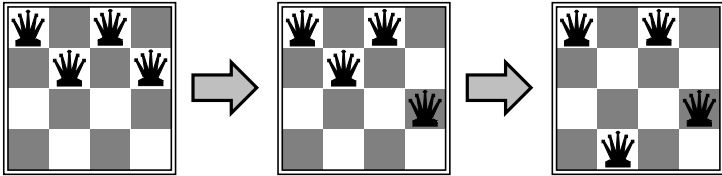


## Review: Basic Concepts

### Example: $n$ -queens

Put  $n$  queens on an  $n \times n$  board with no two queens on the same row, column, or diagonal



- Case 1: Consider one (fixed) cell at a time
- Case 2: Consider one row at a time
- Case 3: Consider one queen at a time

## Artificial Intelligence

### Informed Search and Exploration

Readings: Chapter 4 of Russell & Norvig.

## Review: Tree 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)) return node
    fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

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

## $n$ -queens

- Case 1: Consider one (fixed) cell at a time
- Case 2: Consider one row at a time
- Case 3: Consider one queen at a time

	case1	case 2	case 3
<b>Branching factor:</b>	2	$n$	$n^2$
<b>Maximal depth:</b>	$n^2$	$n$	$n$
<b>State space:</b>	$2^{n^2}$	$n^n$	$n^{2^n}$

## Informed Search Strategies

- Uninformed search strategies look for solutions by *systematically* generating new states and checking each of them against the goal.
- This approach is very inefficient in most cases.
- Most successor states are “obviously” a bad choice.
- Such strategies do not know that because they have minimal *problem-specific knowledge*.
- **Informed** search strategies exploit problem-specific knowledge as much as possible to drive the search.
- They are almost always *more efficient* than uninformed searches and often also *optimal*.

Artificial Intelligence – p.6/52

## Uninformed Search Strategies

Strategies	Time	Space	Complete?
Breadth-first Search	$O(b^d)$	$O(b^d)$	Yes
Depth-first Search	$O(b^m)$	$O(bm)$	No
Depth-limited Search	$O(b^l)$	$O(bl)$	No
Iterative Deepening Search	$O(b^d)$	$O(bd)$	Yes
Uniform Cost Search	$O(b^d)$	$O(b^d)$	Yes

where  $b$  is the branching factor,  $d$  is the depth of the shallowest solution,  $m$  is the length of the longest path,  $l$  is the limit set by the user.

Artificial Intelligence – p.5/52

## Informed Search Strategies

- $f$  is typically an *imperfect measure* of the goodness of the node. The right choice of nodes is not always the one suggested by  $f$ .
- It is possible to build a perfect evaluation function, which will always suggest the right choice.
- How?
- Why don't we use perfect evaluation functions then?

Artificial Intelligence – p.8/52

## Informed Search Strategies

### Main Idea

- Use the knowledge of the problem domain to build an **evaluation function**  $f$ .
- For every node  $n$  in the search space,  $f(n)$  quantifies the *desirability* of expanding  $n$  in order to reach the goal.
- Then use the desirability value of the nodes in the fringe to decide which node to expand next.

Artificial Intelligence – p.7/52

## Best-First Search

- Idea: use an *evaluation function* for each node to estimate of “desirability”
- Strategy: Always expand most desirable unexpanded node
- **Implementation:** *fringe* is a priority queue sorted in decreasing order of desirability
- Special cases:
  - uniform-cost search
  - greedy search
  - A\* search

## Standard Assumptions on Search Spaces

- The cost of a node increases with the node's depth.
- Transitions costs are non-negative and bounded below. That is, there is a  $\delta > 0$  such that the cost of each transition is  $\geq \delta$ .
- Each node has only finitely-many successors.

**Note:** There *are* problems that do *not* satisfy one or more of these assumptions.

## Best-first Search Strategies

- There is a *whole family* of best-first search strategies, each with a different evaluation function.
- Typically, strategies use estimates of the *cost* of reaching the goal and try to *minimize* it.
- Uniform Search also tries to minimize a cost measure.
- Is it a best-first search strategy?
- Not in spirit, because the evaluation function should incorporate a *cost estimate of going from the current state to the closest goal state*.

