Introduction to Artificial Intelligence

LIAW, RUNG-TZUO

Department of Computer Science and Information Engineering

Fu Jen Catholic University, Taiwan

Chapter 4
Search in Complex Environments

Outline

- **Heuristic (informed) search strategies**
- **Heuristic functions**
- **Local search algorithms & optimization problems**
- **Global search algorithm**
- **Online search agents & unknown environment**

Outline

- **Heuristic (informed) search strategies**
  - Best-first search
    - Greedy search
    - A* search
    - RBFS / SMA*
- **Heuristic functions**
- **Local search algorithms & optimization problems**
- **Global search algorithm**
- **Online search agents & unknown environment**

Review: Tree-Search

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

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

Best-First Search

- General approach of informed search
  - Uninformed: Uses only info available in problem definition
  - Informed: When strategies can determine **whether one non-goal state is *better* than another** → need evaluation
- Node is selected for expansion based on an evaluation function $\boldsymbol{f(n)}$
  - Evaluation function measures distance to the goal
  - Best-first: choose node which appears best
- Implementation:
  - Fringe is a queue sorted in decreasing order of desirability
  - Special cases: greedy search, A* search

What if $f(n)$ is unobtainable?

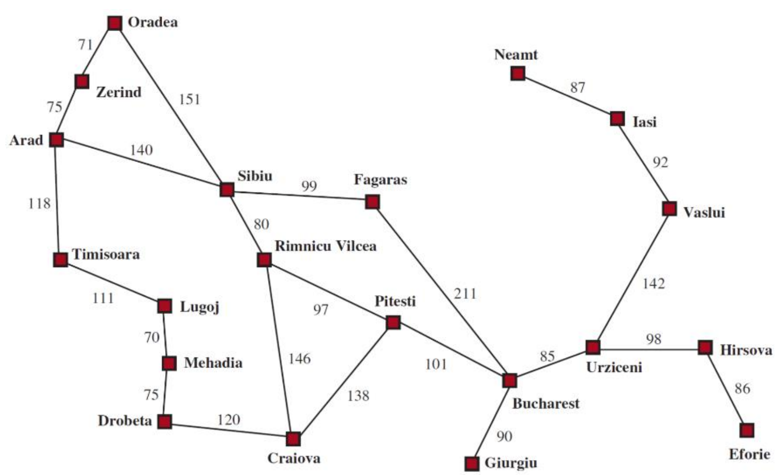
Heuristic

- "A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood."
- E.g. for tree search, heuristic function $h(n)$ = 'estimated' cost of the cheapest path from node n to goal node
  - If n is goal then $h(n) = 0$
  - More information later

Example

- **Romania with step costs in km**
  - h_{SLD} = straight-line distance heuristic
    - h_{SLD} can NOT be computed from the problem description itself from certain experience
  - In this example $f(n) = h(n)$
    - Expand node that is closest to goal = Greedy (best-first) search

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

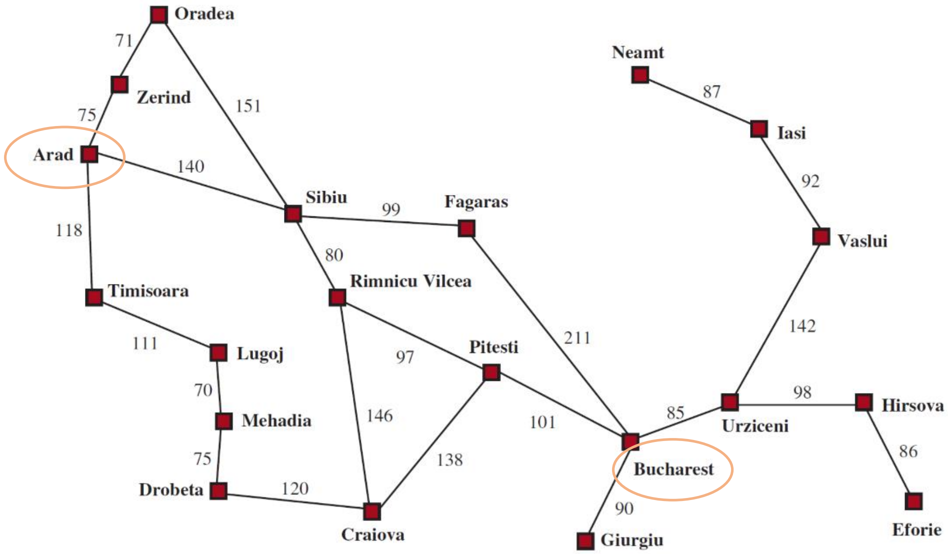


Example: Greedy Search

a) Initial state

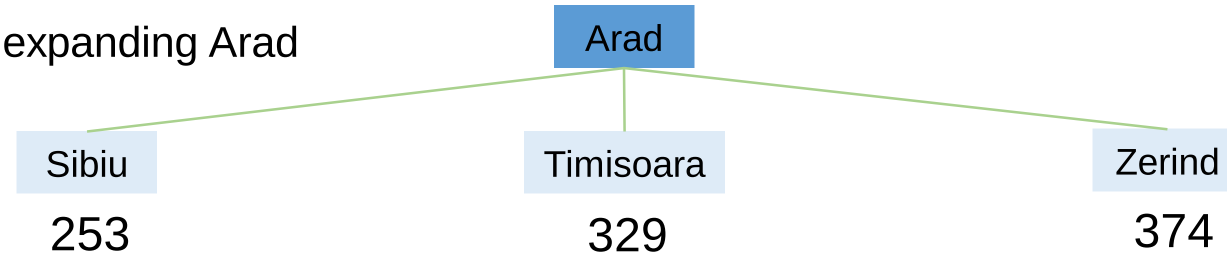
Arad

- **Scenario**
  - Use greedy search to solve the problem of traveling from Arad to Bucharest
- **Initial state = Arad**

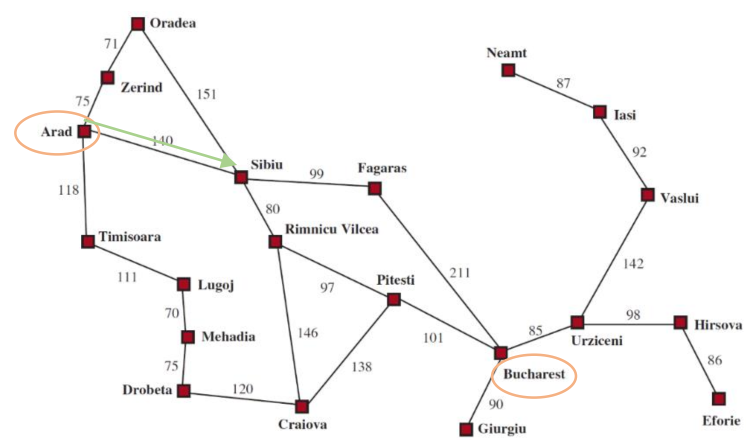


Example: Greedy Search

b) After expanding Arad

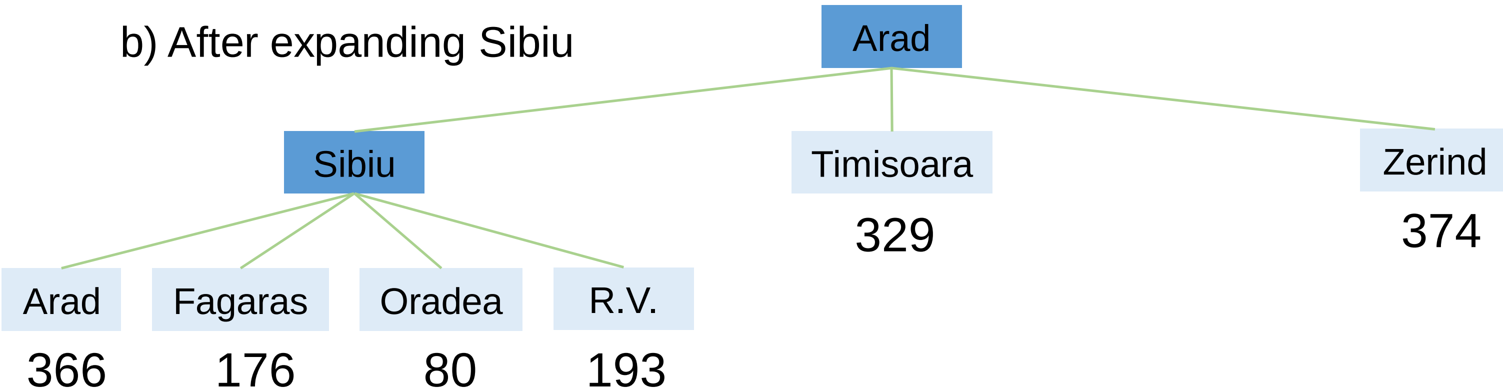


- **Procedure**
  - The first expansion step produces Sibiu (253), Timisoara (329), Zerind (374)
  - Greedy best-first will select Sibiu
    - Use greedy search to solve the problem of traveling from Arad to Bucharest

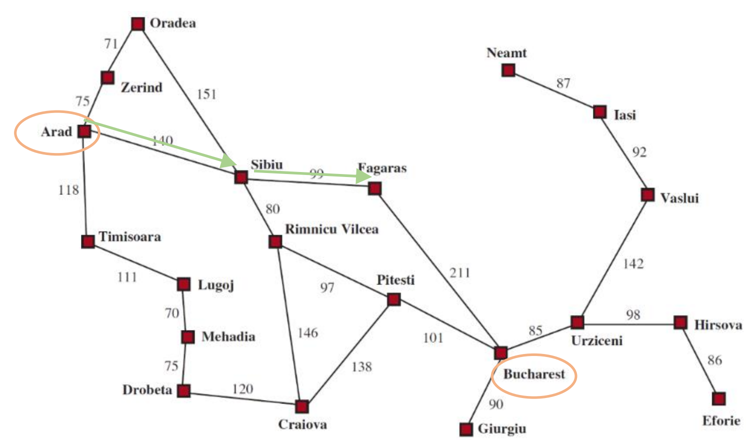


Example: Greedy Search

b) After expanding Sibiu

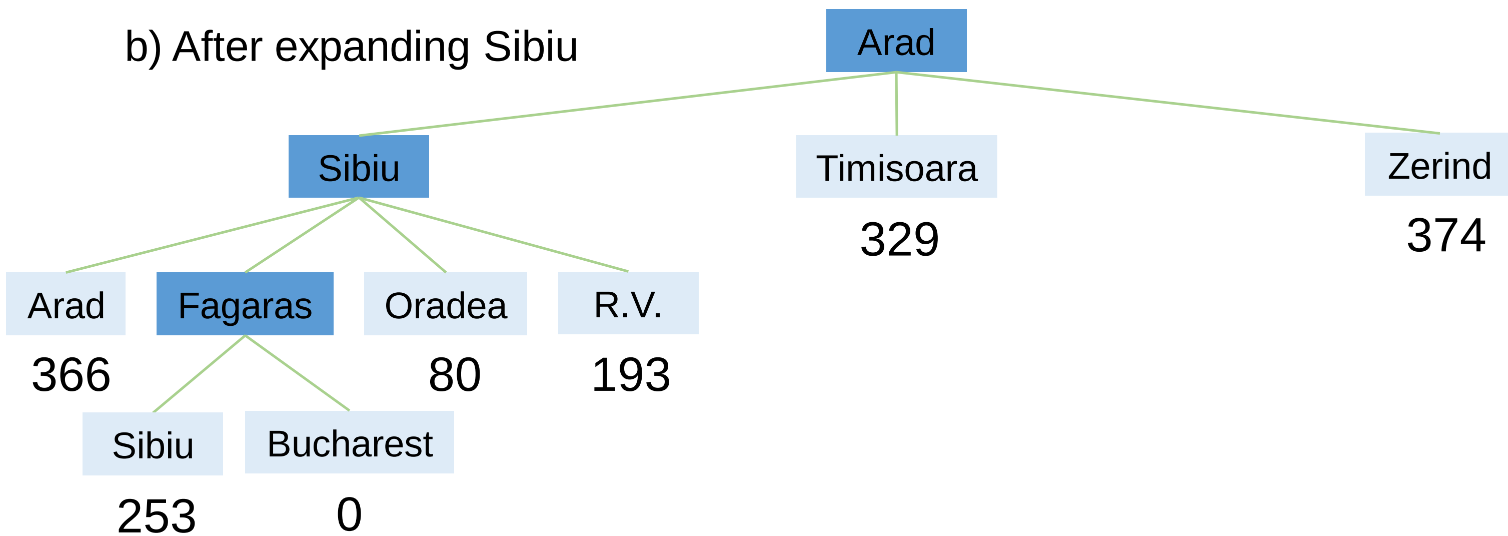


- **Procedure** (cont'd)
  - If Sibiu is expanded we get: Arad, Fagaras, Oradea and R.V.
  - Greedy best-first search will select: Fagaras (176)

