Artificial Intelligence

Search

Marc Toussaint
University of Stuttgart
Winter 2015/16

(slides based on Stuart Russell's AI course)

Outline

- Problem formulation & examples
- Basic search algorithms

Example: Romania

On holiday in Romania; currently in Arad.

Flight leaves tomorrow from Bucharest

Formulate goal:

- be in Bucharest, $\mathcal{S}_{\text{goal}} = \{\text{Bucharest}\}$

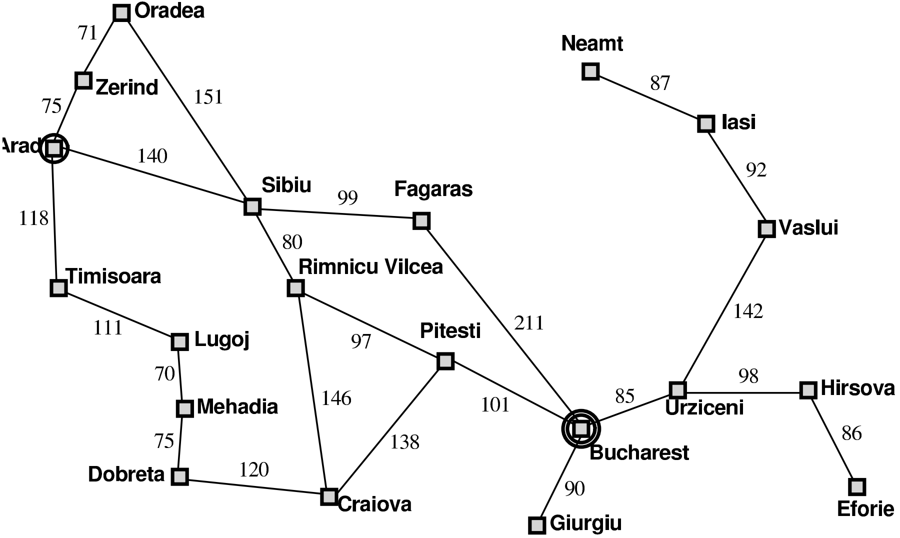
Formulate problem:

- states: various cities, $\mathcal{S} = \{\text{Arad, Timisoara}, \ldots\}$
- actions: drive between cities, $\mathcal{A} = \{\text{edges between states}\}$

Find solution:

- sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
- minimize costs with cost function, $(s, a) \mapsto c$

Example: Romania



Problem types

Deterministic, fully observable *("single-state problem")*

Agent knows exactly which state it will be in; solution is a sequence

First state and world known $\rightarrow$ the agent does not rely on observations

Non-observable *("conformant problem")*

Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable *("contingency problem")*

percepts provide *new* information about current state

solution is a reactive plan or a policy

often *interleave* search, execution

Unknown state space *("exploration problem")*

Deterministic, fully observable problem def.

A deterministic, fully observable problem is defined by four items:

initial state $s_0 \in \mathcal{S}$ e.g., $s_0 =$ Arad

successor function $succ : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$

e.g., $succ($Arad,Arad-Zerind$) =$ Zerind

goal states $\mathcal{S}_{\text{goal}} \subseteq \mathcal{S}$

e.g., $s =$ Bucharest

step cost function $cost(s, a, s')$, assumed to be ≥ 0

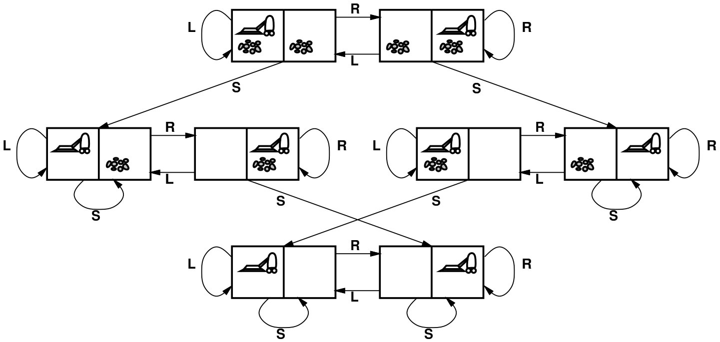
e.g., traveled distance, number of actions executed, etc.

the path cost is the sum of step costs

A solution is a sequence of actions leading from s_0 to a goal

An optimal solution is a solution with minimal path costs

Example: vacuum world



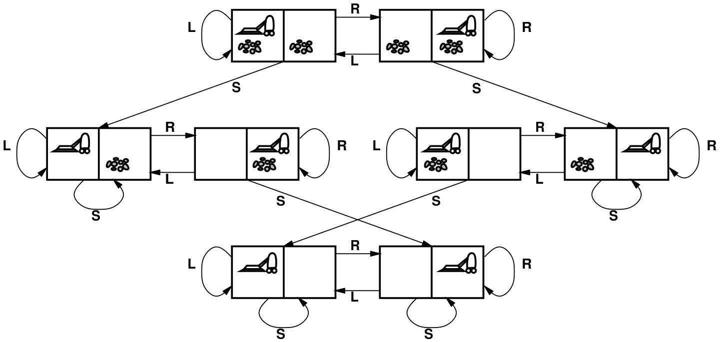
states??

actions??

goal test??

path cost??

