solving lots of 8-puzzles
  - An *inductive learning algorithm* can be used to predict costs for other states that arise during search (using neural networks, decision trees, and other methods).

Local search and optimization

- Local search = no search tree; use single current state and move to neighboring states.
- Advantages:
  - Use very little memory
  - Find often reasonable solutions in large or infinite state spaces.
- Only applicable to problems where the path is irrelevant (e.g., 8-queen), unless the path is encoded in the state
- Also useful for pure optimization problems.
  - Find best state according to some ***objective function***.
  - e.g. survival of the fittest as a metaphor for optimization.

State space landscape

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



Hill-climbing search

- "is a loop that continuously moves in the direction of increasing value", i.e. uphill
  - It terminates when a peak is reached.
- Hill climbing does not look ahead of the *immediate neighbors* of the current state.
- Hill-climbing chooses randomly among the set of best successors, if there is more than one.
- Hill-climbing a.k.a. *greedy local search*

Hill-climbing search

- "Like climbing Everest in thick fog with amnesia"

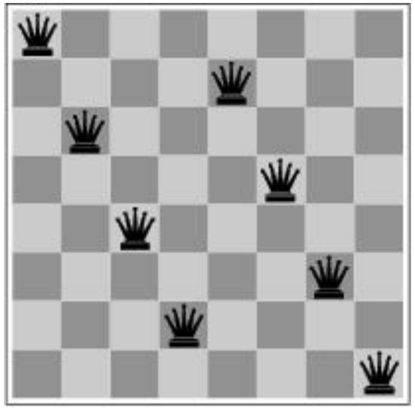
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

  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*

Example: *n*-queens Problem

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

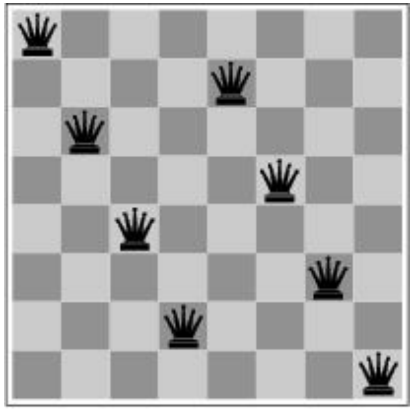
8-queens Problem Incremental or **Uninformed** Formulation



Incremental formulation: augment state description starting with an empty state) vs. complete-state formulation (starts with all 8 queens on board)

- States??
- Initial state??
- Actions??
- Goal test??

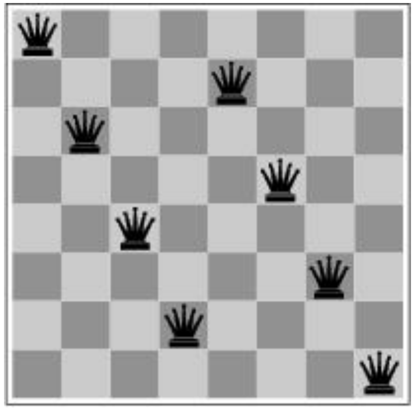
8-queens Problem Incremental or **Uninformed** Formulation



Incremental formulation:

- States? Any arrangement of 0 to 8 queens on the board
- Initial state? No queens on board
- Actions/Successor function? Add queen to any empty square
- Goal test? 8 queens on board and none attacked
- ➔ 64 x 63 x … x 57 possible sequences to investigate ≈ 1.8×10^{14}

8-queens Problem Incremental or **Uninformed** Formulation



Incremental formulation (alternative)

- States? *n* (0≤ *n*≤ 8) queens on the board, one per column in the *n* leftmost columns with no queen attacking another.
- Actions/Successor function? Add queen in leftmost empty column such that is not attacking other queens
- ➔ only 2057 possible sequences to investigate
- Yet makes no difference when *n*=100

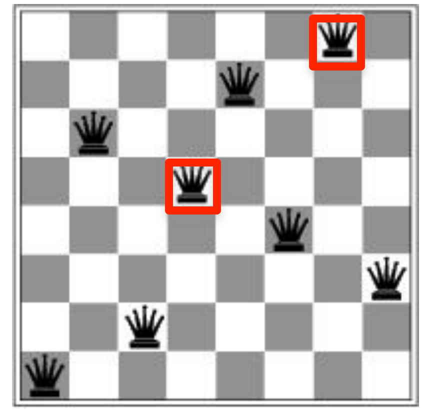
8-queens problem Complete-state or **Informed** Formulation Hill-climbing example

- Complete-state formulation (typically used in local searches): each state has 8 queens on board, one per column.
- **Successor function:** returns all possible states generated by moving a single queen to another square in the same column.
- **Heuristic function *h(n)*:** the number of pairs of queens that are attacking each other (directly or indirectly).

8-queens problem
Hill-climbing example

a)

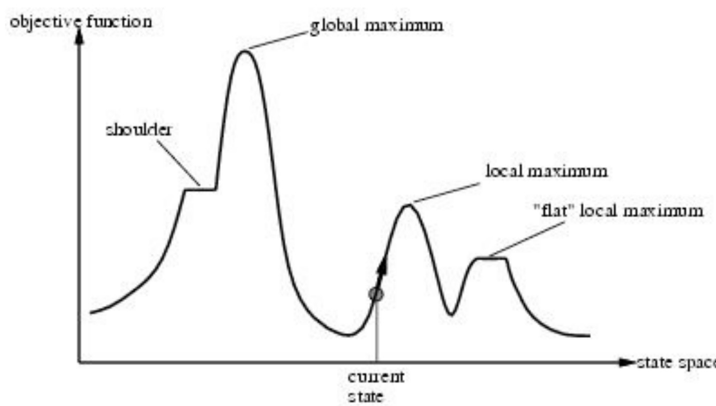
b)



a) Shows a state of *h*=17 and the *h*-value for each possible successor.

b) Shows a local minimum in the 8-queens state space (*h*=1).

