Artificial Intelligence
Planning and Search

E.Giunchiglia, F. Leofante, A. Tacchella

Computer Engineering
Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi

Last updated: May 5, 2024

Agenda - Search

1. Why and What
2. Uninformed Search
   - Breadth-first search & Uniform-cost search
   - Depth-first search & Iterative-deepening search
   - Summary
3. Informed Search
   - Best-first Search
   - A^* search
   - Heuristics
4. Graph search
5. Planning as Search
   - Progressive Planning
   - Regressive Planning
   - Heuristics for state-space search

Agenda - Search

1. Why and What
2. Uninformed Search
   - Breadth-first search & Uniform-cost search
   - Depth-first search & Iterative-deepening search
   - Summary
3. Informed Search
   - Best-first Search
   - A^* search
   - Heuristics
4. Graph search
5. Planning as Search
   - Progressive Planning
   - Regressive Planning
   - Heuristics for state-space search

Why Search

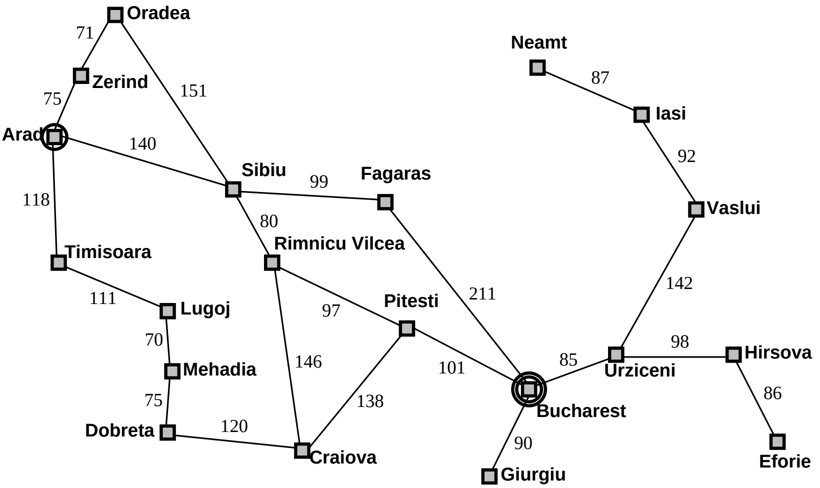
Several problems in AI (including planning) can be formulated as ***search in a graph***

A classical example:

- On holiday in Romania...
- ...currently I'm in Arad...
- ...but my flight leaves tomorrow from Bucharest!

- **Problem**: how can I get from Arad to Bucharest?
  - i.e., want to find a sequence of cities such as, e.g., Arad, Sibiu, Fagaras, Bucharest.

Example: holidays in Romania



Search problems

A search problem is defined by five items:

- **initial state**: e.g., *In(Arad)*
- A description of all possible **actions**: e.g., *Go(Zerind)*
- **successor function** $Res(a, S)$: returns the state that results from doing action a in state S
  - e.g., *Res(Go(Zerind), In(Arad)) = In(Zerind)*
- **goal**: e.g., *In(Bucharest)*
- **cost function** (additive) that assigns a cost to a solution,
  - e.g., sum of distances, number of actions executed, etc.

s . b s'

cost(s)+cost(a,s')

What is search

A solution for the problem we defined is a sequence of actions leading from the initial state to a goal state.

What is search

A solution for the problem we defined is a sequence of actions leading from the initial state to a goal state.

The process of looking for such sequence is called **search**

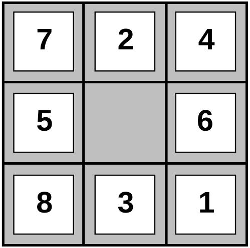
A note on the definition of the state space

The problem formulation we saw is reasonable, but still a **model**!

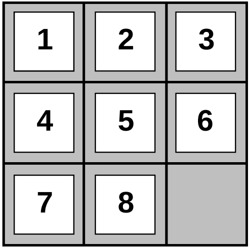
Real world is too complex, state space must be **abstracted**:

- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., “Arad $\rightarrow$ Zerind” represents a complex set of real actions
  - **each** 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!

The 8-puzzle

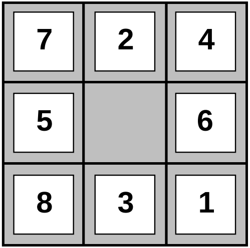


Start State

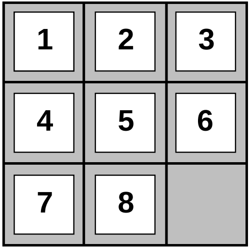


Goal State

The 8-puzzle



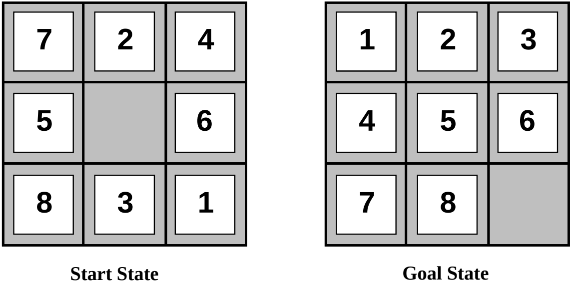
Start State



Goal State

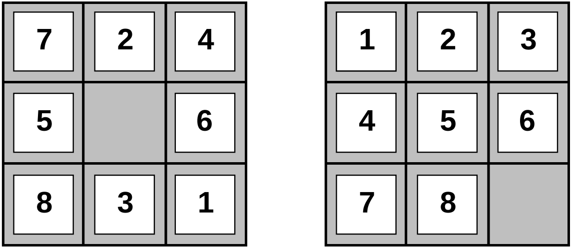
States?

The 8-puzzle



States? integer locations of tiles (ignore intermediate positions)
Actions?

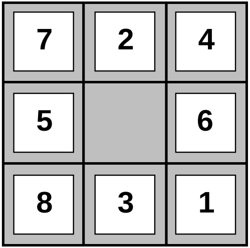
The 8-puzzle



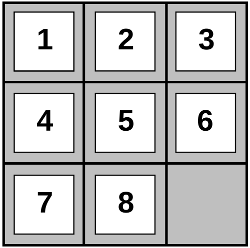
Start State Goal State

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

The 8-puzzle



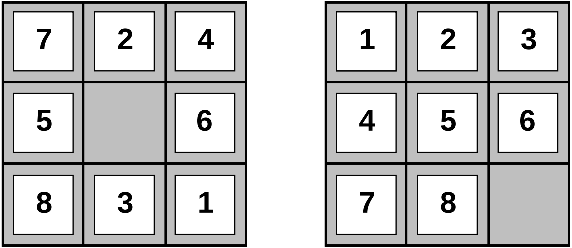
Start State



Goal State

States? integer locations of tiles (ignore intermediate positions)
Actions? move blank left, right, up, down (ignore unjamming etc.)
Goal? goal state given above
Cost?