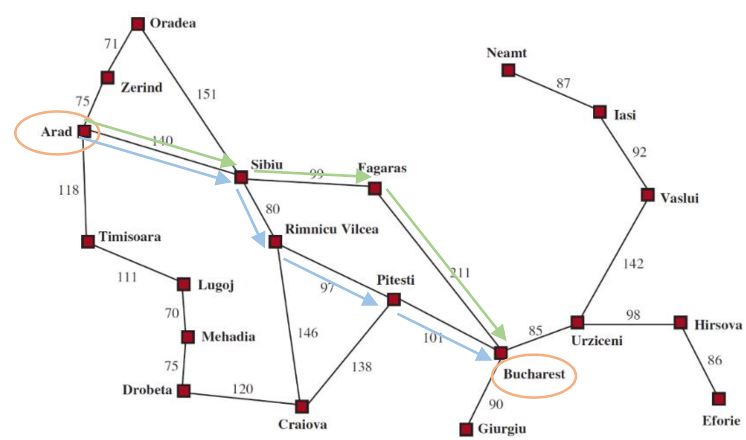


Example: Greedy Search

b) After expanding Sibiu

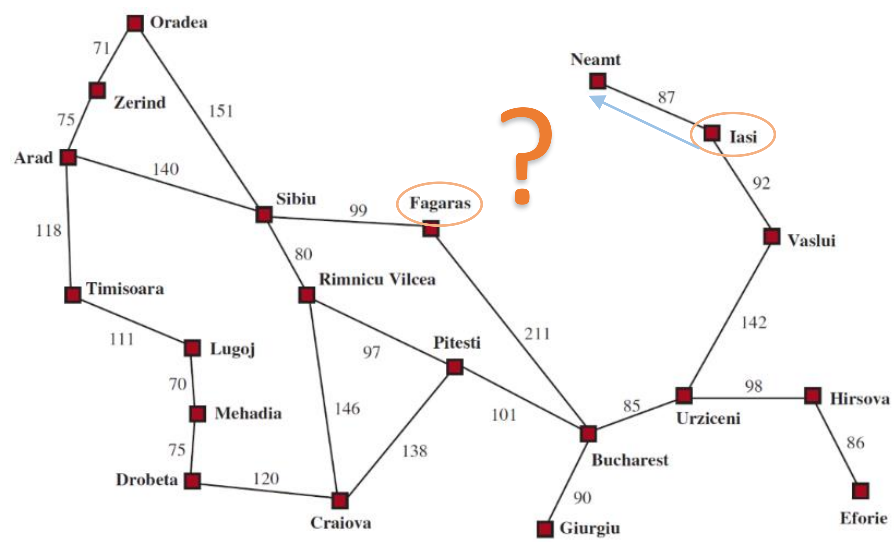


- **Procedure** (cont'd)
  - If Fagaras is expanded we get: Sibiu and Bucharest
  - Goal reached!
    - Yet NOT optimal
    - (cf. Arad, Sibiu, R.V., Pitesti)



Greedy Search

- **Completeness? No**
  - cf. DFS
  - Start down an infinite path and never return
  - Check on repeated states
  - Minimizing h(n) can result in false starts
    e.g. Iasi to Fagaras
- **Optimality? No**
  - Same as DFS



Greedy Search

- **Time complexity:** $O(b^m)$
  - cf. Worst-case DFS
    - (with m is maximum depth of search space)
  - Good heuristic can give dramatic improvement
- **Space complexity:** $O(b^m)$
  - Keeps all nodes in memory

A* Search

- **A* search**
  - Best-known form of best-first search
  - Idea: avoid expanding paths that are already expensive
  - Evaluation function

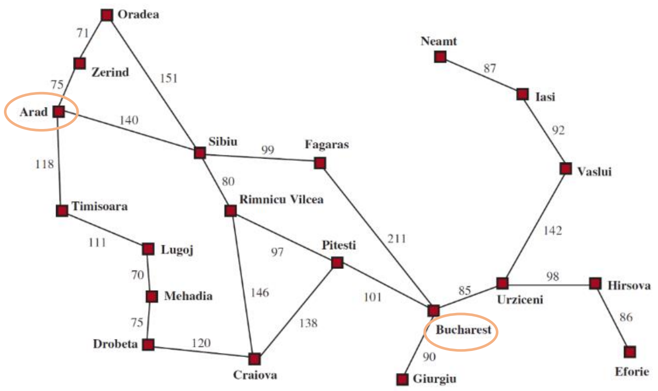
$$f(n) = g(n) + h(n)$$

    - $g(n)$: real cost (so far) to reach the node
    - $h(n)$: estimated cost to get from the node to the goal
    - $f(n)$: estimated total cost of path through node n to goal

A* Search

- **A* search uses an admissible heuristic**
  - A heuristic is admissible if it never overestimates the cost to reach the goal
    - e.g. $h_{SLD}(n)$ never overestimates the actual road distance
  - Are optimistic
  - Formally:
    - $h(n) \leq h^*(n)$where $h^*(n)$ is the true cost from n
    - $h(n) \geq 0 \; so \; h(G) = 0$ for any goal G

Example: A* Search



a) Initial state

Arad

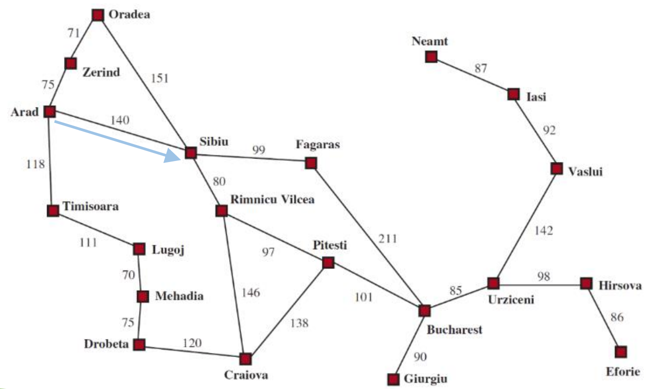
0+366=366

- **Scenario**
  - **Find Bucharest starting at Arad**
  - **Initial state = Arad**
- **Procedure**
  - $\boldsymbol{f(Arad)}$

$$\boldsymbol{= g(Arad, Arad) + h(Arad)}$$
$$\boldsymbol{= 0 + 366 = 366}$$

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

Example: A* Search



b) After expanding Arad

Arad

Sibiu — 140+253=393

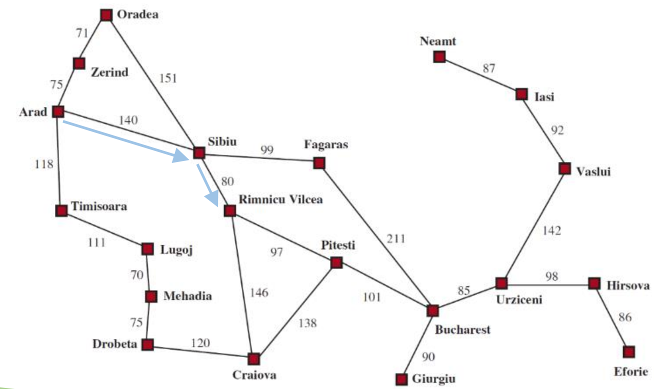
Timisoara — 118+329=447

Zerind — 75+374=449

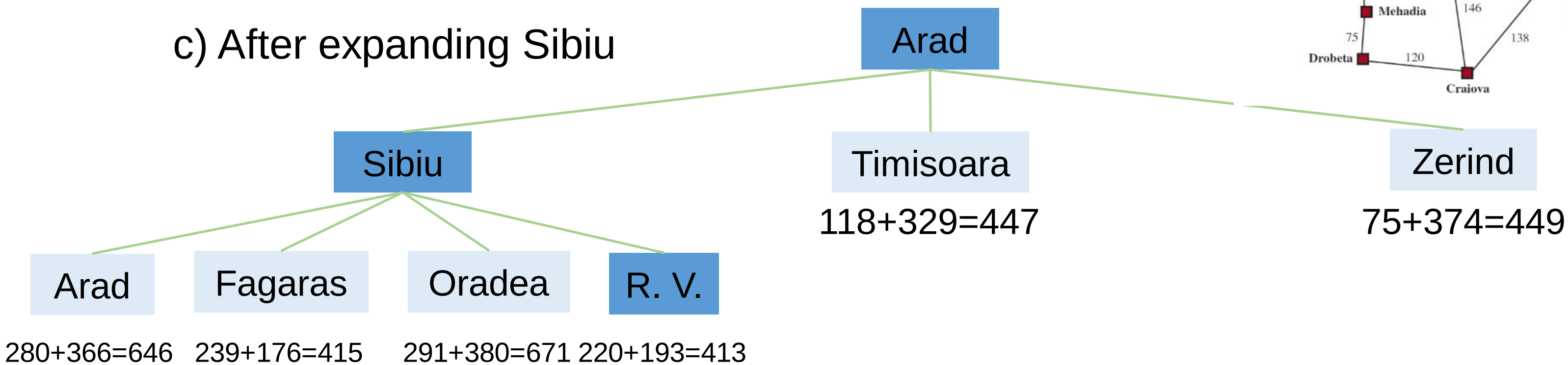
- **Expand Arad and determine $f(n)$ for each node**
  - $f(s) = g(A,S) + h(S) = 140 + 253 = 393$
  - $f(T) = g(A,T) + h(T) = 118 + 329 = 447$
  - $f(Z) = g(A,Z) + h(Z) = 75 + 374 = 449$
- **Best choice is Sibiu**

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

Example: A* Search



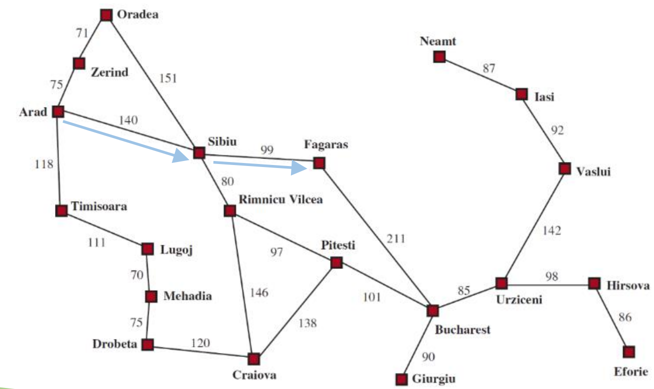
c) After expanding Sibiu



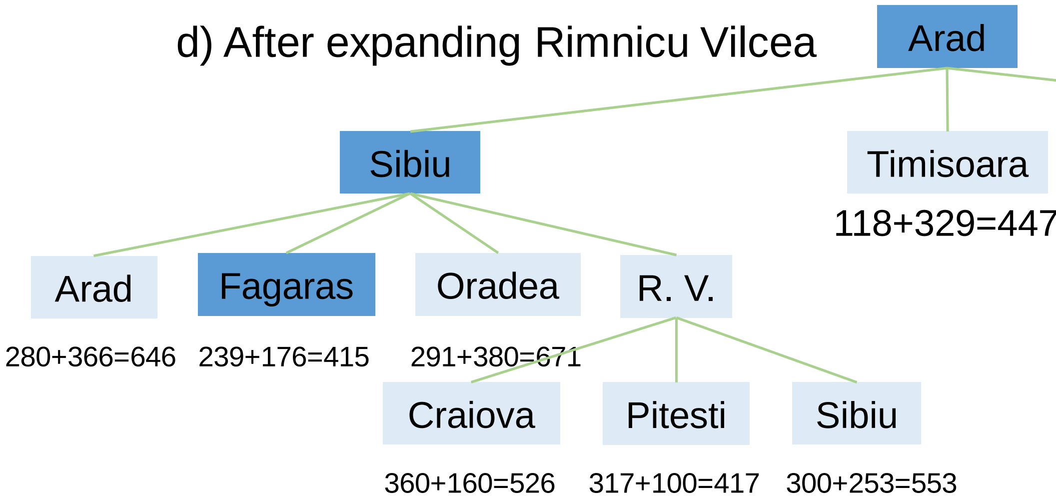
- **Expand Sibiu and determine $f(n)$ for each node**
  - $f(A) = g(S, A) + h(A) = 280 + 366 = 646$
  - $f(F) = g(S, F) + h(F) = 239 + 176 = 415$
  - $f(O) = g(S, O) + h(O) = 291 + 380 = 671$
  - $f(RV) = g(S, RV) + h(RV) = 220 + 193 = 413$
- **Best choice is Rimnicu Vilcea**

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

Example: A* Search



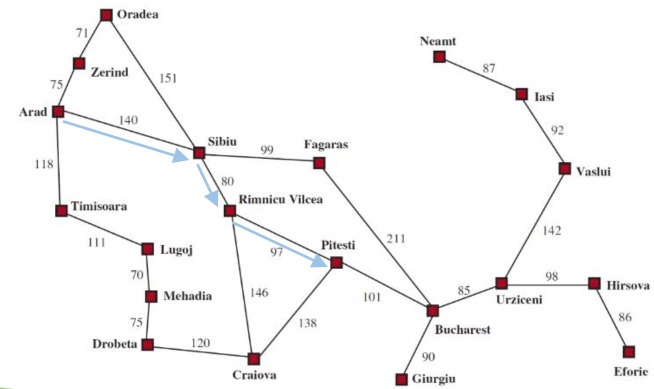
d) After expanding Rimnicu Vilcea



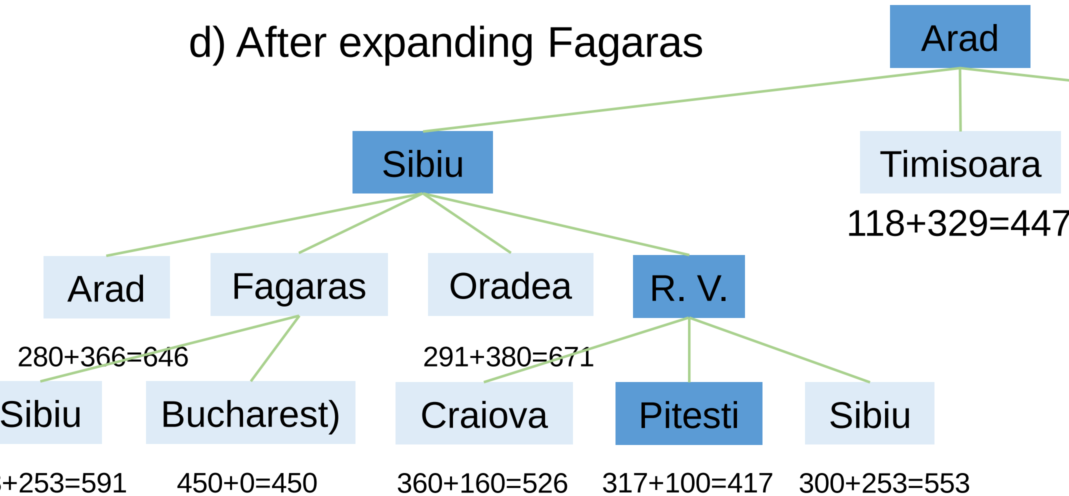
- **Expand Rimnicu Vilcea and determine $f(n)$ for each node**
  - $f(C) = g(RV, C) + h(C) = 360 + 160 = 526$
  - $f(P) = g(RV, P) + h(P) = 317 + 100 = 417$
  - $f(S) = g(RV, S) + h(S) = 300 + 253 = 553$
- **Best choice is Fagaras**

