

# CmpE 540

## Principles of Artificial Intelligence

Pınar Yolum  
[pinar.yolum@boun.edu.tr](mailto:pinar.yolum@boun.edu.tr)

Department of  
Computer Engineering  
Boğaziçi University

## Informed search algorithms

Chapter 4 (Sections 1—3)  
(Based mostly on the course slides  
from <http://aima.cs.berkeley.edu/> and  
<http://www.cmpe.boun.edu.tr/~akin/>)

## Outline

- Best-first search
- Greedy best-first search
- A\* search
- Heuristics
- Local search algorithms
- Hill-climbing search
- Simulated annealing search
- Local beam search
- Genetic algorithms

3

## 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] applied to STATE(node) succeeds return node
    fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

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

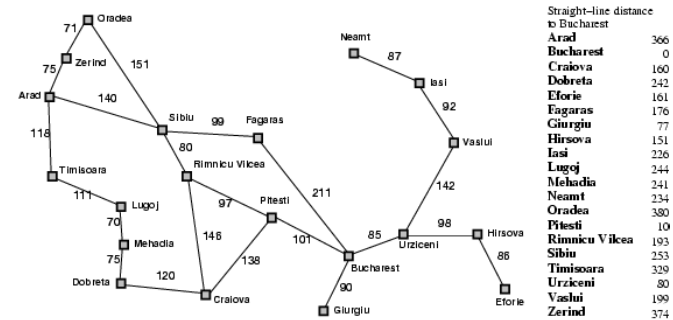
4

## Best-first search

- Idea: use an **evaluation function**  $f(n)$  for each node
  - estimate of "desirability"
  - Expand most desirable unexpanded node
- Implementation:  
Order the nodes in fringe in decreasing order of desirability
- Special cases:
  - greedy best-first search
  - A\* search

5

## Romania with step costs in km



7

## Greedy best-first search

- Evaluation function  $f(n) = h(n)$  (**h**euristic) = 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

8

## Greedy best-first search example



9

## Greedy best-first search example



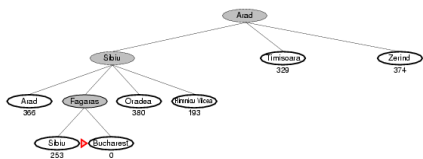
10

## Greedy best-first search example



11

## Greedy best-first search example



12

## Properties of greedy best-first search

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

13

## 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 from  $n$  to goal
- $f(n)$  = estimated total cost of path through  $n$  to goal

14

## A\* search example



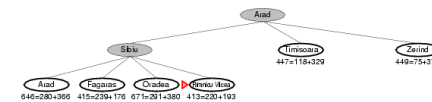
15

## A\* search example



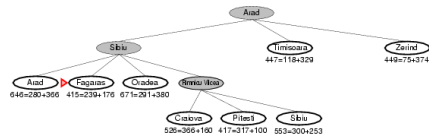
16

## A\* search example



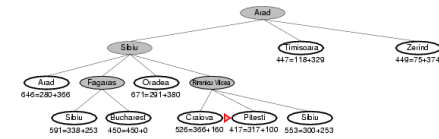
17

## A\* search example



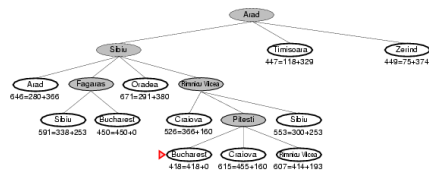
18

## A\* search example



19

## A\* search example



20

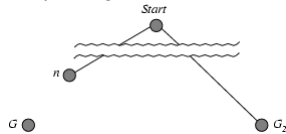
## Admissible heuristics

- 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  $n$ .
- An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is **optimistic**
- Example:  $h_{SLD}(n)$  (never overestimates the actual road distance)
- **Theorem:** If  $h(n)$  is admissible, A\* using TREE-SEARCH is optimal

21

## Optimality of A\* (proof)

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

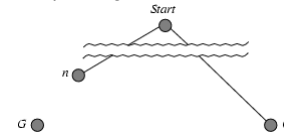


- $f(G_2) = g(G_2)$  since  $h(G_2) = 0$
- $g(G_2) > g(G)$  since  $G_2$  is suboptimal
- $f(G) = g(G)$  since  $h(G) = 0$
- $f(G_2) > f(G)$  from above

22

## Optimality of A\* (proof)

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



- $f(G_2) > f(G)$  from above
  - $h(n) \leq h^*(n)$  since  $h$  is admissible
  - $g(n) + h(n) \leq g(n) + h^*(n)$
  - $f(n) \leq f(G)$
- Hence  $f(G_2) > f(n)$ , and A\* will never select  $G_2$  for expansion

23

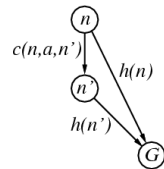
## Consistent heuristics

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

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

- If  $h$  is consistent, we have
- $$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