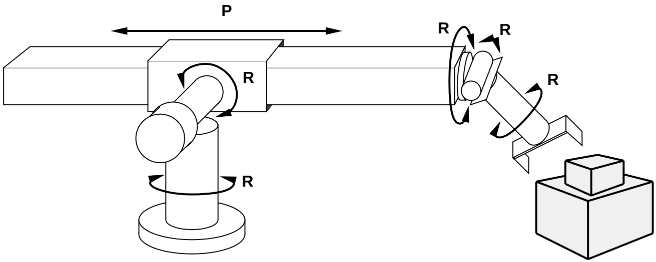
The 8-puzzle



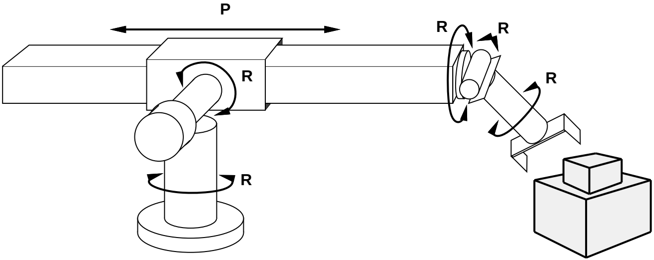
Start State **Goal State**

States? integer locations of tiles (ignore intermediate positions)
Actions? move blank left, right, up, down (ignore unjamming etc.)
Goal? goal state given above
Cost? 1 per move

A robotic assembly

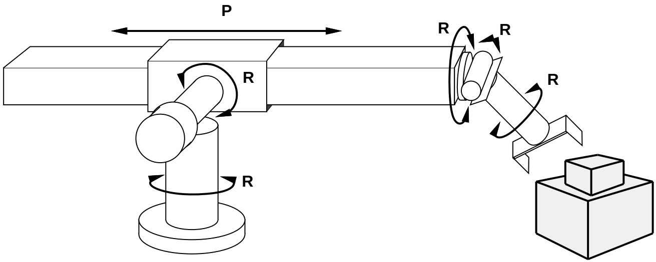


A robotic assembly



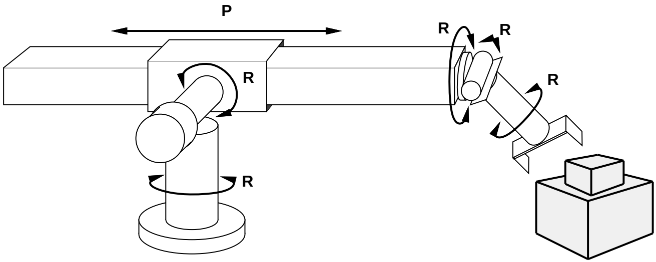
States?

A robotic assembly



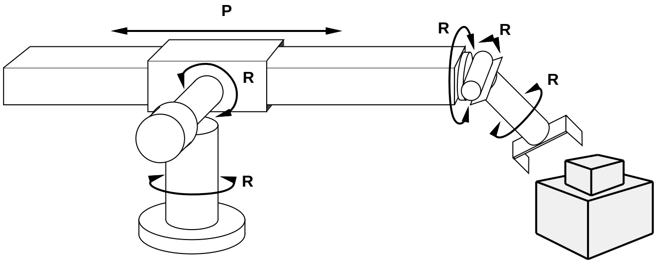
States? coordinates of robot joint, parts of the object to be assembled
Actions?

A robotic assembly



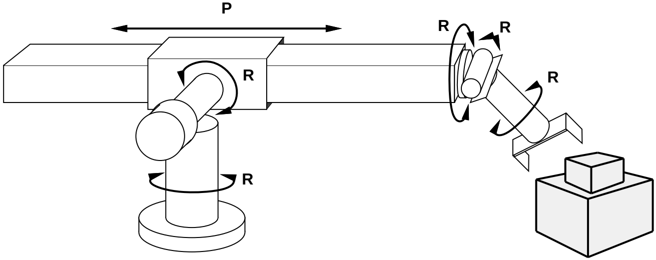
States? coordinates of robot joint, parts of the object to be assembled
Actions? continuous motions of robot joints
Goal?

A robotic assembly



States? coordinates of robot joint, parts of the object to be assembled
Actions? continuous motions of robot joints
Goal? complete assembly
Cost?

A robotic assembly



States? coordinates of robot joint, parts of the object to be assembled
Actions? continuous motions of robot joints
Goal? complete assembly
Cost? time to execute

Searching for solutions

Building a search tree

Possible action sequences from initial state form a **search tree** where:

- **nodes** correspond to states
- the initial state is the **root** node
- **branches** are actions

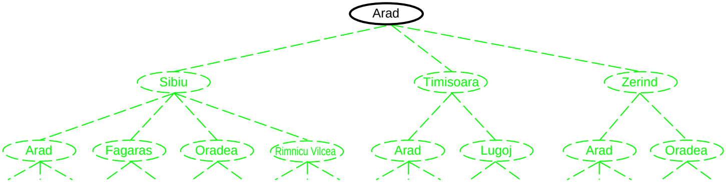
Searching for solutions

Building a search tree

Possible action sequences from initial state form a **search tree** where:

- **nodes** correspond to states
- the initial state is the **root** node
- **branches** are actions

Example:

