Solving Problems by Searching

Chapter 3

Outline

◇ Problem-solving agents

◇ Problem types

◇ Problem formulation

◇ Example problems

◇ Basic search algorithms

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
        goal ← Formulate-Goal(state)
        problem ← Formulate-Problem(state, goal)
        seq ← Search( problem)
        if seq is failure then return a null action
    action ← First(seq)
    seq ← Rest(seq)
    return action
```

Note: this is offline problem solving; solution executed "eyes closed."
Online problem solving involves acting without complete knowledge.

Example: Romania

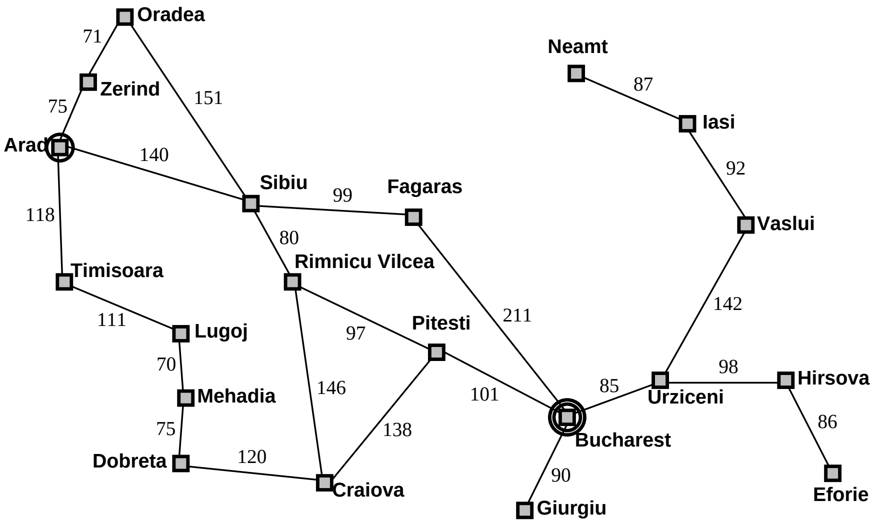
On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

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

Example: Romania



Problem types

Deterministic, fully observable $\Longrightarrow$ single-state problem
    Agent knows exactly which state it will be in; solution is a sequence

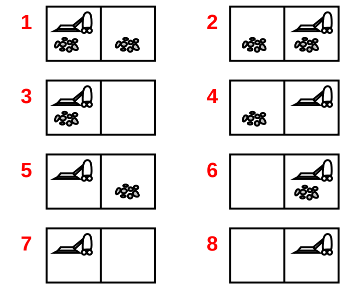
Non-observable $\Longrightarrow$ conformant problem
    Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable $\Longrightarrow$ contingency problem
    percepts provide **new** information about current state
    solution is a contingent plan or a policy
    often **interleave** search, execution

Unknown state space $\Longrightarrow$ exploration problem ("online")

Example: vacuum world

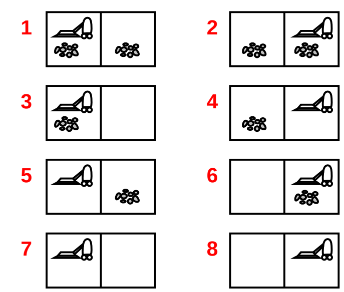
Single-state, start in #5. Solution??



Example: vacuum world

Single-state, start in #5. Solution??
$[Right, Suck]$

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??

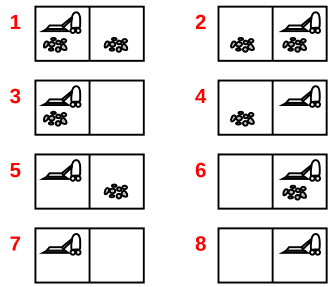


Example: vacuum world

Single-state, start in #5. Solution??
$[Right, Suck]$

Conformant, start in $\{1,2,3,4,5,6,7,8\}$
e.g., $Right$ goes to $\{2,4,6,8\}$. Solution??
$[Right, Suck, Left, Suck]$

Contingency
◇ Murphy's Law (non-deterministic): $Suck$ can dirty a clean carpet; start in #5
◇ Local sensing (partially-observable): dirt, location only, start in {#5,#7}.
Solution??

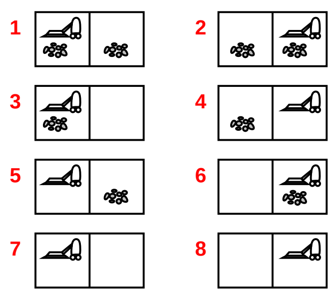


Example: vacuum world

Single-state, start in #5. Solution??
$[Right, Suck]$

Conformant, start in $\{1,2,3,4,5,6,7,8\}$
e.g., $Right$ goes to $\{2,4,6,8\}$. Solution??
$[Right, Suck, Left, Suck]$

Contingency
◇ Murphy's Law (non-deterministic): $Suck$ can dirty a clean carpet; start in #5
◇ Local sensing (partially-observable): dirt, location only, start in {#5,#7}.
Solution??
$[Right, \textbf{while } dirt \textbf{ do } Suck]$
$[Right, \textbf{if } dirt \textbf{ then } Suck]$



Single-state problem formulation

A problem is defined by four items:

◇ initial state e.g., "at Arad"

◇ successor function $S(x)$ = set of action–state pairs
