GHENT UNIVERSITY

FACULTY OF ENGINEERING AND ARCHITECTURE

E016350 - Artificial Intelligence

Lecture 13
Part 2

Problem-solving agents
Search strategies

Aleksandra Pizurica

Ghent University
Fall 2024

Outline

[R&N], Chapter 3

This presentation is based on: S. Russel and P. Norvig: *Artificial Intelligence: A Modern Approach*, (Fourth Ed.), denoted as [R&N] and the resource page http://aima.cs.berkeley.edu/

Problem types

Deterministic, fully observable (topic of this lecture)

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

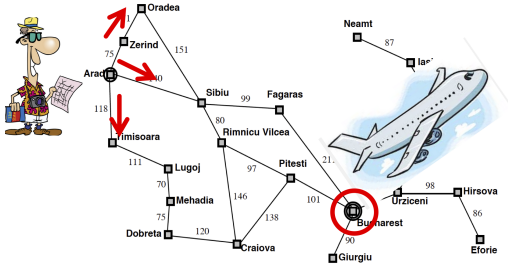
often **interleave** search, execution

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

Example: Romania

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

Formulate goal:
be in Bucharest

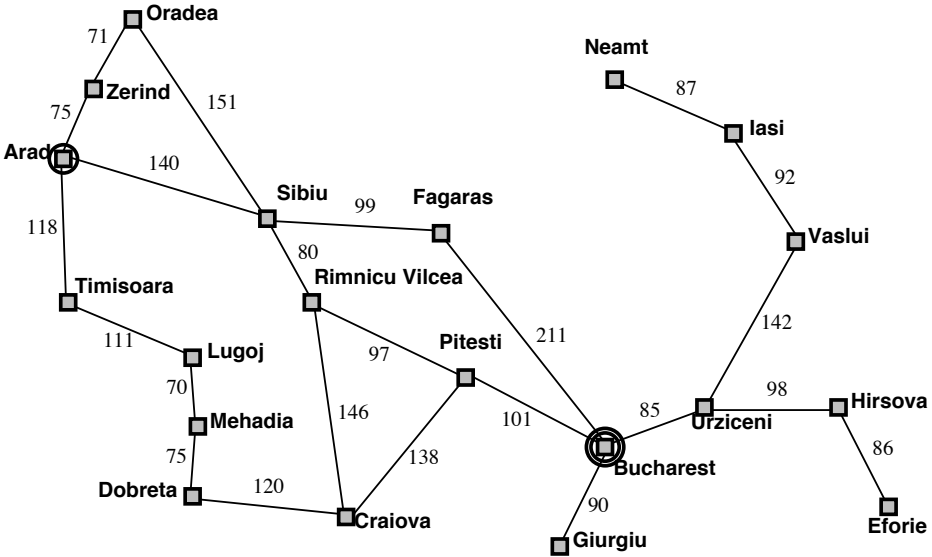


Formulate problem:

- states: various cities
- actions: drive between cities

Find solution:
sequence of cities,
e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Outline

Problem-solving agents

Restricted form of general agent:

```
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)
  action ← RECOMMENDATION(seq, state)
  seq ← REMAINDER(seq, state)
  return action
```

Note: this is offline problem solving; In an online problem solving the agent doesn't know what the state space is, and has to build a model of it as it acts.

Problem formulation

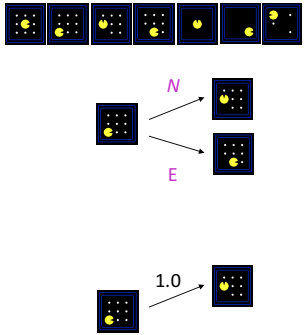
A search problem is formally defined by:

- state space – a set of possible states
  and initial state e.g., $Arad$
- actions available to the agent
  e.g., $\textsc{Actions}(Arad) = [ToSibiu, ToTimisoara, ToZerind]$
  and transition model, i.e., successor function
  describes what each action does
  e.g., $\textsc{Result}(Arad, ToZerind) = Zerind$
- goal test, can be
  explicit, e.g., $x = Bucharest$
  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

Problem formulation

- state space – a set of possible states
  and initial state e.g., particular configuration
- actions available to the agent
  e.g., N, W, E, S
  and transition model, i.e., successor function
  describes what each action does
  e.g., the resulting configuration
- goal test, can be
  Sometimes can be satisfied by multiple states,
  e.g., *"Eat all the dots"*
- path cost (additive)
  e.g., sum of distances, number of actions
  $c(x, a, y)$ 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
$\Rightarrow$ state space must be **abstracted** for problem solving
(Abstract) state = set of real states

(Abstract) action = complex combination of real actions

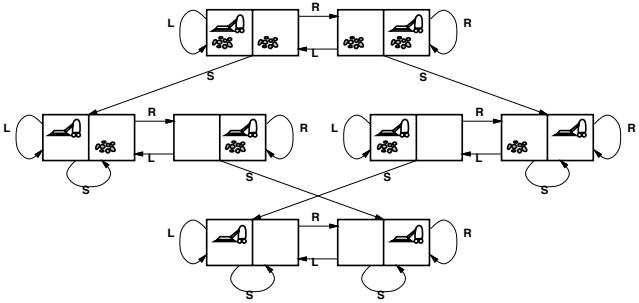
e.g., "Arad $\rightarrow$ Zerind" represents a complex set
of possible routes, detours, rest stops, etc.

For guaranteed realizability, **any** real state "in Arad"
must get to some real state "in Zerind"

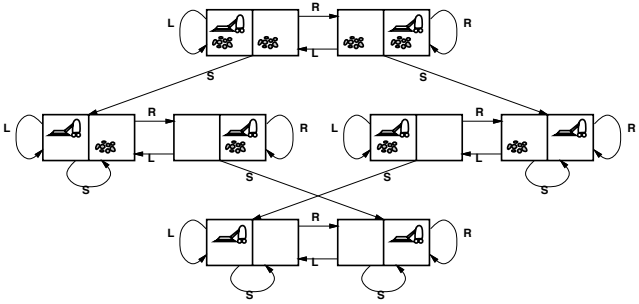
(Abstract) solution =
set of real paths that are solutions in the real world
Each abstract action should be "easier" than the original problem!

Outline

Example: vacuum world state space graph

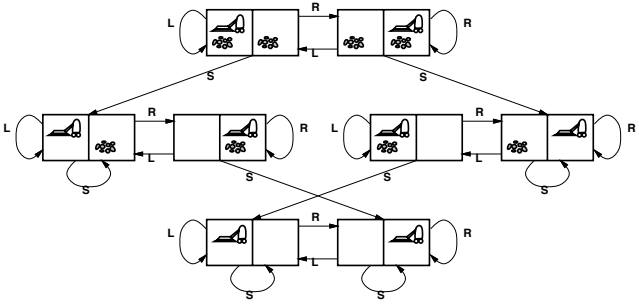


Example: vacuum world state space graph



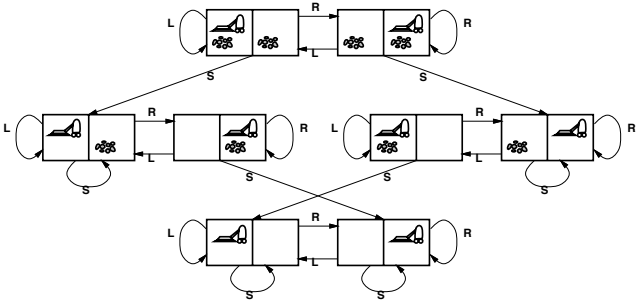
states??

Example: vacuum world state space graph



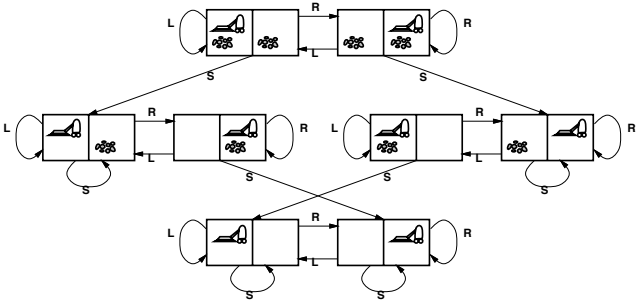
states?? integer dirt and robot locations (ignore dirt amounts etc.)

Example: vacuum world state space graph



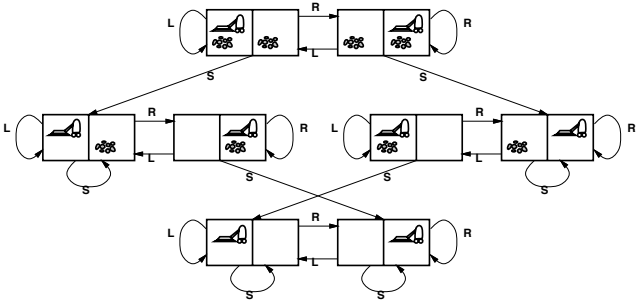
states?? integer dirt and robot locations (ignore dirt amounts etc.)
actions??