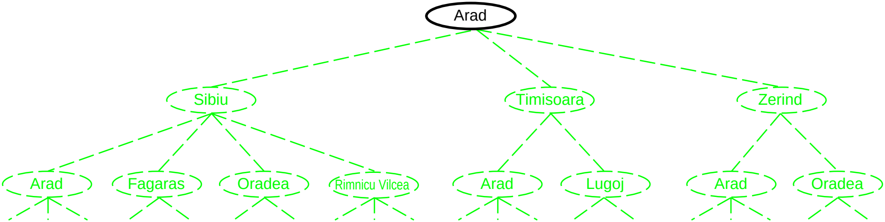


Tree search algorithms

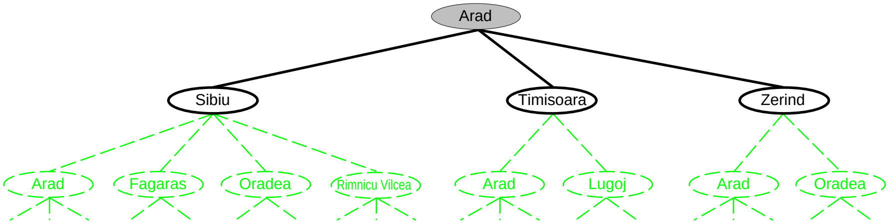
```
function TREE-SEARCH(problem, strategy) returns a solution or failure
   initialize the frontier using the initial state of problem
   loop do
       if the frontier is empty then return failure
       choose a leaf node according to strategy, remove it from frontier
       if the node is a goal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

Frontier: set of current leaf nodes available for expansion at any given point

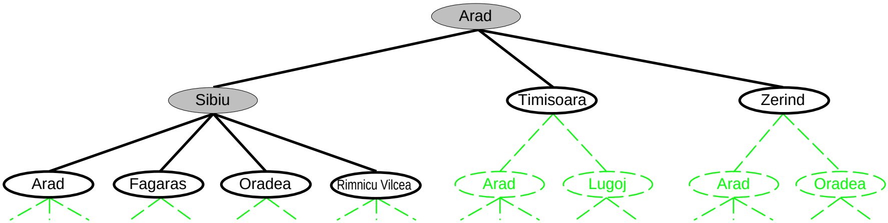
Tree search example



Tree search example



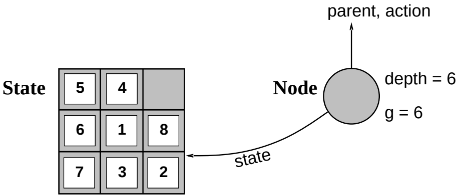
Tree search example



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**

States do not have parents, children, depth, or path cost!



.

Implementation: general tree search

```
function TREE-SEARCH( problem, frontier) returns a solution, or failure
   frontier ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), frontier)
   loop do
      if frontier is empty then return failure
      node ← REMOVE-FRONT(frontier)
      if GOAL-TEST(problem, STATE(node)) then return node
      frontier ← INSERTALL(EXPAND(node, problem), frontier)
```

