

Solving problems by searching

Chapter 3

1

Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
- Informed search algorithms

14 Jan 2004

CS 3243 - Blind Search

2

Problem-solving agents

```

function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
static: seq, an action sequence, initially empty
        state, some description of the current world state
        goal, a goal, initially null
        problem, a problem formulation
        state ← UPDATE-STATE(state, percept)
        if seq is empty then do
            goal ← FORMULATE-GOAL(state)
            problem ← FORMULATE-PROBLEM(state, goal)
            seq ← SEARCH(problem)
        action ← FIRST(seq)
        seq ← REST(seq)
        return action

```

3

Example: Romania

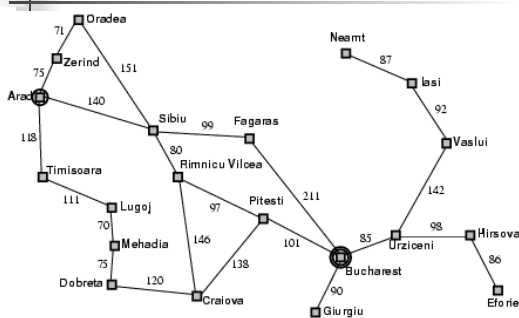
- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal:
 - be in Bucharest
- Formulate problem:
 - states: various cities
 - actions: drive between cities
- Find solution:
 - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

14 Jan 2004

CS 3243 - Blind Search

4

Example: Romania



14 Jan 2004

CS 3243 - Blind Search

5

Problem types

- Deterministic, fully observable → single-state problem
 - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensorless problem (conformant problem)
 - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → contingency problem
 - percepts provide new information about current state
 - often interleave} search, execution
- Unknown state space → exploration problem

6

Example: vacuum world

- Single-state, start in #5.
Solution?

7

Example: vacuum world

- Single-state, start in #5.
Solution? [Right, Suck]
- Sensorless, start in {1,2,3,4,5,6,7,8} e.g., Right goes to {2,4,6,8}
Solution?

8

Example: vacuum world

- Sensorless, start in {1,2,3,4,5,6,7,8}
Right goes to {2,4,6,8}
Solution?
[Right, Suck, Left, Suck]
- Contingency
 - Nondeterministic: Suck may dirty a clean carpet
 - Partially observable: location, dirt at current location.
 - Percept: [L, Clean], i.e., start in #5 or #7
Solution?

9

Example: vacuum world

- Sensorless, start in {1,2,3,4,5,6,7,8}
Right goes to {2,4,6,8}
Solution?
[Right, Suck, Left, Suck]
- Contingency
 - Nondeterministic: Suck may dirty a clean carpet
 - Partially observable: location, dirt at current location.
 - Percept: [L, Clean], i.e., start in #5 or #7
Solution? [Right, if dirt then Suck]

10

Single-state problem formulation

A problem is defined by four items:

- initial state e.g., "at Arad"
- actions or successor function $S(x)$ = set of action-state pairs
 - e.g., $S(\text{Arad}) = \{ \langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots \}$
- goal test, can be
 - explicit, e.g., $x = \text{"at Bucharest"}$
 - implicit, e.g., $\text{Checkmate}(x)$
- path cost (additive)
 - e.g., sum of distances, number of actions executed, etc.
 - $c(x, a, y)$ is the step cost, assumed to be ≥ 0

- A solution is a sequence of actions leading from the initial state to a goal state

11

Selecting a state space

- Real world is absurdly complex
 - state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
 - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"
- (Abstract) solution =
 - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

12

Vacuum world state space graph

- states?
- actions?
- goal test?
- path cost?

13

Vacuum world state space graph

- states? integer dirt and robot location
- actions? *Left, Right, Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

14

Example: The 8-puzzle

- states?
- actions?
- goal test?
- path cost?

15

Example: The 8-puzzle

- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

[Note: optimal solution of *n*-Puzzle family is NP-hard]

16

Example: robotic assembly

- states?: real-valued coordinates of robot joint angles
parts of the object to be assembled
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute

17

Tree search algorithms

- Basic idea:
 - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

```

function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
            
```