Example: vacuum world state space graph



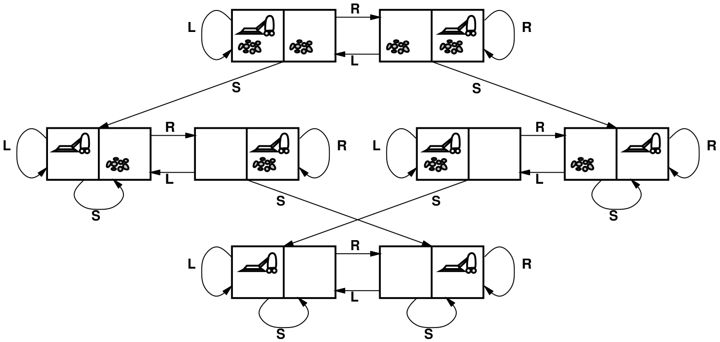
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??

goal test??

path cost??

Example: vacuum world state space graph



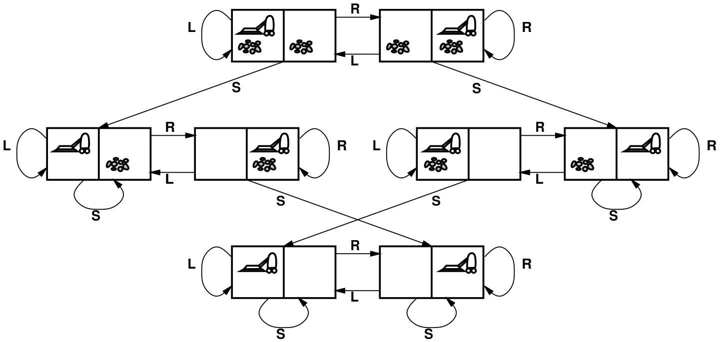
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: $Left$, $Right$, $Suck$, $NoOp$

goal test??

path cost??

Example: vacuum world state space graph



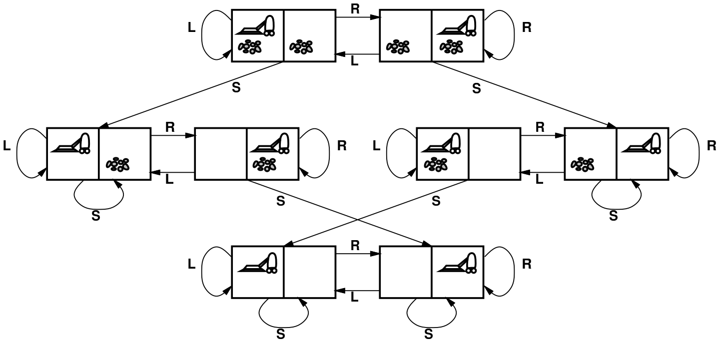
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: $Left$, $Right$, $Suck$, $NoOp$

goal test??: no dirt

path cost??

Example: vacuum world state space graph



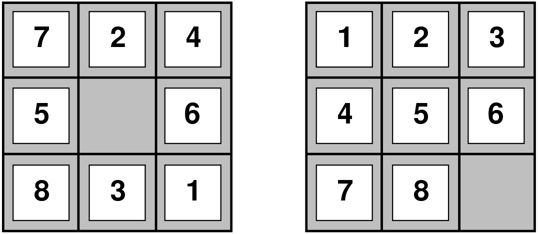
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: $Left$, $Right$, $Suck$, $NoOp$

goal test??: no dirt

path cost??: 1 per action (0 for $NoOp$)

Example: The 8-puzzle



Start State　　　Goal State

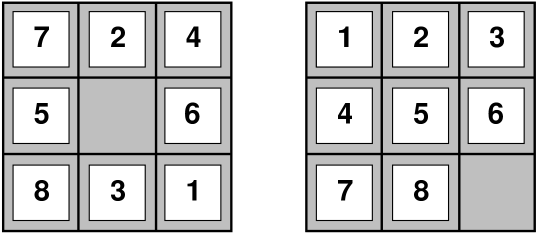
states??

actions??

goal test??

path cost??