```
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(STATE[node], action, result)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
   return successors
```

Search strategies

- A strategy is defined by picking the *order of node expansion*
- if we assume the frontier is implemented as a priority queue and that we pick the node to be visited/expanded from the top, then the strategy depends on how nodes are stored in the queue.

Terminology: A node is

- *generated* when it is inserted in the priority queue,
- *visited* when it is picked from the priority queue,
- *expanded* when its successors are generated.

Evaluation of search strategies

- 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 ∞)

Agenda - Search

Uninformed search strategies

Uninformed (or blind) strategies use only information about the node.

They can only generate successors and distinguish a goal state from a non-goal state[1].

[1] Strategies that use also information about the goal, i.e., to evaluate if a node is *more promising* than another are called informed or heuristic search strategies.

Uninformed search strategies

Uninformed (or blind) strategies use only information about the node.

They can only generate successors and distinguish a goal state from a non-goal state[1].

In the following we will see:

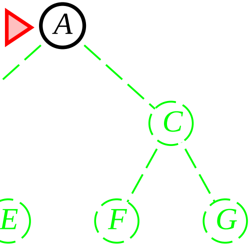
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

[1] Strategies that use also information about the goal, i.e., to evaluate if a node is *more promising* than another are called informed or heuristic search strategies.

Breadth-first

Idea: Expand shallowest unexpanded node

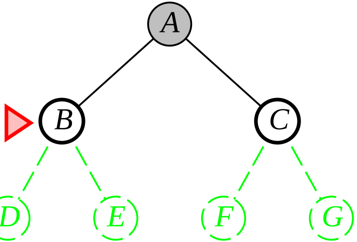
Implementation: *frontier* is a FIFO queue, i.e. new states go at the end



Breadth-first

Idea: Expand shallowest unexpanded node

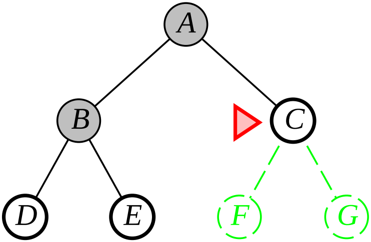
Implementation: *frontier* is a FIFO queue, i.e. new states go at the end



Breadth-first

Idea: Expand shallowest unexpanded node

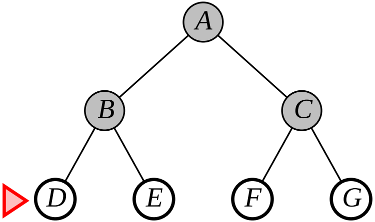
Implementation: *frontier* is a FIFO queue, i.e. new states go at the end



Breadth-first

Idea: Expand shallowest unexpanded node

Implementation: *frontier* is a FIFO queue, i.e. new states go at the 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? $1 + b + b^2 + b^3 + \ldots + b^d = b^{d+1} - 1 = O(b^{d+1})$, i.e., exponential in d
- Space?

Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? $1 + b + b^2 + b^3 + \ldots + b^d = b^{d+1} - 1 = O(b^{d+1})$, i.e., exponential in d
- Space? $O(b^d)$ i.e., exponential in d
- Optimal?

Properties of breadth-first search

good
- Complete? Yes (if b is finite)
- Time? $1 + b + b^2 + b^3 + \ldots + b^d = b^{d+1} - 1 = O(b^{d+1})$, i.e., exponential in d

bad
- Space? $O(b^d)$ i.e., exponential in d
- 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

Define: path cost of a node $g(n)$

Idea: Expand least-cost unexpanded node

Implementation: *frontier* = queue ordered by g, lowest first

Equivalent to breadth-first if step costs all equal

Uniform-cost search

Define: path cost of a node $g(n)$

Idea: Expand least-cost unexpanded node

Implementation: *frontier* = queue ordered by g, lowest first

Equivalent to breadth-first if step costs all equal

- Complete?

Uniform-cost search

Define: path cost of a node $g(n)$

Idea: Expand least-cost unexpanded node

Implementation: *frontier* = queue ordered by g, lowest first

Equivalent to breadth-first if step costs all equal

- Complete? Yes, if step cost is always $\geq \epsilon$ where $\epsilon \in \mathbb{R}^+$ is an arbitrary small positive constant.
- Time?

Uniform-cost search

Define: path cost of a node $g(n)$

Idea: Expand least-cost unexpanded node

Implementation: *frontier* = queue ordered by g, lowest first

Equivalent to breadth-first if step costs all equal

- Complete? Yes, if step cost is always $\geq \epsilon$ where $\epsilon \in \mathbb{R}^+$ is an arbitrary small positive constant.
- Time? # of nodes with $g \leq C^*$, where C^* is the cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
- Space?

Uniform-cost search

Define: path cost of a node $g(n)$

Idea: Expand least-cost unexpanded node

Implementation: *frontier* = queue ordered by g, lowest first

Equivalent to breadth-first if step costs all equal

- Complete? Yes, if step cost is always $\geq \epsilon$ where $\epsilon \in \mathbb{R}^+$ is an arbitrary small positive constant.
- Time? # of nodes with $g \leq C^*$, where C^* is the cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
- Space? # same as time
- Optimal?

Uniform-cost search

Define: path cost of a node $g(n)$

Idea: Expand least-cost unexpanded node

Implementation: *frontier* = queue ordered by g, lowest first

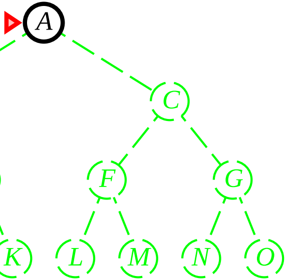
Equivalent to breadth-first if step costs all equal

- Complete? Yes, if step cost is always $\geq \epsilon$ where $\epsilon \in \mathbb{R}^+$ is an arbitrary small positive constant.
- Time? # of nodes with $g \leq C^*$, where C^* is the cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
- Space? # same as time
- Optimal? Yes–nodes expanded in increasing order of $g(n)$

Depth-first search

Idea: Expand deepest unexpanded node

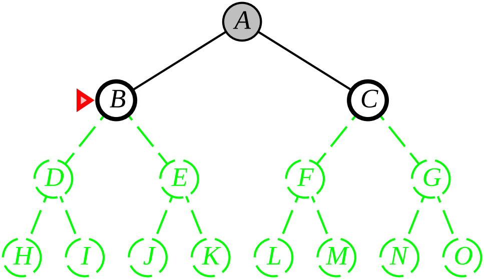
Implementation: *frontier* = LIFO queue, i.e., put successors at the front



Depth-first search

Idea: Expand deepest unexpanded node

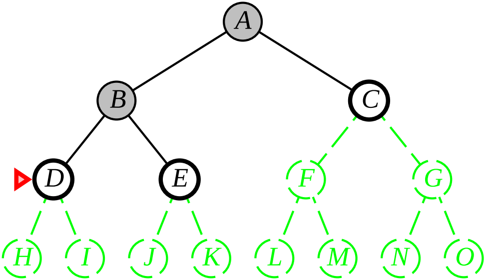
Implementation: *frontier* = LIFO queue, i.e., put successors at the front



Depth-first search

Idea: Expand deepest unexpanded node

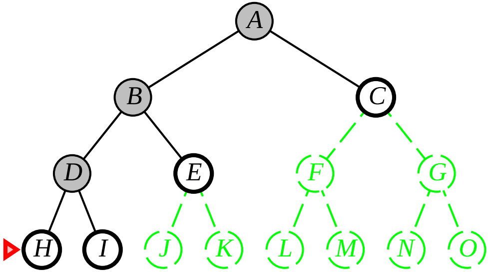
Implementation: *frontier* = LIFO queue, i.e., put successors at the front



Depth-first search

Idea: Expand deepest unexpanded node

Implementation: *frontier* = LIFO queue, i.e., put successors at the front

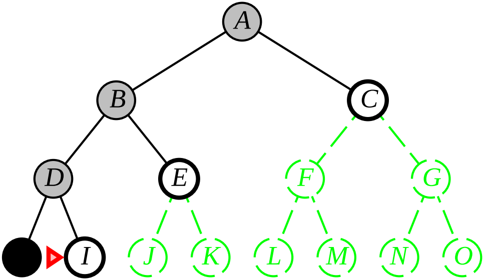


ordine:ABDHIEJKCFLMGNO

Depth-first search

Idea: Expand deepest unexpanded node

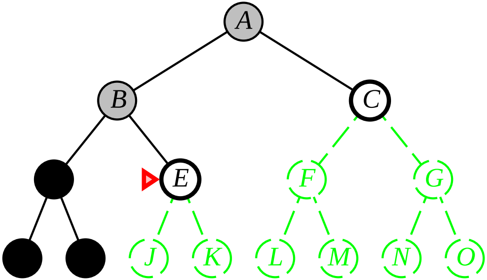
Implementation: *frontier* = LIFO queue, i.e., put successors at the front



Depth-first search

Idea: Expand deepest unexpanded node

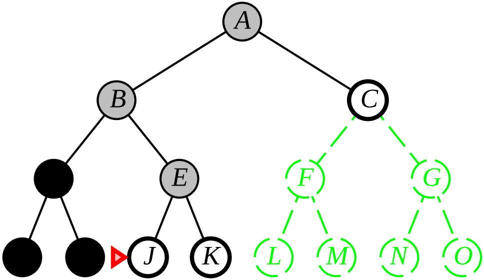
Implementation: *frontier* = LIFO queue, i.e., put successors at the front



Depth-first search

Idea: Expand deepest unexpanded node

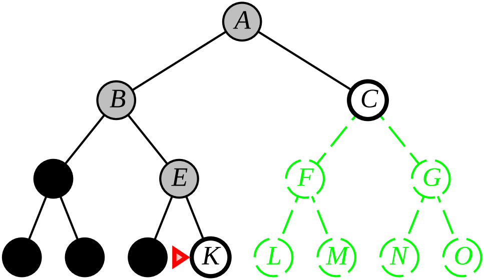
Implementation: *frontier* = LIFO queue, i.e., put successors at the front



Depth-first search

Idea: Expand deepest unexpanded node

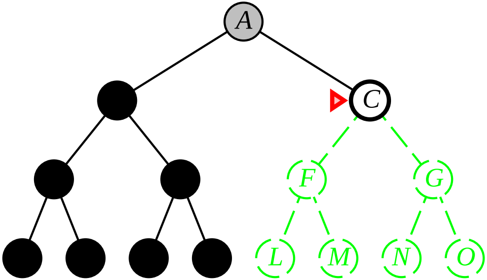
Implementation: *frontier* = LIFO queue, i.e., put successors at the front



Depth-first search

Idea: Expand deepest unexpanded node

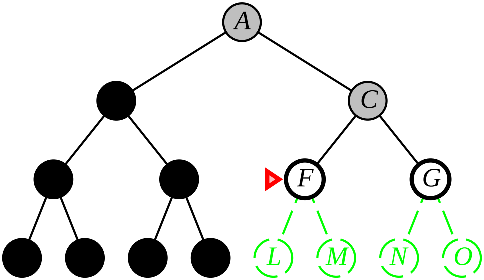
Implementation: *frontier* = LIFO queue, i.e., put successors at the front



Depth-first search

Idea: Expand deepest unexpanded node

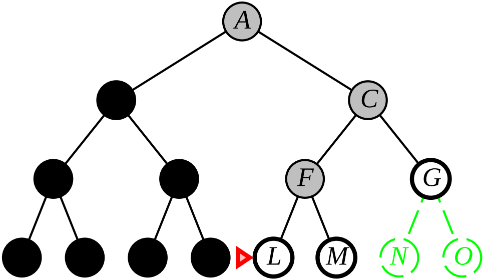
Implementation: *frontier* = LIFO queue, i.e., put successors at the front



Depth-first search

Idea: Expand deepest unexpanded node

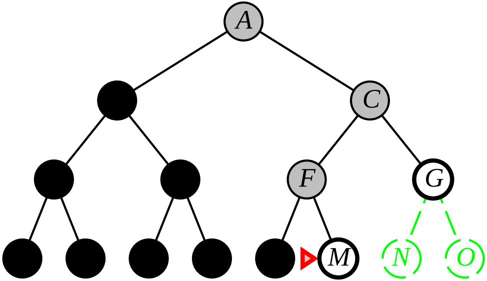
Implementation: *frontier* = LIFO queue, i.e., put successors at the front



Depth-first search

Idea: Expand deepest unexpanded node

Implementation: *frontier* = LIFO queue, i.e., put successors at the front



Properties of depth-first search

- Complete?

Properties of depth-first search

- Complete? No, it fails in infinite-depth spaces, spaces with loops...
- Time?

Properties of depth-first search

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

Modification of DFS to prevent failure in infinite state spaces.

DFS with depth limit l, i.e., nodes at depth l have no successors.

Depth-limited search

Modification of DFS to prevent failure in infinite state spaces.

DFS with depth limit l, i.e., nodes at depth l have no successors.

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

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
   cutoff-occurred? ← false
   if GOAL-TEST(problem, STATE[node]) then return 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
```