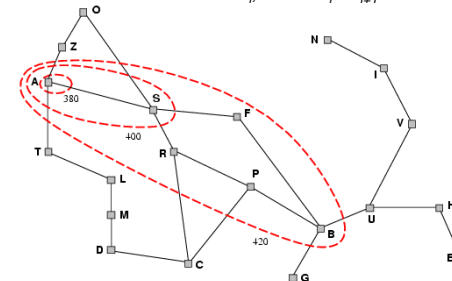
- i.e.,  $f(n)$  is non-decreasing along any path.
- Theorem:** If  $h(n)$  is consistent, A\* using GRAPH-SEARCH is optimal



24

## 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  $f_i < f_{i+1}$



25

## Properties of A\*

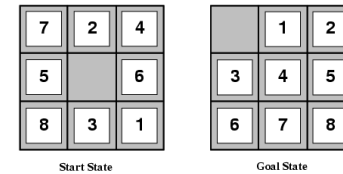
- **Complete?** Yes (unless there are infinitely many nodes with  $f \leq f(G)$ )
- **Time?** Exponential
- **Space?** Keeps all nodes in memory
- **Optimal?** Yes, cannot expand  $f_{i+1}$  until  $f_i$  is finished

26

## Admissible heuristics

E.g., for the 8-puzzle:

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



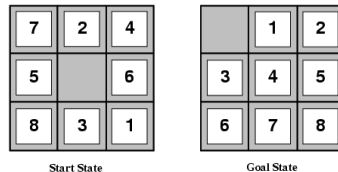
- $h_1(S) = ?$
- $h_2(S) = ?$

27

## Admissible heuristics

E.g., for the 8-puzzle:

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



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

28

## Dominance

- If  $h_2(n) \geq h_1(n)$  for all  $n$  (both admissible) then  $h_2$  **dominates**  $h_1$  and  $h_2$  is better for search
- Typical search costs (average number of nodes expanded):

- $d=12$       IDS = 3,644,035 nodes  
 $A^*(h_1) = 227$  nodes  
 $A^*(h_2) = 73$  nodes
- $d=24$       IDS = too many nodes  
 $A^*(h_1) = 39,135$  nodes  
 $A^*(h_2) = 1,641$  nodes

Given any admissible heuristics  $h_a, h_b$ ,  
 $h(n) = \max(h_a(n), h_b(n))$   
 is also admissible and dominates  $h_a, h_b$ ,

29

## Relaxed problems

- A problem with fewer restrictions on the actions is called a **relaxed problem**
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- 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

30

## IDA\* (Iterative Deepening A\*)

- Memory- bounded Search
  - iterative deepening
  - bounded f- cost
  - most cases: required space = bd
  - if too long time -> epsilon- admissible variant

31

## Iterative Deepening A\*

A\*- fixed- cost( max)

1. Set  $N$  to be the set of initial nodes and set  $lub$  to infinity.
2. If  $N$  is empty, then signal failure and exit.
3. Set  $n$  to be the first node in  $N$  and remove  $n$  from  $N$ .
4. If  $n$  is a goal node, then signal success and exit.
5. If the value of  $n$  is equal to  $max$ , then go to step 2.
6. For each child:
  - Check whether the value is larger than  $max$ .
  - If yes then if the value is smaller than  $lub$ , set  $lub$  to this value.
  - If no then add child to  $N$ .
7. Sort  $N$  and go to step 2.

Call A\*- fixed- cost with increasing  $max$  (using  $lub$ ).

$lub$  = lower upper bound

32

## SMA\* (Simplified Memory- Bounded A\*)

- Memory- bounded Search
    - uses whatever memory is possible
    - avoids repeated states as far as memory allows
    - complete if memory is sufficient to store shallowest solution path
    - optimal if memory is sufficient to store shallowest solution path
- (otherwise best solution with available memory is returned)

33

## Iterative improvement algorithms

- In many optimization problems, path is irrelevant; the goal state itself is the solution
- Then state space = set of "complete" configurations (complete-state formulation vs. incremental formulation)
- In such cases, can use iterative improvement algorithms;
  - keep a single "current" state, try to improve it
- Constant space, suitable for online as well as offline search
- Often want to find optimal configuration, but also works for constraint satisfaction problems, e.g. n-queens, timetabling

34

## Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it

35

## Example: $n$ -queens

- Put  $n$  queens on an  $n \times n$  board with no two queens on the same row, column, or diagonal
- Local search: start with all  $n$ , move a queen to reduce conflicts



36

## Hill-climbing search

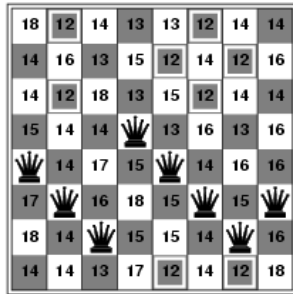
- "Like climbing Everest in thick fog with amnesia"

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
                 neighbor, a node