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

Example: A* Search



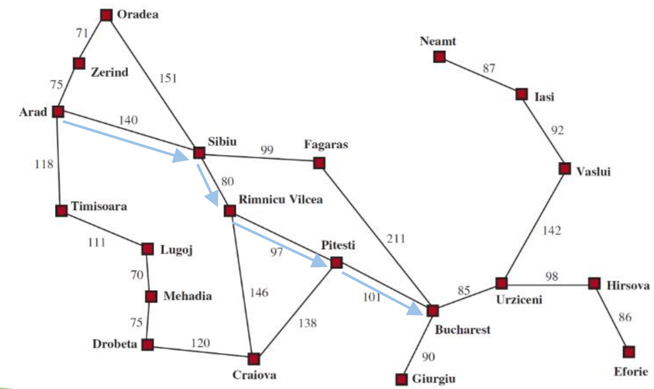
d) After expanding Fagaras



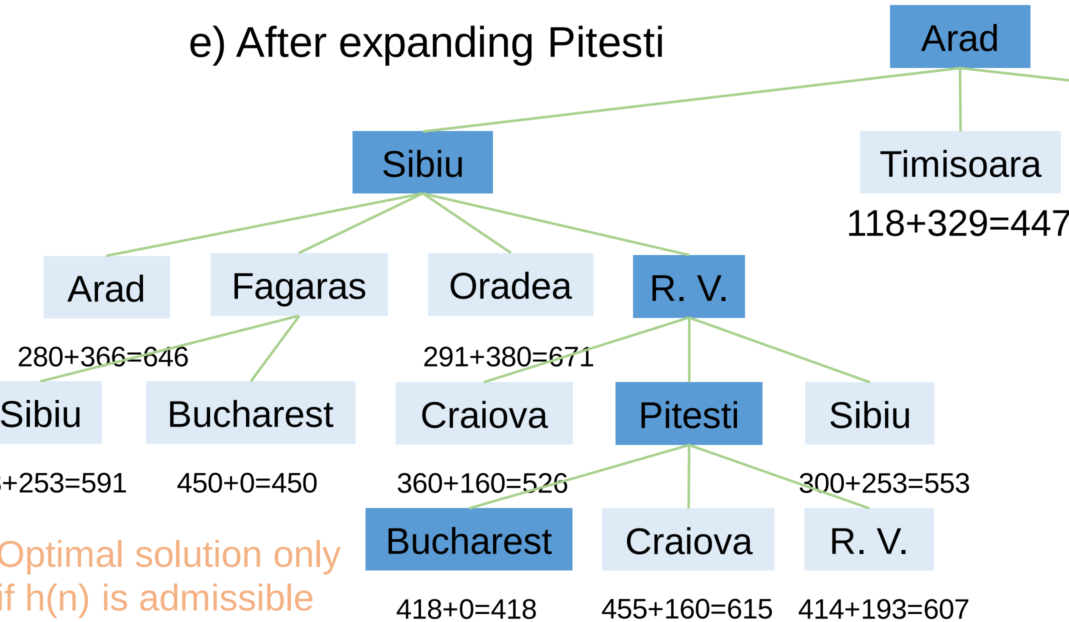
City	h_{SLD}	City	h_{SLD}
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

- **Expand Rimnicu Vilcea and determine $f(n)$ for each node**
  - $f(S) = g(F,S) + h(S) = 338 + 253 = 591$
  - $f(B) = g(F,B) + h(B) = 450 + 0 = 450$
- **Best choice is Pitesti**

Example: A* Search



e) After expanding Pitesti



Optimal solution only if h(n) is admissible

- **Expand Pitesti and determine $f(n)$ for each node**
  - $f(B) = g(P, B) + h(B) = 418 + 0 = 418$
  - $f(C) = g(P, C) + h(C) = 455 + 160 = 615$
  - $f(RV) = g(P, RV) + h(RV) = 414 + 193 = 607$
- **Best choice is Bucharest**

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

A* Search

- **Completeness? Yes**
  - Since bands of increasing f are added
  - Unless there are infinitely many nodes with

$$f < f(G)$$

- **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 $f(n) > C^*$
  - Also optimally efficient (not including ties)

A* Search

- **Time complexity:**
  - Number of nodes expanded is still exponential in the length of the solution
- **Space complexity:**
  - It keeps *all* generated nodes in memory
  - Hence space is the major problem, not time

Memory-Bounded Heuristic Search

- **Some solutions to A* space problems (maintain completeness and optimality)**
  - Iterative-deepening A* (IDA*)
  - Here cutoff information is the f-cost (g+h) instead of depth
  - Recursive best-first search (RBFS)
  - Recursive algorithm that attempts to mimic standard best-first search with linear space
  - (Simple) Memory-bounded A* ((S)MA*)
    - Drop the worst-leaf node when memory is full

Recursive Best-First Search

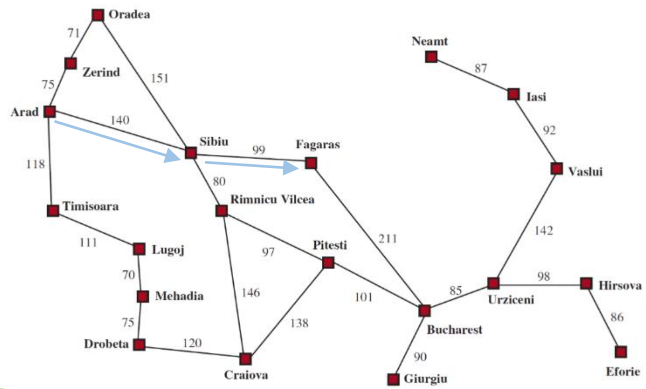
- **Keeps track of the f-value of the best-alternative path available**
  - Idea: change mind timely
  - If current f-values exceeds this alternative f-value than backtrack to alternative path
  - Upon backtracking change f-value to best f-value of its children
  - Re-expansion of this result is thus still possible

Recursive Best-First Search

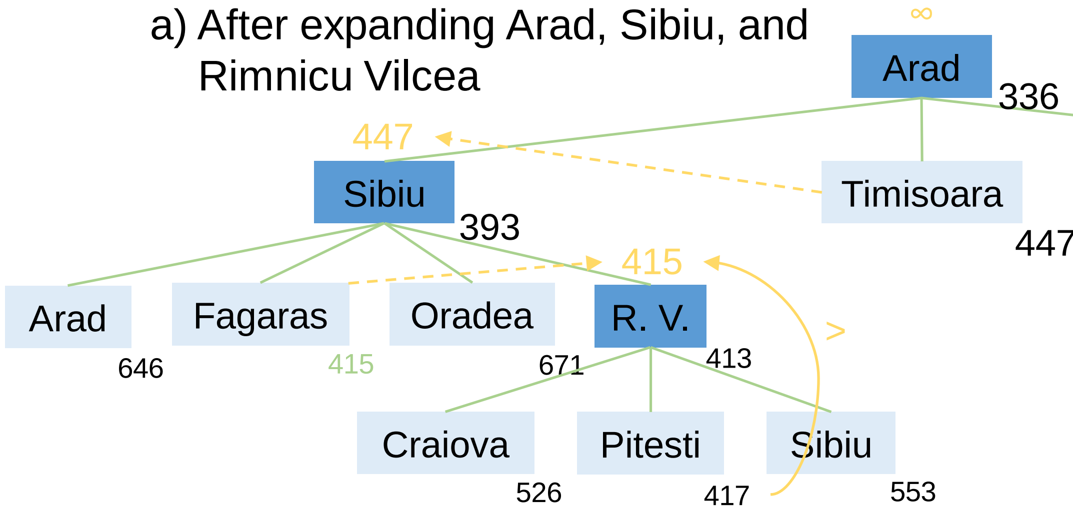
function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **return** a solution, or failure
  return RBFS(problem,MAKE-NODE(INITIAL-STATE[problem]),∞)

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

Example: RBFS



a) After expanding Arad, Sibiu, and Rimnicu Vilcea

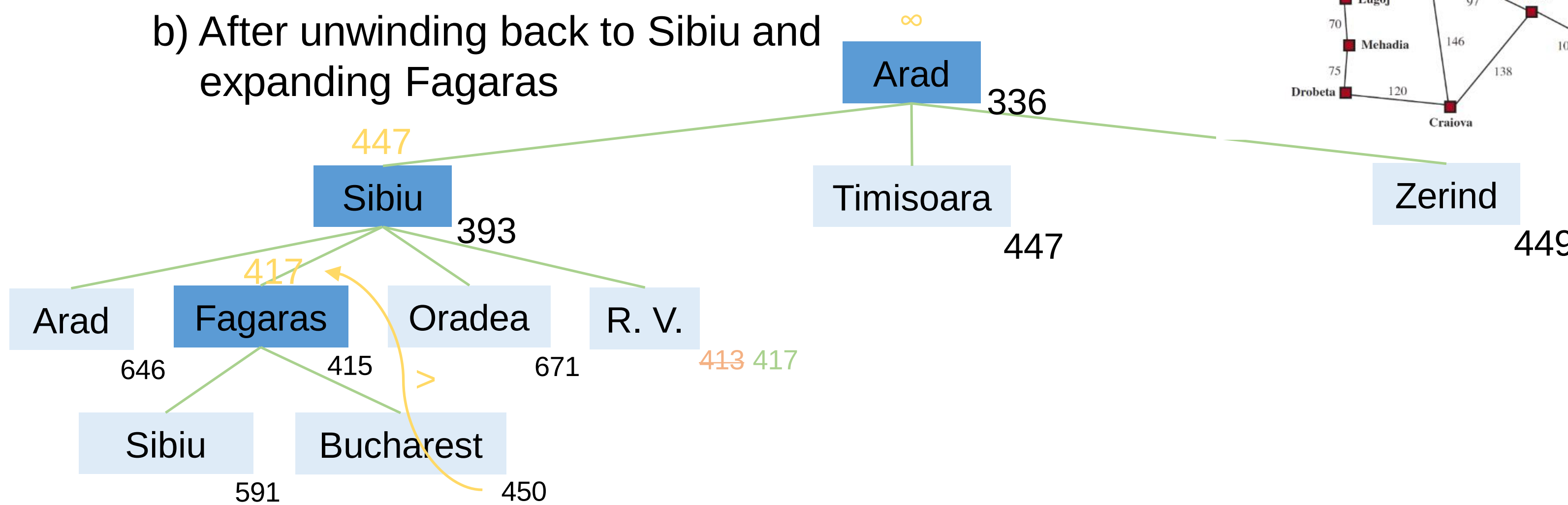


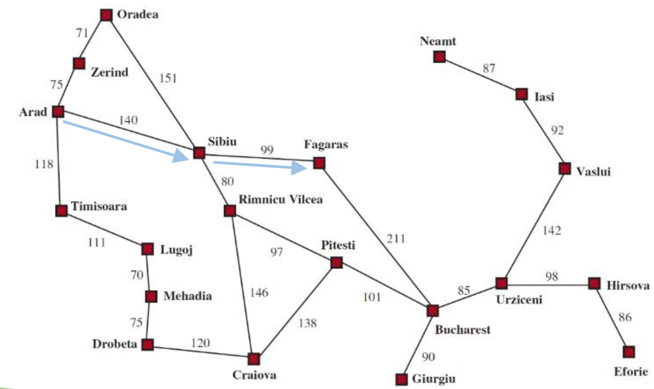
- **Path until Rumnicu Vilcea is already expanded**
  - **Above node: f_{limit} for every recursive call is shown on top**
  - **Below node: $f(n)$**
  - **The path is followed until Pitesti which has a f-value worse than the f-limit**

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

Example: RBFS

b) After unwinding back to Sibiu and expanding Fagaras





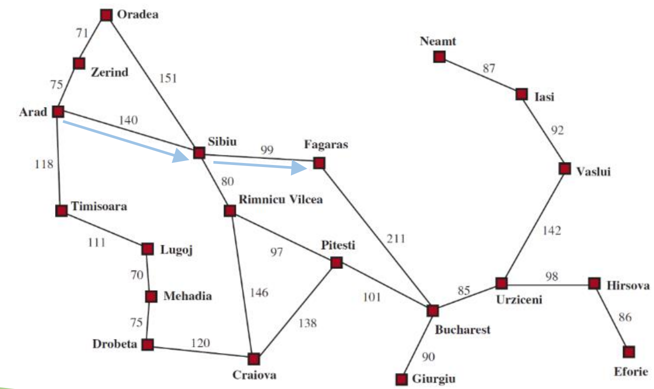
City	h_{SLD}	City	h_{SLD}
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

- **Unwind recursion and store best f-value for current best leaf Pitesti**

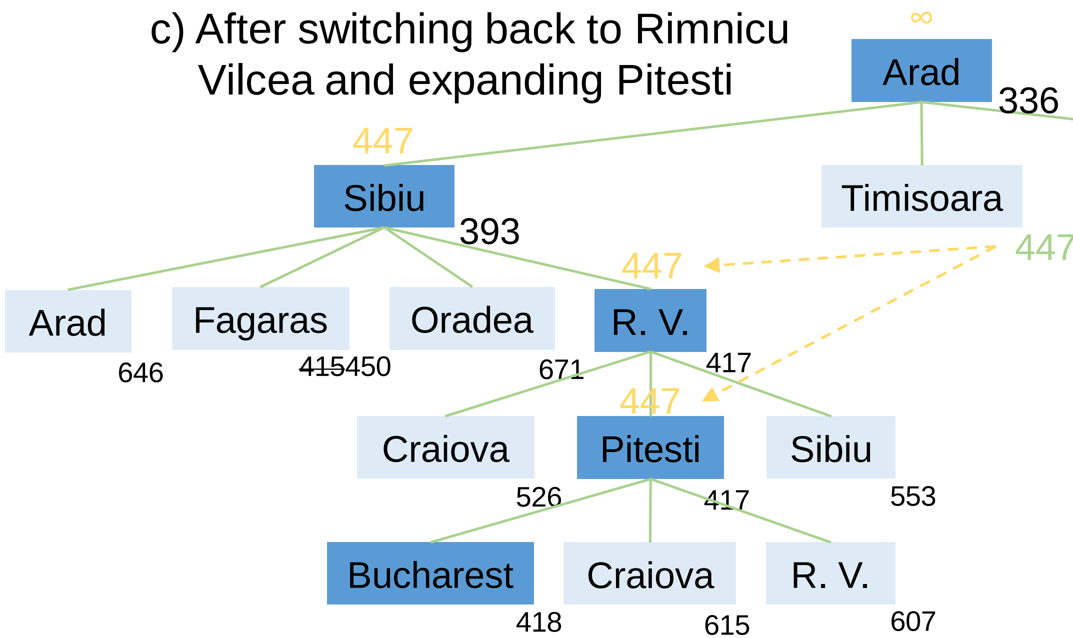
  result, *f* [best] ← RBFS(*problem*, *best*, min(f_{limit}, *alternative*))