e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind\rangle, \ldots\}$

◇ goal test, can be
explicit, e.g., x = "at Bucharest"
implicit, e.g., $NoDirt(x)$

◇ 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

Selecting a state space

Real world is absurdly complex
⇒ state space must be **abstracted** for problem solving

◇ (Abstract) state = set of real states

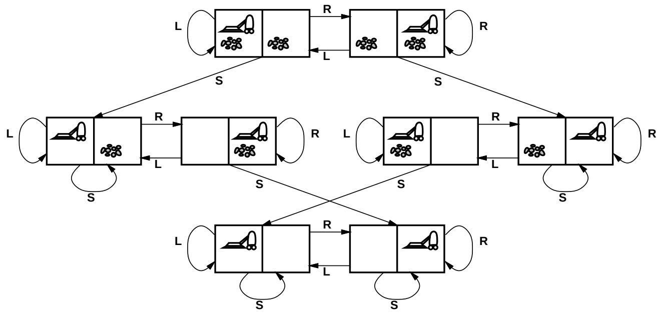
◇ (Abstract) action = complex combination of real actions
e.g., "Arad → Zerind" represents a complex set
of possible routes, detours, rest stops, etc.

◇ 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

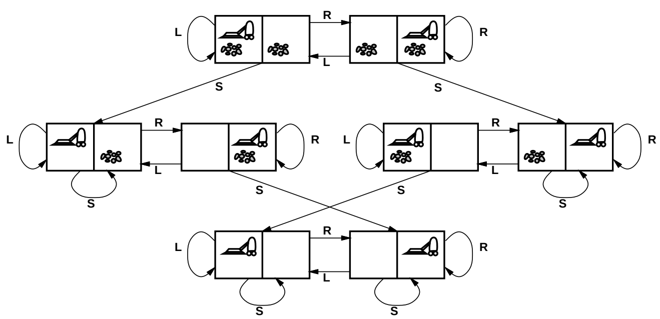
◇ Each abstract action should be "easier" than the original problem!

Example: vacuum world state space graph



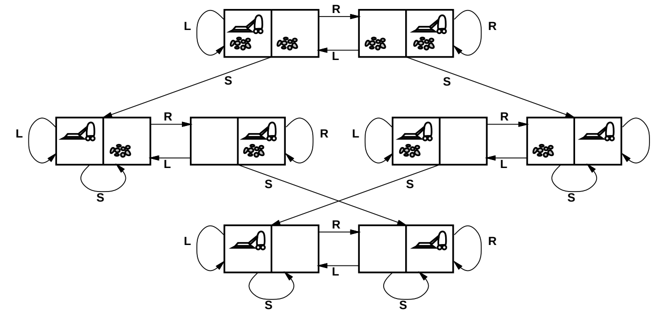
states??
actions??
goal test??
path cost??

Example: vacuum world state space graph



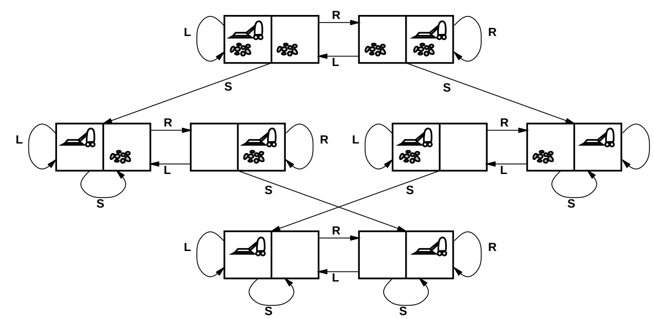
states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??
goal test??
path cost??

Example: vacuum world state space graph



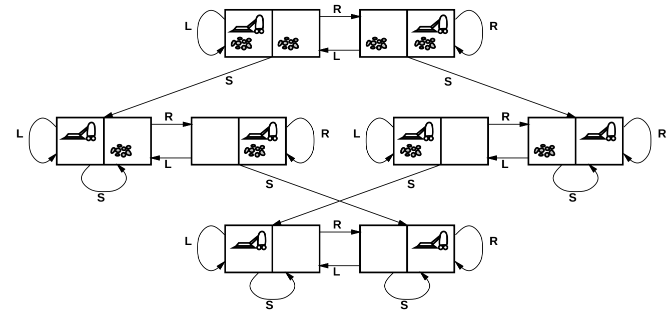
states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??
path cost??

Example: vacuum world state space graph



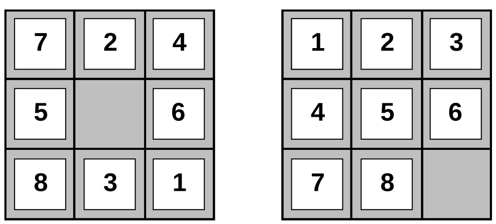
states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??: no dirt
path cost??

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: $Left$, $Right$, $Suck$, $NoOp$
goal test??: no dirt
path cost??: 1 per action (0 for $NoOp$)

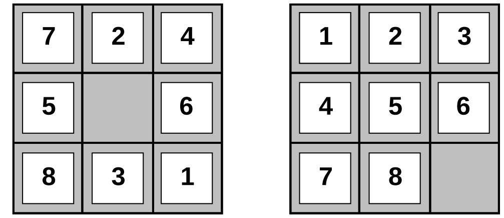
Example: The 8-puzzle



Start State Goal State

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

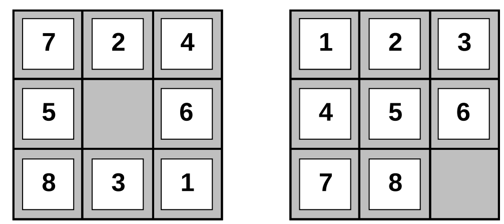
Example: The 8-puzzle



Start State Goal State

states??: integer locations of tiles (ignore intermediate positions)
actions??
goal test??
path cost??

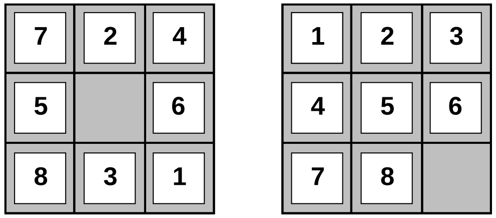
Example: The 8-puzzle



Start State Goal State

states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??
path cost??

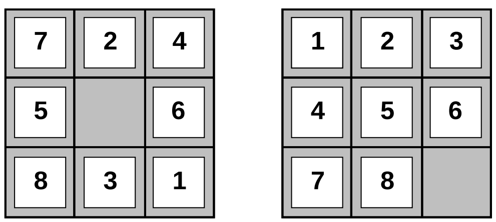
Example: The 8-puzzle



Start State Goal State

states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??

Example: The 8-puzzle

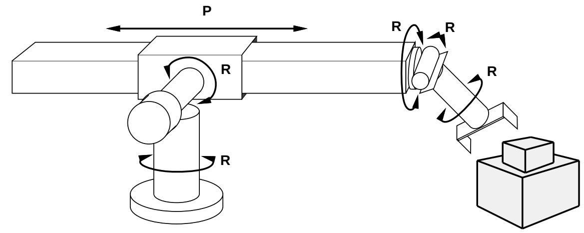


Start State Goal State

states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??: 1 per move

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

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 **with no robot included!**

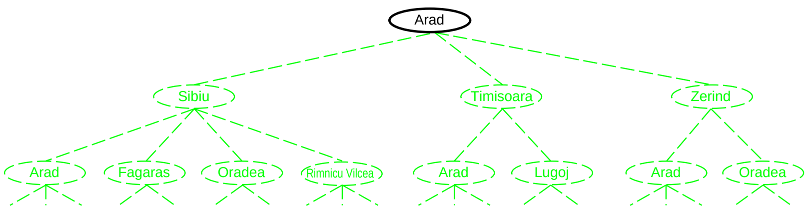
path cost??: time to execute

Tree search algorithms

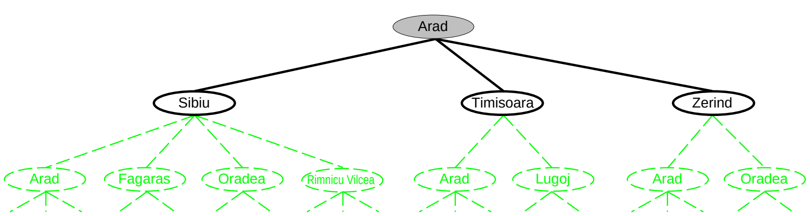
Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding states)