Example: The 8-puzzle



Start State   Goal State

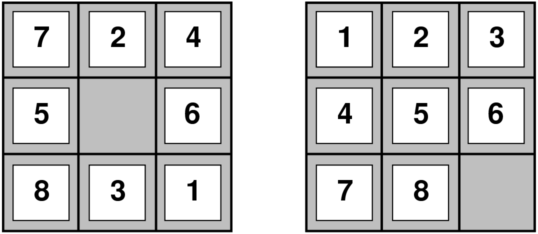
states??: integer locations of tiles (ignore intermediate positions)

actions??

goal test??

path cost??

Example: The 8-puzzle



Start State **Goal State**

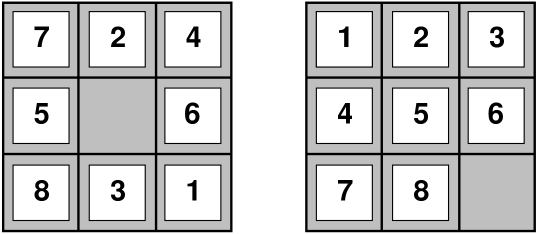
states??: integer locations of tiles (ignore intermediate positions)

actions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??

path cost??

Example: The 8-puzzle



Start State **Goal State**

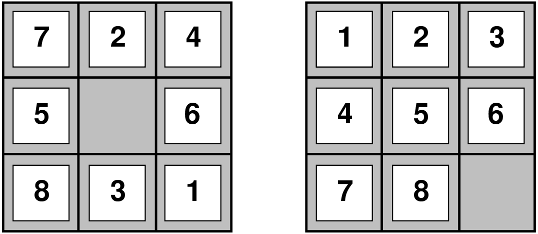
states??: integer locations of tiles (ignore intermediate positions)

actions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??: = goal state (given)

path cost??

Example: The 8-puzzle



Start State Goal State

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

actions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??: = goal state (given)

path cost??: 1 per move

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

Tree Search Algorithms

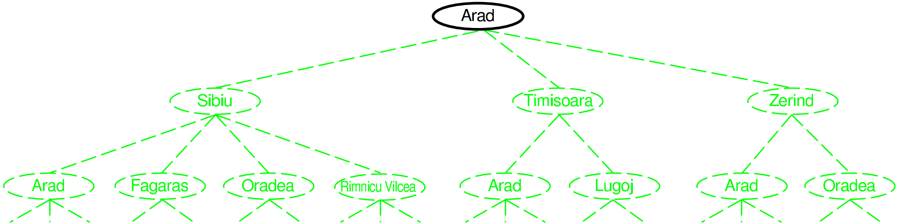
Tree search algorithms

Basic idea:

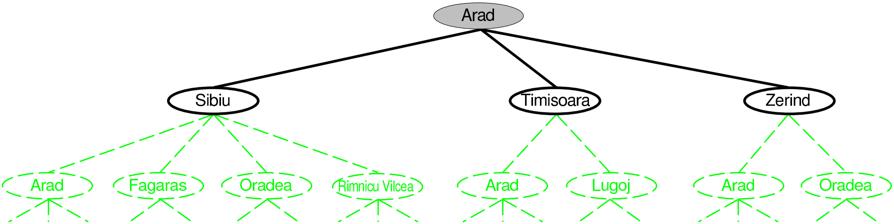
offline, simulated exploration of state space
by generating successors of already-explored states

(a.k.a. expanding states)

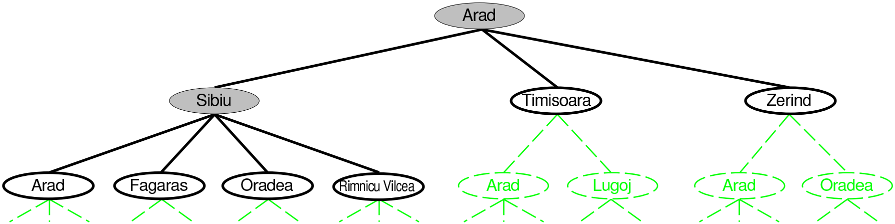
Tree search example



Tree search example



Tree search example



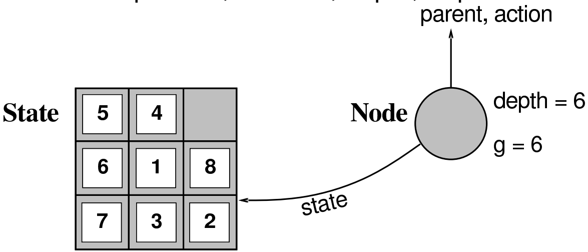
Implementation: states vs. nodes

A state is a (representation of) a physical configuration

A node is a data structure constituting part of a search tree

  includes parent, children, depth, path cost $g(x)$

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



The $\textsc{Expand}$ function creates new nodes, filling in the various fields and using the $\textsc{SuccessorFn}$ of the problem to create the corresponding states.