Example: vacuum world state space graph



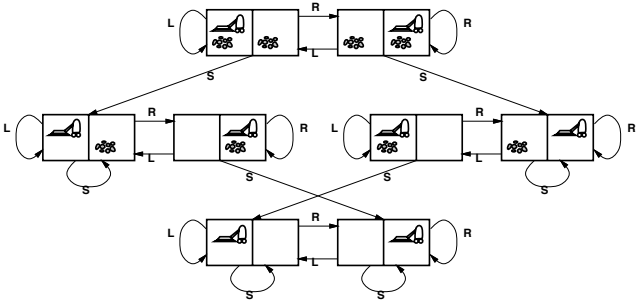
states?? integer dirt and robot locations (ignore dirt amounts etc.)
actions?? $Left$, $Right$, $Suck$, $NoOp$

Example: vacuum world state space graph



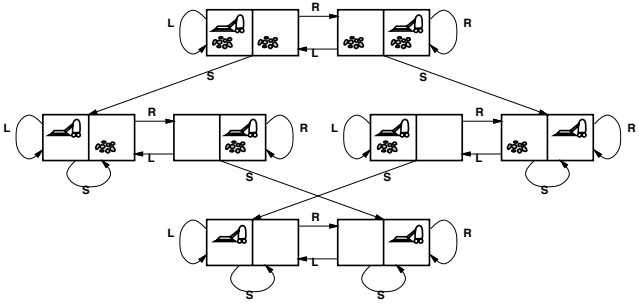
states?? integer dirt and robot locations (ignore dirt amounts etc.)
actions?? $Left$, $Right$, $Suck$, $NoOp$
goal test??

Example: vacuum world state space graph



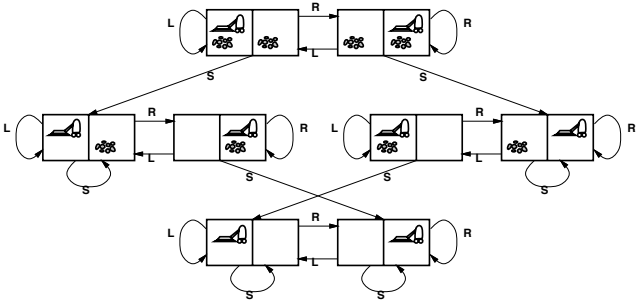
states?? integer dirt and robot locations (ignore dirt amounts etc.)
actions?? $Left$, $Right$, $Suck$, $NoOp$
goal test?? no dirt

Example: vacuum world state space graph



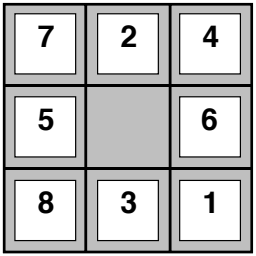
states?? integer dirt and robot locations (ignore dirt amounts etc.)
actions?? $Left$, $Right$, $Suck$, $NoOp$
goal test?? no dirt
path cost??

Example: vacuum world state space graph

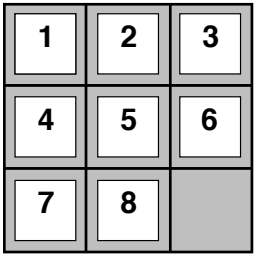


states?? integer dirt and robot locations (ignore dirt amounts etc.)
actions?? $Left$, $Right$, $Suck$, $NoOp$
goal test?? no dirt
path cost?? 1 per action (0 for $NoOp$)

Example: The 8-puzzle



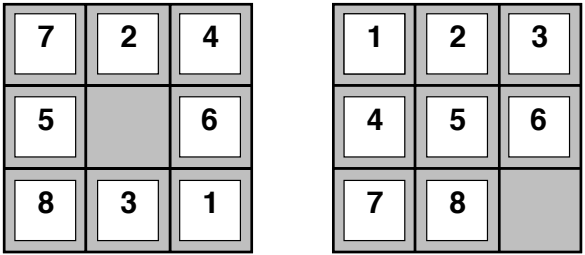
Start State



Goal State

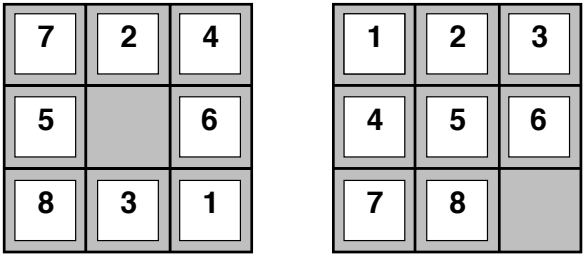
states??

Example: The 8-puzzle



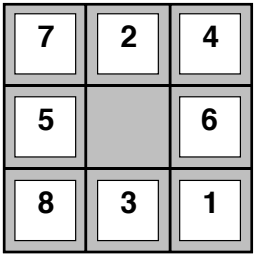
states?? integer locations of tiles (ignore intermediate positions)

Example: The 8-puzzle

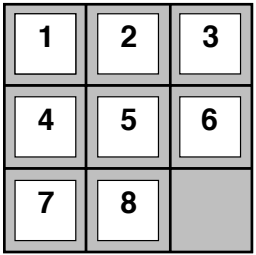


states?? integer locations of tiles (ignore intermediate positions)
actions??

Example: The 8-puzzle



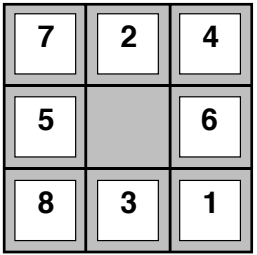
Start State



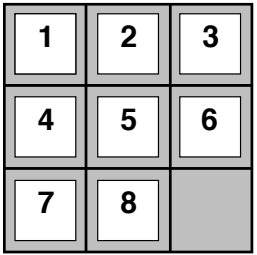
Goal State

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

Example: The 8-puzzle



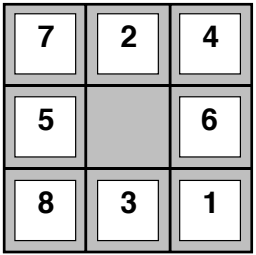
Start State



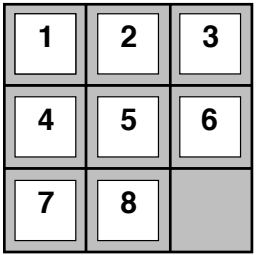
Goal State

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

Example: The 8-puzzle



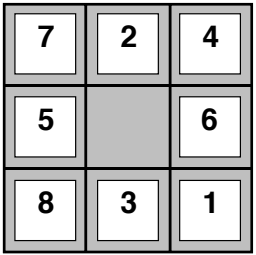
Start State



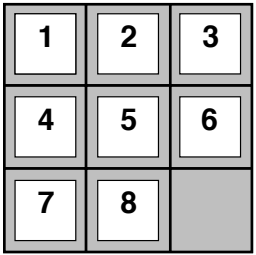
Goal State

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

Example: The 8-puzzle



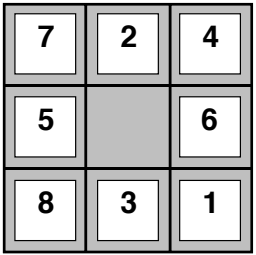
Start State



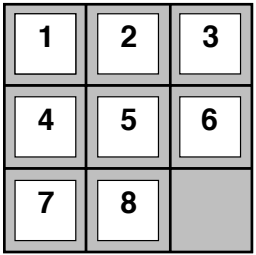
Goal State

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



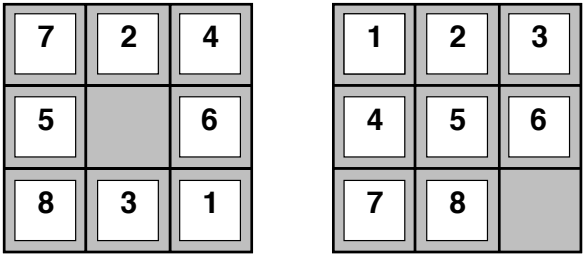
Start State



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

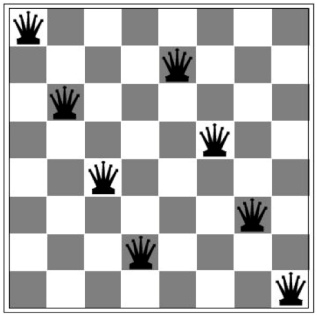
Example: The 8-puzzle



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]

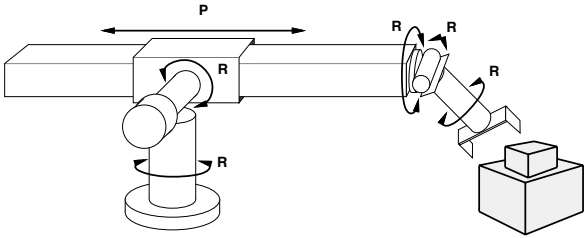
Example: other toy problems



$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

Left: the n-queens problem; Right: the Knuth sequence

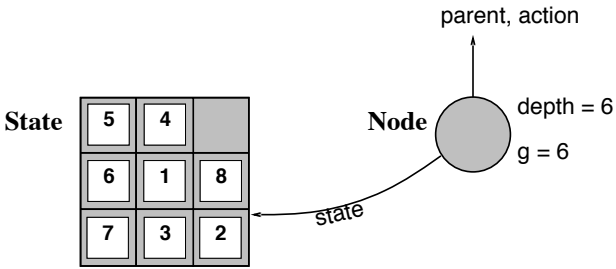
Example: robotic assembly



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

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!