```
function Tree-Search( problem, strategy) returns a solution, or failure
   initialize the frontier using the initial state of problem
   loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier based on strategy
      if the node contains a goal state then return the corresponding solution
      else expand the chosen node and add the resulting nodes to the frontier
   end
```

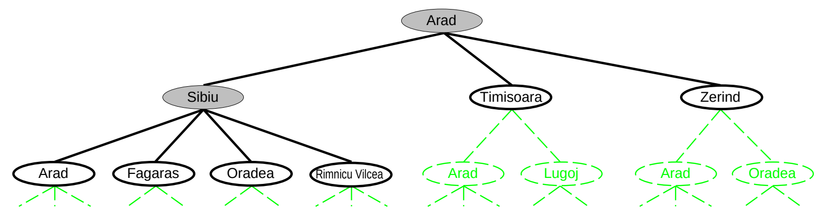
Tree search example



Tree search example

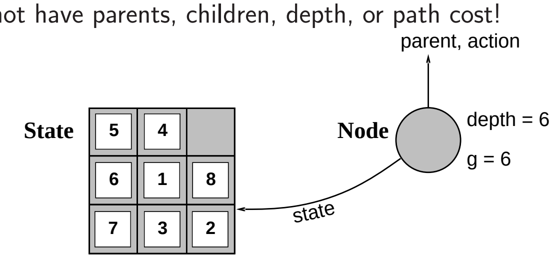


Tree search example



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 parent, children, depth, path cost $g(x)$
States do not have parents, children, depth, or path cost!



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

Search strategies

A 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/expanded
- 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 "shallowest" solution
- m—maximum depth of the state space (may be ∞)

Uninformed search strategies

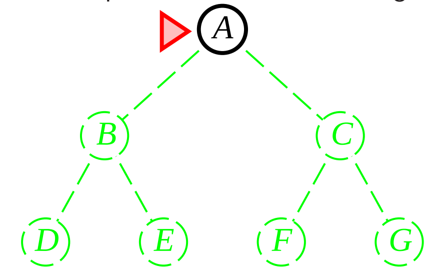
Uninformed strategies use only the information available in the problem definition

- ♢ Breadth-first search
- ♢ Uniform-cost search
- ♢ Depth-first search
- ♢ Depth-limited search
- ♢ Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

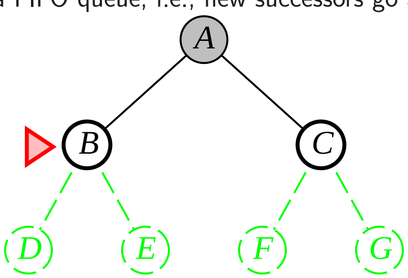
Implementation:
frontier is a FIFO queue, i.e., new successors go at end



Breadth-first search

Expand shallowest unexpanded node

Implementation:
frontier is a FIFO queue, i.e., new successors go at end

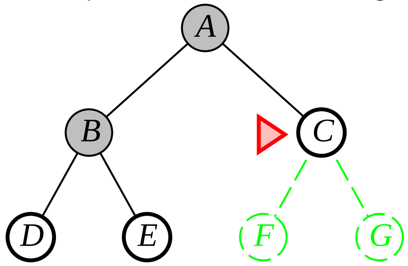


Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

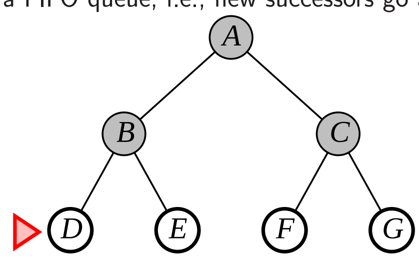


Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

Complete??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time (# of visited nodes)?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$

Time (# of generated nodes)?? $b+b^2+b^3+\ldots+b^d+(b^{d+1}-b) = O(b^{d+1})$

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time (# of generated nodes)?? $b+b^2+b^3+\ldots+b^d+(b^{d+1}-b) = O(b^{d+1})$

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time (# of generated nodes)?? $b+b^2+b^3+\ldots+b^d+(b^{d+1}-b) = O(b^{d+1})$