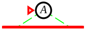
Iterative deepening search

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

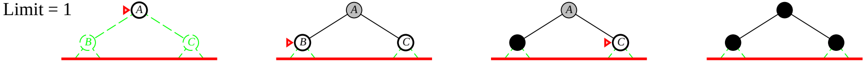
    for *depth* $\leftarrow$ 0 **to** ∞ **do**
        result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
        if *result* $\neq$ cutoff **then return** *result*
    end

Iterative deepening search $l = 0$

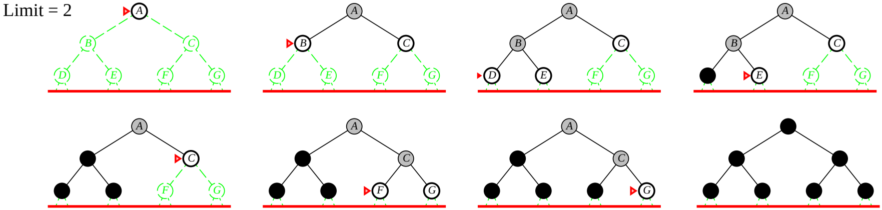
Limit = 0



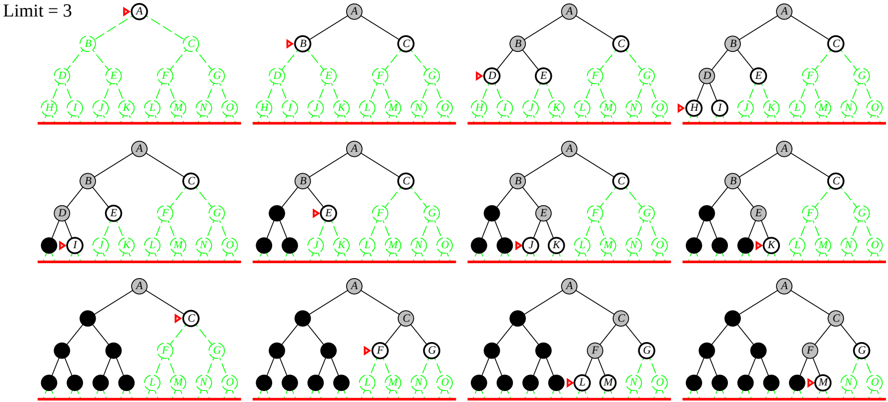
Iterative deepening search $l = 1$



Iterative deepening search $l = 2$



Iterative deepening search $l = 3$



Properties of iterative deepening search

- Complete?

Properties of iterative deepening search

- Complete? Yes
- Time?

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space?

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal?

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- 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:

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

$$\begin{aligned} N(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 \\ &= 1,111,100 \end{aligned}$$

- IDS does better because other nodes at depth d are not expanded

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	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Agenda - Search

Informed Search

We saw that **blind** search can be used to solve arbitrary search problems...

...why do we need **informed** search strategies then?

Informed Search

We saw that **blind** search can be used to solve arbitrary search problems...

...why do we need **informed** search strategies then?

Because we want to be more *efficient*!

Leveraging *problem-specific* knowledge, informed (a.k.a. **heuristic**) search can find solutions more efficiently than blind search.