State-space graphs and search trees

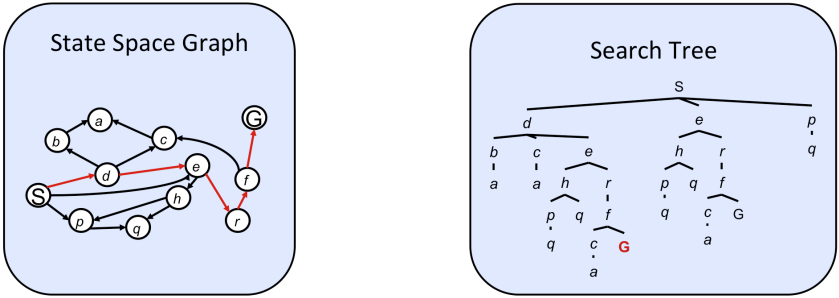
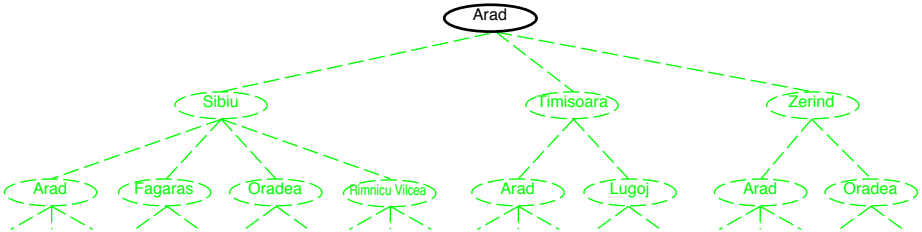


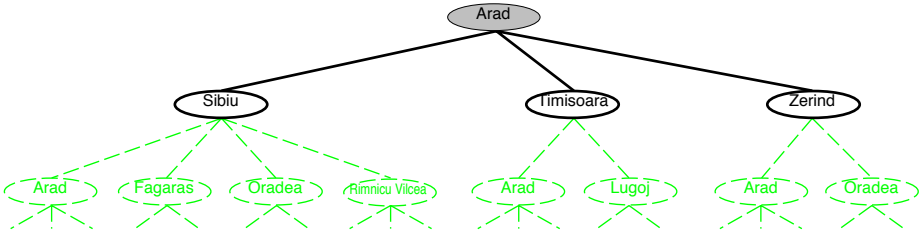
Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

- All possible action sequences starting at the initial state form a search tree
- The branches are actions
- The nodes correspond to the states in the state space of the problem.
- A node with no children is a leaf node
- The set of nodes available for expansion at a given moment is the frontier (fringe)

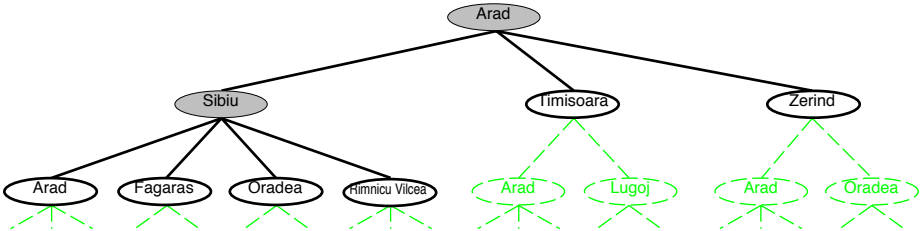
Tree search example



Tree search example



Tree search example



General tree search

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
  end
```

General tree search: implementation