Drawbacks

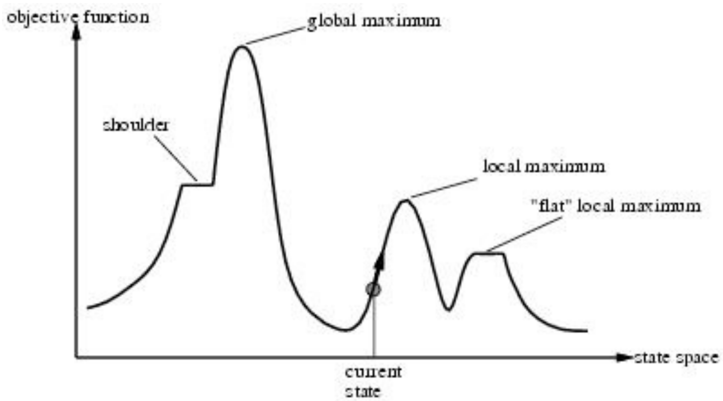


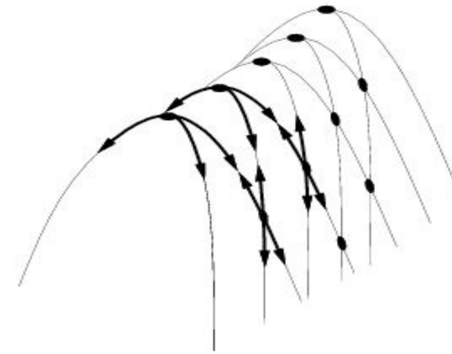
- **Local maxima:** local max is peak that is higher than each of its neighboring states, but lower than global maximum
- **Plateaux:** an area of the state space where the evaluation function is flat.

➔ **Incomplete:** Gets stuck 86% of the time, solving only 14% of problem instances

➔ Works quickly: 4 steps on average when it succeeds and 3 steps when it gets stuck (not bad for state space with $8^8 \approx$ 17 million states)

Drawbacks





- **Local maxima:** local max is peak that is higher than each of its neighboring states, but lower than global maximum
- **Ridge:** sequence of local maxima difficult for greedy algorithms to navigate
- **Plateaux:** an area of the state space where the evaluation function is flat.
- Gets stuck 86% of the time, works quickly (4 steps on average when it succeeds and 3 when it gets stuck – not bad for state space with $8^8 \approx 17$ million states)

Hill-climbing variations

- Stochastic hill-climbing
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- First-choice hill-climbing
  - Stochastic hill climbing by generating successors randomly until a better one is found.
- Random-restart hill-climbing
  - Tries to avoid getting stuck in local maxima.

Simulated annealing search

- Hill-Climbing that *never* makes “downhill” moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck in local maximum.
- In contrast, purely random walk (moving to a successor chosen uniformly at random from set of successor) is complete but extremely inefficient
- How about combining the two? ➔ Simulated annealing, a version of stochastic hill climbing where *some downhill moves* are allowed: they are accepted readily early in annealing schedule and less often as time goes on.
-

Simulated annealing

- Origin; metallurgical annealing (high T to harden metals, then gradually cooling them)
- Switch point of view from hill climbing to **gradient descent** (i.e. minimize cost)
- **Idea:** escape local minima (or local maxima experienced with hill-climbing) by allowing some "bad” random moves
  - but gradually decrease their frequency
- Bouncing ball analogy:
  - Goal: get ball in deepest crevice of bumpy surface
  - If let ball roll, might get stuck in local minimum
  - If shake surface hard (high temperature), ball bounces out of LOCAL min, but if shake too hard, ball will be dislodged from GLOBAL min

  ➔ Best start to shake hard, then gradually reduce intensity (lower the temperature),
- Can prove: If T decreases slowly enough, best state is reached.
- Applied for VLSI layout in 1980s, airline scheduling, etc.

Simulated annealing

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

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

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The $schedule$ input determines the value of T as a function of time.

Properties of simulated annealing search

- One can prove: If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1

- Widely used in VLSI layout, airline scheduling, etc.

Steepest Descent

1) $S \leftarrow$ initial state
2) Repeat:
   a) $S' \leftarrow \arg\min_{S' \in SUCCESSORS(S)}\{h(S')\}$
   b) if GOAL?(S') return S'
   c) if $h(S') < h(S)$ then $S \leftarrow S'$ else return failure