Review: Tree search

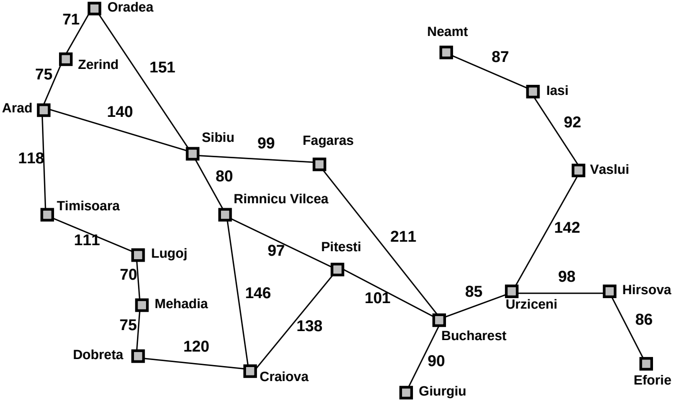
```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
       if fringe is empty then return failure
       node ← REMOVE-FRONT(fringe)
       if GOAL-TEST[problem] applied to STATE(node) succeeds
          then return node
       fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

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

Best-first search

- **Idea**:
  1. introduce an **evaluation function** $f(n)$ taking into account the estimated distance of node n from a goal state.
  2. Do tree search and expand node n with lowest $f(n)$ value
- **Implementation**: *frontier* is a queue sorted in increasing order of $f(n)$
- **Special cases**:
  - greedy search
  - A^* search

Back to Romania - with kilometers



Straight−line distance to Bucharest

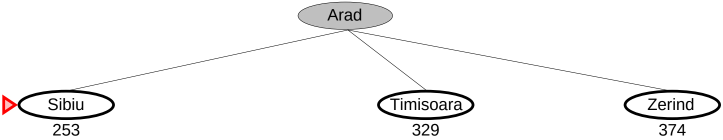
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search

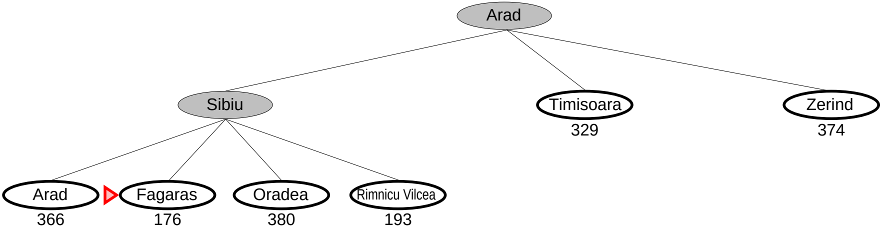
- Tries to expand the node that is **closest** to the goal
- Evaluation function $f(n) = h(n)$
  - $h(n)$ estimate of cost from node n to the closest goal
  - example: $h_{\mathrm{SLD}}(n)$ straight-line distance from n to Bucharest (prev. slide)

Greedy best-first search: example

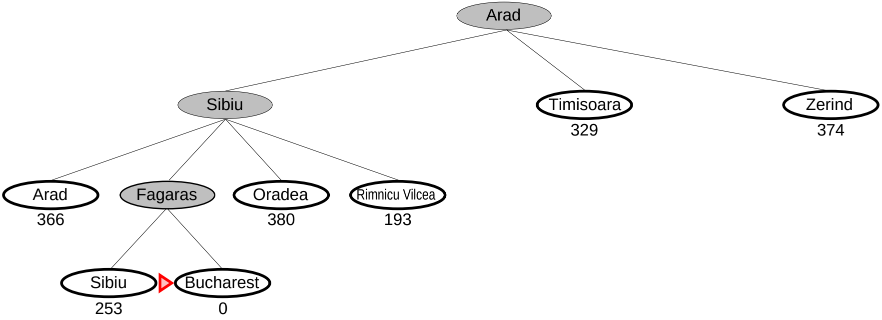
Greedy best-first search: example



Greedy best-first search: example



Greedy best-first search: example



Properties of greedy search

- Complete?

Properties of greedy search

- Complete? No, can get stuck in loops
  - e.g., with Oradea as goal, Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow \ldots$
- Time?

Properties of greedy search

- Complete? No, can get stuck in loops
  - e.g., with Oradea as goal, Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow \ldots$
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space?

Properties of greedy search

- Complete? No, can get stuck in loops
  - e.g., with Oradea as goal, Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow \ldots$
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$, keeps all nodes in memory
- Optimal?

Properties of greedy search

- Complete? No, can get stuck in loops
  - e.g., with Oradea as goal, Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow \dots$
- 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 estimated to be expensive

How? evaluation function $f(n) = g(n) + h(n)$, where:

- $g(n)$ = cost **so far** to reach n
- $h(n)$ = estimated **cost to** goal from n

Theorem

A^* is optimal if $h(n)$ is admissible

Admissible heuristic

An heuristic $h(n)$ is **admissible** if it never overestimates the cost to reach the goal, i.e.,

$$h(n) \leq h^*(n)$$

where $h^*(n)$ is the *true* cost to goal from n.

Example:

1. $h(n) = 0$ is admissible,
2. $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance.

Admissible heuristic

An heuristic $h(n)$ is **admissible** if it never overestimates the cost to reach the goal, i.e.,

$$h(n) \leq h^*(n)$$

where $h^*(n)$ is the *true* cost to goal from n.

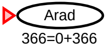
Example:

1. $h(n) = 0$ is admissible,
2. $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance.

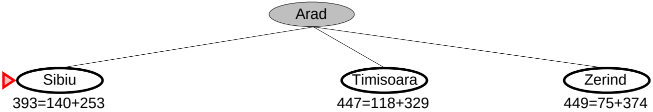
Questions: What can we conclude about A*, when, for each node n,

1. $h(n) = 0$? or
2. $h(n) = h^*(n)$?

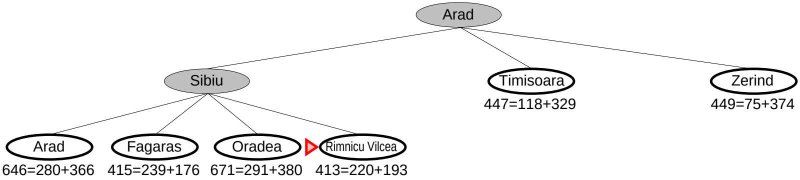
A* search example using $h_{\mathrm{SLD}}(n)$



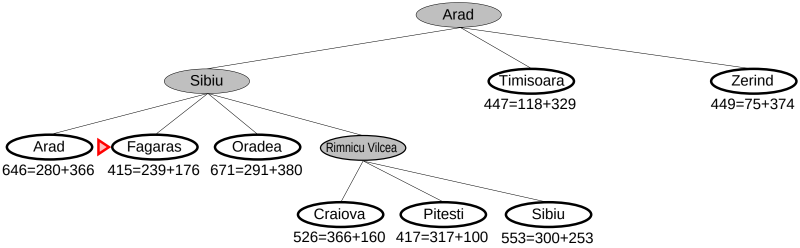
A^* search example using $h_{\mathrm{SLD}}(n)$



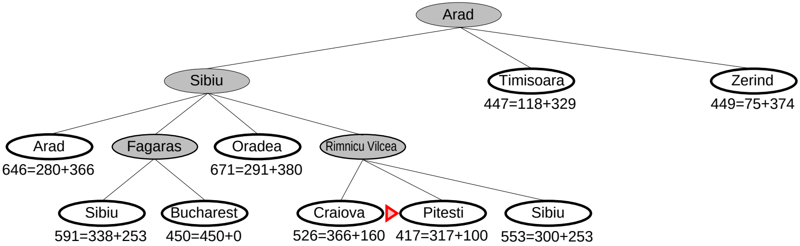
A^* search example using $h_{\mathrm{SLD}}(n)$



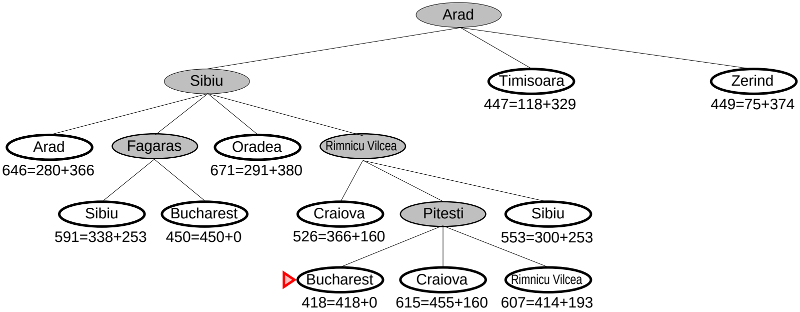
A* search example using $h_{\mathrm{SLD}}(n)$



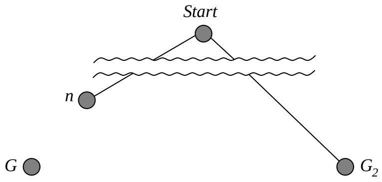
A* search example using $h_{\mathrm{SLD}}(n)$



A* search example using $h_{\mathrm{SLD}}(n)$



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 an optimal path to an optimal goal G.