```
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 node
      fringe ← INSERTALL(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
```

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 least-cost solution
- m – maximum depth of the state space (may be ∞)

Outline

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

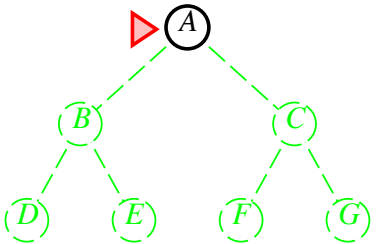
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-first search (BFS)

Expand shallowest unexpanded node

Implementation:

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

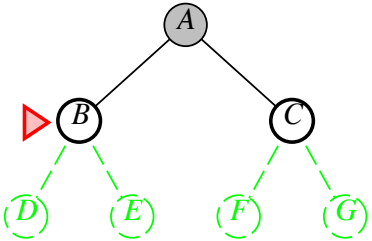


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

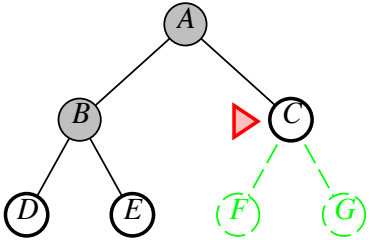


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

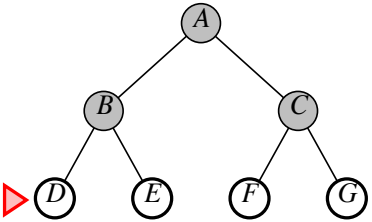


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe 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)

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exp. in d

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exp. in d

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exp. in d

Space?? $O(b^d)$ (keeps every node in memory)

Properties of breadth-first search

Complete?? Yes (if b is finite)
Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exp. in d
Space?? $O(b^d)$ (keeps every node in memory)
Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)
Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exp. in d
Space?? $O(b^d)$ (keeps every node in memory)
Optimal?? Yes (if cost $= 1$ per step); not optimal in general

Properties of breadth-first search

Complete?? Yes (if b is finite)
Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exp. in d
Space?? $O(b^d)$ (keeps every node in memory)
Optimal?? Yes (if cost $= 1$ per step); not optimal in general

Space complexity is the big problem!

Properties of breadth-first search

Complete?? Yes (if b is finite)
Time?? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exp. in d
Space?? $O(b^d)$ (keeps every node in memory)
Optimal?? Yes (if cost = 1 per step); not optimal in general

Space complexity is the big problem!

Note: the goal test is applied when the nodes are generated.

Example: time and memory requirements for breadth-first search

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

The numbers in the table correspond to:
branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

fringe = queue ordered by path cost, lowest first

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

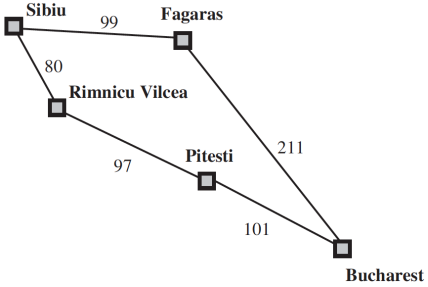
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

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

Uniform-cost search

Expand least-cost unexpanded node
Optimal in general (when costs per step can differ)

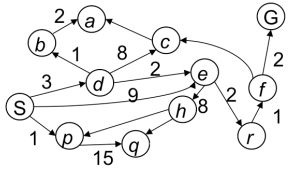


Next to queue-ordering by path cost, two important differences wrt. BFS:

- The goal test is applied when the node is **selected for expansion**
- Before accepting a better candidate path, the frontier nodes are tested

The complexity can be much greater than $O(b^d)$, but not easily characterized in terms of b and d (depends on the cost of the optimal solution C^*)

Uniform-cost search: example



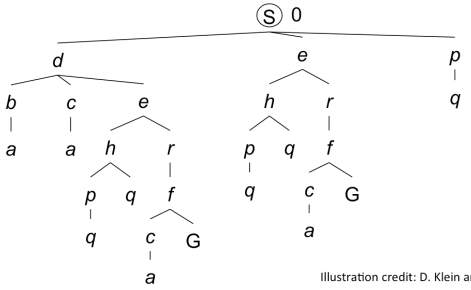
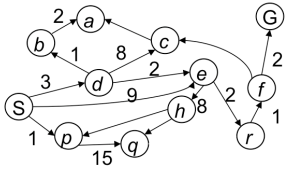


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Uniform-cost search: example



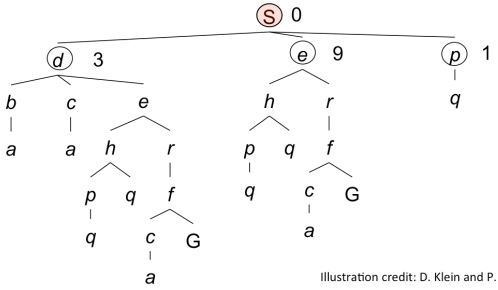


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Uniform-cost search: example

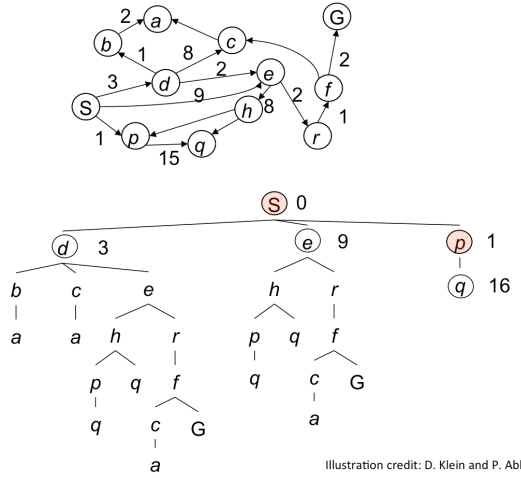
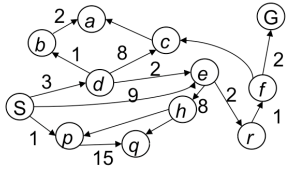


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Uniform-cost search: example



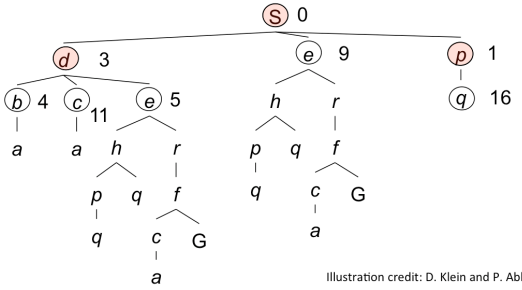
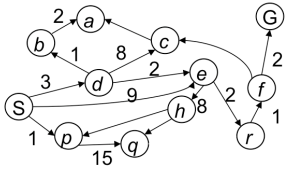


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Uniform-cost search: example



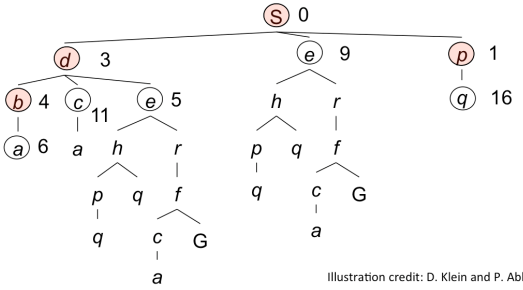
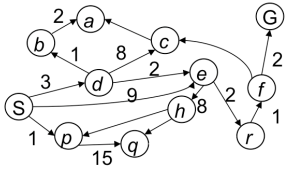


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Uniform-cost search: example



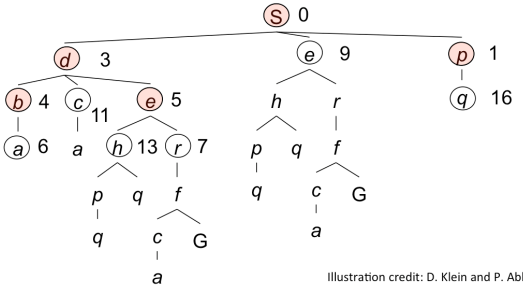
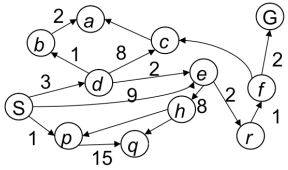