  - **best is now Fagaras. Call RBFS for new best (best value is now 450)**

Example: RBFS



c) After switching back to Rimnicu Vilcea and expanding Pitesti



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

- **Unwind recursion and store best f-value current best leaf Fagaras**
  - **best is now Rimnicu Viclea (again). Call RBF for new best**
    - **Subtree is again expanded**
    - **Best alternative subtree is now through Timisoara**
  - **Solution is found since 447 > 417**

Recursive Best-First Search

- **Completeness: Yes**
- **Optimality: Yes**
  - Like A*, RBFS is optimal if $h(n)$ is admissible
- **Time complexity:**
  - Depends on accuracy of $h(n)$ and how often best path changes
- **Space complexity:** $O(bd)$

RBFS vs. IDA* (Iterative Deepening A*)

- **RBFS is a bit more efficient than IDA*** (Iterative Deepening A*)
  - Still excessive node generation (mind changes)
  - IDA* retains only one single number (the current f-cost limit)
  - IDA* and RBFS suffer from too little memory

(Simplified) Memory-Bounded A*

- **Use all available memory**
  - i.e. expand best leaves until available memory is full
  - When full, SMA* drops worst leaf node (highest f-value)
  - Like RFBS backup forgotten node to its parent
- **What if all leaves have the same f-value?**
  - Same node could be selected for expansion and deletion
  - SMA* solves this by expanding newest best leaf and deleting oldest worst leaf
- **SMA* is complete if solution is reachable, optimal if optimal solution is reachable**

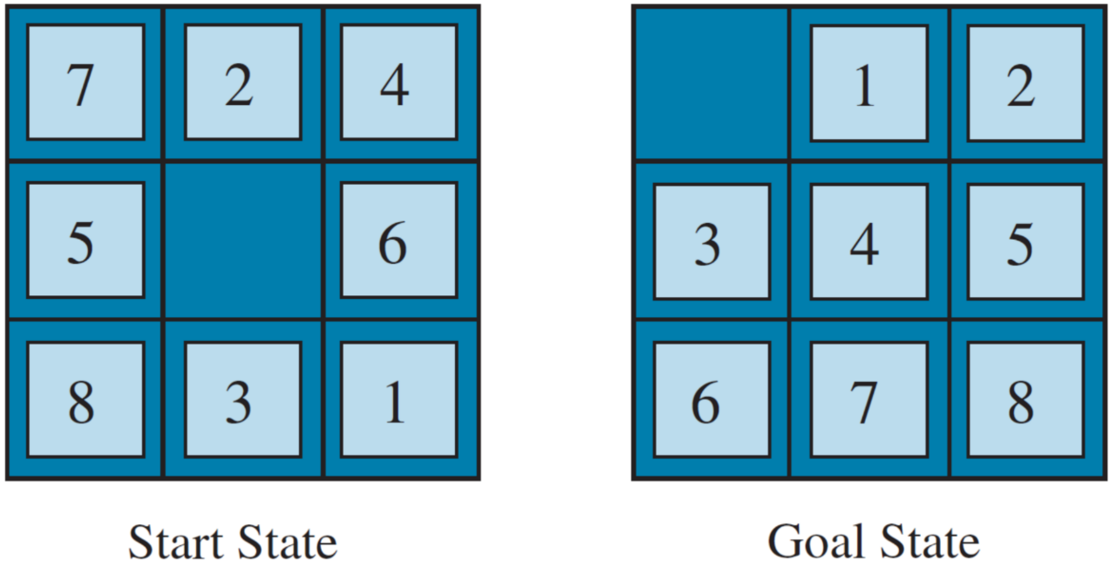
Learning to Search Better

- Background: All previous algorithms use fixed strategies
- **Could agents learn how to search better?**
  - **YES**. Agents can learn to improve their search by exploiting the meta-level state space
    - Each meta-level state is a internal (computational) state of a program that is searching in the object-level state space
    - In A* such a state consists of the current search tree
  - A meta-level learning algorithm can learn from experiences to avoid exploring unpromising subtrees

Outline

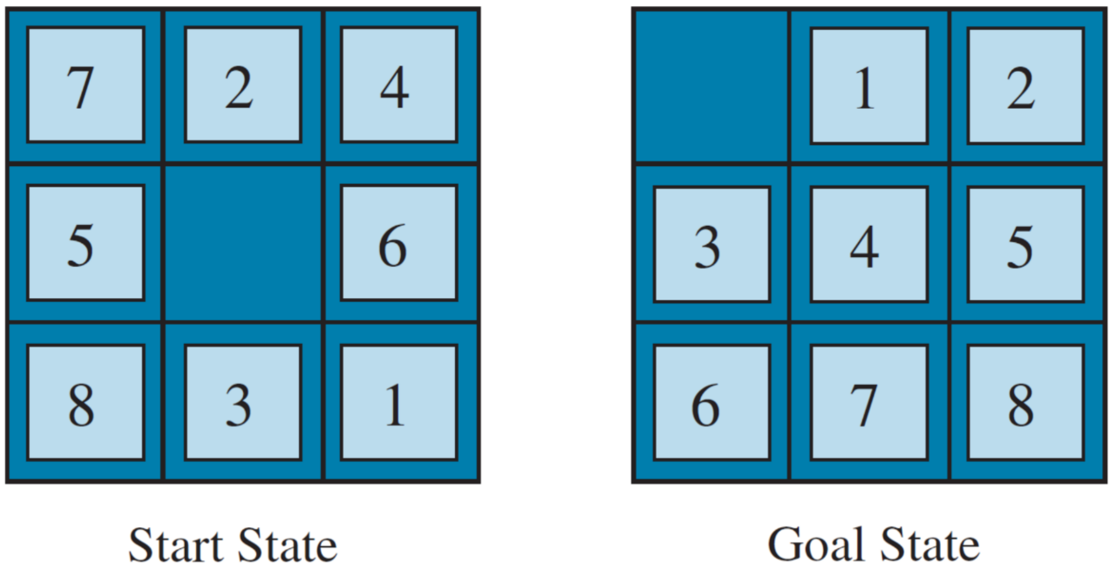
- **Heuristic (informed) search strategies**
- **Heuristic functions**
- **Local search algorithms & optimization problems**
- **Global search algorithm**
- **Online search agents & unknown environment**

Heuristic Functions



- **The 8-puzzle problem**
  - Avg. solution cost is about 22 steps (branching factor ≈ 3)
  - Exhaustive search to depth 22: $\mathbf{3.1 \times 10^{10}}$ states
  - A good heuristic function can reduce the search process

Heuristic Functions



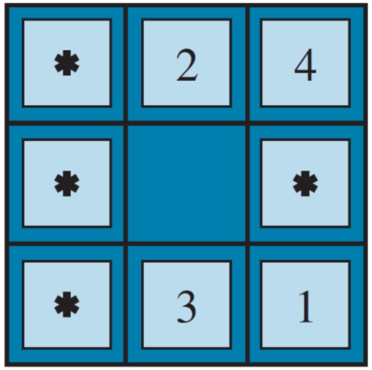
- **Two common heuristics for 8-puzzle**
  - h_1 = the number of misplaced tiles:
    $h_1(s) = 8$
  - h_2 = the sum of the distances of the tiles from their goal positions (Manhattan distance):
    $h_2(s) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$

Inventing Admissible Heuristics

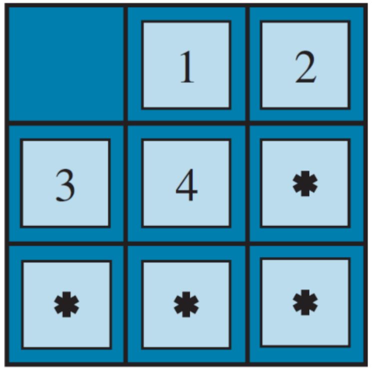
- **Admissible heuristics**
  - Required by A*, RBFS, etc.
- **Can be derived from the exact solution cost of a relaxed version of the problem:**
  - Relaxed 8-puzzle for h_1: a tile can move anywhere
    - As a result, $h_1(n)$ gives the shortest solution
  - Relaxed 8-puzzle for h_2: a tile can move to any adjacent square
    - As a result, $h_2(n)$ gives the shortest solution
  - The cost of optimal solution to a relaxed problem is no greater than the cost of optimal solution to the real problem

Inventing Admissible Heuristics

- **Can also be derived from the solution cost of a subproblem of a given problem**
  - A lower bound on the cost of the real problem
  - Pattern databases store the exact solution to for every possible subproblem instance
  - The complete heuristic is constructed using the patterns in the database



Start State



Goal State

Inventing Admissible Heuristics

- **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

Heuristic Quality

- **How good is a heuristic?**
- **A way is effective branching factor *b****
  - which is the branching factor that a uniform tree of depth d would have, in order to contain N + 1 nodes

$$N + 1 = 1 + b^* + {b^*}^2 + \cdots + {b^*}^d$$

  - 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

- Performance comparison
  - On 1200 random problems with solution lengths from 2 to 24
  - Dominance: If $h_2(n) >= h_1(n)$ for all n (both admissible) then h_2 dominates h_1 and is better for search

	Search Cost (nodes generated)			Effective Branching Factor		
d	BFS	$A^*(h_1)$	$A^*(h_2)$	BFS	$A^*(h_1)$	$A^*(h_2)$
6	128	24	19	2.01	1.42	1.34
8	368	48	31	1.91	1.40	1.30
10	1033	116	48	1.85	1.43	1.27
12	2672	279	84	1.80	1.45	1.28
14	6783	678	174	1.77	1.47	1.31
16	17270	1683	364	1.74	1.48	1.32
18	41558	4102	751	1.72	1.49	1.34
20	91493	9905	1318	1.69	1.50	1.34
22	175921	22955	2548	1.66	1.50	1.34
24	290082	53039	5733	1.62	1.50	1.36
26	395355	110372	10080	1.58	1.50	1.35
28	463234	202565	22055	1.53	1.49	1.36

Outline

- **Heuristic (informed) search strategies**
- **Heuristic functions**
- **Local search algorithms & optimization problems**
- **Online search agents & unknown environment**

Optimization Problem

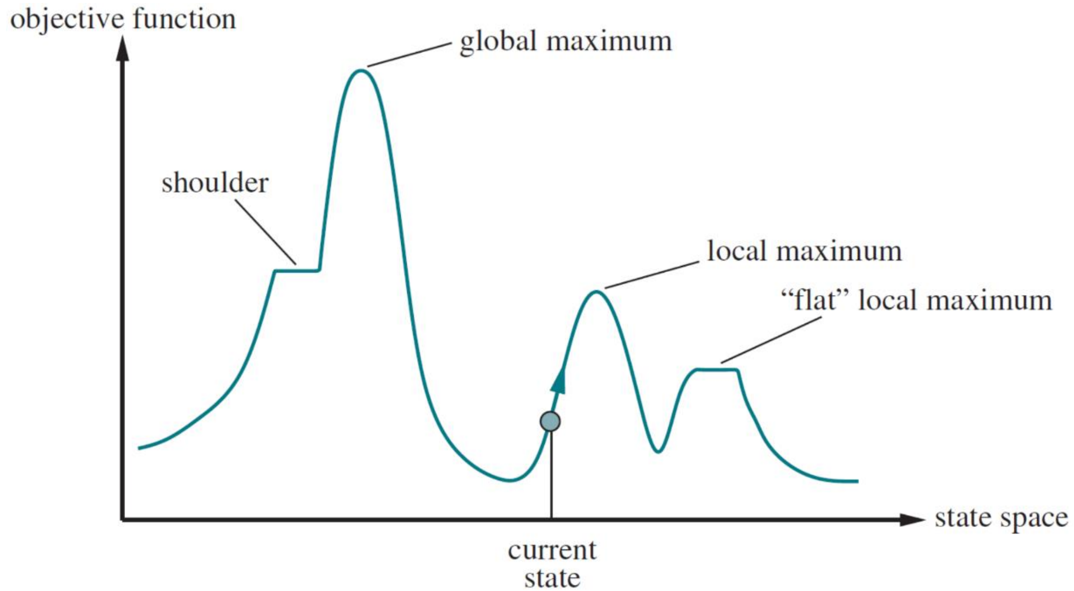
- Definition (minimization problem)
  - Given a search space Ω which represents feasible solutions
  - Given an objective function $f: \Omega \rightarrow \mathbb{R}$
  - Find a solution $\boldsymbol{x}^* = \underset{\boldsymbol{x} \in \boldsymbol{\Omega}}{\operatorname{argmin}} f(\boldsymbol{x})$
    - $\boldsymbol{x}^*$ is known as global optimum
- Optimization problem can always be reduced to a decision problem.
  - E.g. The traveling salesman problem:
    - Optimization: Find an optimal Hamiltonian tour which optimizes the total distance.
    - Decision: Given a distance D, is there a Hamiltonian tour with a distance less than or equal to D?

Complexity Classes

- Class P: decision problems solved by a deterministic machine in polynomial time
- Class NP: decision problems solved by a non-deterministic algorithm in polynomial time
  - There is a short certificate for any solution which can be checked in polynomial time
- Class NP-hard: if all problems of the class NP can be reduced in polynomial time to the problem
- NP-complete: problems in NP class and NP-hard class
- Important open question: P = NP?