Implementation: general tree search

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

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 ∞)

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

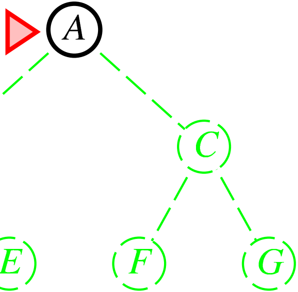
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

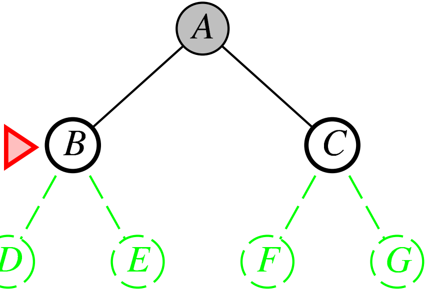


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

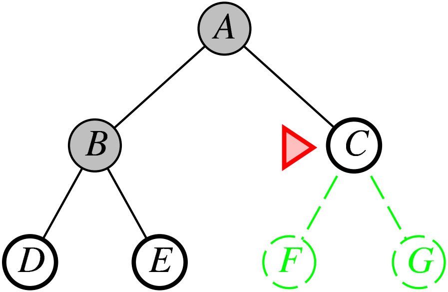


Breadth-first search

Expand shallowest unexpanded node

Implementation:

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

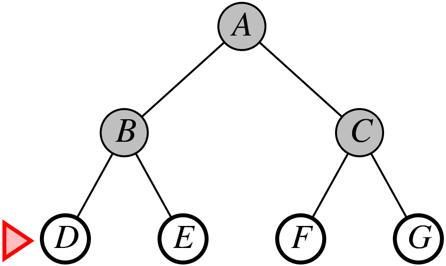


Breadth-first search

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)

Time??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

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

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

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

Optimal?? Yes, if cost-per-step=1; not optimal otherwise

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

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

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

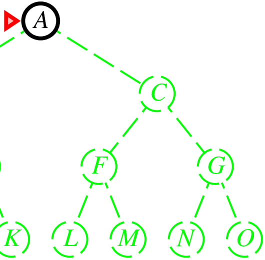
Optimal?? Yes: nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:

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

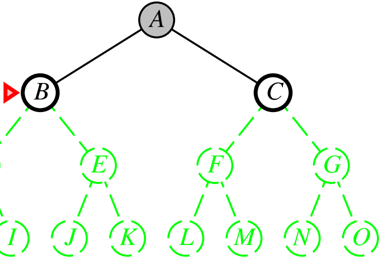


Depth-first search

Expand deepest unexpanded node

Implementation:

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

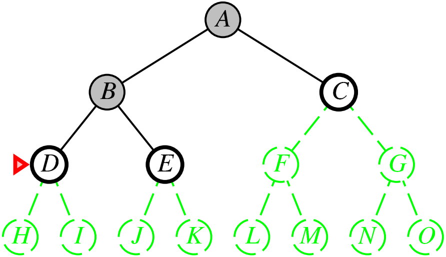


Depth-first search

Expand deepest unexpanded node

Implementation:

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

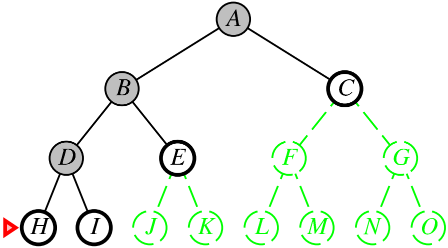


Depth-first search

Expand deepest unexpanded node

Implementation:

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

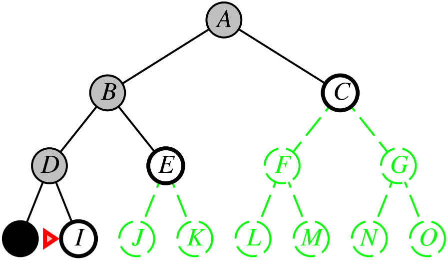


Depth-first search

Expand deepest unexpanded node

Implementation:

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

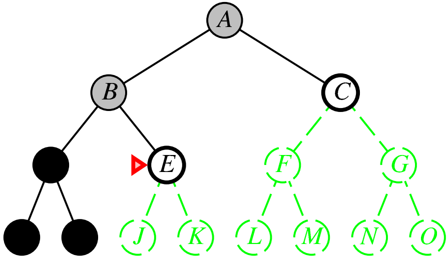


Depth-first search

Expand deepest unexpanded node

Implementation:

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

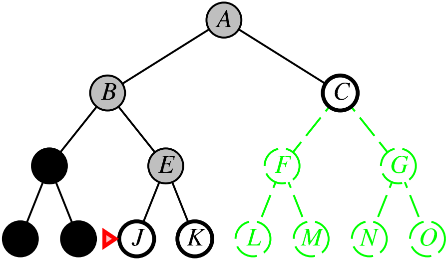


Depth-first search

Expand deepest unexpanded node

Implementation:

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

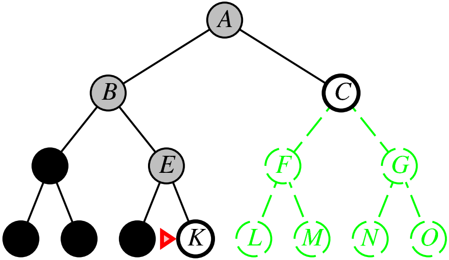


Depth-first search

Expand deepest unexpanded node

Implementation:

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

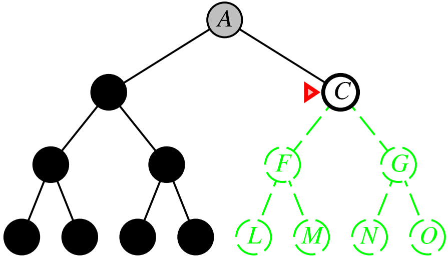


Depth-first search

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

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

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

Depth-limited search

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

Recursive implementation using the stack as LIFO:

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

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

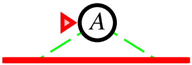
Iterative deepening search

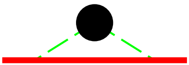
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

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

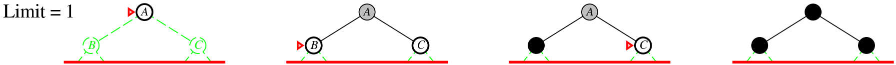
Iterative deepening search $l = 0$

Limit = 0



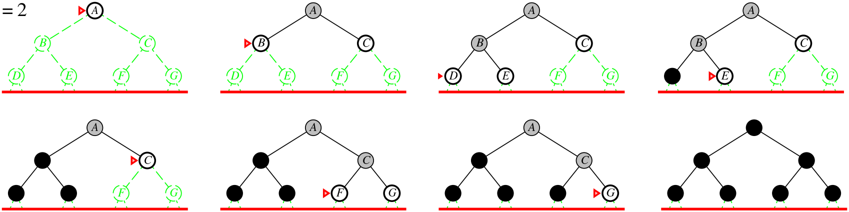


Iterative deepening search $l = 1$

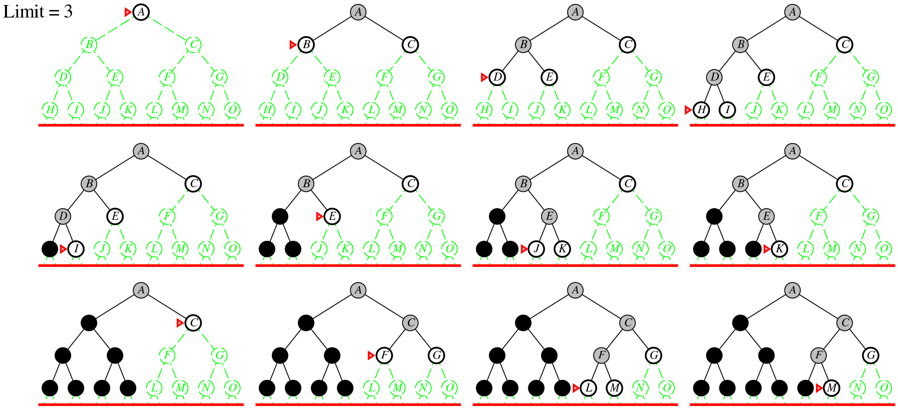


Iterative deepening search $l = 2$

Limit = 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 left leaf:

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

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

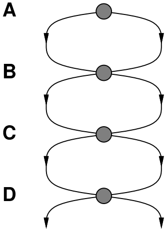
BFS can be modified to apply goal test when a node is *generated*

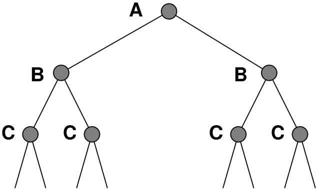
Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	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*