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Uniform-cost search: example



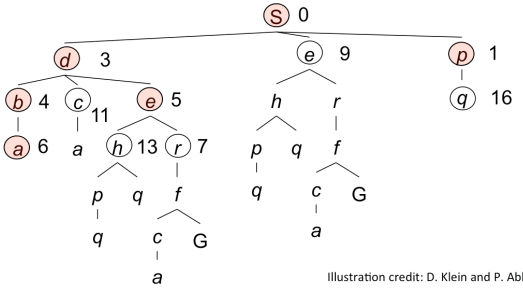
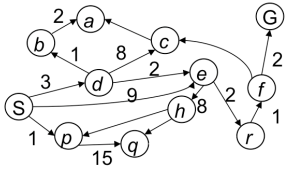


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Uniform-cost search: example



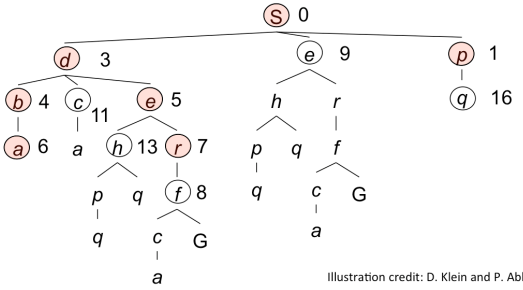
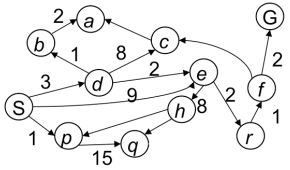


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Uniform-cost search: example



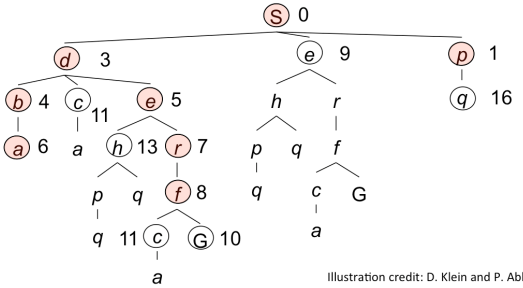
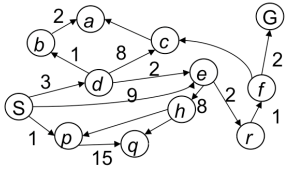


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Uniform-cost search: example



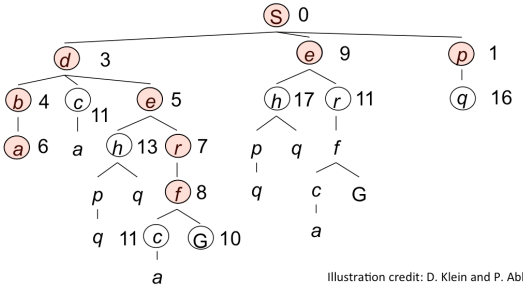


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Uniform-cost search: example

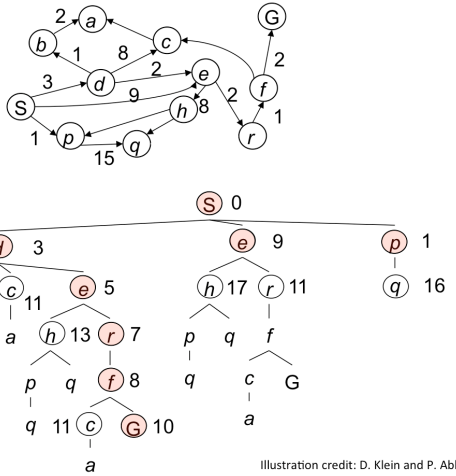


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Uniform-cost search: example

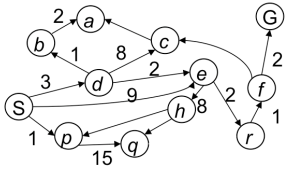


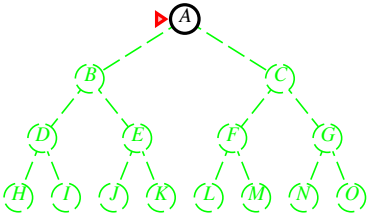
Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Depth-first search (DFS)

Expand deepest unexpanded node

Implementation:

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

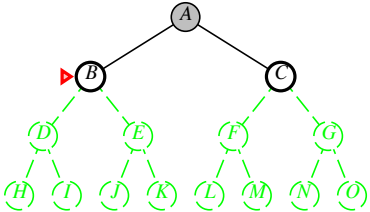


Depth-first search

Expand deepest unexpanded node

Implementation:

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

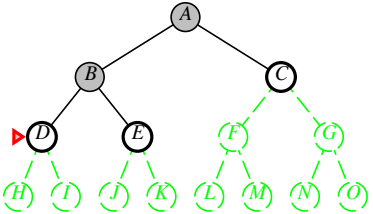


Depth-first search

Expand deepest unexpanded node

Implementation:

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

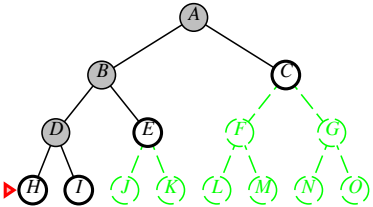


Depth-first search

Expand deepest unexpanded node

Implementation:

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

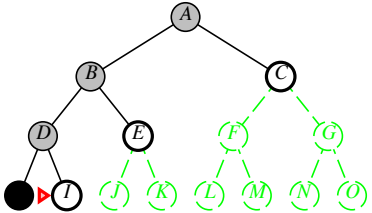


Depth-first search

Expand deepest unexpanded node

Implementation:

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

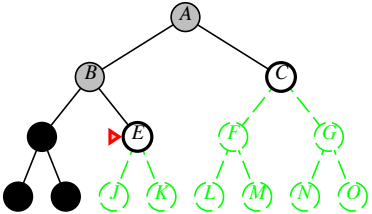


Depth-first search

Expand deepest unexpanded node

Implementation:

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

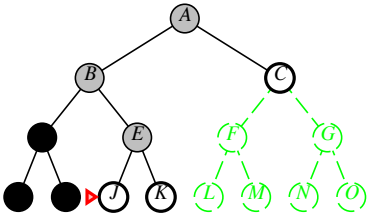


Depth-first search

Expand deepest unexpanded node

Implementation:

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

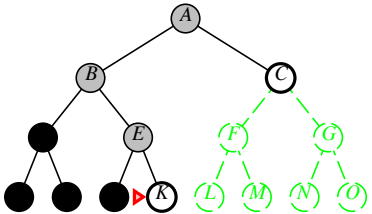


Depth-first search

Expand deepest unexpanded node

Implementation:

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

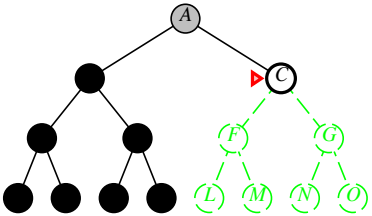


Depth-first search

Expand deepest unexpanded node

Implementation:

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

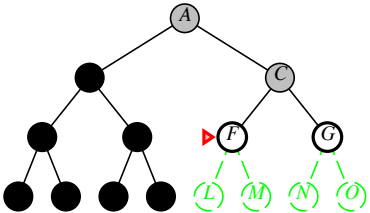


Depth-first search

Expand deepest unexpanded node

Implementation:

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

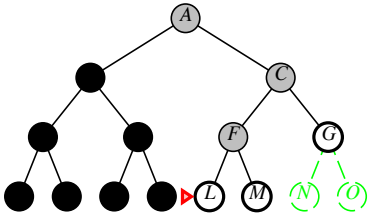


Depth-first search

Expand deepest unexpanded node

Implementation:

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

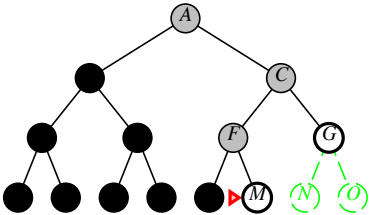


Depth-first search

Expand deepest unexpanded node

Implementation:

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



Properties of depth-first search

Complete??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ 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

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ 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??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ 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!

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ 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??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ 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

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ 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

Example: $b = 10$, 1000 bytes/node, $d = 16$ and assume that nodes at the same depth as the goal node have no successors.

$\rightarrow$ Depth-first search requires 160 kilobytes instead of 10 exabytes with BFS!

Depth-first tree search has thus become the basic "workhorse" of many areas of AI

Depth-limited search

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

Recursive implementation:

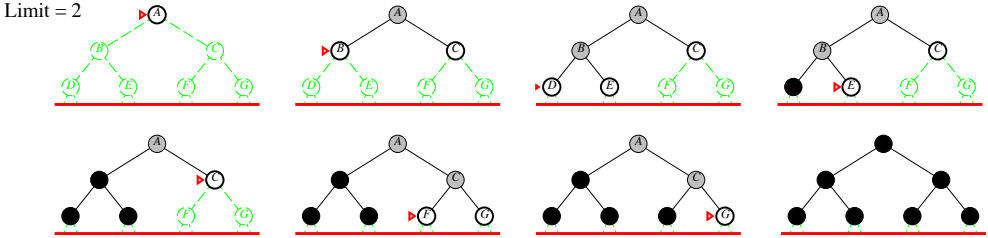
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