18

Tree search example

19

Tree search example

20

Tree search example

21

Implementation: general 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]) then return SOLUTION(node)
        fringe ← INSERT ALL(EXPAND(node, problem), fringe)

function EXPAND(node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
    
```

22

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost $g(x)$, depth

- The Expand function creates new nodes, filling in the various fields and using the SuccessorFn of the problem to create the corresponding states.

23

Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
 - completeness: does it always find a solution if one exists?
 - time complexity: number of nodes generated
 - space complexity: maximum number of nodes in memory
 - optimality: does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b : maximum branching factor of the search tree
 - d : depth of the least-cost solution
 - m : maximum depth of the state space (may be ∞)

24

Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

25

Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
 - fringe* is a FIFO queue, i.e., new successors go at end

26

Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
 - fringe* is a FIFO queue, i.e., new successors go at end

27

Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
 - fringe* is a FIFO queue, i.e., new successors go at end

28

Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
 - fringe* is a FIFO queue, i.e., new successors go at end

29

Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\dots+b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
- Space is the bigger problem (more than time)

30

Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
 - fringe = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost $\geq \epsilon$
- Time? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$ where C^* is the cost of the optimal solution
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

31

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe = LIFO queue, i.e., put successors at front

32

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe:

33

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe = LIFO queue, i.e., put successors at front

34

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe = LIFO queue, i.e., put successors at front

35

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe = LIFO queue, i.e., put successors at front

36

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe:

37

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe:

38

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe:

39

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe:

40

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe:

41

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe:

42

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe:

43

Properties of depth-first search

- Complete?** No: fails in infinite-depth spaces, spaces with loops
 - Modify to avoid repeated states along path
→ **complete in finite spaces**
- Time?** $O(b^m)$: terrible if m is much larger than d
 - but if solutions are dense, may be much faster than breadth-first
- Space?** $O(bm)$, i.e., linear space!
- Optimal?** No

44

Depth-limited search

= depth-first search with depth limit l ,
i.e., nodes at depth l have no successors

```

function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE([INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result ← RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred? ← true
    else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
    
```

45

Iterative deepening search

```

function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  inputs: problem, a problem
  for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
    
```

46

Iterative deepening search $l = 0$

Limit = 0

47

Iterative deepening search $l = 1$

Limit = 1

48

Iterative deepening search / =2

Limit = 2

49

Iterative deepening search / =3

Limit = 3

50

Iterative deepening search

- Number of nodes generated in a depth-limited search to depth d with branching factor b :

$$N_{DLS} = b^0 + b^1 + b^2 + \dots + b^{d-2} + b^{d-1} + b^d$$
- Number of nodes generated in an iterative deepening search to depth d with branching factor b :

$$N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \dots + 3b^{d-2} + 2b^{d-1} + 1b^d$$
- For $b = 10, d = 5$,
 - $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 = 111,111$
 - $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 = 123,456$
- Overhead = $(123,456 - 111,111)/111,111 = 11\%$

14 Jan 2004 CS 3243 - Blind Search 51

Properties of iterative deepening

- Complete? Yes
- Time? $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

52

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{(C^*/\epsilon)})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{(C^*/\epsilon)})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

53

Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!

54

Graph search