Problem Solving as Optimization

- Optimization problem
  - The aim is to find the best state according to an objective function
  - State space landscape
    - Very useful for understanding local search and the problem

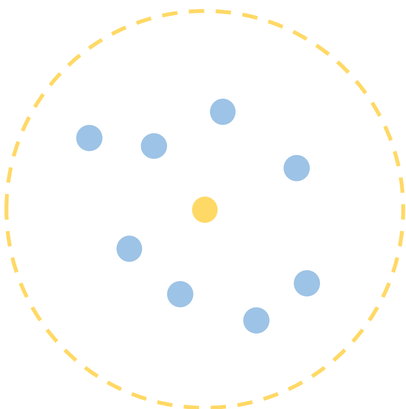


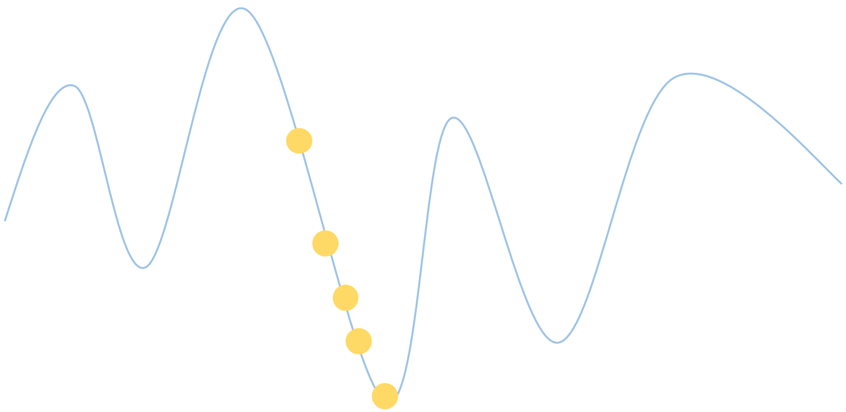
Problem Solving as Optimization

- **Completeness**
  - A complete algorithm always finds a goal if one exists
- **Optimality**
  - An optimal algorithm always finds a global minimum (or maximum)

Local Search

- A.k.a **single solution-based search**
  - “Improvement”of a single solution
  - Generate-and-test
    - Use single *current* state and **move** to *neighboring* states
    - Walks through neighborhoods or search trajectories
  - Exploitation Oriented:
    - Iterative exploration of the neighborhood. (intensification)





Template of local search

Input: Initial solution s_0.

Output: Best solution found

1 $t \leftarrow 0$;

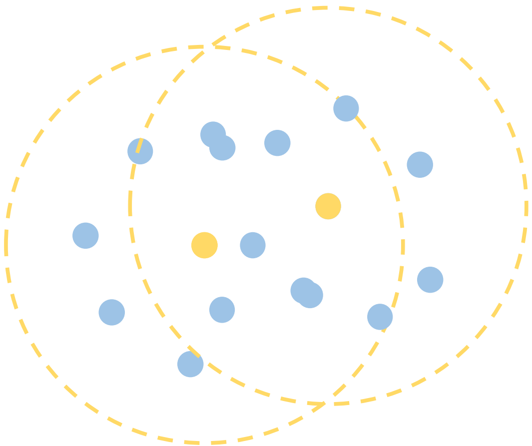
2 Repeat

3 $\quad C_t \leftarrow$Generate(s_t) ; ◂ Generate candidates C_t from s_t

4 $\quad s_{t+1}$ = Select(C_t) ; ◂ Select a solution in C_t to replace s_t

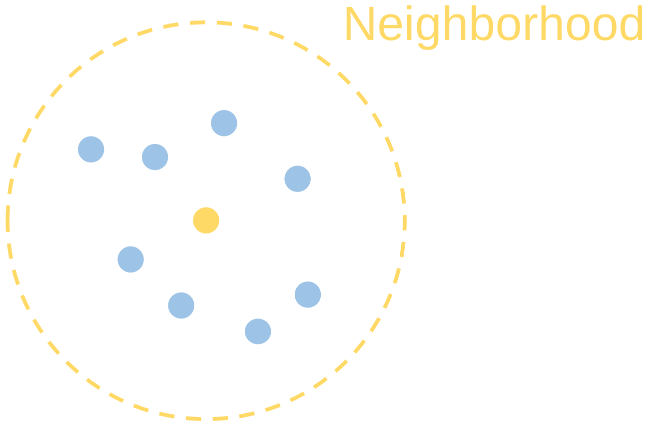
5 $\quad t \leftarrow t + 1$

6 Until stopping criteria satisfied



Elements of local search

- Representation of the solution
- Evaluation function
- Neighborhood function
  - To define solutions which can be considered close to a given solution
- Neighborhood search strategy
  - Random search
  - Systematic search
- Acceptance criterion
  - First improvement
  - Best improvement
  - Random



Features of local search

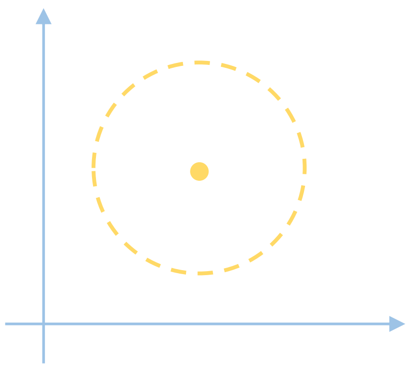
- **Advantages**
  - Use very little memory
  - Find often reasonable solutions in large or infinite state spaces
  - Guarantee of local optimality in little computational time
- **Disadvantages**
  - No guarantee of global optima
  - Poor quality of solution due to getting stuck in poor local optima

Neighborhoods

- **Neighborhood.** *A neighborhood function* N *is a mapping* $N: S \rightarrow 2^S$ *that assigns to each solution* s *of* S *a set of solutions* $N(s) \subset S$*.*
  - A solution s' is a neighbor of s iff $s' \in N(s)$
  - Defined by a *move* operator
- **Locality**
  - The effect on the solution (phenotype) when performing a move (small perturbation) to the representation (genotype)
  - Strong locality: small changes on representation → small changes on solution
  - Weak locality may lead to random search

Neighborhoods

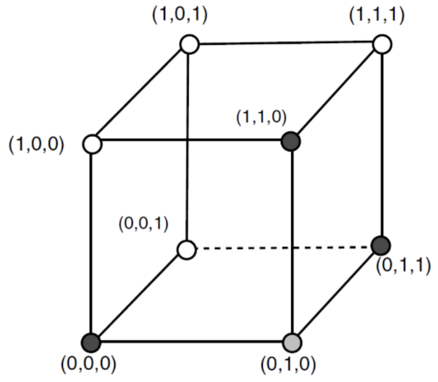
- *The neighborhood* $N(s)$ *of a solution* s *in a continuous space is the ball with center* s *and radius equal to* ϵ *with* $\epsilon > 0$:
$$N(s) = \{s' \in \mathbb{R}^n \mid \|s' - s\| < \epsilon\}$$
- If objective function is continuous and differentiable
  - Gradient descent with different step size
- Otherwise,
  - Gaussian perturbation $N(0, \sigma_i)$ can be considered



The circle represents the neighborhood of s in a continuous problem with two dimensions.

Neighborhoods

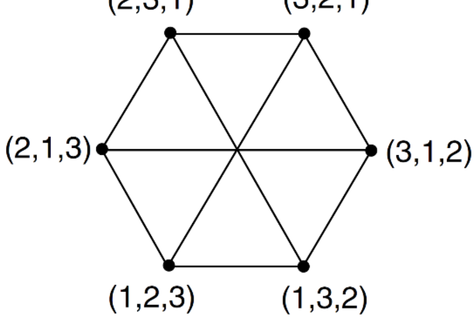
- *In a discrete optimization problem, the neighborhood $N(s)$ of a solution s is represented by the set $\{s' \in S | d(s', s) < \epsilon\}$, where* d *represents a given distance that is related to the move operator.* E.g.,
  - Hamming distance for binary representation
    - $|N(S)| = n$
    - Can be extended to any discrete vector defined by some alphabet Σ with $|N(S)| = n(k-1)$, $\mathrm{k} = |\Sigma|$



➢ Nodes of the hypercube represent solutions of the problem.
➢ The neighbors of a solution (e.g., (0,1,0)) are the adjacent nodes in the graph.

Neighborhoods

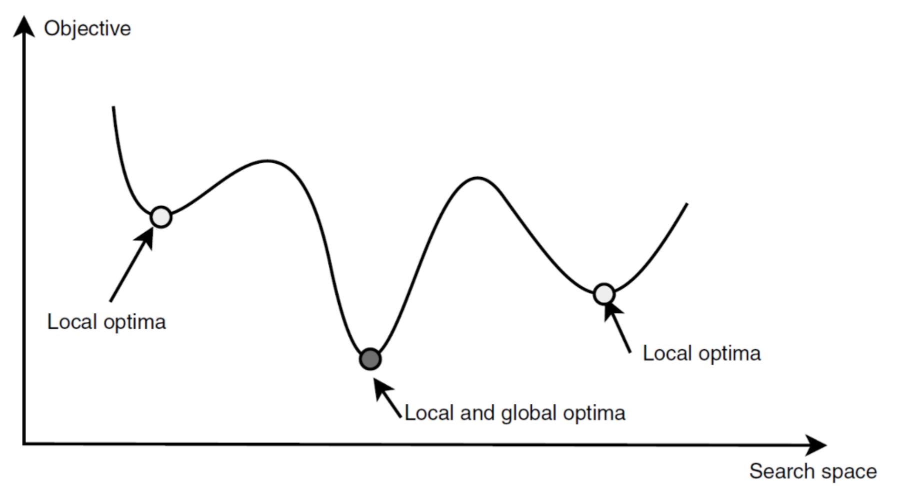
- Permutation
  - Defined by permutation-based operators
    - E.g., swap, inversion, insertion, …
  - Scheduling problems
    - Permutation represents a priority.
    - The relative order in the sequence is important (2-opt → weak locality).
  - Routing problems
    - adjacency of the element is important (2-opt → strong locality).



An example of neighborhood for a permutation problem of size 3. For instance, the neighbors of the solution (2, 3, 1) are (3, 2, 1), (2, 1, 3), and (1, 3, 2).

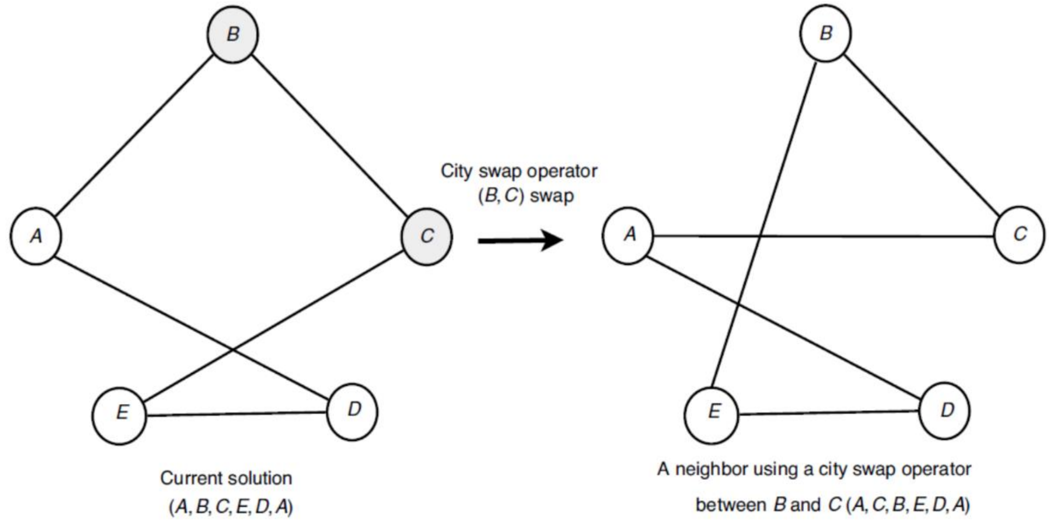
Local Optimum

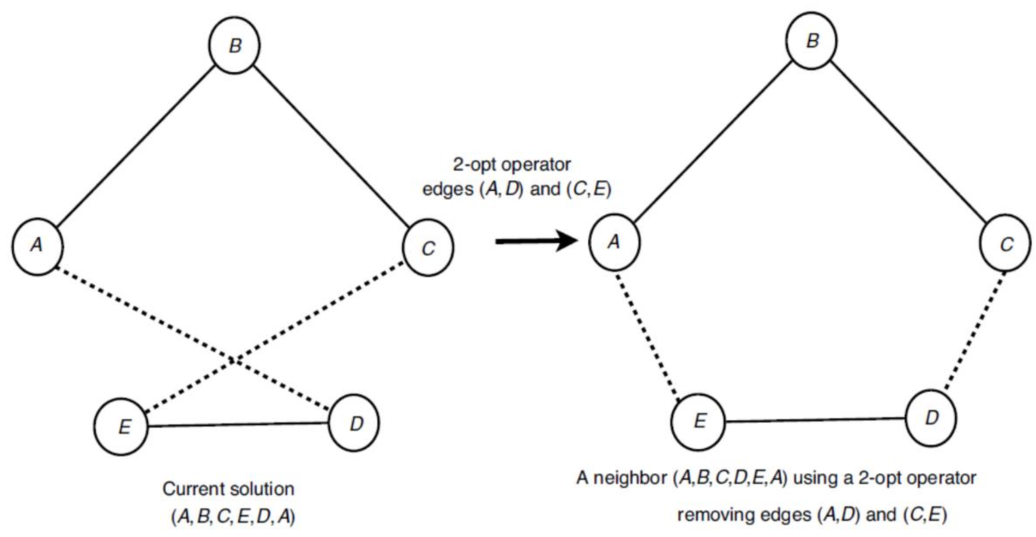
- **Local optimum.** *Relatively to a given neighboring function N, a solution $s \in S$ is a local optimum if it has a better quality than all its neighbors; that is, $f(s) \leq f(s') \; \forall s' \in N(s)$.*
  - Local optimum for a neighborhood N_1 may not be a local optimum for another neighborhood N_2



Local optimum and global optimum in a search space.

k-distance versus k-exchange neighborhoods





Permutation Neighborhoods

- Position-based neighborhoods
  - Insertion
- Order-based neighborhoods
  - Swap
  - Inversion