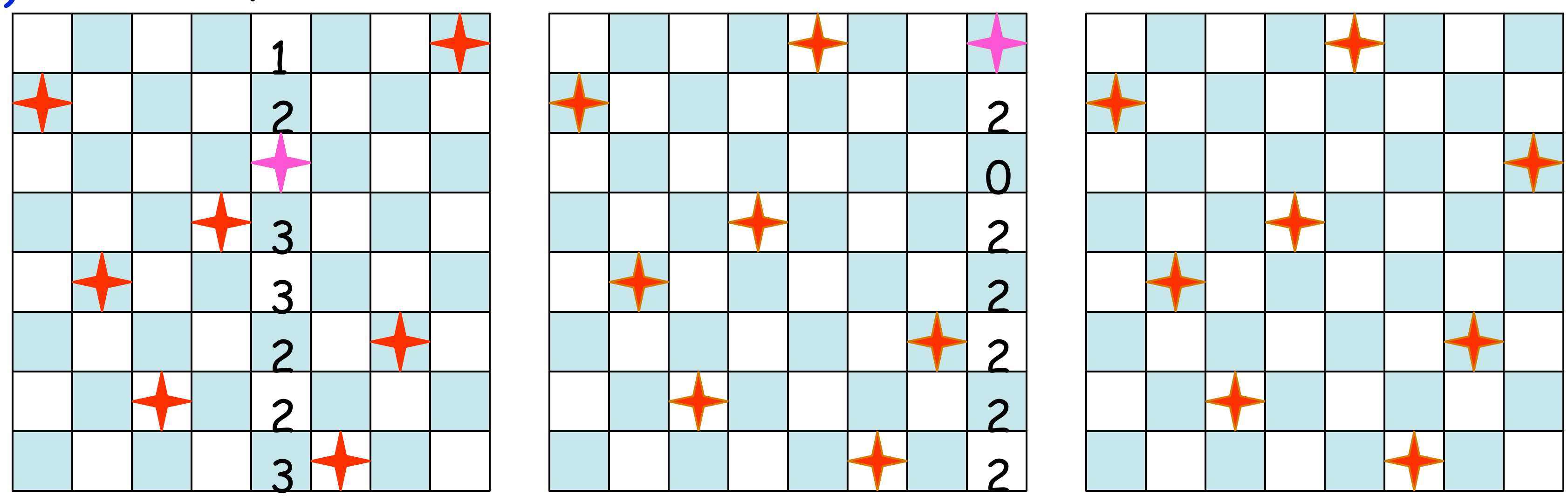
Similar to:

- hill climbing with –h
- gradient descent over continuous space

Application: 8-Queen

Repeat n times:

1) Pick an initial state S at random with one queen in each column
2) Repeat k times:
   a) If GOAL?(S) then return S
   b) Pick an attacked queen Q at random
   c) Move Q in its column to minimize the number of attacking queens → new S [min-conflicts heuristic]
3) Return failure



Application: 8-Queen

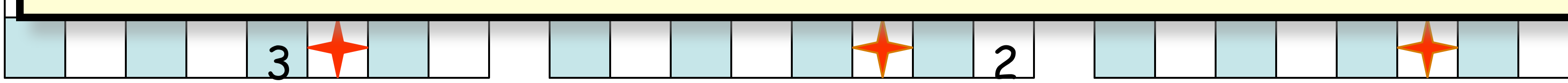
Repeat n times:

1)

2)

Why does it work ???

1) There are **many** goal states that are well-distributed over the state space
2) If no solution has been found after a few steps, it's better to start it all over again. Building a search tree would be much less efficient because of the high branching factor
3) Running time almost independent of the number of queens



Steepest Descent

1) S ← initial state
2) Repeat:
   a) S′ ← $\arg\min_{S' \in SUCCESSORS(S)}\{h(S')\}$
   b) if GOAL?(S′) return S′
   c) if h(S′) < h(S) then S ← S′ else return failure

may easily get stuck in local minima

→ Random restart (as in n-queen example)

→ Monte Carlo descent

Monte Carlo Descent

1) S ← initial state
2) Repeat k times:
   a) If GOAL?(S) then return S
   b) S' ← successor of S picked at random
   c) if h(S') ≤ h(S) then S ← S'
   d) else
      - Δh = h(S')-h(S)
      - with probability ~ exp(−Δh/T), where T is called the "temperature" S ← S' [Metropolis criterion]
3) Return failure

Simulated annealing lowers T over the k iterations.

It starts with a large T and slowly decreases T

"Parallel" Local Search Techniques

They perform several local searches concurrently, but not independently:

- Beam search
- Genetic algorithms

See R&N, local search

Local Beam Search

- Keep track of *k* states rather than just one
  Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated

If any one is a goal state,

stop;

Else

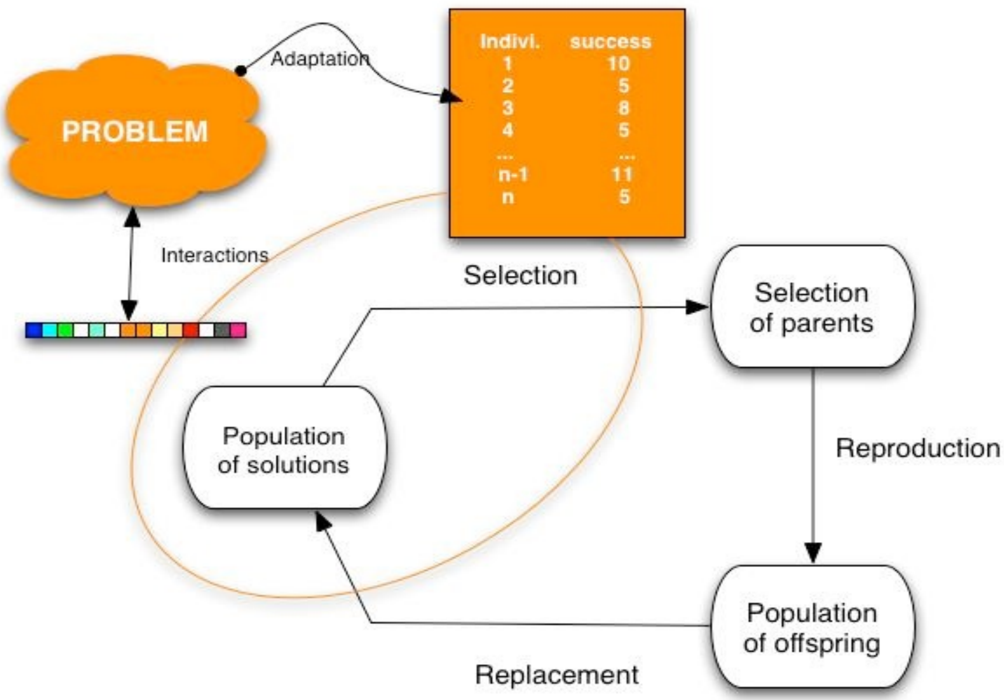
select the *k* best successors from the complete list and repeat.