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 (IDS) $l = 0$

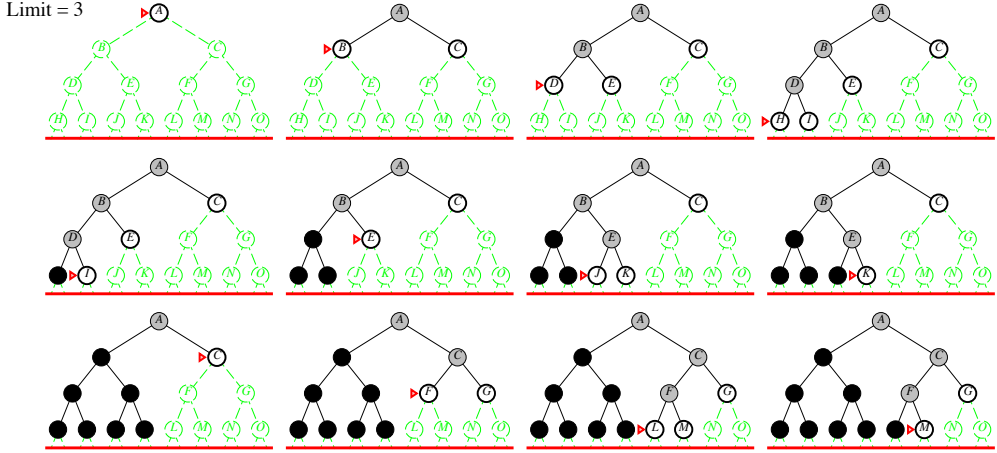
Limit = 0

Iterative deepening search $l = 1$

Iterative deepening search $l = 2$



Iterative deepening search $l = 3$



Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Properties of iterative deepening search

Complete?? Yes

Time??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d)b^1 + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$

Properties of iterative deepening search

Complete?? Yes

Time?? $(d)b^1 + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$

Space??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d)b^1 + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$

Space?? $O(bd)$

Properties of iterative deepening search

Complete?? Yes

Time?? $(d)b^1 + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$

Space?? $O(bd)$

Optimal??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d)b^1 + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$

Can be modified to explore uniform-cost tree

Properties of iterative deepening search

Complete?? Yes

Time?? $(d)b^1 + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$

Can be modified to explore uniform-cost tree

Numerical comparison of the worst-case IDS (solution at far right leaf) and BFS for $b = 10$ and $d = 5$:

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

Properties of iterative deepening search

Complete?? Yes

Time?? $(d)b^1 + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$

Can be modified to explore uniform-cost tree

Numerical comparison of the worst-case IDS (solution at far right leaf) and BFS for $b = 10$ and $d = 5$:

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

In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

[1] complete if b is finite, and the state space either has a solution or is finite.
[2] complete if all action costs are $\geq \epsilon \geq 0$.
[3] cost-optimal if action costs are all identical.
[4] if both directions are breadth-first or uniform-cost

Repeated states

How big is the search tree for this problem (from the start state S)?

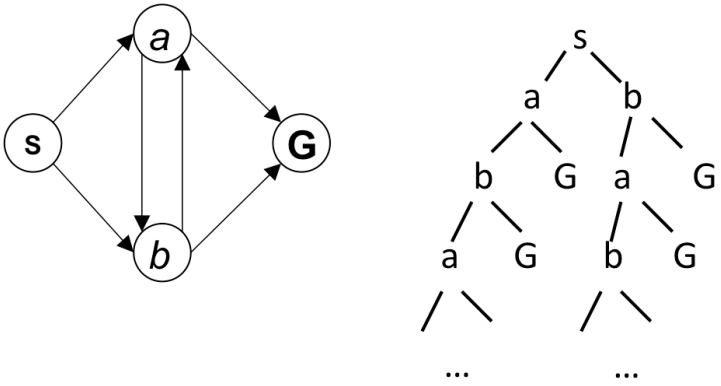


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

Problems with tree search in general

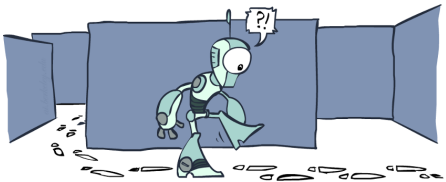


Illustration credit: D. Klein and P. Abbeel: Intro to AI, http://ai.berkeley.edu

- Can get stuck searching the same cycle in the state space graph forever
- Even without loops, we can visit the same node multiple times
  (multiple ways to reach it)
  exponentially more work!

Solution? Keep track of states that you've visited (maintain a **closed set**)

This is the idea of graph search

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 node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Avoids repeated states by keeping a closed set (of already visited nodes)

Outline

Informed search strategies

Idea: use an evaluation function
to estimate desirability of each node

$\Rightarrow$ Expand most desirable unexpanded node

Implementation:

– Fringe is a queue sorted in decreasing order of desirability

Strategies:

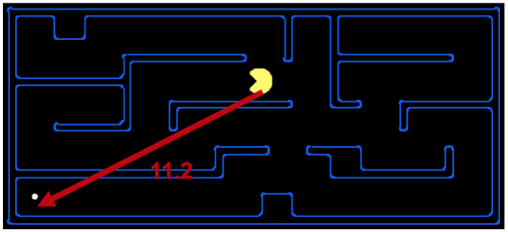
- Greedy best-first search
- A^* search

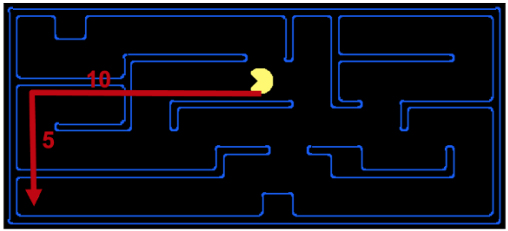
Search Heuristics

Heuristics are functions that

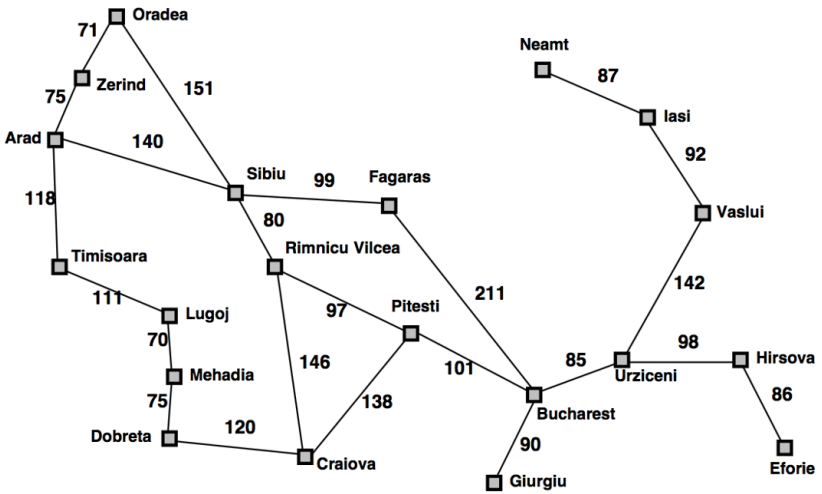
- estimate how close a state is to a goal
- are designed for a particular search problem

Examples: Euclidean distance, Manhattan distance





Search Heuristics



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 best-first search

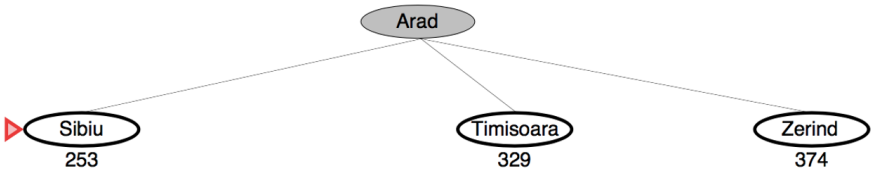
- Evaluation function $h(n)$ (heuristic) = estimate of cost from n to the closest goal

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

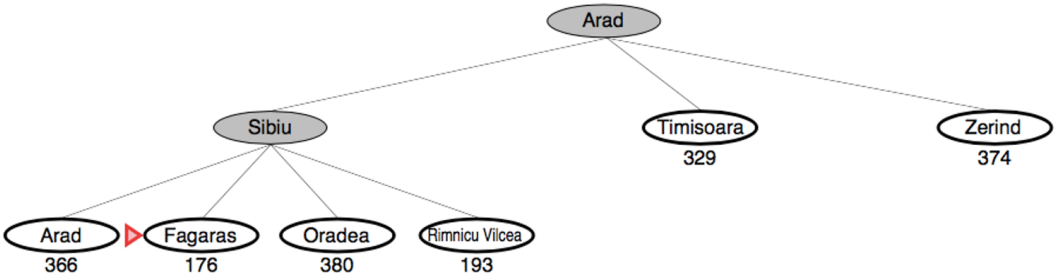
- Greedy search expands the node that appears to be closest to goal.

Greedy search example

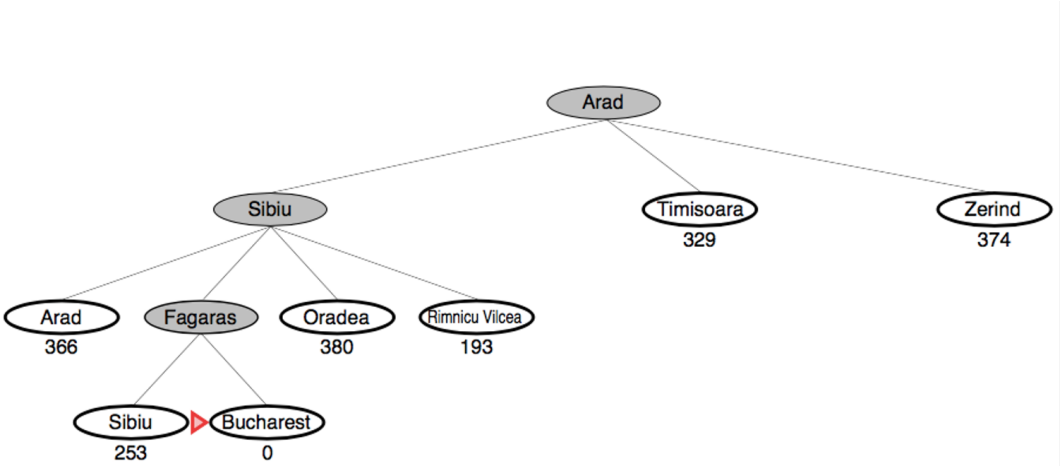
Greedy search example



Greedy search example



Greedy search example



Properties of greedy search

Complete?? – No, can get stuck in loops