```

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 ← INSERT-ALL(EXPAND(node, problem), fringe)
  
```

55

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

56

INFORMED SEARCH

14 Jan 2004 CS 3243 - Blind Search 57

Best-first search

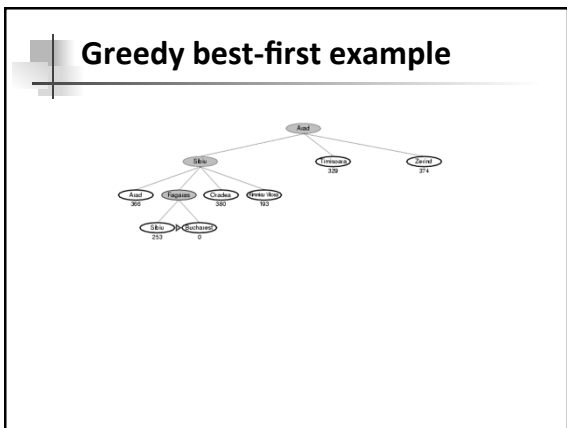
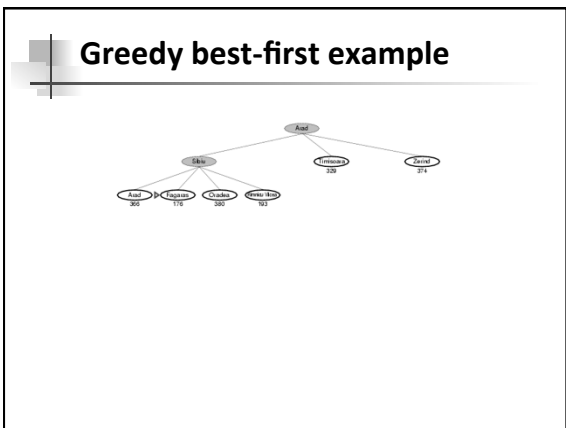
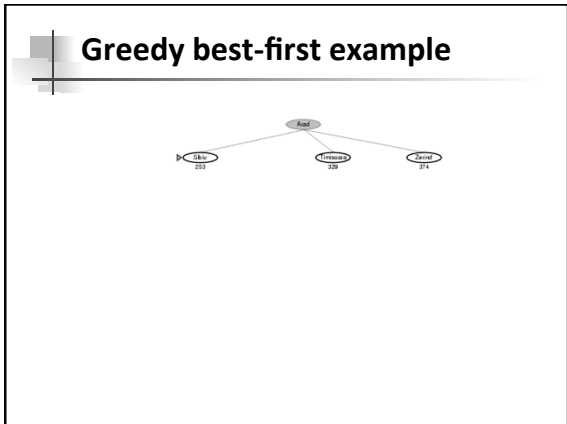
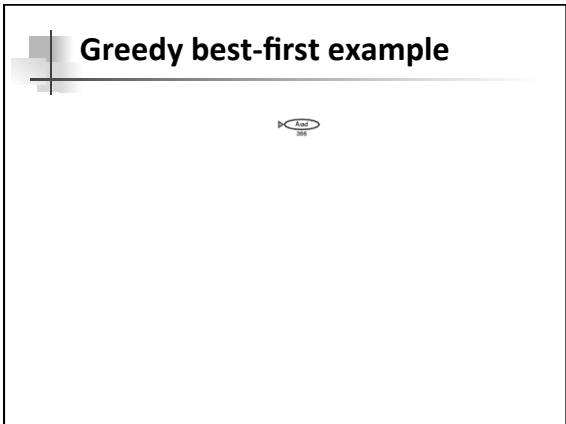
- Idea: use an evaluation function $f(n)$ for each node
 - estimate of "desirability"
 - Expand most desirable unexpanded node
- Implementation:
Order the nodes in fringe in decreasing order of desirability
- Special cases:
 - greedy best-first search
 - A* search

Romania with step costs in km

Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Cluj	160
Drobeta	242
Eforie	161
Giurgiu	176
Hirsova	77
Iasi	151
Lugoj	226
Mehadia	244
Neamt	234
Oradea	380
Pitesti	109
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

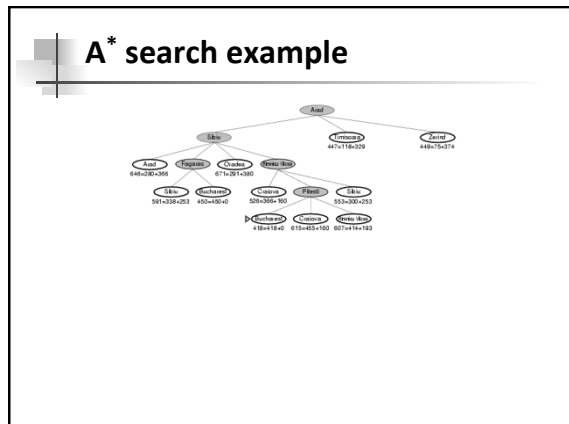
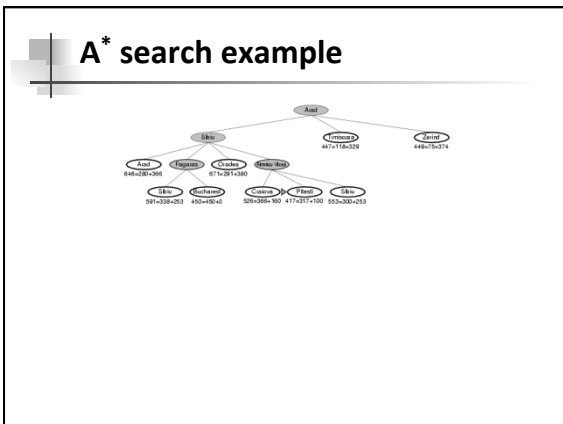
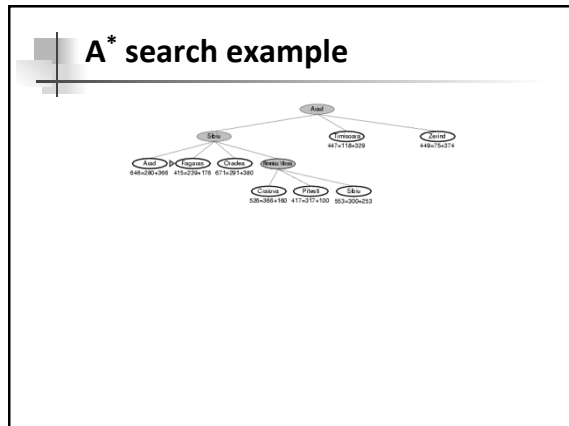
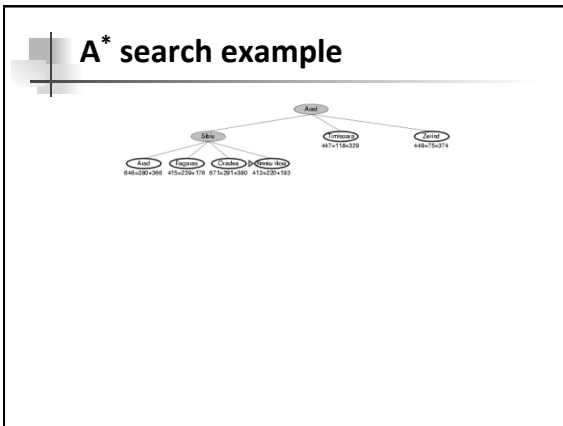
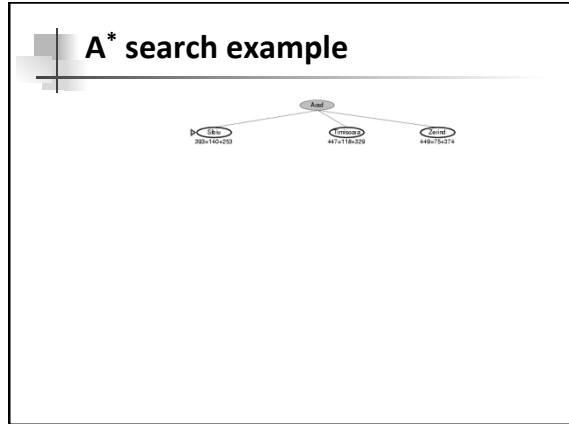
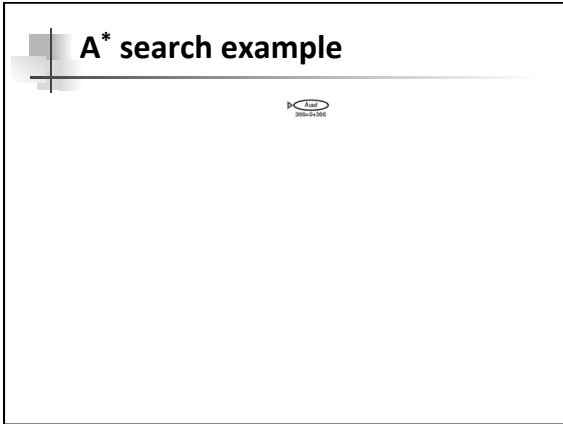
Greedy best-first search

- 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



- ### Properties of greedy best-first
- 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

- ### 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
 - $f(n)$ = est. total cost of path through n to goal