Local Beam Search

- Algorithms different
  - In a random-restart search, each search process runs independently of the others.
  - *In a local beam search, useful information is passed among the parallel search threads.*
- States that generate the best successors tell others
- Effect: algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made

Genetic algorithms

- Variant of local beam search with "*sexual*" *recombination.*



Genetic algorithms (GA)

- A successor state is generated by combining *two* parent states (vs. modifying a single state in local beam search)
- Start with *k* randomly generated states (population)
- Each state, or individual, is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function in GA terminology)
  - Returns higher values for better states (e.g. # of non-attacking pair of queens)
- Produce the next generation of states by selection, crossover, and mutation

Genetic algorithms

- In this instance:
  - Fitness function: number of non-attacking pairs of queens (min = 0, max = (8 × 7)/2 = 28)
  - Probability of being selected for reproduction is directly proportional to fitness score:
  - 24/(24+23+20+11) = 31%
  - 23/(24+23+20+11) = 29% etc

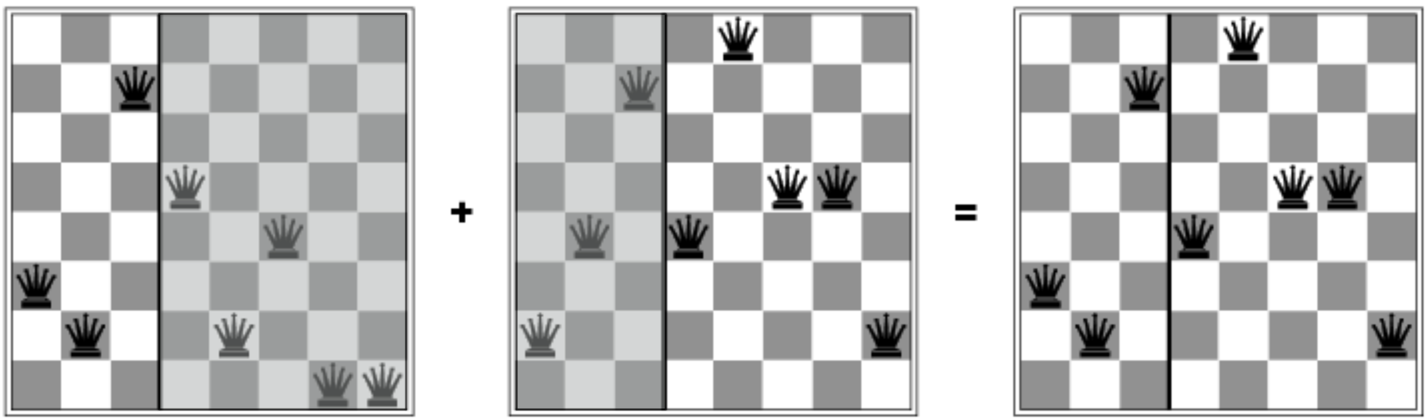
Genetic algorithms

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
     weights ← WEIGHTED-BY(population, fitness)
     population2 ← empty list
     for i = 1 to SIZE(population) do
        parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
        child ← REPRODUCE(parent1, parent2)
        if (small random probability) then child ← MUTATE(child)
        add child to population2
     population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

Figure 4.7 A genetic algorithm. Within the function, *population* is an ordered list of individuals, *weights* is a list of corresponding fitness values for each individual, and *fitness* is a function to compute these values.

Genetic algorithms



Reproduction step example

Search problems

Blind search (uninformed search)

Heuristic search:
Best-First and A*

Construction of Heuristics

Variants of A*

Local search

When to Use Search Techniques?

1) The search space is small, and
   - No other technique is available, or
   - Developing a more efficient technique is not worth the effort

2) The search space is large, and
   - No other available technique is available, and
   - There exist “good” heuristics

Exploration problems

- Until now all algorithms were offline.
  - Offline = solution is determined before executing it.
  - Online = interleaving computation and action

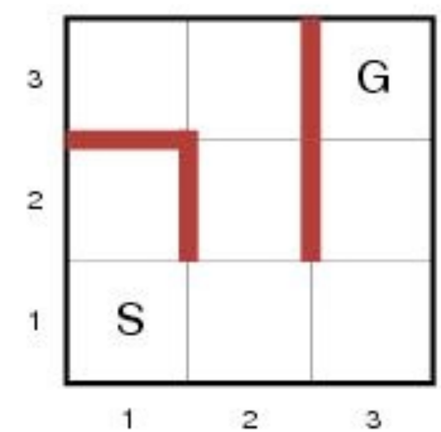
- Online search is necessary for dynamic and semi-dynamic environments
  - It is impossible to take into account all possible contingencies.