Permutation Neighborhoods

- Insertion
  - Preserves most of the order and the adjacency information
  - Procedure:
    1. Pick two allele values at random
    2. Move the second to follow the first, shifting the rest along to accommodate

Permutation Neighborhoods

- Swap
  - Most widely used
  - Preserves most of adjacency information (4 links broken), disrupts order more
  - Procedure:
    - Pick two alleles at random and swap their positions

Permutation Neighborhoods

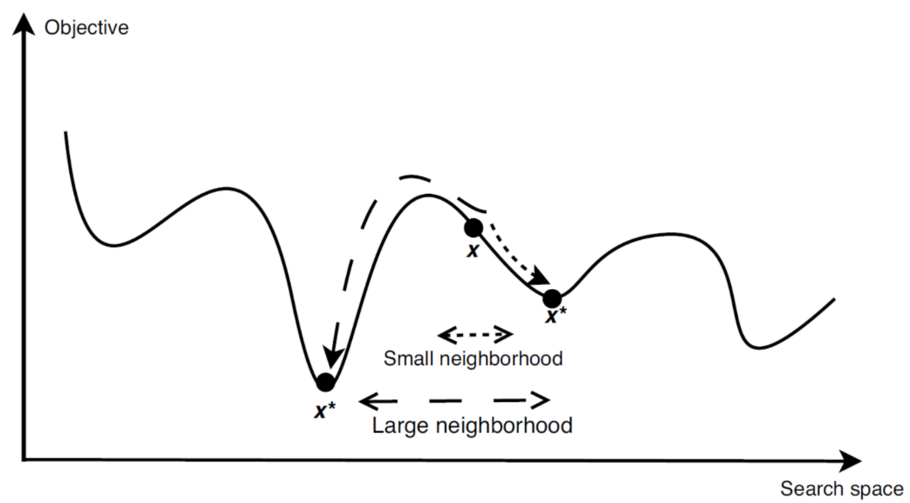
- Scramble
  - Procedure:
    1. Pick a subset of genes at random
    2. Randomly rearrange the alleles in those positions

Permutation Neighborhoods

- Inversion
  - Preserves most adjacency information (only breaks two links) but disruptive of order information
  - Procedure:
    - Pick two alleles at random and then invert the substring between them.

Neighborhood size

- Neighborhood size = #neighbors
- Compromise between
  - Size of the neighborhood (computational complexity)
    - Usually linear or quadratic
  - Quality of solutions



Design efficient procedures to explore large neighborhoods!

Very large neighborhood size

- Size of the neighborhood: high-order polynomial (n>2) or exponential to the problem size
- Main issue: identify improving neighbors or the best neighbor without enumeration of the whole neighborhood

Neighborhood search strategy

- Systematic search
  - Search the whole neighborhood
- Random (partial) search
  - Search partial set of the neighborhood
  - Finding the best neighbor (local optimum) is **not** guaranteed
  - E.g.,
    - Lin–Kernighan (LK) heuristic the TSP
      - Generalized k-opt (variable depth)
      - Only consider nearest (top 5) cities

Initial Solution

- Trade-off: quality vs. computational time
  - Random solution
  - Heuristic solution (e.g. Greedy)
- Using better initial solutions will not always lead to better local optima.
- Hybrid strategy for improving the robustness
- Generating random solutions may be difficult for highly constrained problems
- Partially or completely initialized by a user (e.g. expert) for real-world application

Evaluation of the neighborhood

- Evaluation
  - Most expensive part of a metaheuristic
  - Naive evaluation
    - Direct evaluation of $f(s')$ using objective function f
    - Complete evaluation of each solution in the neighborhood
  - Incremental evaluation
    - An essential issue in design single solution-based metaheuristics
    - Evaluate the difference $\Delta(s, m)$ only
      $f(s') = f(s \oplus m) = f(s) + \Delta(s, m)$
      - s: current solution
      - m: applied move
  - Approximated evaluation
    - Approximate $f(s')$ by calling its surrogates g
    - Trade-off: complexity versus accuracy

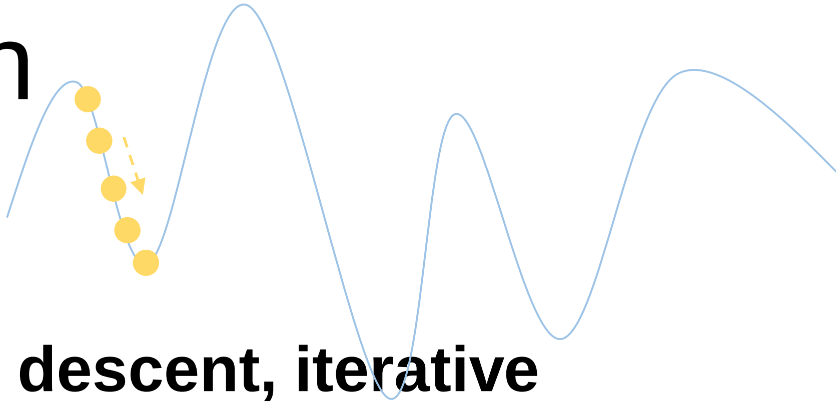
Acceptance Criterion

- First improvement
  - Replace the current solution with the first improved solution in neighborhood
  - **Deterministic** and **partial** search
- Best improvement
  - Replace the current solution with the best solution in neighborhood
  - **Deterministic** and **fully** search
- Random selection
  - Replace the current solution with a random better solution in neighborhood
  - **Stochastic** and **partial** search

Outline

- **Heuristic (informed) search strategies**
- **Heuristic functions**
- **Local search algorithms & optimization problems**
  - **Hill Climbing**
  - **Simulated Annealing**
  - **Local Beam Search**
- **Global search algorithm**
- **Online search agents & unknown environment**

Hill-Climbing Search



- **Hill-climbing (HC)**
  - a.k.a. **greedy local search, descent, iterative improvement**
  - is a loop that continuously moves in the direction of increasing value
    - It terminates when a peak is reached
    - Like climbing a hill
  - Chooses the **best** successors, but not look ahead of the immediate neighbors of the current state
    - Randomly chooses among the set of best successors, if there is more than one

Hill-Climbing Search

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

input: *problem*, a problem
local **variables**: *current*, a node.
neighbor, a node.

current ← MAKE-NODE(INITIAL-STATE[*problem*])

loop do

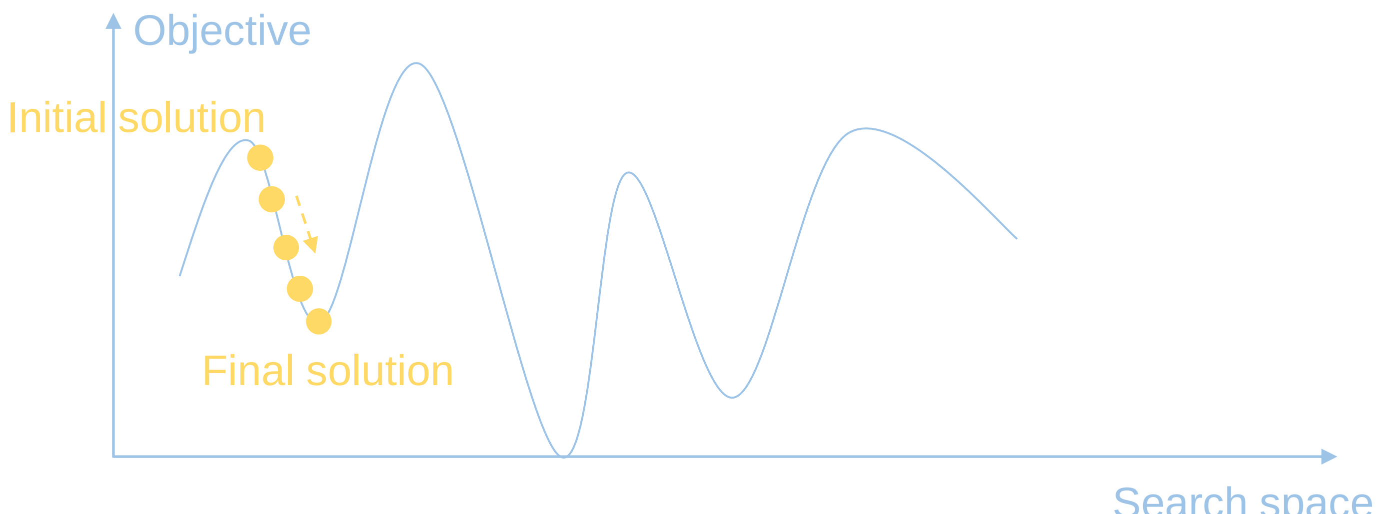
neighbor ← highest valued successor of *current*

if VALUE [*neighbor*] ≤ VALUE[*current*] **then**
return STATE[*current*]

current ← *neighbor*

HC: Towards a local optimum

- From a solution s_0, HC will generate a sequence $s_1, s_2, \dots, s_k$
  - Length of the sequence k is unknown a priori
  - $s_{i+1} \in N(s_i) \forall i \in \{0,1,\dots,k-1\}$
  - $f(s_{i+1}) < f(s_i) \forall i \in \{0,1,\dots,k-1\}$ ◄ Minimization
  - s_k is a local optimum: $f(s_k) \leq f(s) \forall s \in N(s_k)$

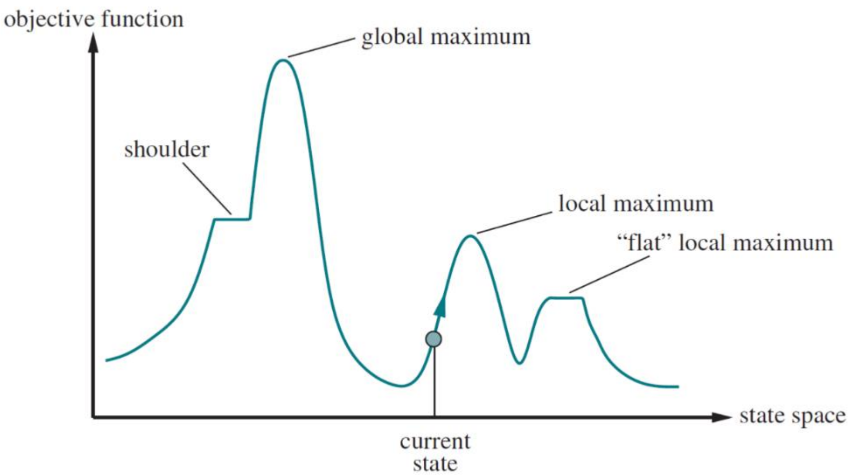


Pros and Cons

- Pros
  - Easy to implement
  - Acceptable time complexity
- Cons
  - It only leads to local optima.
  - The found optima depends on the initial solution.
  - No mean to estimate the relative error from the global optimum
  - No mean to have an upper bound of the computation time: the worst case is exponential

Pros and Cons

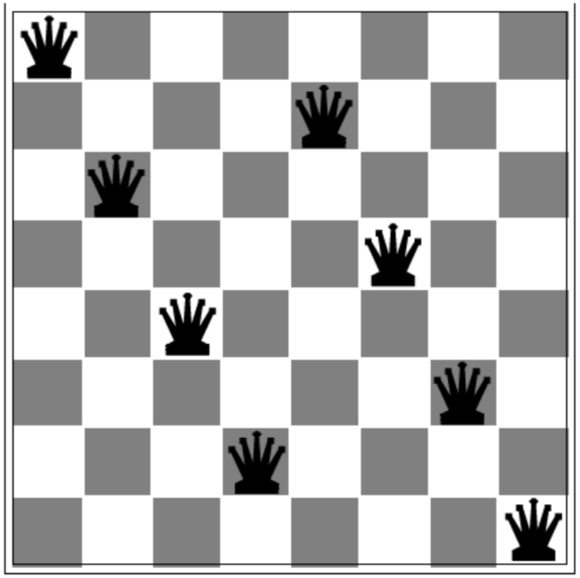
- **HC often gets stuck since**
  - Local maxima
    - A local max. is a peak higher than its neighbors, but lower than global max.
  - Ridge
    - Sequence of local max. that is difficult for greedy algorithms to navigate
  - Plateaus
    - An area of the state space where the evaluation function is flat



Hill-Climbing Variations

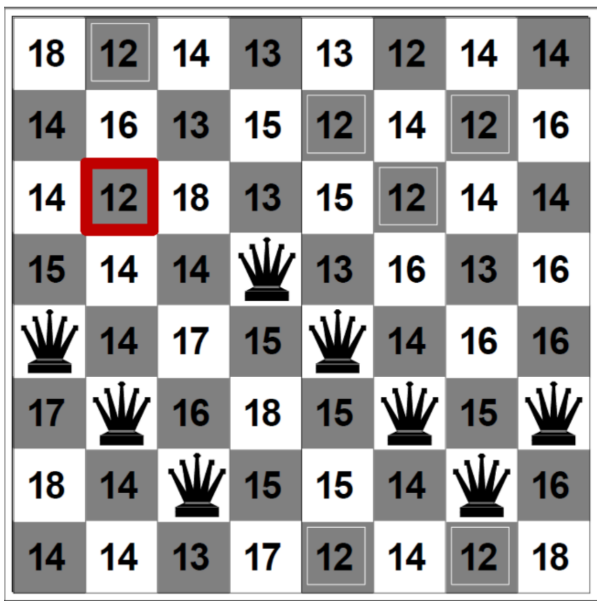
- **Stochastic hill-climbing**
  - Random selection among the uphill moves
  - The selection probability can vary with the steepness of the uphill move
- **First-choice hill-climbing**
  - cf. Stochastic hill climbing by generating successors randomly until a better one is found
- **Random-restart hill-climbing**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

Example: Hill-Climbing (1)