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:
frontier = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$ (lowest step cost)

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, $O(b^{\lceil C^*/\epsilon \rceil})$

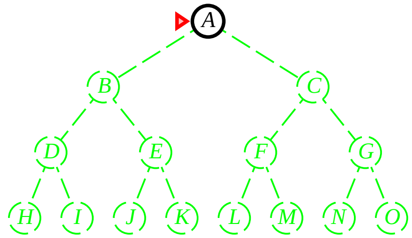
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

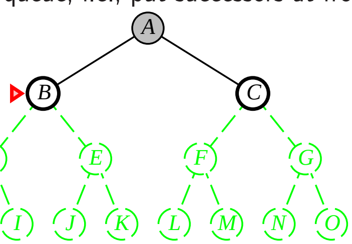


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

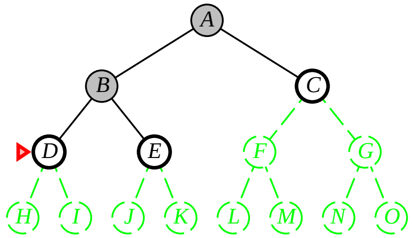


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

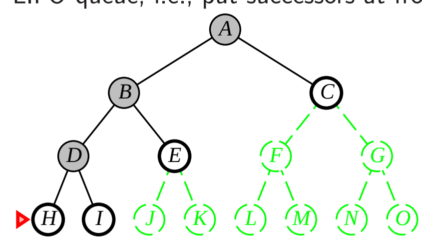


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

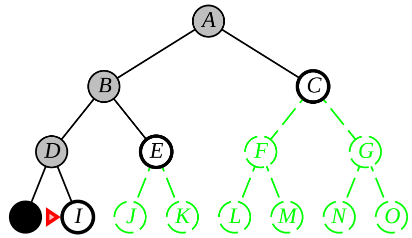


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

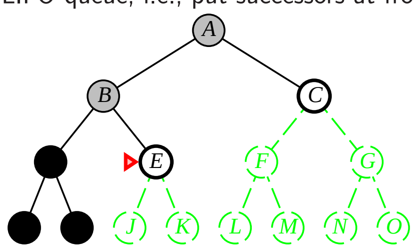


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

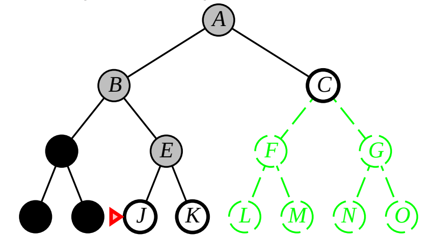


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

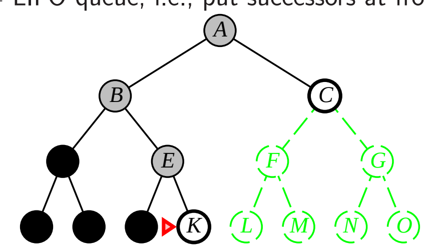


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

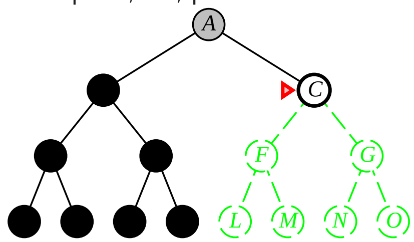


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

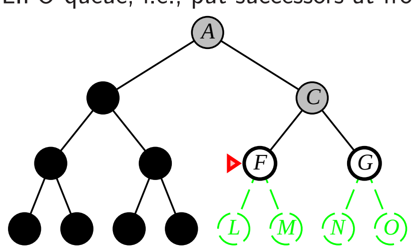


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

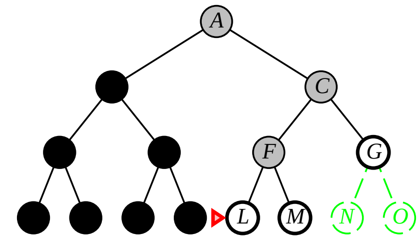


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

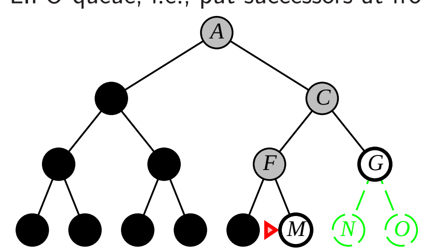


Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front



Properties of depth-first search

Complete??

Properties of depth-first search

Complete?? Yes: in finite spaces
No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path

Time??

Properties of depth-first search

Complete?? Yes: in finite spaces
No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path

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??

Properties of depth-first search

Complete?? Yes: in finite spaces
No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path

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??

Properties of depth-first search

Complete?? Yes: in finite spaces
No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path

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

Depth-limited search

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

Recursive implementation:

```
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 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
```


Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
   inputs: problem, a problem

   for depth ← 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH( problem, depth)
      if result ≠ cutoff then return result
   end
```


Iterative deepening search $l = 0$

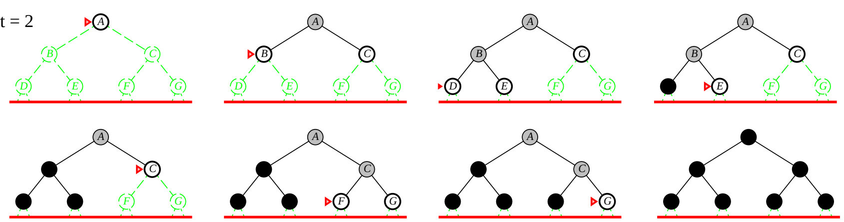
Limit = 0

Iterative deepening search $l = 1$

Limit = 1

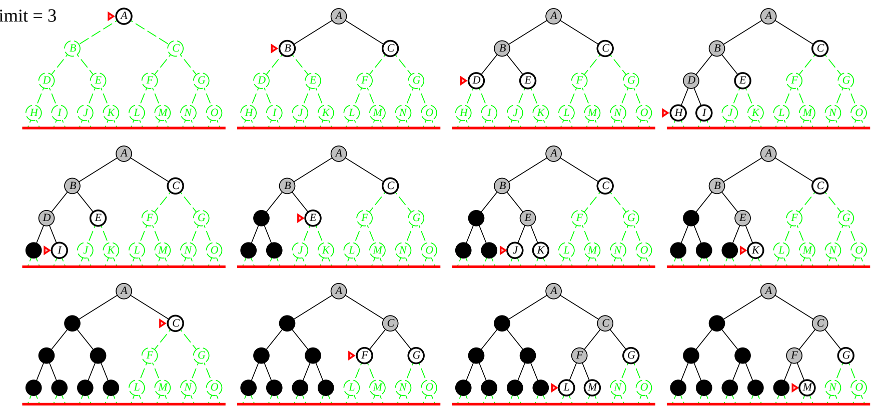
Iterative deepening search $l = 2$

Limit = 2



Iterative deepening search $l = 3$

Limit = 3



Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Time??

Properties of iterative deepening search

Complete?? Yes

Time (# of generated nodes)?? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space??

Properties of iterative deepening search

Complete?? Yes

Time (# of generated nodes)?? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal??

Properties of iterative deepening search

Complete?? Yes

Time (# of generated nodes)?? $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost = 1
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$\begin{aligned} N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N_{visited}(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110 \\ N_{generated}(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100 \end{aligned}$$

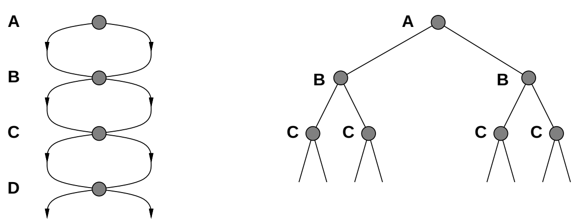
BFS can be modified to apply goal test when a node is **generated**

Summary of algorithms

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

Repeated states

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



Graph search

```
function GRAPH-SEARCH( problem) returns a solution, or failure
   frontier ← a list with node from the initial state of problem
   explored ← an empty set
   loop do
      if frontier is empty then return failure
      node ← REMOVE-FRONT(frontier)
      if node contains a goal state then return SOLUTION(node)
      add STATE[node] to explored
      expand node
      add to frontier the resulting nodes that are
         not in explored or
         not in frontier or
         [better than the corresponding nodes in frontier in some algs]
   end
```

frontier (aka *fringe* or *open*); *explored* (aka *visited* or *closed*)

Informed Search

◇ So far the search algorithms are "uninformed"—independent to the problems

◇ Informed search–incorporating knowledge related to the problem for guiding search