- Used for *exploration problems*:
  - Unknown states and actions.
  - e.g. any robot in a new environment, a newborn baby,…

Online search problems

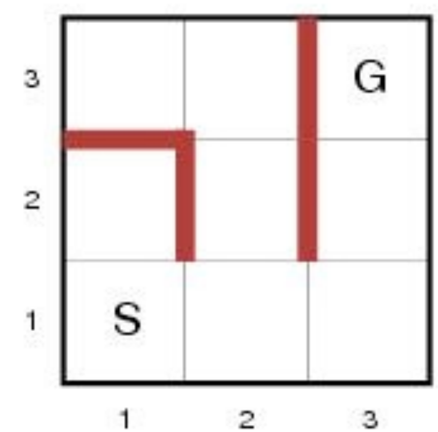
- Agent knowledge:
  - ACTION(s): list of allowed actions in state s
  - C(s,a,s'): step-cost function (! After s' is determined)
  - GOAL-TEST(s)
- An agent can recognize previous states
- Actions are deterministic.
- Access to admissible heuristic *h(s)*

  e.g. manhattan distance

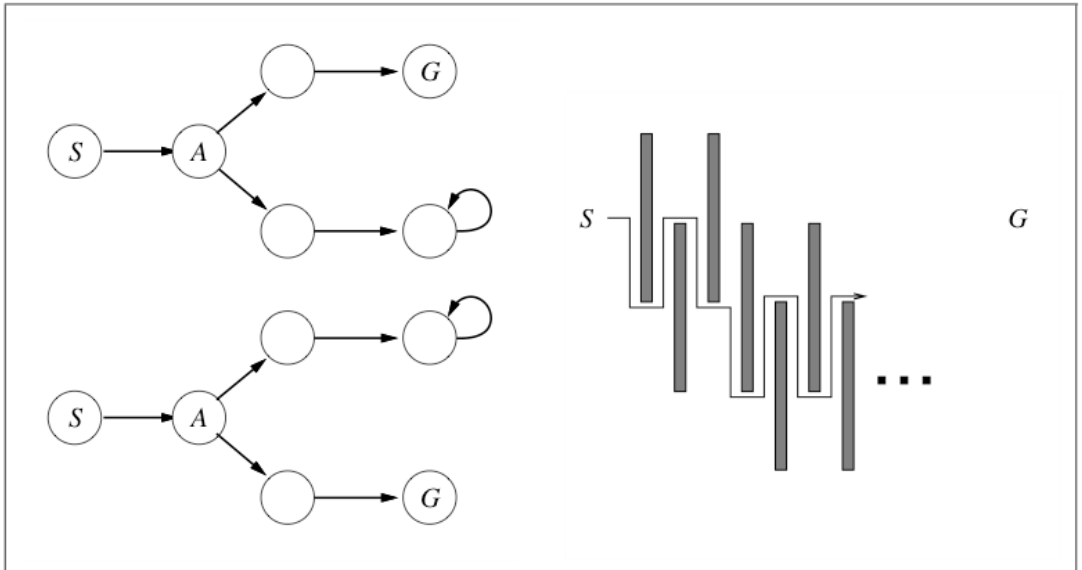


Online search problems

- Objective: reach goal with minimal cost
  - Cost = total cost of travelled path
  - Competitive ratio=comparison of cost with cost of the solution path if search space is known.
  - Can be infinite in case of the agent accidentally reaches dead ends



The adversary argument



(a) (b)

- Assume an adversary who can construct the state space while the agent explores it
  - Visited states S and A. What next?
    - Fails in one of the state spaces
- No algorithm can avoid dead ends in all state spaces.

Online search agents

- The agent maintains a map of the environment.
  - Updated based on percept input.
  - This map is used to decide next action.

Note difference with e.g. A*

An online version can only expand the node it is physically in (local order)

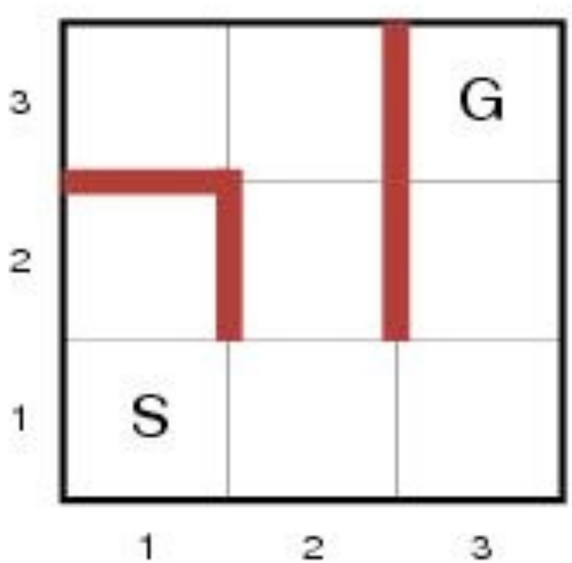
Online DF-search

```
function ONLINE-DFS-AGENT(s) returns an action
   inputs: s, a percept that identifies the current state
   static: result, a table, indexed by action and state, initially empty
           unexplored, a table listing, for each visited state, the actions not yet tried
           unbacktracked, a table listing, for each visited state, the backtracks not yet tried
           s⁻, a⁻, the previous state and action, initially null

   if GOAL-TEST(s) then return stop
   if s is a new state then unexplored[s] ← LEGAL-ACTIONS(s)
   if s⁻ is not null then do
      result[a⁻, s⁻] ← s
      add s⁻ to the front of unbacktracked[s]
   if unexplored[s] is empty then
      if unbacktracked[s] is empty then return stop
      else action ← the a such that result[a, s] = POP(unbacktracked[s])
   else action ← POP(unexplored[s])
   s⁻ ← s; a⁻ ← action
   return action
```

Figure 4.20 An online search agent that uses depth-first exploration. ONLINE-DFS-AGENT is applicable only in bidirected search spaces.