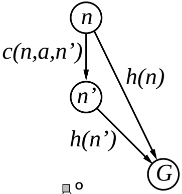
$$
\begin{aligned}
f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\
&> g(G) && \text{since } G_2 \text{ is suboptimal} \\
&= g(n) + h^*(n) && \text{since } n \text{ is on an optimal path} \\
&\geq g(n) + h(n) && \text{since } h \text{ is admissible} \\
&= f(n)
\end{aligned}
$$

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

Optimality of A* (more useful)

A heuristic is **consistent** iff

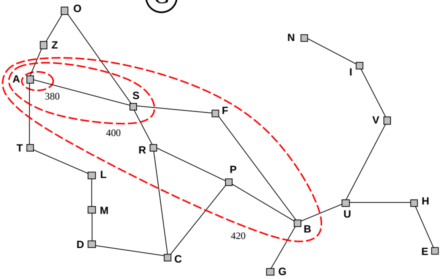
$$h(n) \leq c(n, a, n') + h(n')$$



(Most admissible heuristics are also consistent.)

Lemma

If h is consistent, then A expands nodes in order of increasing f value*.*

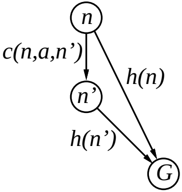


Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$

Proof of Lemma

Recall that h is consistent iff

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



If h is consistent, we have

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

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

Properties of A*

- Complete?

Properties of A*

- Complete? Yes, assuming all actions have a positive cost
- Time?

Properties of A*

- Complete? Yes, assuming all actions have a positive cost
- Time? Exponential in [relative error in h $\times$ length of soln.]
- Space?

Properties of A*

- Complete? Yes, assuming all actions have a positive cost
- Time? Exponential in [relative error in $h \times$ length of soln.]
- Space? Keeps all nodes in memory
- Optimal?

Properties of A*

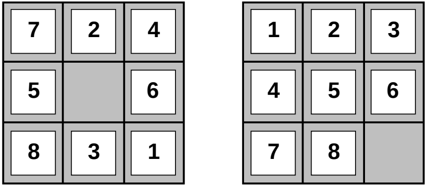
- Complete? Yes, assuming all actions have a positive cost
- Time? Exponential in [relative error in $h \times$ length of soln.]
- Space? Keeps all nodes in memory
- Optimal? Yes, cannot expand f_{i+1} until f_i is finished
  - A* expands all nodes with $f(n) < C^*$
  - A* may expand some nodes with $f(n) = C^*$
  - A* expands no nodes with $f(n) > C^*$

Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance (number of moves to put a tile in the right position assuming there are no other tiles).



Start State **Goal State**

$h_1(S) =?$ 6
$h_2(S) =?$ 2 + 0 + 3 + 1 + 0 + 1 + 3 + 4 = 15

Dominance

What is better for search, h_1 or h_2?

- 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:

$$\begin{array}{ll} d = 14 & \text{IDS = 3,473,941 nodes} \\ & \text{A}^*(h_1) \text{ = 539 nodes} \\ & \text{A}^*(h_2) \text{ = 113 nodes} \\ d = 24 & \text{IDS} \approx \text{54,000,000,000 nodes} \\ & \text{A}^*(h_1) \text{ = 39,135 nodes} \\ & \text{A}^*(h_2) \text{ = 1,641 nodes} \end{array}$$

- 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 *optimal* 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 optimal solution
- If the rules are relaxed so that a tile can move to *any adjacent square*, then $h_2(n)$ gives the optimal solution
- **Key point:** the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem
- **Important:** the optimal solution cost of the relaxed problem should be "easy" to compute.

Summary

- Heuristic functions estimate costs of optimal 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 optimal solution of relaxed problems

Agenda - Search

1. Why and What
2. Uninformed Search
   - Breadth-first search & Uniform-cost search
   - Depth-first search & Iterative-deepening search
   - Summary
3. Informed Search
   - Best-first Search
   - A^* search
   - Heuristics
4. **Graph search**
5. Planning as Search
   - Progressive Planning
   - Regressive Planning
   - Heuristics for state-space search

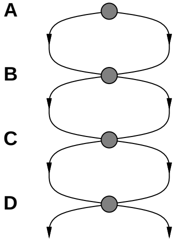
Repeated states

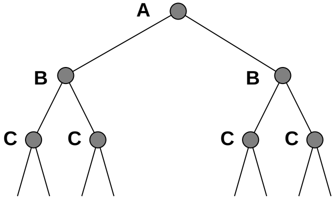
Algorithms that forget their history are doomed to repeat it

Repeated states

Algorithms that forget their history are doomed to repeat it

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





Graph search

Idea: augment TREE-SEARCH with **closed list** which remembers every expanded node.