Best-first search

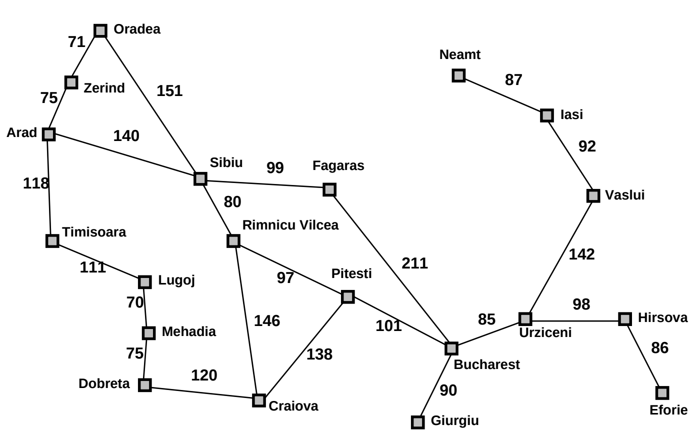
Idea: use an evaluation function for each node
 – estimate of "desirability"

$\Rightarrow$ Expand most desirable unexpanded node

Implementation:
frontier is a queue sorted in decreasing order of desirability

Special cases:
 greedy search
 A* search

Romania with step costs in km



Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search

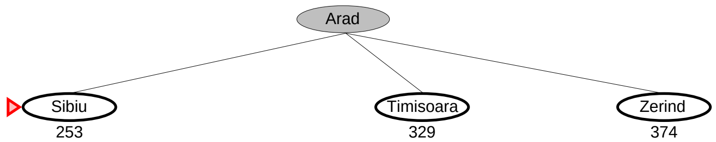
Evaluation function $h(n)$ (**h**euristic)
 $=$ estimate of cost from n to the closest goal

E.g., $h_{\mathrm{SLD}}(n)$ $=$ straight-line distance from n to Bucharest

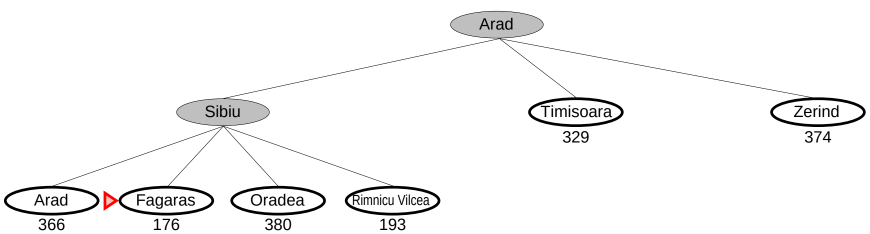
Greedy search expands the node that **appears** to be closest to goal

Greedy search example

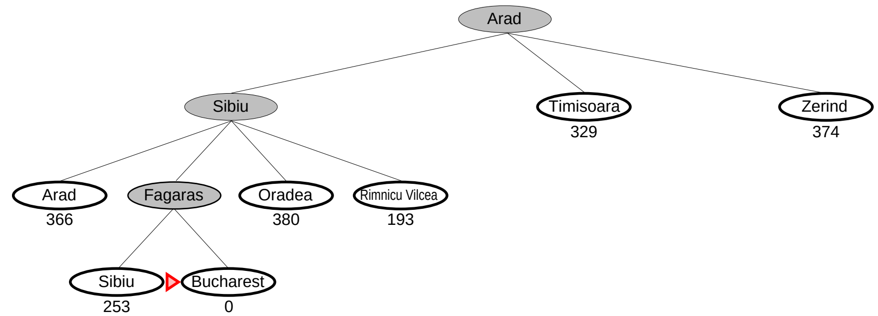
Greedy search example



Greedy search example



Greedy search example



Properties of greedy search

Complete??

Properties of greedy search

Complete?? Yes–Complete in finite space with repeated-state checking
No–can get stuck in loops, e.g., with Oradea as goal,
Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Time??

Properties of greedy search

Complete?? Yes–Complete in finite space with repeated-state checking
No–can get stuck in loops, e.g.,
Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space??

Properties of greedy search

Complete?? Yes–Complete in finite space with repeated-state checking
No–can get stuck in loops, e.g.,
Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$, but a good heuristic can give dramatic improvement

Optimal??

Properties of greedy search

Complete?? Yes–Complete in finite space with repeated-state checking
No–can get stuck in loops, e.g.,
Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$, but a good heuristic can give dramatic improvement

Optimal?? No

A* search

Idea: avoid expanding paths that are already expensive

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

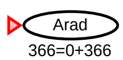
$g(n)$ = cost so far to reach n
$h(n)$ = estimated cost to goal from n
$f(n)$ = estimated total cost of path through n to goal

A* search uses an admissible heuristic
i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the **true** cost from n.
(Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)

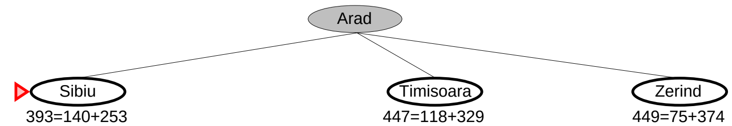
E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance

Theorem: A* search is optimal

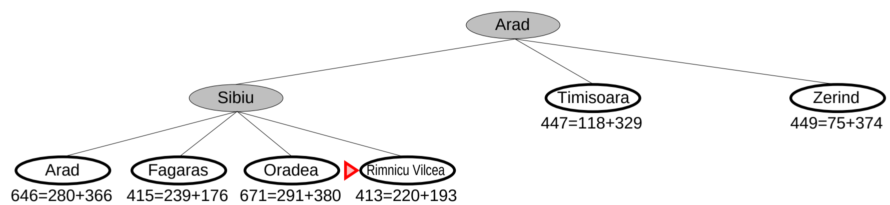
A* search example



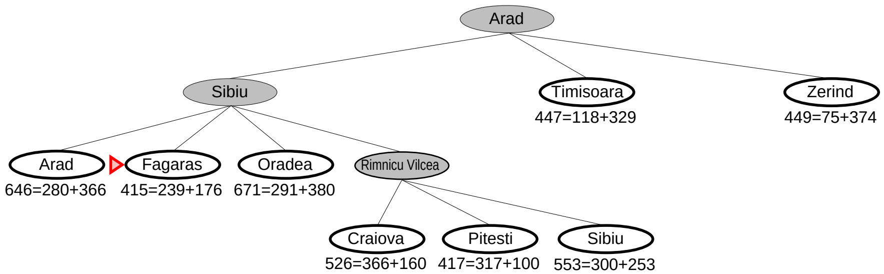
A* search example



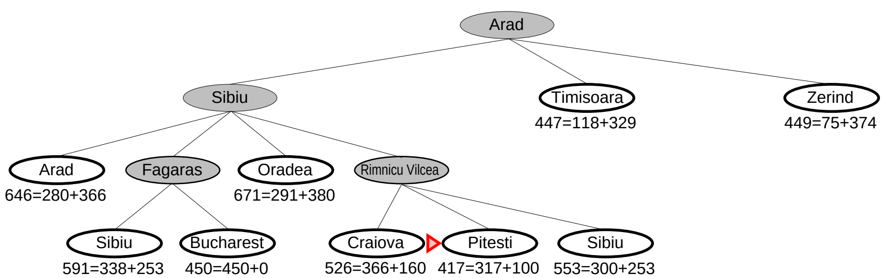
A* search example



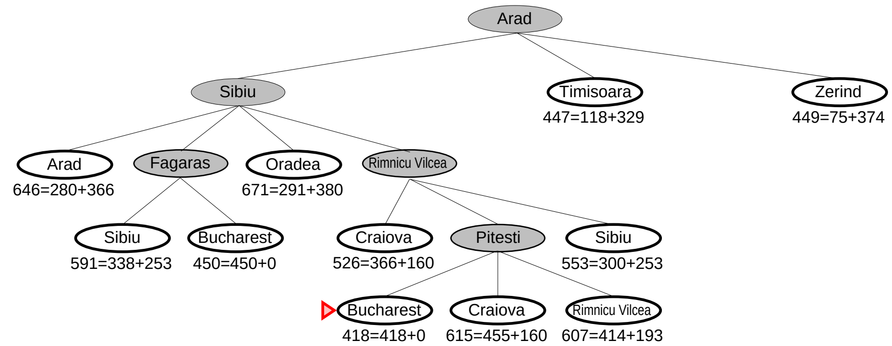
A* search example



A* search example

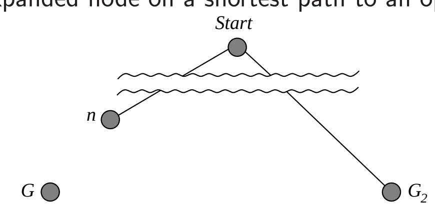


A* search example



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.



Want to prove: $f(G_2) > f(n)$ [A* will never select G_2 for expansion]

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

Properties of A*

Complete??

Properties of A*

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

Time??

Properties of A*

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

Time?? Exponential in [relative error in $h \times d$]

Space??

Properties of A*

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

Time?? Exponential in [relative error in $h \times d$]

Space?? Exponential

Optimal??

Properties of A*

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

Time?? Exponential in [relative error in $h \times d$]

Space?? Exponential

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished, where $f_{i+1} > f_i$

C^* is the cost for the optimal solution:

A* expands all nodes with $f(n) < C^*$
A* expands some nodes with $f(n) = C^*$
A* expands no nodes with $f(n) > C^*$

A* vs Uniform-cost Search

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

Isn't UCS just A* with $h(n)$ being zero (admissible)?

Both are optimal, why is A* usually "faster?"

A* vs Uniform-cost Search

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

Isn't UCS just A* with $h(n)$ being zero (admissible)?

Both are optimal, why is A* usually "faster?"

Consider $h(n)$ is perfect, $f(n)$ is ?

A* vs Uniform-cost Search

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

Isn't UCS just A* with $h(n)$ being zero (admissible)?

Both are optimal, why is A* usually "faster?"

Consider $h(n)$ is perfect, $f(n)$ is the actual total path cost.

A* vs Uniform-cost Search

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

Isn't UCS just A* with $h(n)$ being zero (admissible)?

Both are optimal, why is A* usually "faster?"

Consider $h(n)$ is perfect, $f(n)$ is the actual total path cost.

- If n doesn't lead to a goal state, $f(n) = \infty$
  - A* ?
  - UCS ?

A* vs Uniform-cost Search

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

Isn't UCS just A* with $h(n)$ being zero (admissible)?

Both are optimal, why is A* usually "faster?"