Online DF-search, example

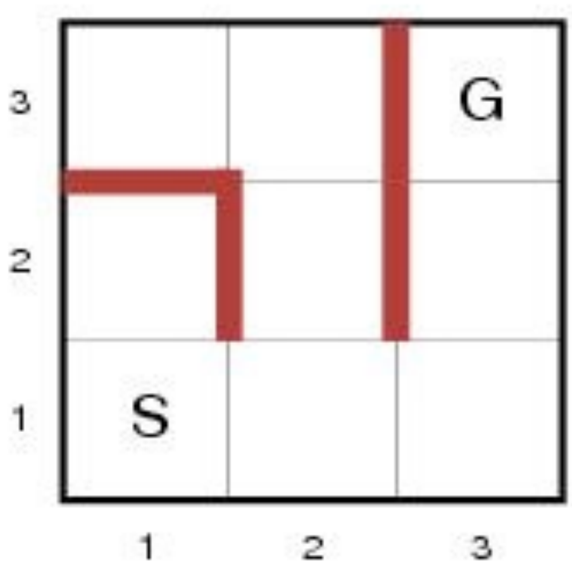


- Assume maze problem on 3x3 grid.
- s' = (1,1) is initial state
- Result, unexplored (UX), unbacktracked (UB), …

  are empty
- S,a are also empty

Online DF-search, example

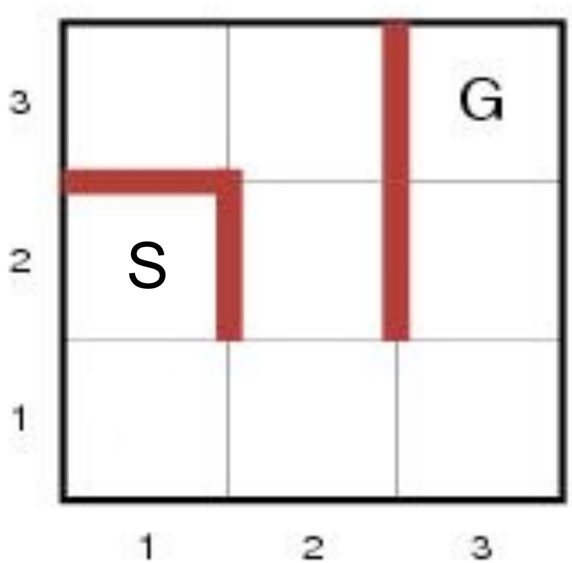
S'=(1,1)



- GOAL-TEST((,1,1))?
  - S not = G thus false
- (1,1) a new state?
  - True
  - ACTION((1,1)) -> UX[(1,1)]
    - {RIGHT,UP}
- s is null?
  - True (initially)
- UX[(1,1)] empty?
  - False
- POP(UX[(1,1)])->a
  - A=UP
- s = (1,1)
- Return a

Online DF-search, example

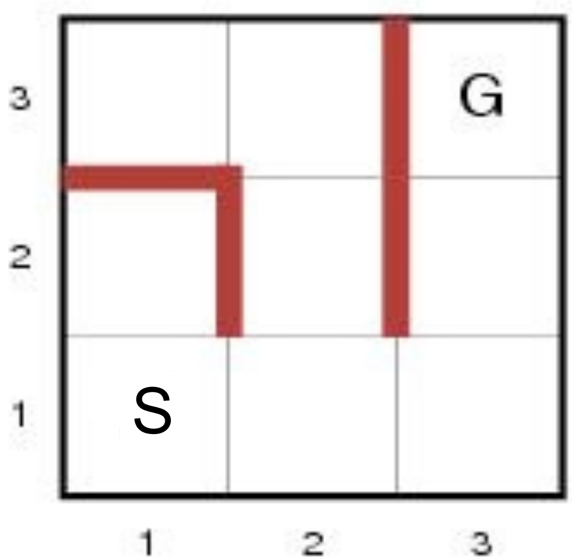
S'=(2,1)



- GOAL-TEST((2,1))?
  - S not = G thus false
- (2,1) a new state?
  - True
  - ACTION((2,1)) -> UX[(2,1)]
    - {DOWN}
- s is null?
  - false (s=(1,1))
  - result[UP,(1,1)] <- (2,1)
  - UB[(2,1)]={(1,1)}
- UX[(2,1)] empty?
  - False
- A=DOWN, s=(2,1) return A