Loops: Repeated states

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





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
```

But: storing all visited nodes leads again to exponential space complexity (as for BFS)

Summary

In BFS (or uniform-cost search), the fringe propagates layer-wise, containing nodes of similar distance-from-start (cost-so-far), leading to optimal paths but exponential space complexity $O(B^{d+1})$

In DFS, the fringe is like a deep light beam sweeping over the tree, with space complexity $O(bm)$. Iteratively deepening it also leads to optimal paths.

Graph search can be exponentially more efficient than tree search, but storing the visited nodes may lead to exponential space complexity as BFS.

Greedy and A*Search

Best-first search

Idea: use an arbitrary priority function $f(n)$ for each node

– actually $f(n)$ is neg-priority: nodes with lower $f(n)$ have higher priority

$f(n)$ should reflect which nodes *could* be on an optimal path

– *could* is optimistic – the lower $f(n)$ the more optimistic you are that n is on an optimal path

$\Rightarrow$ Expand the unexpanded node with highes priority

Implementation:

fringe is a queue sorted in decreasing order of priority

Special cases:

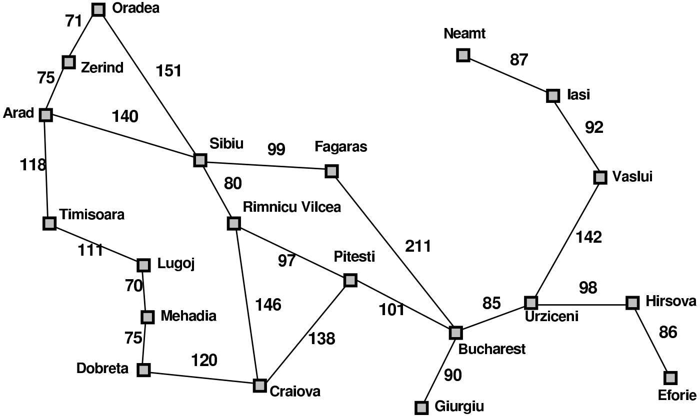
greedy search

A* search

Uniform-Cost Search as special case

- Define $g(n)$ = cost-so-far to reach n
- Then Uniform-Cost Search is Prioritized Search with $f = g$

Romania with step costs in km



Straight−line distance to Bucharest

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

Greedy search

We set the priority function equal to a heuristic $f(n) = h(n)$

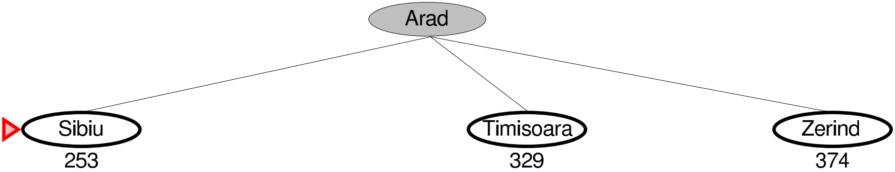
$h(n)$ = estimate of cost from n to the closest goal

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

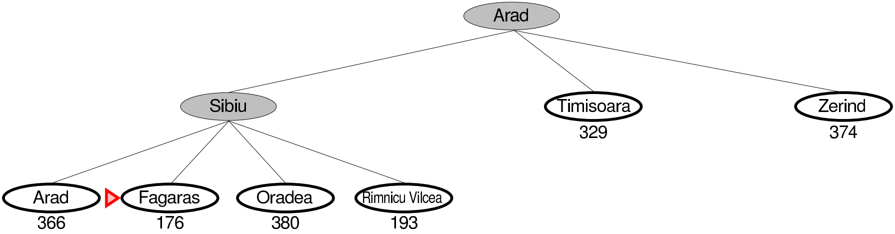
Greedy search expands the node that *appears* to be closest to goal

Greedy search example

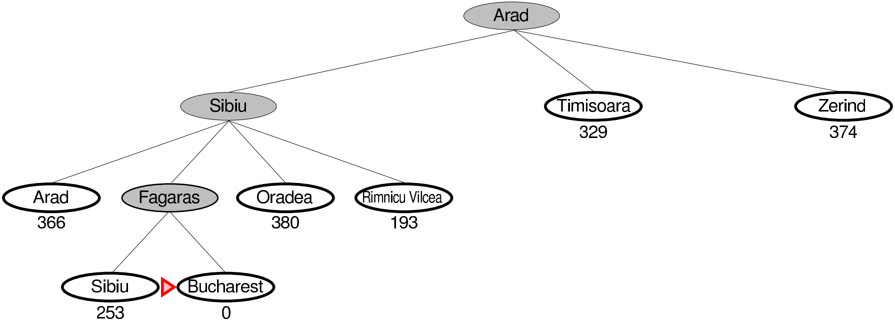
Greedy search example



Greedy search example



Greedy search example



Properties of greedy search

Complete??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g., with Oradea as goal,

Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Complete in finite space with repeated-state checking

Time??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,

Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Complete in finite space with repeated-state checking

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.,

Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Complete in finite space 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??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,

Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Complete in finite space 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

Greedy search does not care about the 'past' (the cost-so-far).

A* search

Idea: *combine information from the past and the future*

neg-priority = cost-so-far + estimated cost-to-go

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

$g(n)$ = cost-so-far to reach n

$h(n)$ = estimated cost-to-go from n

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

A* search uses an admissible (=optimistic) heuristic

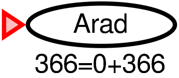
i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost-to-go from n.

(Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)

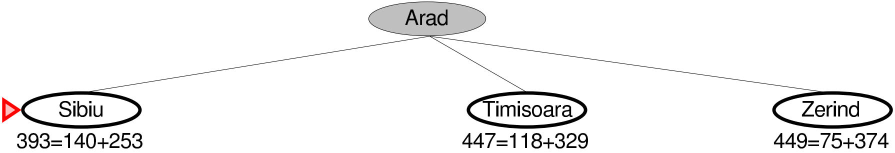
E.g., $h_{\mathrm{SLD}}(n)$ never overestimates the actual road distance

Theorem: A* search is optimal (=finds the optimal path)

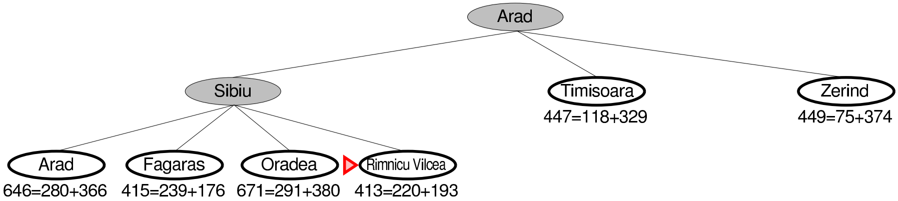
A* search example



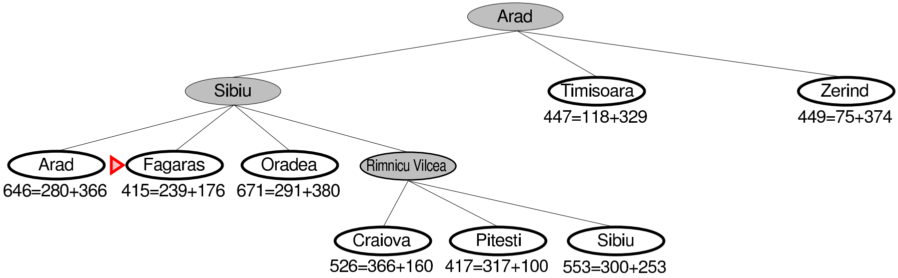
A* search example



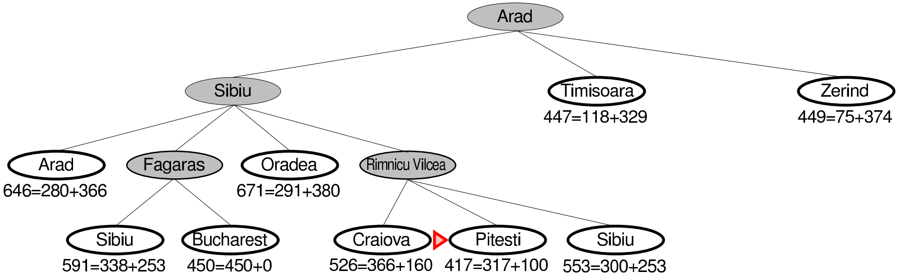
A* search example



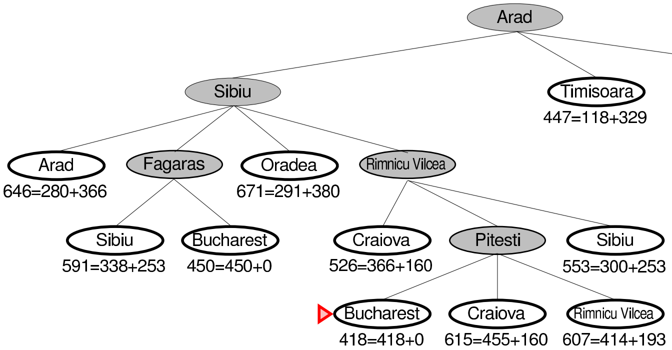
A* search example



A* search example



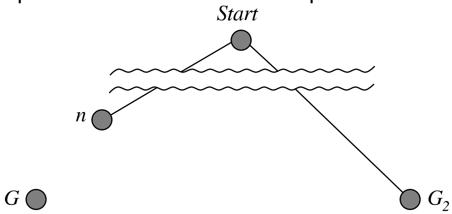
A* search example



Proof of optimality of A*

Suppose some suboptimal goal G_2 has been generated and is in the fringe (but has not yet been selected to be tested for goal condition!). We want to proof:
Any node on a shortest path to an optimal goal G will be expanded before G_2.

Let n be an unexpanded node on a shortest path to G.



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

Since $f(n) < f(G_2)$, A* will expand n *before* G_2. This is true for any n on the shortest path. In particular, at some time G is added to the fringe, and since $f(G) = g(G) < f(G_2) = g(G_2)$ it will select G *before* G_2 for goal testing.

Properties of A*

Complete??

Properties of A*

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

Time??

Properties of A*

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

Properties of A*

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

Time?? Exponential in [relative error in h $\times$ length of soln.]

Space?? Exponential. Keeps all nodes in memory

Optimal??

Properties of A*

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

Time?? Exponential in [relative error in $h \times$ length of soln.]

Space?? Exponential. Keeps all nodes in memory

Optimal?? Yes

A* expands all nodes with $f(n) < C^*$

A* expands some nodes with $f(n) = C^*$