```
function GRAPH-SEARCH( problem, frontier) returns a solution, or failure

   closed ← an empty set
   frontier ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), frontier)
   loop do
      if frontier is empty then return failure
      node ← REMOVE-FRONT(frontier)
      if GOAL-TEST(problem, STATE[node]) then return node
      if STATE[node] is not in closed then
         add STATE[node] to closed
         frontier ← INSERTALL(EXPAND(node, problem), frontier)
   end
```

Summary

- Graph search can be exponentially more efficient than tree search
- Graph search can use exponentially more space than tree search
- Graph search is guaranteed to terminate if state space is finite

Agenda - Search

Planning as Search

Apply standard search techniques to the state space induced by planning problem:

- each node in the graph denotes a state of the search and
- arcs connect states of the search that can be reached by executing a single action.

Two possible approaches:

- Progression through search states
- Regression through search states

ProgWS: a Progressive, World-State Planner

For applying search procedure, we need to be able to compute the successor states of a given state.
In progressive planning, given the set of actions $\mathcal{A}$:

1. A state S is represented by the set of fluents that are true in S.
2. Starting from the initial state (represented as a set of fluents), the search tree/graph is generated according to the selected search algorithm.
3. The set of successor states of the current state are the states (each represented by a set of fluents) that result from the execution of an applicable actions. More precisely, given an action $A = P_A, E_A, D_A$, the successor states of S are the states

$$\{(S \cup E_A) \setminus D_A : A \in \mathcal{A}, P_A \subseteq S\}.$$

Given the above, standard search algorithms can be applied.

REGWS: a Regressive, Search-State Planner

Definition (Goal Regression)

*The result of **regressing** a formula, say cur-goals, through an action Act is a logical sentence that encodes the weakest preconditions that must be true **before** Act is executed in order to assure that cur-goals will be true **after** Act is executed. In symbols:*

$$\mathit{preconditions}(\mathit{Act}) \cup (\mathit{cur\text{-}goals} \setminus \mathit{goals\text{-}added\text{-}by}(\mathit{Act}))$$

Note: For the regression of *cur-goals* through *Act* to be defined it is necessary that the effects of *Act* do not conflict with *cur-goals*.

REGWS: a Regressive, Search-State Planner (Cont)

In regressive planning, given the set of actions $\mathcal{A}$:

1. A state S is represented by the set of fluents that are true in S.
2. Starting from the goal state (represented as a set of fluents), the search tree/graph is generated according to the selected search algorithm.
3. The set of successor states of the current state are the states (each represented by a set of fluents) that could lead to that state by applying an action. More precisely, given an action $A = P_A, E_A, D_A$, the successor states of S are the states

$$\{(S \setminus E_A) \cup P_A : A \in \mathcal{A}, E_A \cap S \neq \{\,\}, D_A \cap S = \{\,\}\}.$$

Given the above, standard search algorithms can be applied.

Analysis of PROGWS and REGWS

- Both PROGWS and REGWS are complete or not depending on the strategy used to visit nodes.
- The complexity of any deterministic implementation is in $O(b^d)$, where b is the branching factor (i.e. the max number of choices to be considered at each nondeterministic branch point) and d is the depth of the optimal plan.
- Under the (plausible) assumption that goals involve only a small fraction of the atoms used to describe the initial states, then regression planning is likely to have (initially, at least) a much smaller branching factor.

Heuristics for state-space search

- We recall that an effective way to design an admissible heuristics for a search problem is to consider a *relaxed* version of the problem for which the cost of the optimal solution can be "easily" computed.
- In planning the availability of an explicit representation for actions simplifies the task.
- Here we focus on two heuristics:
  - **Empty-delete-list heuristics** and
  - **Empty-delete-list heuristics with empty preconditions**.

  These heuristics are admissible if cost is unitary.
- Non admissible heuristics may be taken into account if optimality is not important.

Empty-delete-list heuristics

- **Idea:** Remove all negated literals from the effects of all the actions we get a simplified planning problem.
- The heuristics is quite accurate, but computing it involves actually running a (simple) planning problem.
- In practice, the search in the relaxed problem is often fast enough that the cost is worthwhile.

Example: Let us consider the planning problem with:

- Current State: $At(Home) \wedge Sells(SM, Milk)$
- Goal: $At(Home) \wedge Have(Milk)$
- Operators:

Empty-delete-list heuristics: Example continued

- The operators of the relaxed problem are:

- According to the heuristics the estimated optimal cost for the current state is 2 (corresponding to the plan [Go(Home,SM),Buy(Milk)]) whereas the actual optimal cost is 3 (corresponding to the plan [Go(Home,SM),Buy(Milk),Go(SM,Home)]).

Empty-delete-list heuristics with empty preconditions

- **Idea:** Remove all preconditions and all the negated literals form the effects of all the actions.
- All actions become always applicable.
- Count the minimum number of actions required such that the union of the current state with the actions' (positive) effects satisfies the goal.
- This counting amounts to solving a minimal set cover problem which is NP-hard
- The set cover problem can be quickly solved in practice by greedy (and hence suboptimal) algorithms. However, in this way the admissibility of the heuristics (and hence the optimality of the plan length) is compromised.

Empty-delete-list heuristics with empty preconditions: Example

- With reference to the previous example the operators of the relaxed problem are:

- According to the heuristics the estimated optimal cost for the current state is 1 (corresponding to the set of actions {Buy(Milk)}) whereas the actual optimal cost is 3 (corresponding to the plan [Go(Home,SM),Buy(Milk),Go(SM,Home)]).