Complete in finite spaces 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?? – 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

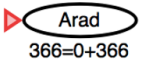
$f(n) =$ estimated total cost of path through n to goal

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)

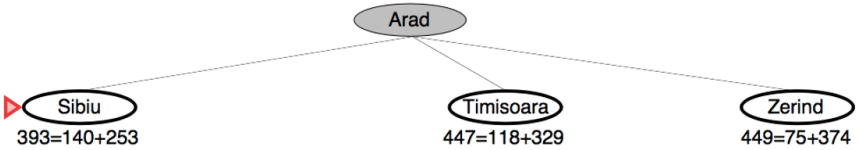
E.g. $h_{SLD}(n)$ never overestimates the actual road distance

Theorem: A^* search is optimal

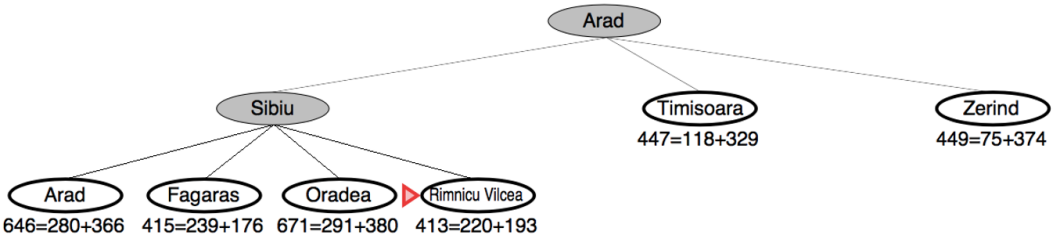
A^* search example



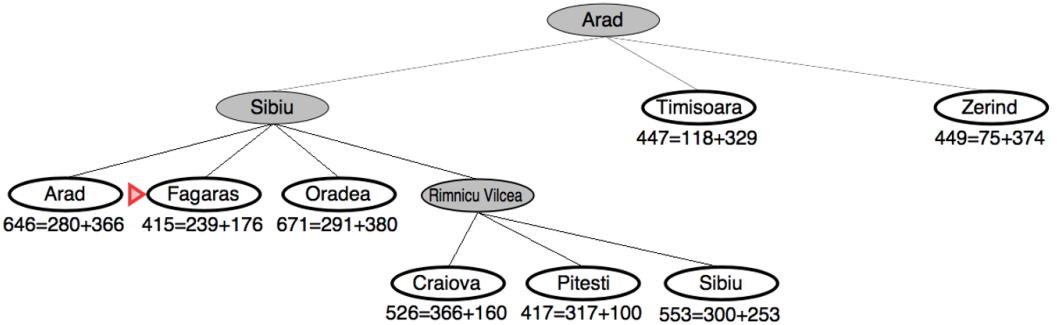
A^* search example



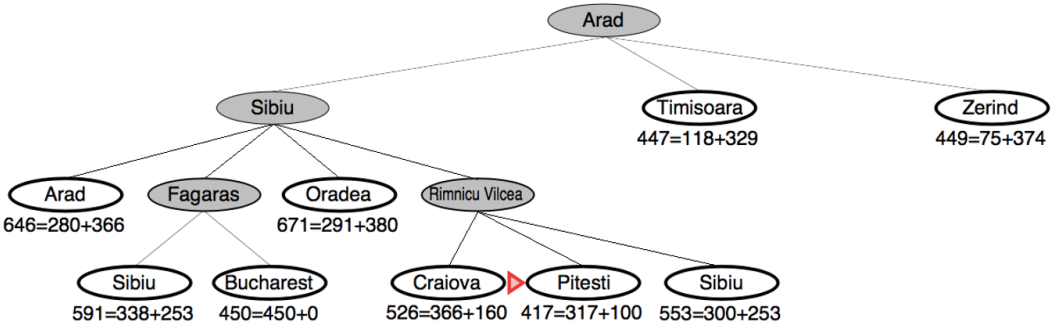
A^* search example



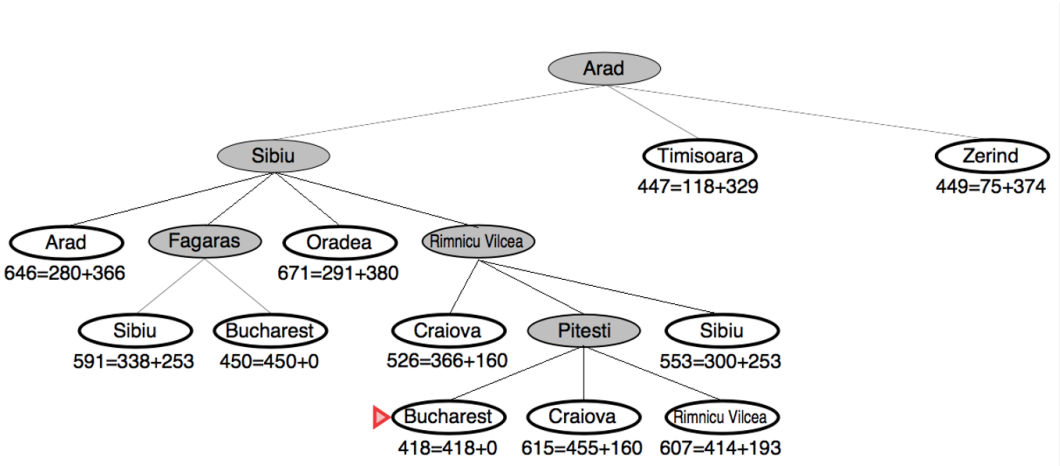
A^* search example



A^* search example

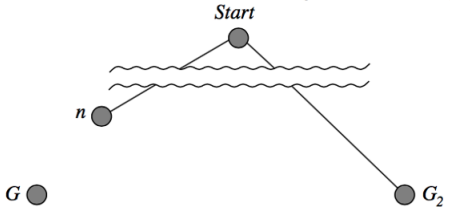


A^* search example



Optimality of A^*

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_1.



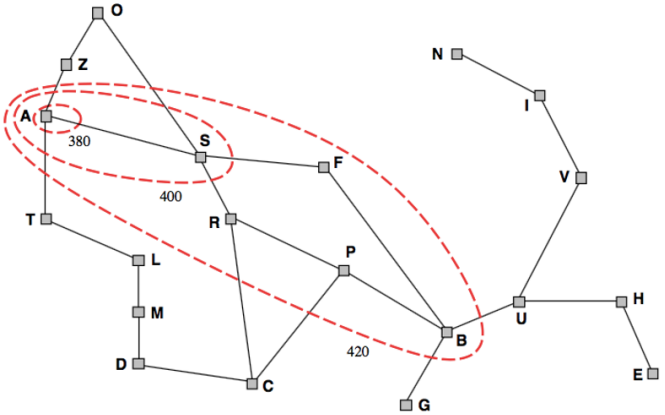
$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\
&\geq f(n) && \text{since } h \text{ is admissible}
\end{aligned}
$$

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

Optimality of A^*

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^*

Optimality of A^*

Complete??

Optimality of A^*

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

Optimality of A^*

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

Time??

Optimality of A^*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$
Time?? Exponential in [relative error in $h \times$ length of soln.]

Optimality of A^*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$
Time?? Exponential in [relative error in $h \times$ length of soln.]
Space??

Optimality of A^*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$
Time?? Exponential in [relative error in $h \times$ length of soln.]
Space?? Keeps all nodes in memory

Optimality of A^*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$
Time?? Exponential in [relative error in $h \times$ length of soln.]
Space?? Keeps all nodes in memory
Optimal??

Optimality of A^*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$
Time?? Exponential in [relative error in $h \times$ length of soln.]
Space?? Keeps all nodes in memory
Optimal?? Yes – cannot expand f_{i+1} until f_{i+1} is finished

Optimality of A^*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$
Time?? Exponential in [relative error in $h\times$ length of soln.]
Space?? Keeps all nodes in memory
Optimal?? Yes – cannot expand f_{i+1} until f_{i+1} is finished

Let C^* be the optimal path cost. Then

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

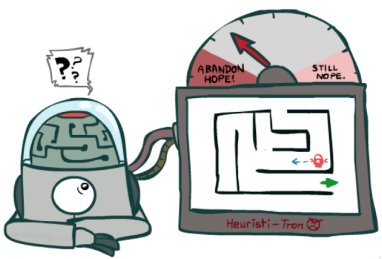
Choosing a good heuristic function

Consider these three questions:

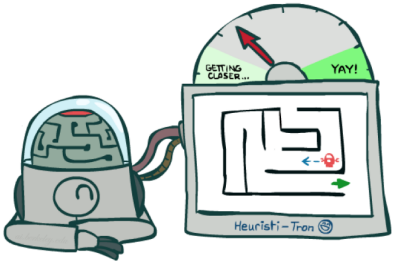
1. Which conditions h needs to satisfy to guarantee completeness and optimality of A^* under tree search?
2. Are these the same in the case of graph search?
3. If multiple h guarantee the optimality, how to choose?

Admissibility

Admissibility of h guarantees completeness and optimality of A^* under tree search



Pessimistic h may trap good plans



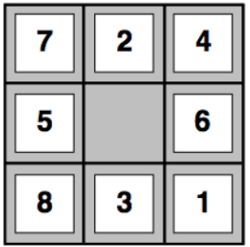
Admissible heuristics are optimistic!

Picture credit: D. Klein & P. Abbeel, Berkeley AI course CS188.

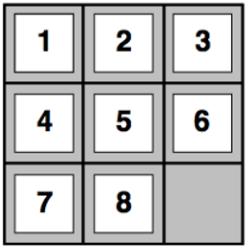
Admissible heuristics: Examples

E.g., for the 8-puzzle:

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



Start State

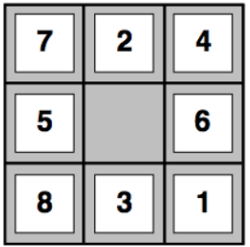


Goal State

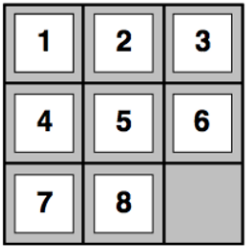
Admissible heuristics: Examples

E.g., for the 8-puzzle:

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



Start State



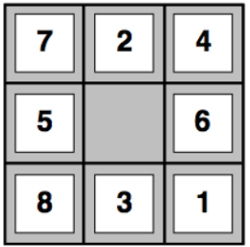
Goal State

$h_1(S)$??

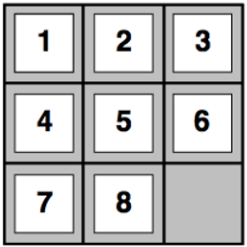
Admissible heuristics: Examples

E.g., for the 8-puzzle:

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



Start State



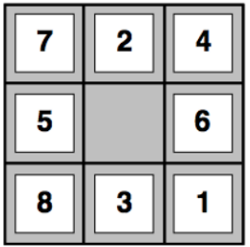
Goal State

$h_1(S)$?? 6

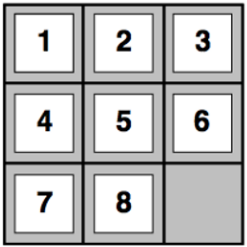
Admissible heuristics: Examples

E.g., for the 8-puzzle:

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



Start State



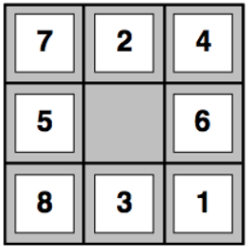