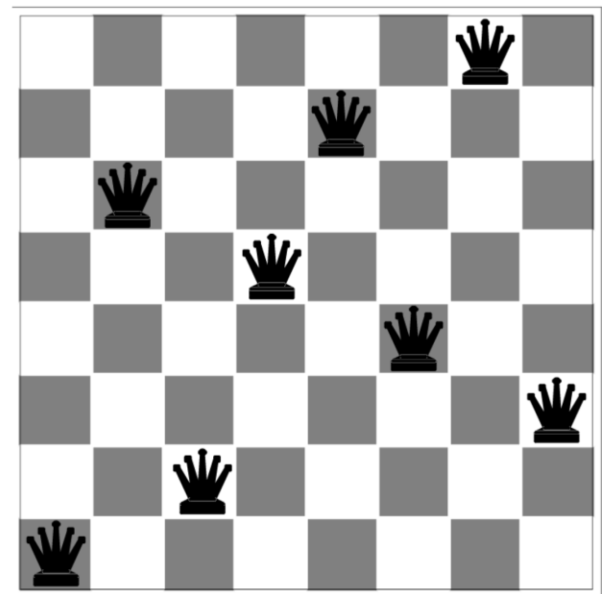


- **8-queens problem**
  - Complete-state formulation
  - Successor function: move a single queen to another square in the same column
  - Heuristic function $h(n)$: the number of pairs of queens that are attacking each other (directly or indirectly)

Example: Hill-Climbing (2)



A state of h=17 and the h-value for each possible successor

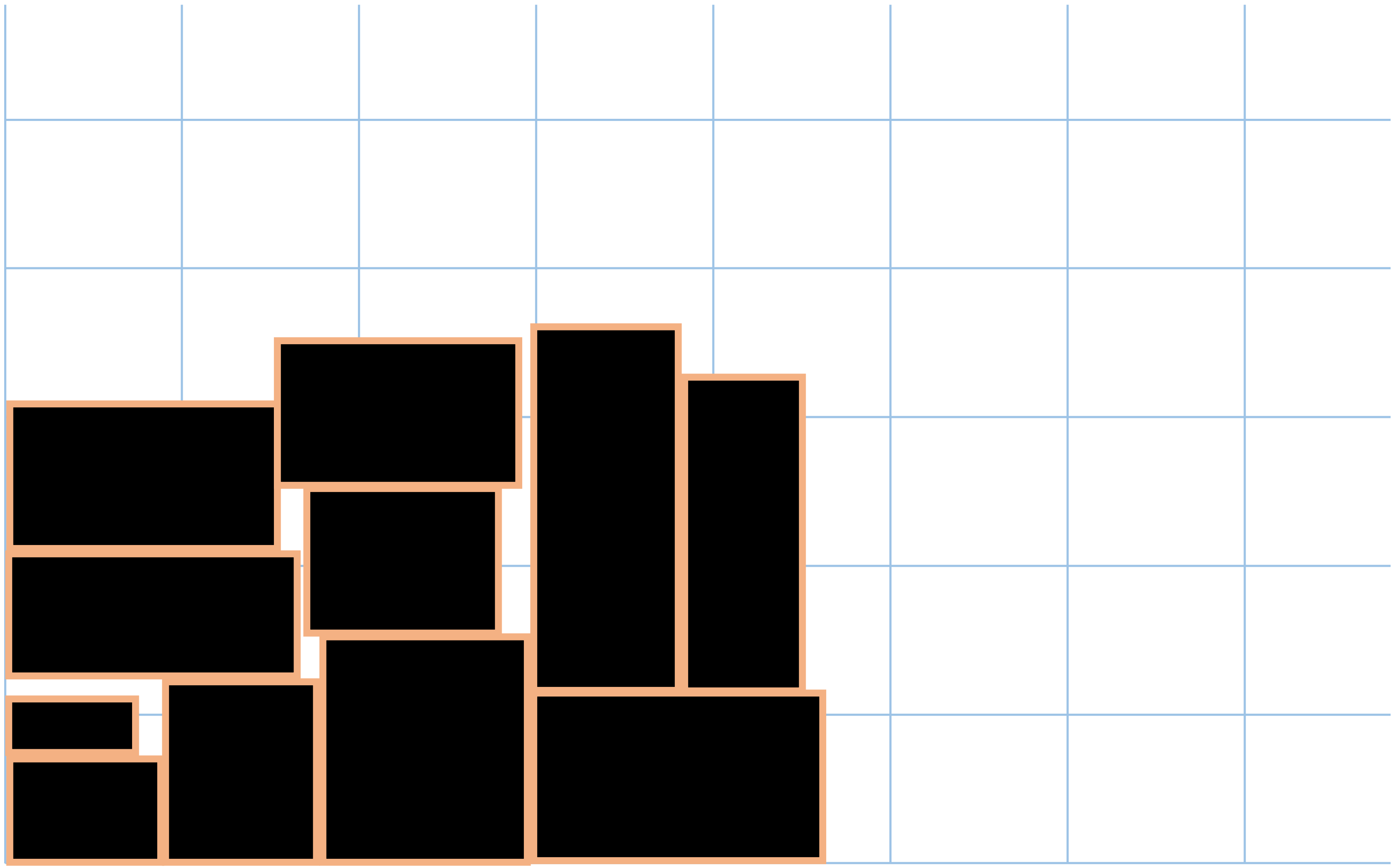


A local minimum in the 8-queens state space (h=1)

2-Dimensional packing problem

- **Input**: A set of n rectangles $I = \{1,2,\ldots,n\}$ characterized by (width, length, ...)
- **Output**: Coordinates *x*, *y* of rectangles. Place all the rectangles on a plan without overlap to minimize the Surface used
- Applications : textile, metal, wood, factory layout planning

2-Dimensional packing problem



Outline

- **Heuristic (informed) search strategies**
- **Heuristic functions**
- **Local search algorithms & optimization problems**
  - **Hill Climbing**
  - **Simulated Annealing**
  - **Local Beam Search**
- **Global search algorithm**
- **Online search agents & unknown environment**

Simulated Annealing

- **Simulated annealing (SA)**
  - Escape local maxima by allowing “bad” moves
    - Idea: but gradually decrease their size and frequency
  - Origin: metallurgical annealing
    - A heat treatment causing changes in its properties such as strength and hardness
    - If temperature T decreases slowly enough, best state is reached
    - Quenching (淬火): harden (+brittle)
Annealing (退火): harden
Temping (回火): soften (+tougher)

Simulated Annealing

- **Simulated annealing** (cont'd)
  - Bouncing ball analogy:
    - Shaking hard (= high temperature)
    - Shaking less (= lower the temperature)
  - Widely applied in industrial area, e.g., VLSI layout, airline scheduling, etc.

Simulated Annealing

function SIMULATED-ANNEALING(*problem*, *schedule*) **return** a solution state

input: *problem*, a problem
schedule, a mapping from time to temperature
local variables: *current*, a node
next, a node
T, a “temperature” controlling the downward steps prob.

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 $\Delta E > 0$ **then** *current* ← *next*

else *current* ← *next* only with **probability** $e^{\Delta E/T}$

Outline

- **Heuristic (informed) search strategies**
- **Heuristic functions**
- **Local search algorithms & optimization problems**
  - **Hill Climbing**
  - **Simulated Annealing**
  - **Local Beam Search**
- **Online search agents & unknown environment**

Local Beam Search

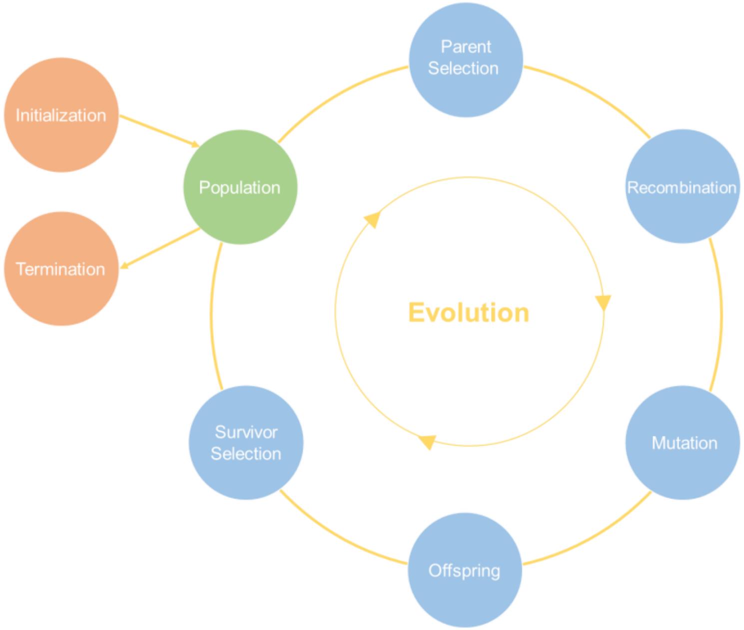
- **Keep track of k states instead of one**
  - Initially: k random states
  - Next: determine all successors of k states
  - If any of successors is goal $\rightarrow$ finished
  - Else select k best from successors and repeat
- **Major difference with random-restart search**
  - Information is shared among k search threads
  - e.g. one state may generates several good while others do bad $\rightarrow$ “come over here, guys!”
- **Can suffer from lack of diversity**
  - Stochastic variant: choose k successors in proportional to state success

Outline

- **Heuristic (informed) search strategies**
- **Heuristic functions**
- **Local search algorithms & optimization problems**
- **Global search algorithm**
- **Online search agents & unknown environment**

Genetic Algorithm

- Genetic algorithm (GA)
  - Variant of local beam search with sexual recombination
  - Simulates natural evolution, based on “survival of the fittest”



Genetic Algorithm

function GENETIC_ALGORITHM(*population*, FITNESS-FN) **return** an individual

input: *population*, a set of individuals
FITNESS-FN, a function that measures the fitness of an individual

repeat

new_population ← empty set

loop for i **from** 1 **to** SIZE(*population*) **do**

x ← RANDOM_SELECTION(*population*, FITNESS_FN)

y ← RANDOM_SELECTION(*population*, FITNESS_FN)

child ← REPRODUCE(*x*, *y*)

if (small random probability) **then** *child* ← MUTATE(*child*)

add *child* to *new_population*

population ← *new_population*

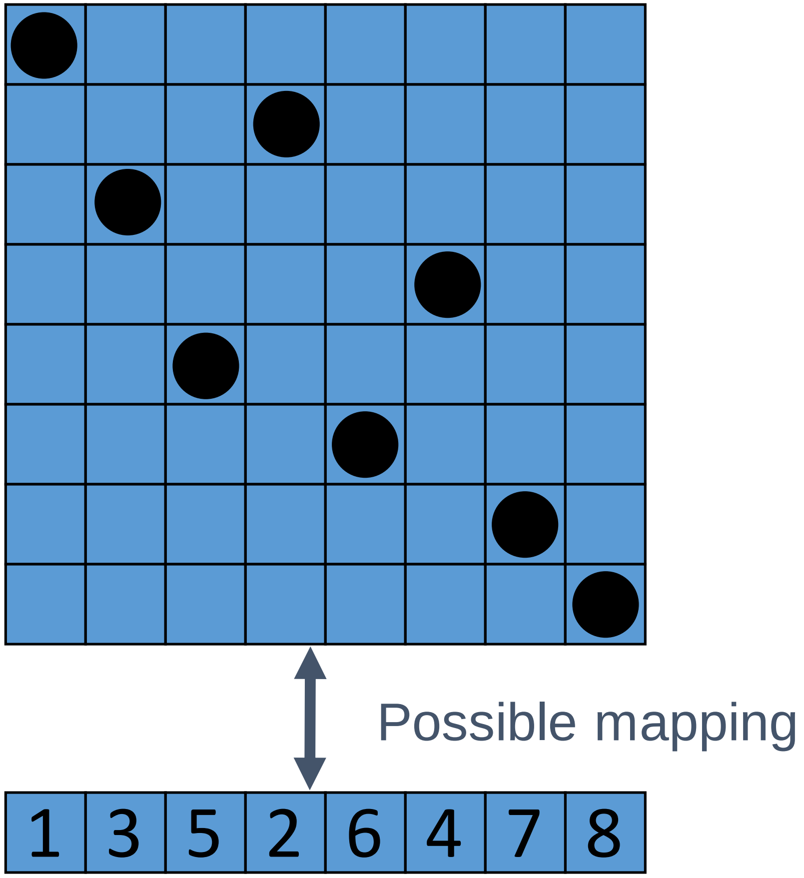
until some individual is fit enough, or enough time has elapsed