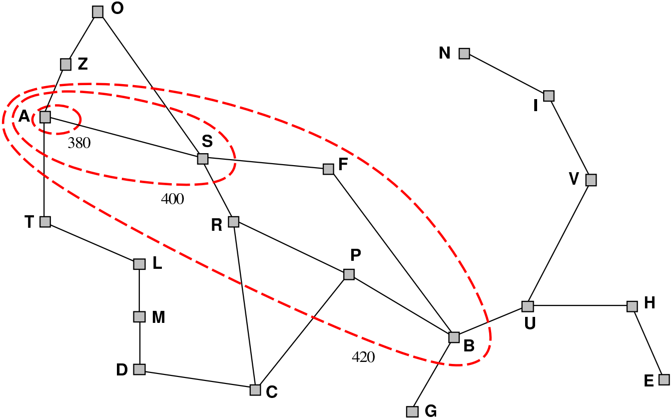
A* expands no nodes with $f(n) > C^*$

Optimality of A* (more useful)

Lemma: A* expands nodes in order of increasing f value*

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)

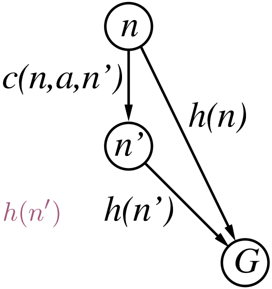
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Proof of lemma: Consistency

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 nondecreasing along any path.

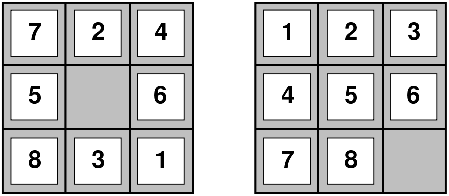
Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



Start State Goal State

$\underline{h_1(S) =}$??

$\underline{h_2(S) =}$??

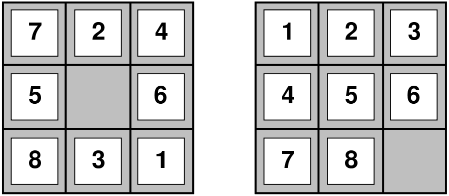
Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



Start State Goal State

$h_1(S)$ =?? 6

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

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 dominates h_1 and is better for search
Typical search costs:

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

Relaxed problems

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

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

If the rules are relaxed so that a tile can move to *any adjacent square*, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

Memory-bounded A*

As with BFS, A* has exponential space complexity

Iterative-deepening A*, works for integer path costs, but problematic for real-valued

(Simplified) Memory-bounded A* (SMA*):

– Expand as usual until a memory bound is reach

– Then, whenever adding a node, remove the *worst* node n' from the tree

– worst means: the n' with highest $f(n')$

– To not loose information, *backup* the measured step-cost $cost(\tilde{n}, a, n')$ to improve the heuristic $h(\tilde{n})$ of its parent

SMA* is complete and optimal if the depth of the optimal path is within the memory bound

Summary

Combine information from the past and the future

A heuristic function $h(n)$ represents information about the future

  – it estimates cost-to-go optimistically

Good heuristics can dramatically reduce search cost

Greedy best-first search expands lowest h

– incomplete and not always optimal

A* search expands lowest $f = g + h$

– neg-priority = cost-so-far + estimated cost-to-go

– complete and optimal

– also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems

Memory-bounded startegies exist

Outlook

Tree search with *partial observations*

– rather discuss this in a fully probabilistic setting later

Tree search for *games*

– minimax extension to tree search

– discuss state-of-the-art probabilistic Monte-Carlo tree search methods later