Goal State

$h_1(S)$?? 6
$h_2(S)$??

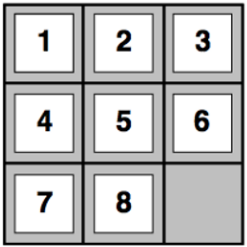
Admissible heuristics: Examples

E.g., for the 8-puzzle:

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



Start State

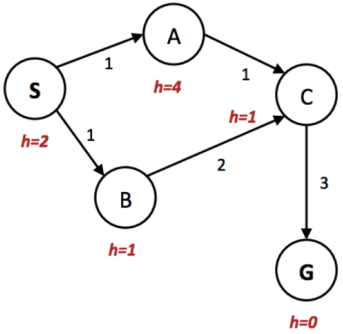


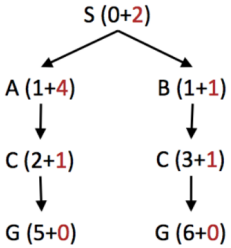
Goal State

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

Conditions under graph search

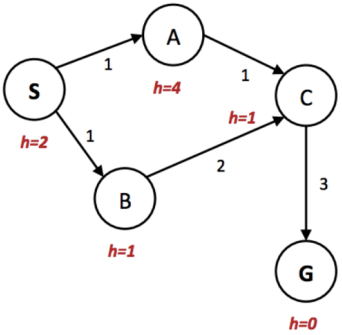
Consider the example below and apply A^* with graph search. Do you reach the goal?

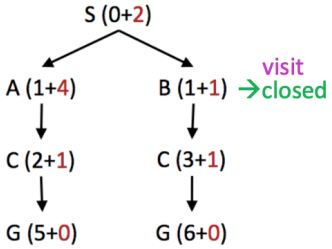




Conditions under graph search

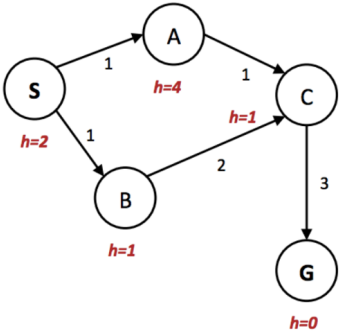
Consider the example below and apply A^* with graph search. Do you reach the goal?

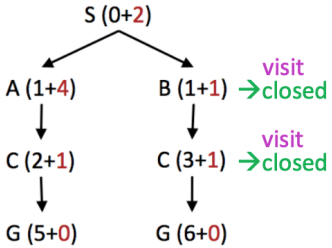




Conditions under graph search

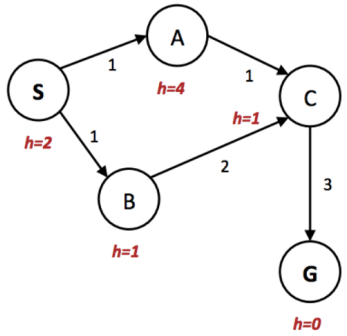
Consider the example below and apply A^* with graph search. Do you reach the goal?

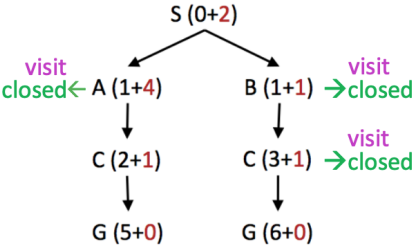




Conditions under graph search

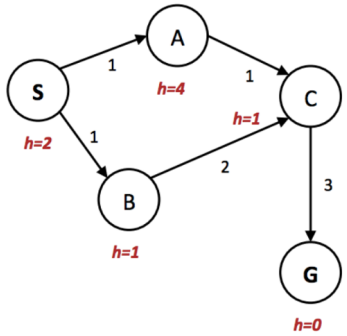
Consider the example below and apply A^* with graph search. Do you reach the goal?

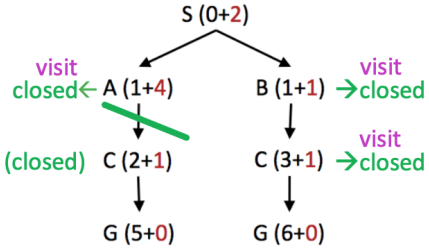




Conditions under graph search

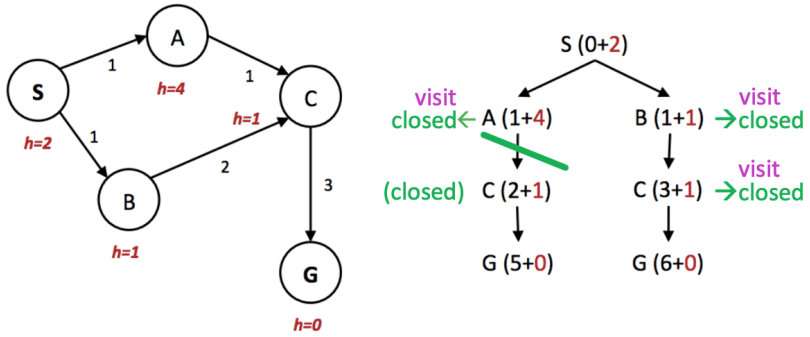
Consider the example below and apply A^* with graph search. Do you reach the goal?





Conditions under graph search

Consider the example below and apply A^* with graph search. Do you reach the goal?



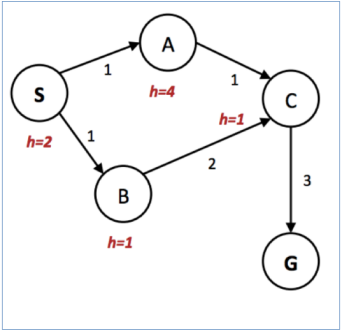
Although h was admissible, A^* didn't find the optimal solution!

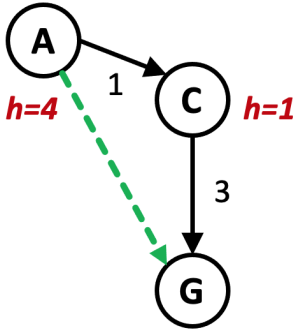
$\Rightarrow$ A stronger condition on h is required for A^* under graph search.

h needs to be **consistent**

Conditions under graph search

What was wrong here?





✓ $h(A) \leq$ cost (A to G)

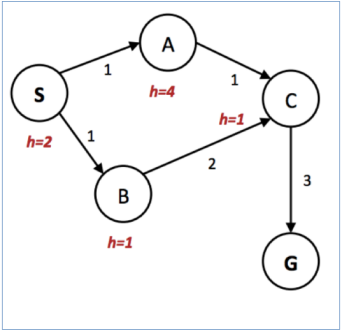
but,

✗ $h(A) - h(C) >$ cost(A to C)

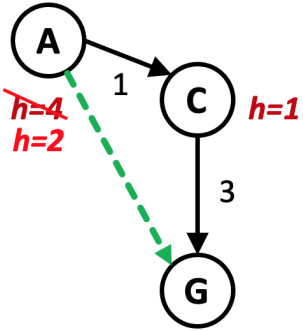
$f(n) = g(n) + h(n)$ appears smaller through C than from A

Conditions under graph search

What was wrong here?



How to fix it?



✓ $h(A) \leq$ cost (A to G)

but,

✗ $h(A) - h(C) >$ cost(A to C)

$f(n) = g(n) + h(n)$ appears smaller through C than from A

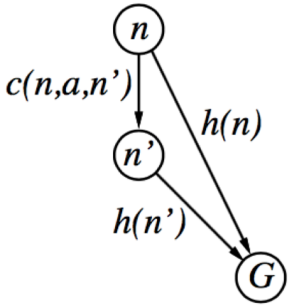
Consistent heuristics

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 non decreasing along any path.

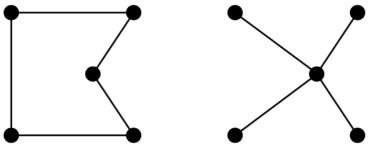
How to come up with admissible heuristics?

Derive admissible heuristics from the exact solution cost of a relaxed problem

Example: take 8-puzzle and previously defined $h_1(n)$ and $h_2(n)$
Consider two relaxed versions of the problem:

- If a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

Example: travelling salesperson problem (TSP)



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

Dominance

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

then h_2 **dominates** h_1 and is better for search

Typical search costs (8-puzzle):

$d = 14$ IDS $= 3{,}473{,}941$ nodes
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

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

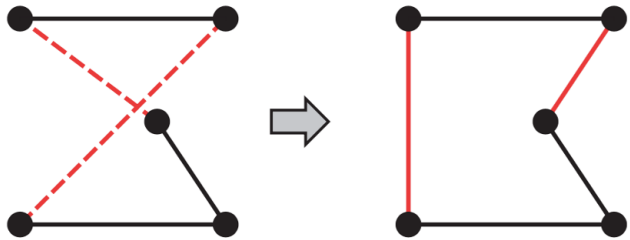
Outline

Iterative improvement algorithms

- In many optimization problems, **path** is irrelevant,
  the goal state itself is the solution
- Then the state space = set of "complete" configurations;
  find **optimal** configuration, e.g., TSP
  or, find configuration satisfying constraints, e.g., timetable
- in such cases, can use iterative improvement algorithms;
  keep a single "current" state, try to improve it

Example: Travelling Salesperson Problem

Start with any complete tour, perform pairwise exchanges

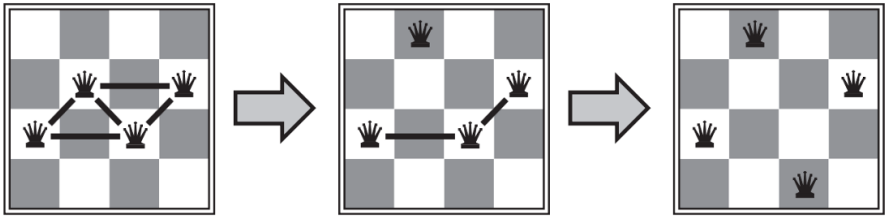


Variants of this approach get within 1% of optimal very quickly with thousands of cities

Example: n-queens

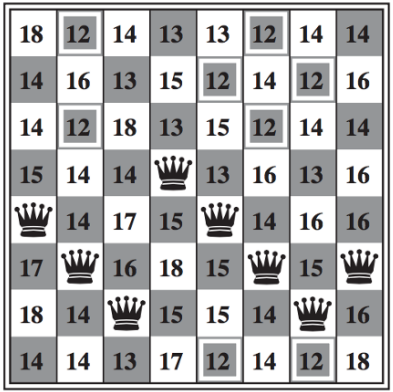
Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

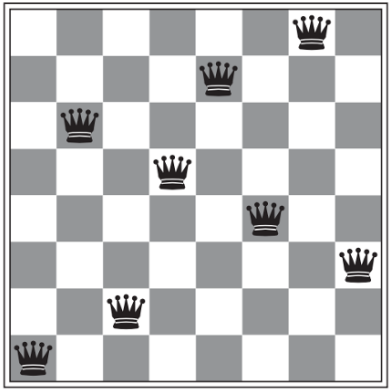
Move a queen to reduce number of conflicts



Almost always solves n-queens problems almost instantaneously for very large n, e.g., $n = 1million$

Example: n-queens





h: number of attacking pairs