return the best individual in *population*, according to FITNESS-FN

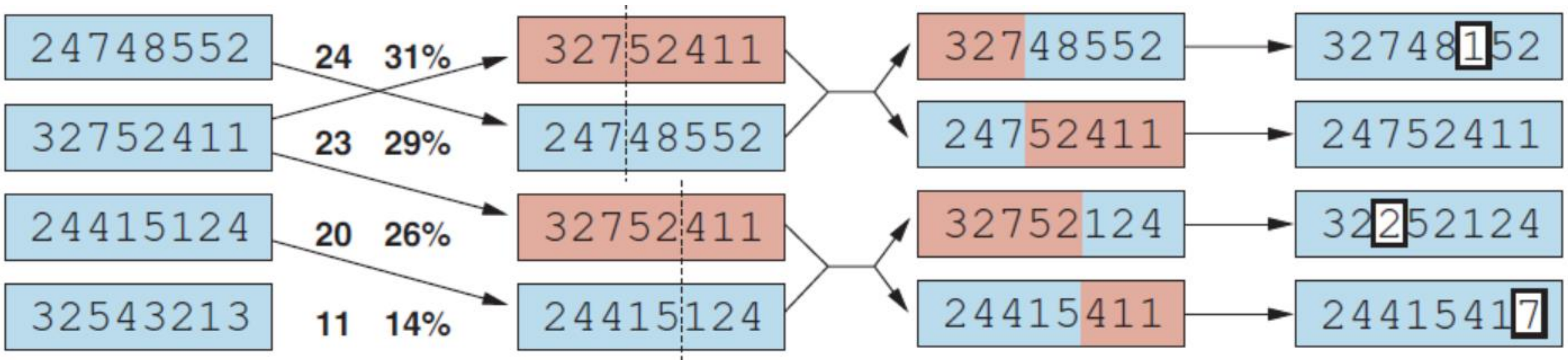
GA for 8 queens puzzle: Representation

Phenotype:
a board configuration

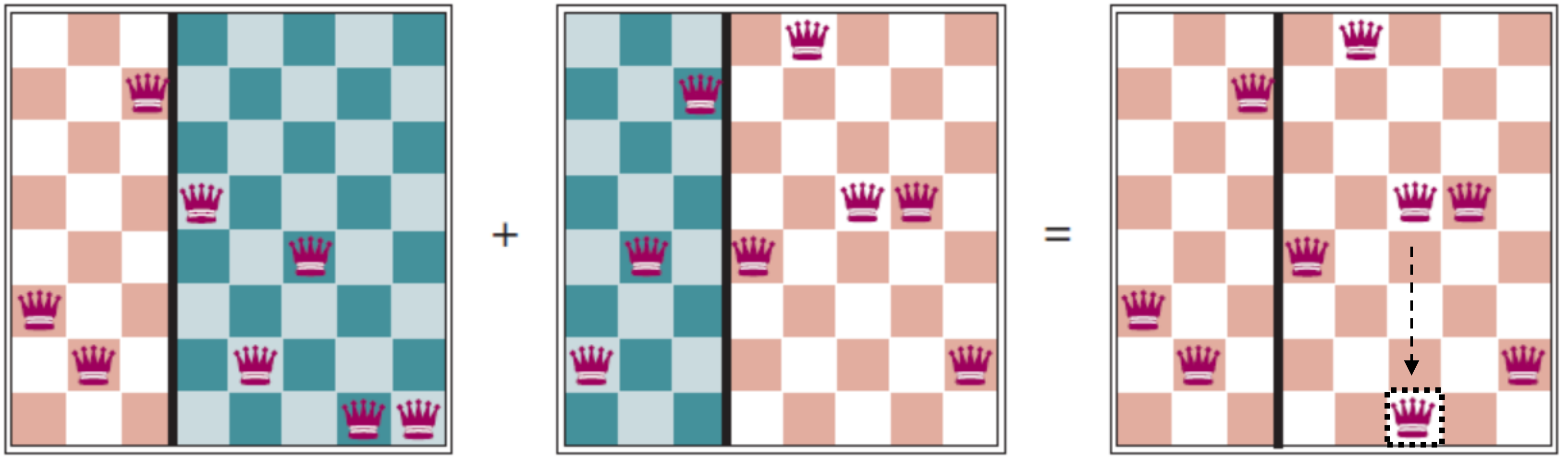
Genotype:
a permutation
(integer vector) of
the numbers 1–8



GA for 8 queens puzzle: Reproduction and mutation



One-point crossover and mutation on genotype



Result of one-point crossover and mutation on phenotype

Outline

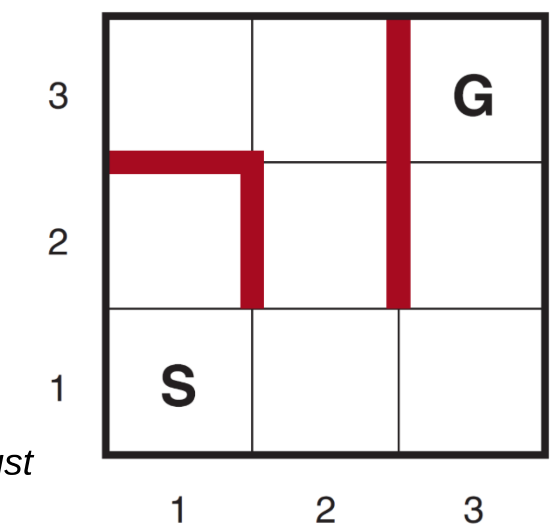
- **Heuristic (informed) search strategies**
- **Heuristic functions**
- **Local search algorithms & optimization problems**
- **Global search algorithm**
- **Online search agents & unknown environment**

Exploration Problems

- **Until now all algorithms were offline**
  - Offline = solution is determined before executing it
  - Online = interleaving computation and action
- **However, 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)
- **Notes**
  - Cannot access the successor of a state unless trying all actions in it
    - e.g. the agent doesn't know UP from (1,1) leads to (1,2)
  - Can recognize previous states
  - Actions are deterministic
  - Access to admissible heuristic $h(s)$
    - e.g. Manhattan distance



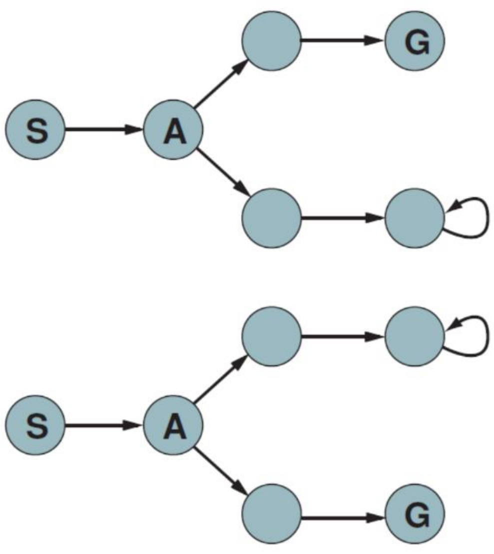
A simple maze problem. The agent starts at S and must reach G but knows nothing of the environment.

Online Search Problems

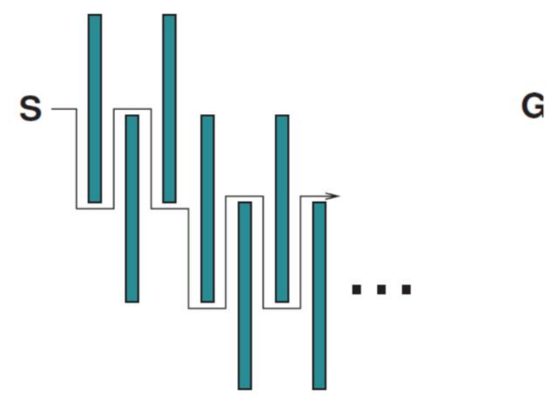
- **Objective: reach goal with minimal cost**
  - Cost = total cost of traveled path
  - Competitive ratio = comparison of cost *with* cost of the solution path if search space is known
  - Can be infinite in case of the agent accidentally reaches dead ends

The Adversary Argument

- **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



(a)



(b)

No algorithm can avoid dead ends in all state spaces.

Online Search Agents

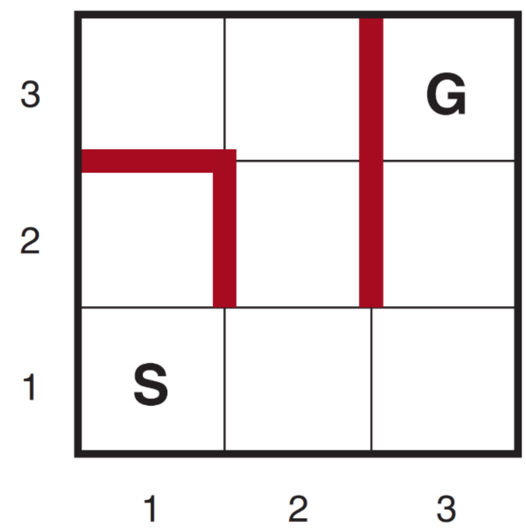
- **The agent maintains a map of the environment**
  - This map is used to decide next action
  - Updated based on percept input
  - Note difference with e.g. A*
  - An online version can only expand the node it is physically in (local order)

Online DFS

```
function ONLINE-DFS-AGENT(problem, s′) returns an action
        s, a, the previous state and action, initially null
    persistent: result , a table mapping (s, a) to s′, initially empty
                untried, a table mapping s to a list of untried actions
                unbacktracked, a table mapping s to a list of states
                               never backtracked to
    if problem.IS-GOAL(s′) then return stop
    if s′ is a new state (not in untried) then
        untried[s′]←problem.ACTIONS(s′)
    if s is not null then
        result [s, a]←s′
        add s to the front of unbacktracked[s′]
    if untried[s′] is empty then
        if unbacktracked[s′] is empty then return stop
        else a←an action b s.t. result [s′, b] = POP(unbacktracked[s′])
    else a←POP(untried[s′])
    s ←s′
    return a
```

Example: Online DFS (1)

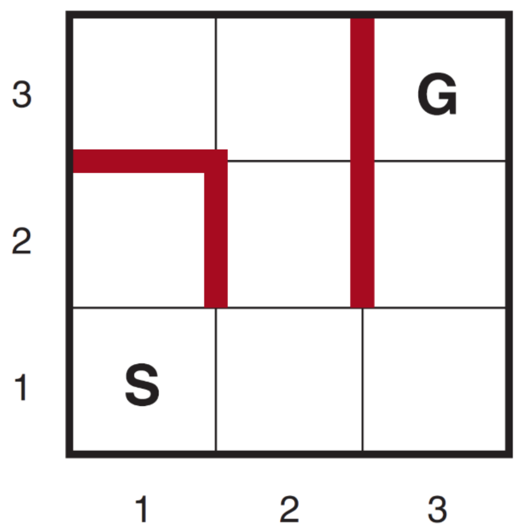
- **Maze problem on 3x3 grid**
  - s' = (1,1) is initial state
  - Result, unexplored (UX), unbacktracked (UB), …are empty
  - s, a are also empty



Example: Online DFS (2)

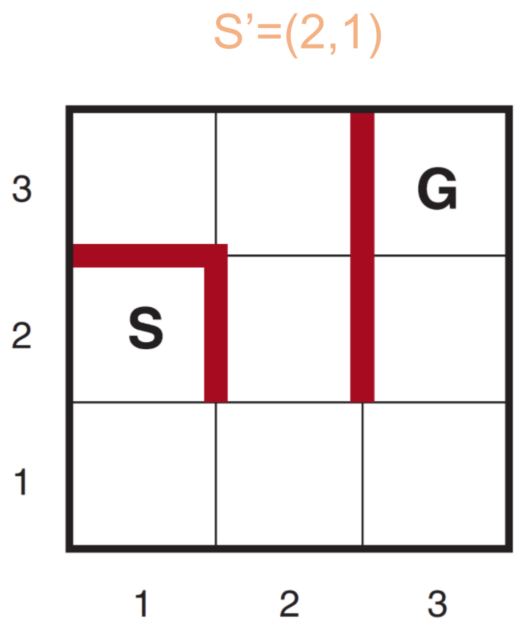
- **Procedure**
  - GOAL-TEST((1,1))?
    - S != G thus false
  - (1,1) a new state?
    - True
    - ACTION((1,1)) → UX[(1,1)]
      - Action: {**UP**, **RIGHT**}
  - s is null?
    - True (initially)
  - UX[(1,1)] empty?
    - False
  - POP(UX[(1,1)])->a
    - a=UP
  - s = (1,1)
  - Return a

S’=(1,1)



Example: Online DFS (3)

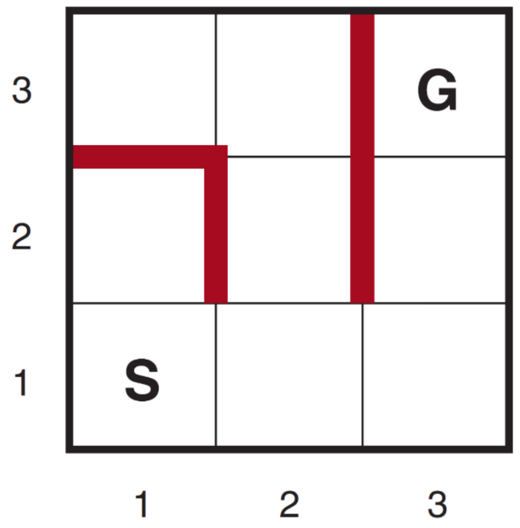
- **Procedure**
  - GOAL-TEST((2,1))?
    - S != G thus false
  - (2,1) a new state?
    - True
    - ACTION((2,1)) → UX[(2,1)]
      - Action: {**DOWN**}
  - s is null?
    - false (s=(1,1))
    - result[UP,(1,1)] ← (2,1)
    - UB[(2,1)]={(1,1)}
  - UX[(2,1)] empty?
    - False
  - a=DOWN, s=(2,1) return a



Example: Online DFS (4)

- **Procedure**
  - 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

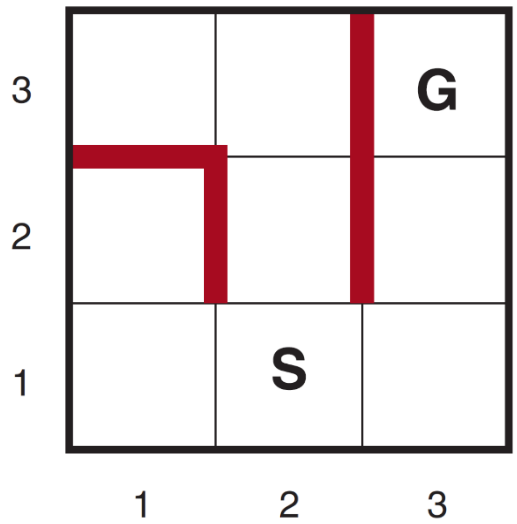
S'=(1,1)



Example: Online DFS (5)

- **Procedure**
  - GOAL-TEST((1,2))?
    - S not = G thus false
  - (1,2) a new state?
    - True, UX[(1,2)]={**LEFT**, **RIGHT**, **UP**}
  - 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

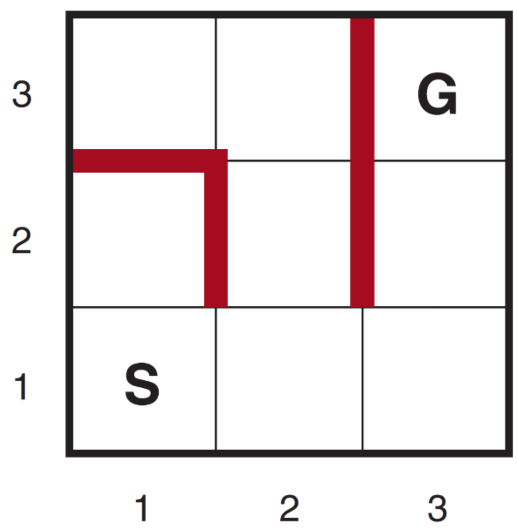
S'=(1,2)



Example: Online DFS (6)

- **Procedure**
  - 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) …

S'=(1,1)



Example: Online DFS (7)

- **Notes**
  - 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 DFS works only when actions are reversible