## Implementing Best-first Search

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
inputs: problem, a problem
         Eval-Fn, an evaluation function

Queueing-Fn ← a function that orders nodes by EVAL-FN
return GENERAL-SEARCH(problem, Queueing-Fn)
```

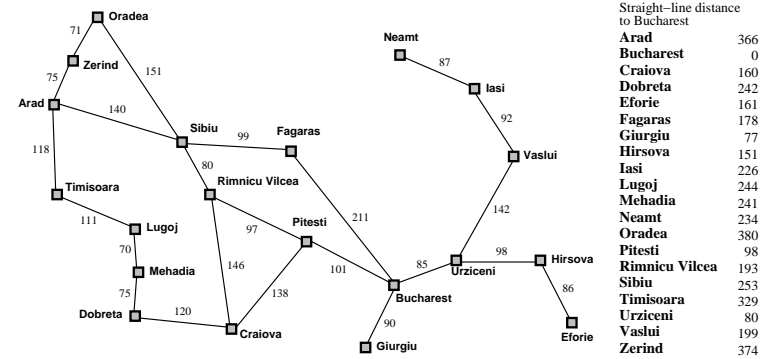
```
function GENERAL-SEARCH(problem, QUEUEING-FN) returns a solution, or failure

nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
loop do
  if nodes is empty then return failure
  node ← REMOVE-FRONT(nodes)
  if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
  nodes ← QUEUEING-FN(nodes, EXPAND(node, OPERATORS[problem]))
end
```

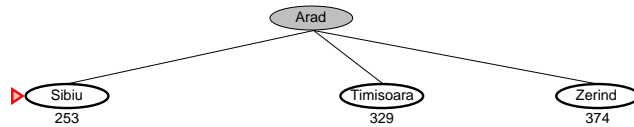
# Greedy Search

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

# Romania with Step Costs in km



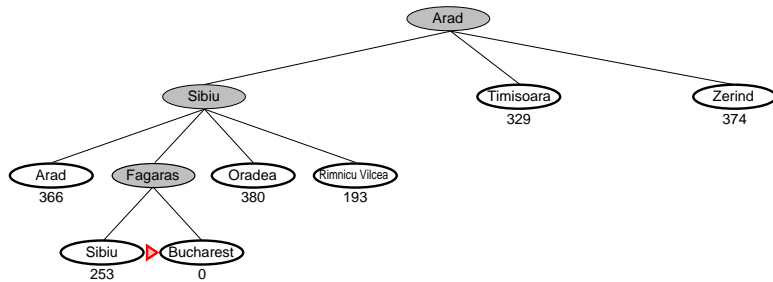
# Greedy Search Example



# Greedy Search Example

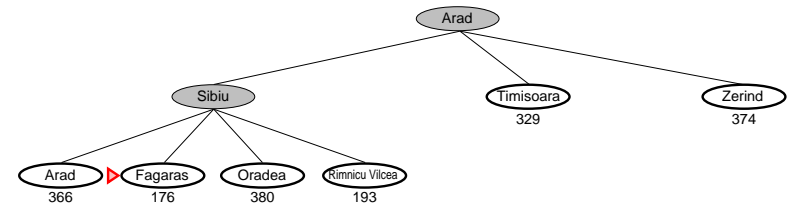


## Greedy search example



Artificial Intelligence – p.18/52

## Greedy Search Example



Artificial Intelligence – p.17/52

## Properties of greedy search

- Complete?? No—can get stuck in loops, e.g., with Oradea as goal, Iasi → Neamt → Iasi → Neamt →
- Complete in finite space with repeated-state checking
- Time??

Artificial Intelligence – p.20/52

## Properties of Greedy Search

- Complete??

Artificial Intelligence – p.19/52

## Properties of Greedy Search

- **Complete??** No—can get stuck in loops, e.g., lasi → Neamt → lasi → Neamt →  
Complete in finite space with repeated-state checking
- **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 Search

- **Complete??** No—can get stuck in loops, e.g., lasi → Neamt → lasi → Neamt →  
Complete in finite space with repeated-state checking
- **Time??**  $O(b^m)$ , but a good heuristic can give dramatic improvement
- **Space??**

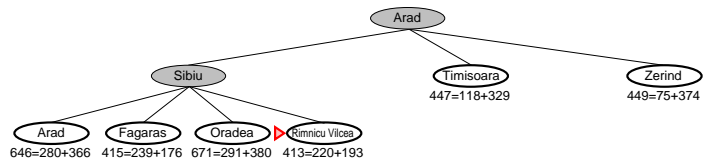
## A\* Search Example



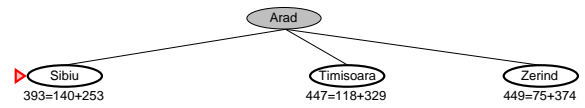
## A\* Search

- Idea: avoid expanding paths that are already expensive
- Evaluation function  $f(n) = g(n) + h(n)$
- $g(n)$  = cost so far to reach  $n$   
 $h(n)$  = estimated cost to goal from  $n$   
 $f(n)$  = estimated total cost of path through  $n$  to goal
- A\* search uses an **admissible** heuristic  
i.e.,  $h(n) \leq h^*(n)$  where  $h^*(n)$  is the **true** cost from  $n$  to a goal.  
(Also require  $h(n) \geq 0$ , so  $h(G) = 0$  for any goal  $G$ .)  
E.g.,  $h_{\text{SLD}}(n)$  never overestimates the actual road distance
- **Theorem:** A\* search is optimal if  $h$  is admissible.

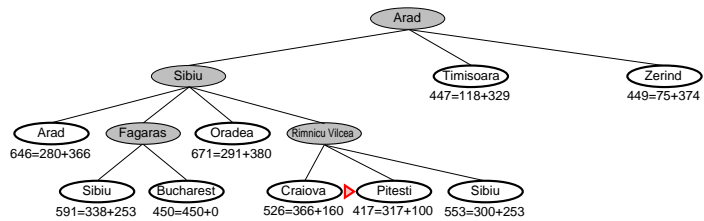