Online DF-search, example

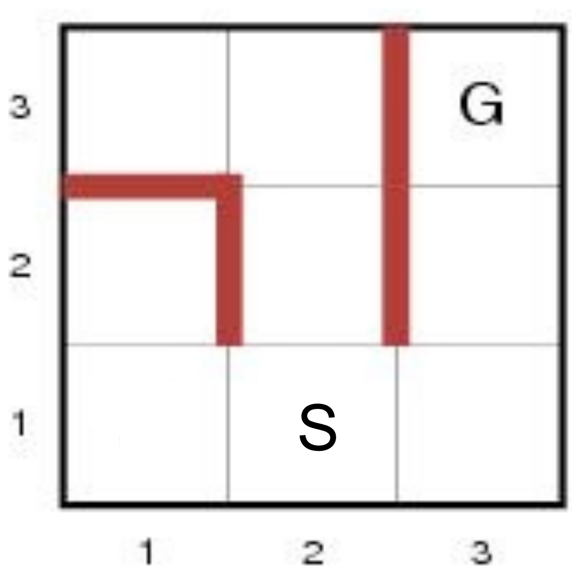
S’=(1,1)



- GOAL-TEST((1,1))?
  - S not = G thus false
- (1,1) a new state?
  - false
- s is null?
  - false (s=(2,1))
  - result[DOWN,(2,1)] <- (1,1)
  - UB[(1,1)]={(2,1)}
- UX[(1,1)] empty?
  - False
- A=RIGHT, s=(1,1) return A

Online DF-search, example

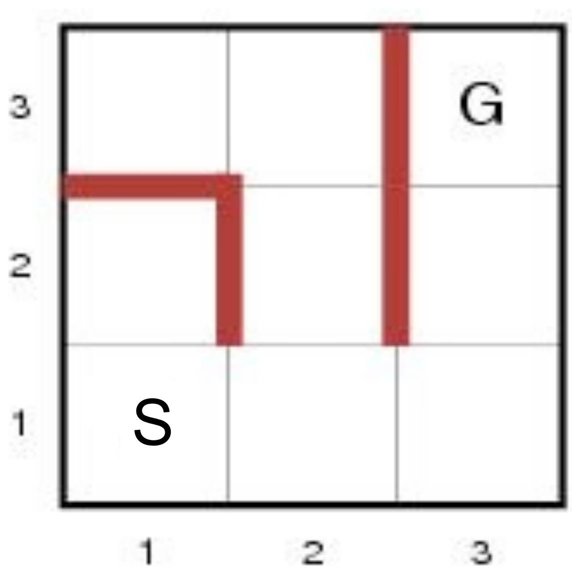
S'=(1,2)



- GOAL-TEST((1,2))?
  - S not = G thus false
- (1,2) a new state?
  - True, UX[(1,2)]={RIGHT,UP,LEFT}
- s is null?
  - false (s=(1,1))
  - result[RIGHT,(1,1)] <- (1,2)
  - UB[(1,2)]={(1,1)}
- UX[(1,2)] empty?
  - False
- A=LEFT, s=(1,2) return A

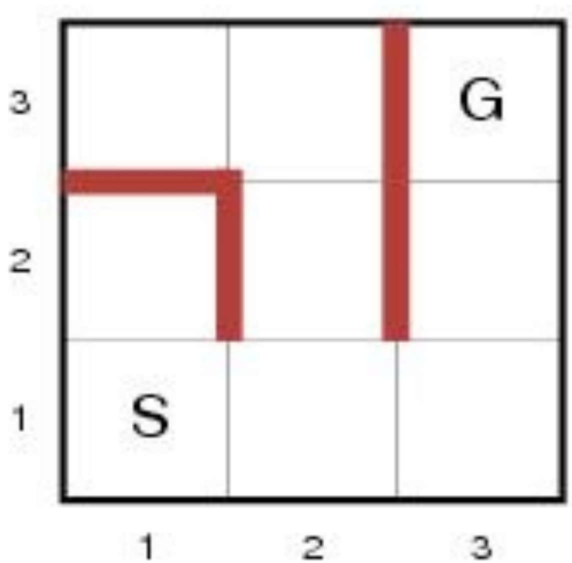
Online DF-search, example

S'=(1,1)



- GOAL-TEST((1,1))?
  - S not = G thus false
- (1,1) a new state?
  - false
- s is null?
  - false (s=(1,2))
  - result[LEFT,(1,2)] <- (1,1)
  - UB[(1,1)]={(1,2),(2,1)}
- UX[(1,1)] empty?
  - True
  - UB[(1,1)] empty? False
- A= b for b in result[b,(1,1)]=(1,2)
  - B=RIGHT
- A=RIGHT, s=(1,1) …

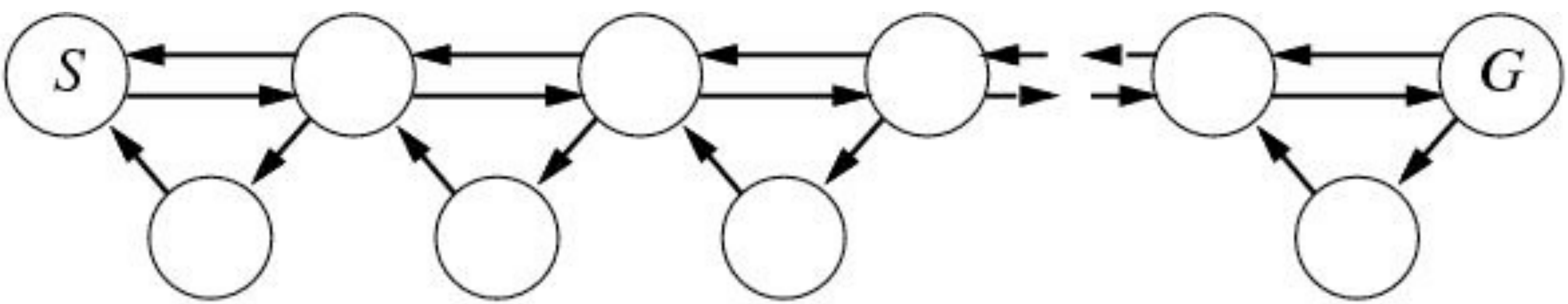
Online DF-search



- Worst case each node is visited twice.
- An agent can go on a long walk even when it is close to the solution.
- An online iterative deepening approach solves this problem.
- Online DF-search works only when actions are reversible.

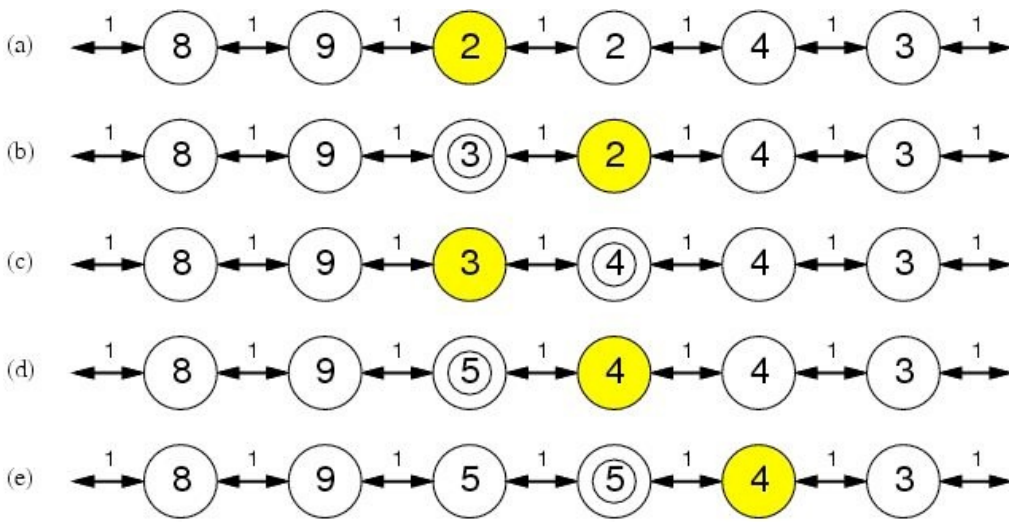
Online local search

- Hill-climbing is already online
  - One state is stored.
- Bad performance due to local maxima
  - Random restarts impossible.
- Solution: Random walk introduces exploration (can produce exponentially many steps)



Online local search

- Solution 2: Add memory to hill climber
  - Store current best estimate *H(s)* of cost to reach goal
  - *H(s)* is initially the heuristic estimate *h(s)*
  - Afterward updated with experience (see below)
- Learning real-time A* (LRTA*)



Learning real-time A*

function LRTA*-AGENT(s) **returns** an action
  inputs: s, a percept that identifies the current state
  static: *result*, a table, indexed by action and state, initially empty
    H, a table of cost estimates indexed by state, initially empty
    s^-, a^-, the previous state and action, initially null

  if GOAL-TEST(s) **then return** *stop*
  if s is a new state (not in H) **then** $H[s] \leftarrow h(s)$
  unless s^- is null
    $result[a^-, s^-] \leftarrow s$
    LRTA*-UPDATE($H, s^-, result$)
  $action \leftarrow$ the action a in LEGAL-ACTIONS(s) that minimizes LRTA*-COST($a, s, result, H$)
  $s^- \leftarrow s$; $a^- \leftarrow action$
  return *action*

procedure LRTA*-UPDATE($H, s^-, result$)
  $H[s^-] \leftarrow \min_{a \in \text{LEGAL-ACTIONS}(s^-)}$ LRTA*-COST($a, s^-, result, H$)

function LRTA*-COST($a, s, result, H$) **returns** a cost estimate
  if $result[a, s]$ is unknown **then return** $h(s)$
  else return $c(s, a, result[a, s]) + H[result[a, s]]$

Figure 4.23 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

Summary

- **Heuristics** to reduce search costs
- Algorithms that use heuristics, optimality comes with price in terms of search costs:
  - **Best-first search** is just GRAPH-SEARCH where the minimum-cost unexpanded nodes are selected for expansion. Best-first algorithms typically use a heuristic function *h(n)* that estimates the cost of a solution from *n*
  - **Greedy best-first search** expands nodes with minimal *h(n)*. It is not optimal but is often efficient.
  - **A* search** expands nodes with minimal *f(n)=g(n)+h(n)*. A* is complete and optimal, provided that we guarantee that *h(n)* is admissible (for TREE-SE ARCH) or consistent (for GRAPH-SEARCH). The space complexity of A*is still prohibitive.
  - The performance of heuristic search algorithms depends on the quality of the heuristic function.

Summary (2)

- *Local search* methods such as the classical **hill-climbing** algorithm operate on complete-state formulations. Several stochastic algorithms have been developed, including **simulated annealing**, which returns optimal solutions when given an appropriate cooling schedule.

- A **genetic algorithm** is a stochastic hill-climbing search in which a large population of states is maintained. New states are generated by **mutation** and by **crossover**, which combines of pairs of states from the population.

- **Exploration problems** arise when the agent has no idea about the states and actions of its environment. For safely explorable environments, online search agents can build a map and find a goal if one exists. Updating heuristic estimates from experience provides an effective method to escape from local minima.