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
  neighbor ← a highest-valued successor of current
  if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
  current ← neighbor
```

37

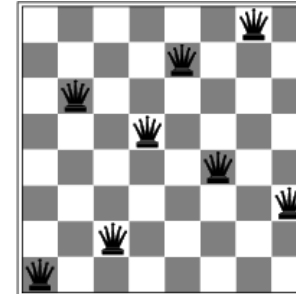
## Hill-climbing search: 8-queens problem



- $h$  = number of pairs of queens that are attacking each other, either directly or indirectly
- $h = 17$  for the above state

38

## Hill-climbing search: 8-queens problem

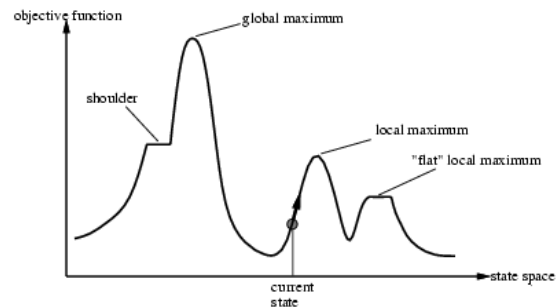


- A local minimum with  $h = 1$

39

## Hill-climbing search

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



40

## Local Optima (Foothill Problem)

- **Problem Definition**
  - Point reached by hill-climbing may be maximal but not maximum
  - Maximal
    - Definition: not dominated by any neighboring point (with respect to criterion measure)
  - Maximum
    - Definition: dominates all neighboring points (wrt criterion measure)
- **Ramifications**
  - Steepest ascent hill-climbing will become trapped (why?)
  - Need some way to break out of trap state
    - Accept transition (i.e., search move) to dominated neighbor
    - Start over: random restarts

41

## Lack of Gradient (Plateau Problem)

- **Problem Definition**
  - Function space may contain points whose neighbors are indistinguishable (with respect to criterion measure)
  - Effect: “flat” search landscape
- **Ramifications**
  - Steepest ascent hill-climbing will become trapped
  - Need some way to break out of zero gradient
    - Accept transition (i.e., search move) to random neighbor
    - Random restarts
    - Take bigger steps

42

## Simulated annealing search

- Idea: escape local maxima by allowing some “bad” moves but **gradually decrease** their frequency

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
inputs: problem, a problem
       schedule, a mapping from time to “temperature”
local variables: current, a node
                next, a node
                T, a “temperature” controlling prob. of downward steps

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

43

## 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

44

## Tabu Search

- Tabu: prevent returning quickly to same state.
- Implementation: Keep fixed length queue (“tabu list”): add most recent step to queue; drop “oldest” step.
- Never make step that's currently on the tabu list.
- Quite powerful; competitive with simulated annealing

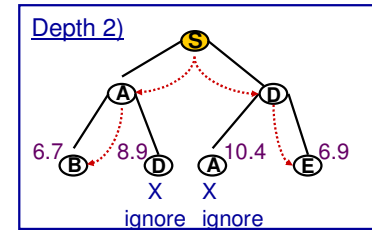
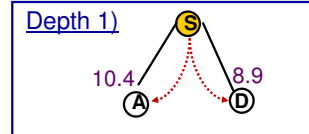
45

## 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.

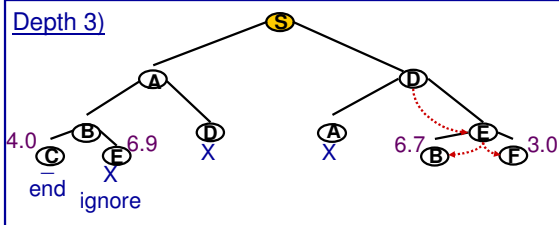
46

## Beam search

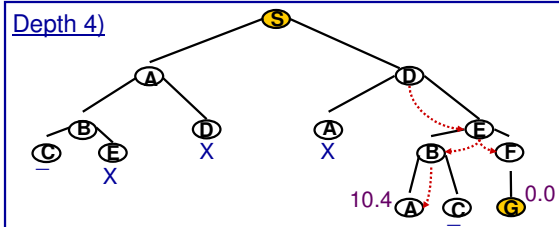


- Assume a pre-fixed WIDTH (example: 2)
- Perform breadth-first, BUT:
- Only keep the WIDTH best new nodes depending on heuristic at each new level.

47



- Optimization: ignore leaves that are not goal nodes (see C)



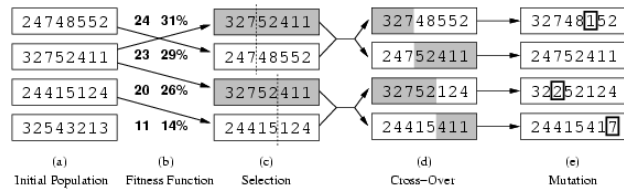
48

## Genetic algorithms

- A successor state is generated by combining two parent states
- Start with  $k$  randomly generated states (**population**)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (**fitness function**). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation

49

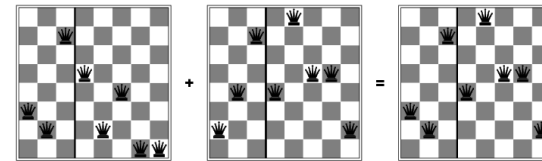
## Genetic algorithms



- Fitness function: number of non-attacking pairs of queens (min = 0, max =  $8 \times 7/2 = 28$ )
- $24/(24+23+20+11) = 31\%$
- $23/(24+23+20+11) = 29\%$  etc

50

## Genetic algorithms



51