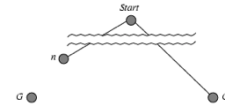


Admissible heuristics

- 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

Optimality of A^* (proof)

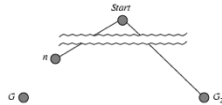
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G .



- $f(G_2) = g(G_2)$ 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

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 .



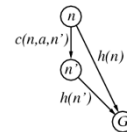
- $f(G_2) > f(G)$ from above
- $h(n) \leq h^*(n)$ since h is admissible
- $g(n) + h(n) \leq g(n) + h^*(n)$
- $f(n) \leq f(G)$

Hence $f(G_2) > f(n)$, and A^* will never select G_2 for expansion

Consistent heuristics

- A heuristic h 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

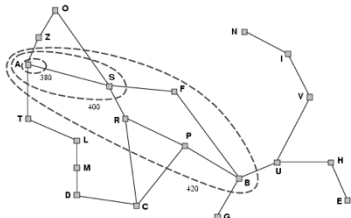
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

- i.e., $f(n)$ is non-decreasing along any path.

- Theorem: If $h(n)$ is consistent, A^* using GRAPH-SEARCH is optimal

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 $f_i < f_{i+1}$



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

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)

■ $h_1(S) = ?$
 ■ $h_2(S) = ?$

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

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

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

THE END

14 Jan 2004
CS 3243 - Blind Search
81

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)
- then h_2 dominates h_1
- 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