# A\* Search Example



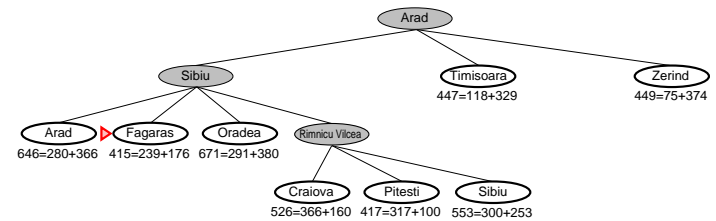
# A\* Search Example



# A\* Search Example



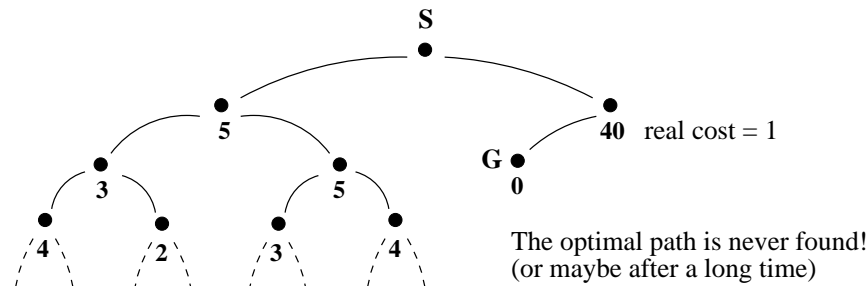
# A\* Search Example



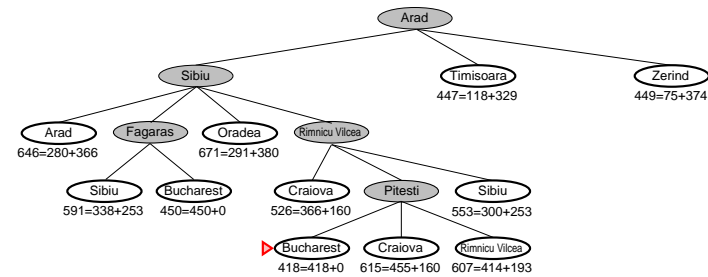
# A\* Search with Admissible Heuristic

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

Overestimates are dangerous (h values are shown)



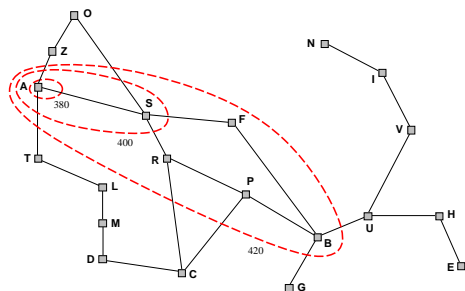
# A\* Search Example



# Optimality of A\* (more useful)

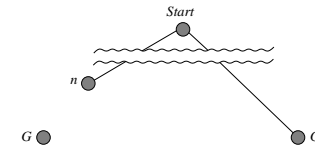
**Lemma:** A\* expands nodes in order of increasing  $f$  value\*  
Gradually adds “ $f$ -contours” of nodes (cf. breadth-first adds layers)

Contour  $i$  has all nodes with  $f = f_i$ , where  $f_i < f_{i+1}$



# Optimality of A\* (standard proof)

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



$$\begin{aligned}
 f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
 &> g(G) && \text{since } G_2 \text{ is suboptimal} \\
 &\geq f(n) && \text{since } h \text{ is admissible}
 \end{aligned}$$

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

## Properties of A\*

- Complete?? Yes, unless there are infinitely many nodes with  $f \leq f(G)$
- Time??

## Properties of A\*

- Complete??

## Properties of A\*

- Complete?? Yes, unless there are infinitely many nodes with  $f \leq f(G)$
- Time??  $O(f * |\{n \mid f(n) \leq f(G)\}|)$  (exponential in general in terms of the length of solutions)
- Space??  $O(|\{n \mid f(n) \leq f(G)\}|)$
- Optimal??

## Properties of A\*

- Complete?? Yes, unless there are infinitely many nodes with  $f \leq f(G)$
- Time??  $O(f * |\{n \mid f(n) \leq f(G)\}|)$  (exponential in general in terms of the length of solutions)
- Space??

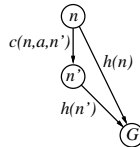
# Consistency

A heuristic is *consistent* if

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

If  $h$  is consistent, we have

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



I.e.,  $f(n)$  is nondecreasing along any path.

# Properties of A\*

- **Complete??** Yes, unless there are infinitely many nodes with  $f \leq f(G)$
- **Time??**  $O(f * |\{n \mid f(n) \leq f(G)\}|)$  (exponential in general in terms of the length of solutions)
- **Space??**  $O(|\{n \mid f(n) \leq f(G)\}|)$
- **Optimal??** 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  $f(n) > C^*$

# Admissible Heuristics

For the 8-puzzle:

- $h_1(n)$  = number of misplaced tiles
- $h_2(n)$  = total **Manhattan** distance (i.e., number of squares from desired location of each tile)

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

- $h_1(S) = ??$  7
- $h_2(S) = ??$   $4+0+3+3+1+0+2+1 = 14$

# Admissible Heuristics

For the 8-puzzle:

- $h_1(n)$  = number of misplaced tiles
- $h_2(n)$  = total **Manhattan** distance (i.e., number of squares from desired location of each tile)

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

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

## Optimality/Completeness of A\* Search

If the problem is solvable, A\* always finds an optimal solution when

- the standard assumptions are satisfied,
- the heuristic function is admissible.

A\* is **optimally efficient** for any heuristic function  $h$ : No other optimal strategy expands fewer nodes than A\* when using the same  $h$ .

## Dominance

- **Definition:** If  $h_2(n) \geq h_1(n)$  for all  $n$  (both admissible) then  $h_2$  **dominates**  $h_1$ .
- For 8-puzzle,  $h_2$  indeed dominates  $h_1$ .
  - $h_1(n)$  = number of misplaced tiles
  - $h_2(n)$  = total **Manhattan** distance
- If  $h_2$  dominates  $h_1$ , then  $h_2$  is better for search.
- For 8-puzzle, search costs:

$d = 14$  IDS = 3,473,941 nodes (IDS = Iterative Deepening Search)

A\*( $h_1$ ) = 539 nodes

A\*( $h_2$ ) = 113 nodes

$d = 24$  IDS  $\approx$  54,000,000,000 nodes

A\*( $h_1$ ) = 39,135 nodes

A\*( $h_2$ ) = 1,641 nodes

## IDA\* and SMA\*

- **IDA\*** (Iterative Deepening A\*): Set a limit and store only those nodes  $x$  whose  $f(x)$  is under the limit. The limit is increased by some value if no goal is found.
- **SMA\*** (Simplified Memory-bound A\*): Work like A\*; when the memory is full, drop the node with the highest  $f$  value before adding a new node.

## Complexity of A\* Search

- **Worst-case time complexity:** still exponential ( $O(b^d)$ ) unless the error in  $h$  is bounded by the logarithm of the actual path cost. That is, unless

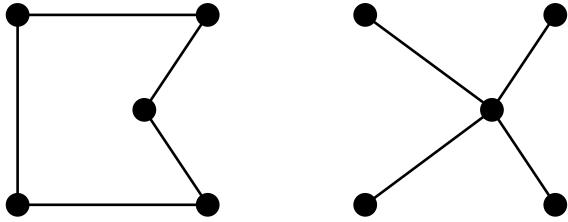
$$|h(n) - h^*(n)| \leq O(\log h^*(n))$$

where  $h^*(n)$  = actual cost from  $n$  to goal.

- **Worst-Case Space Complexity:**  $O(b^m)$  as in greedy best-first.
- A\* generally runs out of memory before running out of time. (Improvements: IDA\*, SMA\*).

## Relaxed Problems

Well-known example: **traveling salesperson problem** (TSP)  
Find the shortest tour visiting all cities exactly once



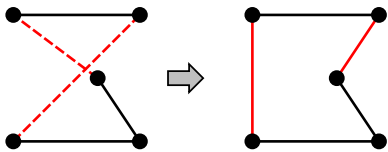
**Minimum spanning tree** can be computed in  $O(n^2)$   
and is a lower bound on the shortest (open) tour

## Relaxed Problems

- Admissible heuristics can be derived from the **exact** solution cost of a **relaxed** version of the problem
- If the rules of the 8-puzzle are relaxed so that a tile can move **anywhere**, then  $h_1(n)$  gives the shortest solution
- If the rules are relaxed so that a tile can move to **any adjacent square**, then  $h_2(n)$  gives the shortest solution
- Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

## Local Search Example: TSP

- TSP: Traveling Salesperson Problem
- Start with any complete tour, perform pairwise exchanges



For  $n$  cities, each state has  $n(n-1)/2$  neighbors.

## Local Search Algorithms

- In many optimization problems, **path** is irrelevant; the goal state itself is the solution.
- Define state space as a set of “complete” configurations; find **optimal** configuration, e.g., TSP or, find configuration satisfying constraints, e.g., timetable
- State space = set of “complete” configurations.
- In such cases, can use **local search** (or iterative improvement) algorithms; keep a single “current” state, try to improve it.
- Constant space, suitable for online as well as offline search

## Local Search Example: 8-queens

Standard and Compact Representations:

	1	2	3	4	5	6	7	8
1					*			
2			*					
3							*	
4						*		
5	*							
6				*				
7		*						
8								*

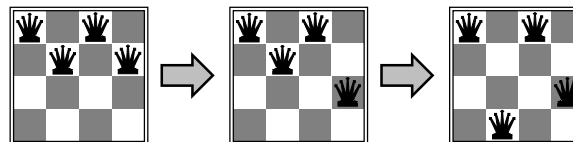
$c$	1	2	3	4	5	6	7	8
$r$	5	7	2	6	1	4	3	8
$c - r$	-4	-5	1	-2	4	2	4	0
$c + r$	6	9	5	10	6	10	10	16

Operation: Switching two columns.

Neighbors of each state:  $8 \cdot 7 / 2 = 28$ .

## Local Search Example: $n$ -queens

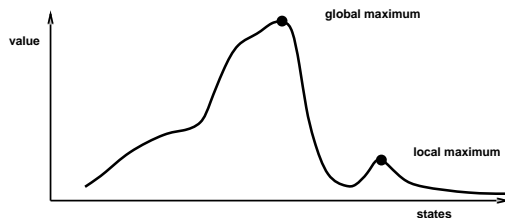
- Put  $n$  queens on an  $n \times n$  board with no two queens on the same row, column, or diagonal
- Change the row of a queen in a given column to reduce number of conflicts



For  $n$  queens, each state has  $n(n - 1)$  neighbors.

## Performance of Hill-Climbing

- Quality of the solution  
Problem: depending on initial state, can get stuck on local maxima
- Time to get the solution  
In continuous spaces, problems may be slow to converge.  
Choose a good initial solution; find good ways to compute the cost function



Improvements: Simulated annealing, tabu search, etc.

## Hill-Climbing (or Gradient Descent)

“Like climbing Everest in thick fog with amnesia”

```
function Hill-Climbing(problem) return state
node: current, neighbor;
current := Make-Node(Initial-State(problem));
loop do
    neighbor := highest-value-successor(current)
    if (Value(neighbor) < Value(current))
        then return State(current)
        else current := neighbor
    end loop
end function
```

The returned state is a local maximum state.