Consider $h(n)$ is perfect, $f(n)$ is the actual total path cost.

- If n doesn't lead to a goal state, $f(n) = \infty$
  - A* doesn't explore n.
  - UCS doesn't know and keeps on exploring n (and its sucessors).
- If n doesn't lead to the optimal goal, but n^* does: $f(n) > f(n^*)$
  - A* ?
  - UCS ?

A* vs Uniform-cost Search

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

Isn't UCS just A* with $h(n)$ being zero (admissible)?

Both are optimal, why is A* usually "faster?"

Consider $h(n)$ is perfect, $f(n)$ is the actual total path cost.

- If n doesn't lead to a goal state, $f(n) = \infty$
  - A* doesn't explore n.
  - UCS doesn't know and keeps on exploring n (and its sucessors).
- If n doesn't lead to the optimal goal, but n^* does: $f(n) > f(n^*)$
  - A* doesn't explore n and *only* explores n^* !
  - UCS doesn't know and keeps on exploring n (and its sucessors).

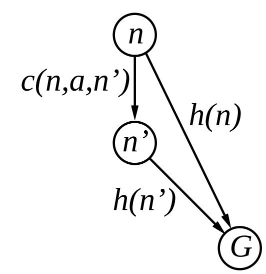
In terms of speed, the worst case for A* is when $h(n)$ is zero, but we don't use $h(n) = 0$.

Consistency

Consider n' is a successor of n, a heuristic is consistent if

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

Let's find the relationship between $f(n)$ and $f(n')$:



$$\begin{aligned} f(n') &= g(n') + h(n') \\ f(n') &= g(n) + c(n, a, n') + h(n') \\ f(n') &\ge g(n) + h(n) \qquad \text{[since } h \text{ is consistent]} \\ f(n') &\ge f(n) \end{aligned}$$

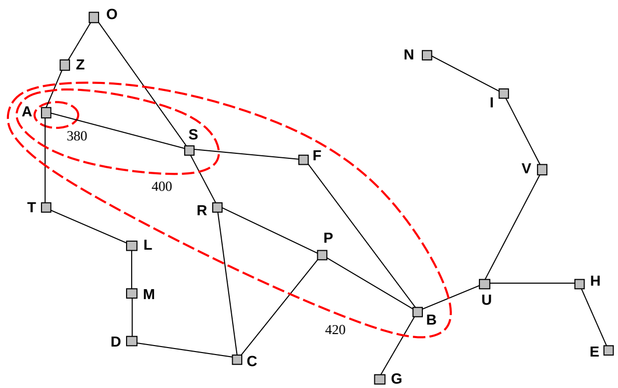
i.e. $f(n)$ values for a sequence of nodes along *any* path are nondecreasing (similar to $g(n)$ values in UCS).

A* using GRAPH-SEARCH is optimal if $h(n)$ is consistent (using a similar argument as UCS).

Optimality of A* (consistent heuristics)

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 $f_i < f_{i+1}$



Admissible vs Consistent Heuristics

Consistency is a slightly stronger/stricter requirement than admissibility.

$consistentHeuristics \subset admissibleHeuristics$

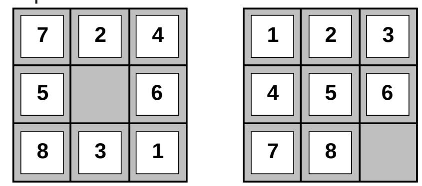
Admissible heuristics are usually consistent.

Not easy to concort admissible, but not consistent heuristics.

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)



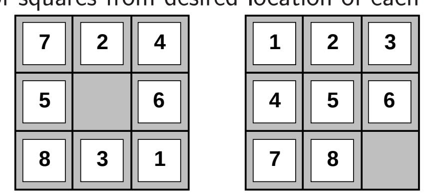
Start State   Goal State

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

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)



Start State   Goal State

$h_1(S) =$?? 6
$h_2(S) =$?? 4+0+3+3+1+0+2+1 = 14

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 dominates h_1 and is faster for search

Typical search costs:

$d = 14$ IDS = 3,473,941 nodes
A$^*(h_1)$ = 539 nodes
A$^*(h_2)$ = 113 nodes
$d = 24$ IDS ≈ 54,000,000,000 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

Relaxed problems

Admissible heuristics can be derived from the **exact**
solution cost of a **relaxed** version of the problem

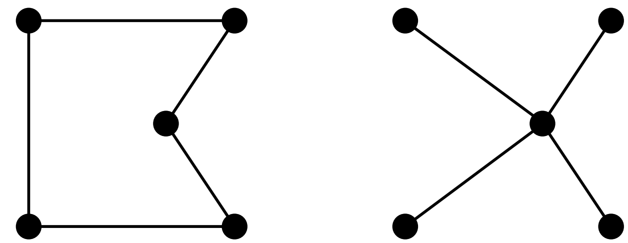
If the rules of the 8-puzzle are relaxed so that a tile can move **anywhere**,
then $h_1(n)$ gives the shortest solution

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

Relaxed problems contd.

Well-known example: travelling 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

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

Graph search can be exponentially more efficient than tree search

Summary

Heuristic functions estimate costs of shortest paths

Good heuristics can dramatically reduce search cost

Greedy best-first search expands lowest h
– incomplete and not always optimal

A* search expands lowest $g + h$
– complete and optimal
– also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems