AI Planning for Autonomy

2. Search Algorithms

Basic Stuff You're Gonna Need to Search for a Solution
Where To Search Next?

Chris Ewin & Tim Miller

THE UNIVERSITY OF MELBOURNE

With slides by Nir Lipovetsky

Basic State Model: Classical Planning

Ambition:

Write one program that can solve all classical search problems.

State Model $\mathcal{S}(P)$:

- finite and discrete state space S
- a known initial state $s_0 \in S$
- a set $S_G \subseteq S$ of goal states
- actions $A(s) \subseteq A$ applicable in each $s \in S$
- a deterministic transition function $s' = f(a, s)$ for $a \in A(s)$
- positive action costs $c(a, s)$

$\rightarrow$ A **solution** is a sequence of applicable actions that maps s_0 into S_G, and it is **optimal** if it minimizes **sum of action costs** (e.g., # of steps)

$\rightarrow$ Different **models** and **controllers** obtained by relaxing assumptions in **blue** . . .

Example

Criteria for Evaluating Search Strategies

Guarantees:

Completeness: Is the strategy guaranteed to find a solution when there is one?

Optimality: Are the returned solutions guaranteed to be optimal?

Criteria for Evaluating Search Strategies

Guarantees:

Completeness: Is the strategy guaranteed to find a solution when there is one?

Optimality: Are the returned solutions guaranteed to be optimal?

Complexity:

Time Complexity: How long does it take to find a solution? (Measured in generated states.)

Space Complexity: How much memory does the search require? (Measured in states.)

Criteria for Evaluating Search Strategies

Guarantees:

Completeness: Is the strategy guaranteed to find a solution when there is one?

Optimality: Are the returned solutions guaranteed to be optimal?

Complexity:

Time Complexity: How long does it take to find a solution? (Measured in generated states.)

Space Complexity: How much memory does the search require? (Measured in states.)

Typical state space features governing complexity:

Branching factor b: How many successors does each state have?

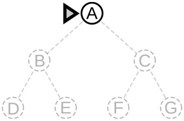
Goal depth d: The number of actions required to reach the shallowest goal state.

Agenda

Breadth-First Search: Illustration and Guarantees

Strategy: Expand nodes in the order they were produced (FIFO frontier).

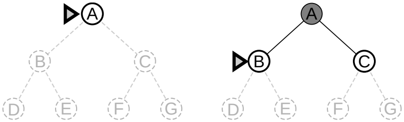
Illustration:



Breadth-First Search: Illustration and Guarantees

Strategy: Expand nodes in the order they were produced (FIFO frontier).

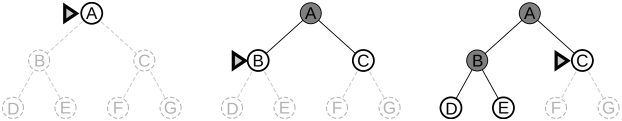
Illustration:



Breadth-First Search: Illustration and Guarantees

Strategy: Expand nodes in the order they were produced (FIFO frontier).

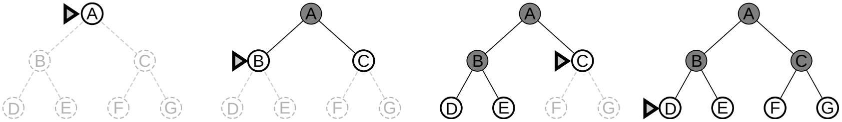
Illustration:



Breadth-First Search: Illustration and Guarantees

Strategy: Expand nodes in the order they were produced (FIFO frontier).

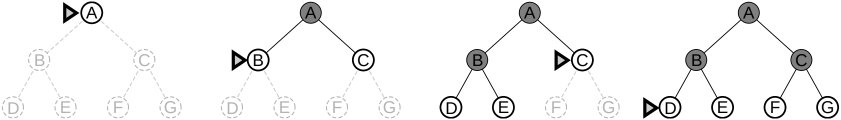
Illustration:



Breadth-First Search: Illustration and Guarantees

Strategy: Expand nodes in the order they were produced (FIFO frontier).

Illustration:



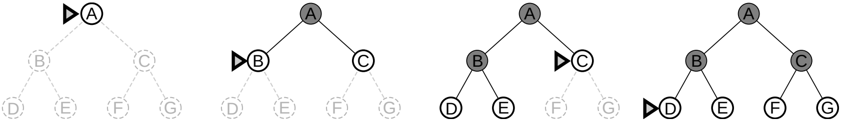
Guarantees:

- Completeness?

Breadth-First Search: Illustration and Guarantees

Strategy: Expand nodes in the order they were produced (FIFO frontier).

Illustration:



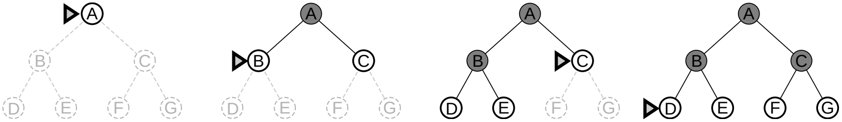
Guarantees:

- Completeness? Yes.
- Optimality?

Breadth-First Search: Illustration and Guarantees

Strategy: Expand nodes in the order they were produced (FIFO frontier).

Illustration:



Guarantees:

- Completeness? Yes.
- Optimality? Yes, for uniform action costs. Breadth-first search always finds a shallowest goal state. If costs are not uniform, this is not necessarily optimal.

Breadth-First Search: Complexity

Time Complexity: Say that b is the maximal branching factor, and d is the goal depth (depth of shallowest goal state).

- Upper bound on the number of generated nodes?

Breadth-First Search: Complexity

Time Complexity: Say that b is the maximal branching factor, and d is the goal depth (depth of shallowest goal state).

- Upper bound on the number of generated nodes? $b + b^2 + b^3 + \cdots + b^d$: In the worst case, the algorithm generates all nodes in the first d layers.
- So the time complexity is $O(b^d)$.
- And if we were to apply the goal test at node-expansion time, rather than node-generation time?

Breadth-First Search: Complexity

Time Complexity: Say that b is the maximal branching factor, and d is the goal depth (depth of shallowest goal state).

- Upper bound on the number of generated nodes? $b + b^2 + b^3 + \cdots + b^d$: In the worst case, the algorithm generates all nodes in the first d layers.
- So the time complexity is $O(b^d)$.
- And if we were to apply the goal test at node-expansion time, rather than node-generation time? $O(b^{d+1})$ because then we'd generate the first $d + 1$ layers in the worst case.

Space Complexity:

Breadth-First Search: Complexity

Time Complexity: Say that b is the maximal branching factor, and d is the goal depth (depth of shallowest goal state).

- Upper bound on the number of generated nodes? $b + b^2 + b^3 + \cdots + b^d$: In the worst case, the algorithm generates all nodes in the first d layers.
- So the time complexity is $O(b^d)$.
- And if we were to apply the goal test at node-expansion time, rather than node-generation time? $O(b^{d+1})$ because then we'd generate the first $d + 1$ layers in the worst case.

Space Complexity: Same as time complexity since all generated nodes are kept in memory.

Breadth-First Search: Example Data

Setting: $b = 10$; 10000 nodes/second; 1000 bytes/node.

Yields data: (inserting values into previous equations)

Depth	Nodes	Time		Memory	
2	110	.11	milliseconds	107	kilobytes
4	11110	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

$\rightarrow$ So, which is the worse problem, time or memory?

Breadth-First Search: Example Data

Setting: $b = 10$; 10000 nodes/second; 1000 bytes/node.

Yields data: (inserting values into previous equations)

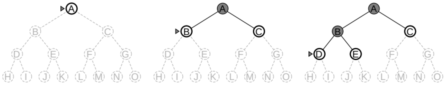
Depth	Nodes	Time		Memory	
2	110	.11	milliseconds	107	kilobytes
4	11110	11	milliseconds	10.6	megabytes
6	10^6	1.1	seconds	1	gigabyte
8	10^8	2	minutes	103	gigabytes
10	10^{10}	3	hours	10	terabytes
12	10^{12}	13	days	1	petabyte
14	10^{14}	3.5	years	99	petabytes

$\rightarrow$ So, which is the worse problem, time or memory? Memory. (In my own experience, typically exhausts RAM memory within a few minutes.)

Depth-First Search: Illustration

Strategy: Expand the most recent nodes in (LIFO frontier).

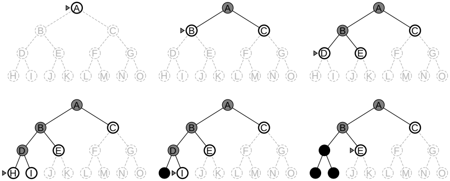
Illustration: (Nodes at depth 3 are assumed to have no successors)



Depth-First Search: Illustration

Strategy: Expand the most recent nodes in (LIFO frontier).

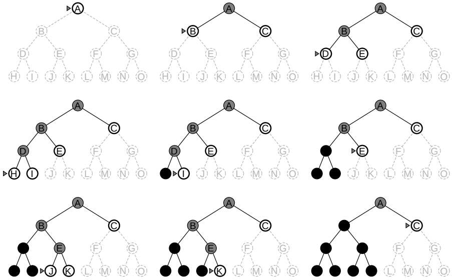
Illustration: (Nodes at depth 3 are assumed to have no successors)



Depth-First Search: Illustration

Strategy: Expand the most recent nodes in (LIFO frontier).

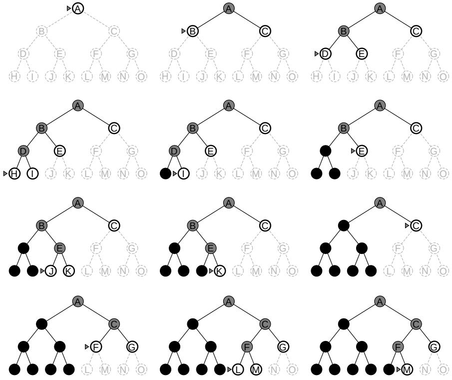
Illustration: (Nodes at depth 3 are assumed to have no successors)



Depth-First Search: Illustration

Strategy: Expand the most recent nodes in (LIFO frontier).

Illustration: (Nodes at depth 3 are assumed to have no successors)



Depth-First Search: Guarantees and Complexity

Guarantees:
Question: A) Complete and optimal B) Complete but may not be optimal C) Optimal but may not be complete D) Neither complete nor optimal

Depth-First Search: Guarantees and Complexity

Guarantees:

Question: A) Complete and optimal B) Complete but may not be optimal C) Optimal but may not be complete D) Neither complete nor optimal

- Optimality?

Depth-First Search: Guarantees and Complexity

Guarantees:

Question: A) Complete and optimal B) Complete but may not be optimal C) Optimal but may not be complete D) Neither complete nor optimal

- Optimality? No. After all, the algorithm just “chooses some direction and hopes for the best”. (Depth-first search is a way of “hoping to get lucky”.)
- Completeness?

Depth-First Search: Guarantees and Complexity

Guarantees:

Question: A) Complete and optimal B) Complete but may not be optimal C) Optimal but may not be complete D) Neither complete nor optimal

- Optimality? No. After all, the algorithm just "chooses some direction and hopes for the best". (Depth-first search is a way of "hoping to get lucky".)
- Completeness? No, because search branches may be infinitely long: No check for cycles along a branch!

  $\rightarrow$ Depth-first search is complete in case the state space is acyclic, e.g., Constraint Satisfaction Problems. If we do add a cycle check, it becomes complete for finite state spaces.

Depth-First Search: Guarantees and Complexity

Guarantees:

Question: A) Complete and optimal B) Complete but may not be optimal C) Optimal but may not be complete D) Neither complete nor optimal

- Optimality? No. After all, the algorithm just "chooses some direction and hopes for the best". (Depth-first search is a way of "hoping to get lucky".)
- Completeness? No, because search branches may be infinitely long: No check for cycles along a branch!

  $\rightarrow$ Depth-first search is complete in case the state space is acyclic, e.g., Constraint Satisfaction Problems. If we do add a cycle check, it becomes complete for finite state spaces.

Complexity:

- Space: Stores nodes and applicable actions on the path to the current node. So if m is the maximal depth reached, the complexity is

Depth-First Search: Guarantees and Complexity

Guarantees:

Question: A) Complete and optimal B) Complete but may not be optimal C) Optimal but may not be complete D) Neither complete nor optimal

- Optimality? No. After all, the algorithm just "chooses some direction and hopes for the best". (Depth-first search is a way of "hoping to get lucky".)
- Completeness? No, because search branches may be infinitely long: No check for cycles along a branch!

  $\rightarrow$ Depth-first search is complete in case the state space is acyclic, e.g., Constraint Satisfaction Problems. If we do add a cycle check, it becomes complete for finite state spaces.

Complexity:

- Space: Stores nodes and applicable actions on the path to the current node. So if m is the maximal depth reached, the complexity is $O(b\,m)$.
- Time: If there are paths of length m in the state space, $O(b^m)$ nodes can be generated. Even if there are solutions of depth 1!

Depth-First Search: Guarantees and Complexity

Guarantees:

Question: A) Complete and optimal B) Complete but may not be optimal C) Optimal but may not be complete D) Neither complete nor optimal

- Optimality? No. After all, the algorithm just “chooses some direction and hopes for the best”. (Depth-first search is a way of “hoping to get lucky”.)
- Completeness? No, because search branches may be infinitely long: No check for cycles along a branch!

  $\rightarrow$ Depth-first search is complete in case the state space is acyclic, e.g., Constraint Satisfaction Problems. If we do add a cycle check, it becomes complete for finite state spaces.

Complexity:

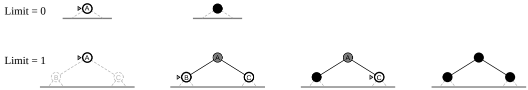
- Space: Stores nodes and applicable actions on the path to the current node. So if m is the maximal depth reached, the complexity is $O(b\,m)$.
- Time: If there are paths of length m in the state space, $O(b^m)$ nodes can be generated. Even if there are solutions of depth 1!

  $\rightarrow$ If we happen to choose “the right direction” then we can find a length-l solution in time $O(b\,l)$ regardless how big the state space is.

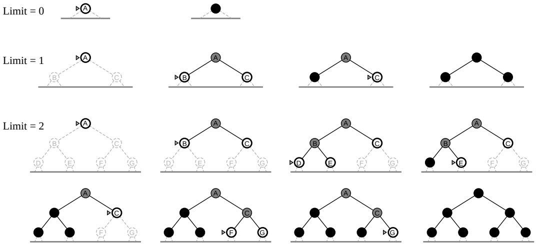
Iterative Deepening Search: Illustration

Limit = 0

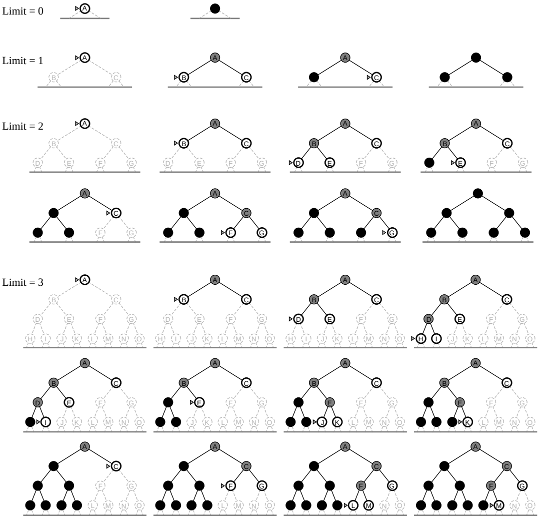
Iterative Deepening Search: Illustration



Iterative Deepening Search: Illustration



Iterative Deepening Search: Illustration



Iterative Deepening Search: Guarantees and Complexity

"Iterative Deepening Search=
Keep doing the same work over again until you find a solution."

Iterative Deepening Search: Guarantees and Complexity

"Iterative Deepening Search=
Keep doing the same work over again until you find a solution."

BUT: Optimality?

Iterative Deepening Search: Guarantees and Complexity

"Iterative Deepening Search=
Keep doing the same work over again until you find a solution."

BUT: Optimality? Yes! (assuming uniform costs) Completeness?

Iterative Deepening Search: Guarantees and Complexity

*"Iterative Deepening Search=
Keep doing the same work over again until you find a solution."*

BUT: Optimality? Yes! (assuming uniform costs) Completeness? Yes! Space complexity?

Iterative Deepening Search: Guarantees and Complexity

"Iterative Deepening Search=
Keep doing the same work over again until you find a solution."

BUT: Optimality? Yes! (assuming uniform costs) Completeness? Yes! Space complexity? $O(b\,d)$.

Iterative Deepening Search: Guarantees and Complexity

*"Iterative Deepening Search=
Keep doing the same work over again until you find a solution."*

BUT: Optimality? Yes! (assuming uniform costs) Completeness? Yes! Space complexity? $O(b\,d)$.

Time complexity:

Breadth-First-Search	$b + b^2 + \cdots + b^{d-1} + b^d \in O(b^d)$
Iterative Deepening Search	$(d)b + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + 1b^d \in O(b^d)$

Iterative Deepening Search: Guarantees and Complexity

*"Iterative Deepening Search=
Keep doing the same work over again until you find a solution."*

BUT: Optimality? Yes! (assuming uniform costs) Completeness? Yes! Space complexity? $O(b\,d)$.

Time complexity:

Breadth-First-Search	$b + b^2 + \cdots + b^{d-1} + b^d \in O(b^d)$
Iterative Deepening Search	$(d)b + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + 1b^d \in O(b^d)$

Example: $b = 10, d = 5$

Breadth-First Search	$10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$
Iterative Deepening Search	$50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$

$\rightarrow$ IDS combines the advantages of breadth-first and depth-first search. It is the preferred blind search method in large state spaces with unknown solution depth.

Agenda

1. Blind Systematic Search Algorithms

2. Informed Systematic Search Algorithms

Greedy Best-First Search

Greedy Best-First Search (with duplicate detection)

open $:=$ **new** priority queue ordered by ascending $h(\textit{state}(\sigma))$
open.insert(make-root-node(init()))
closed $:= \emptyset$
while not *open*.empty():
    $\sigma :=$ *open*.pop-min() /* get best state */
    if $\textit{state}(\sigma) \notin \textit{closed}$: /* check duplicates */
        closed $:= \textit{closed} \cup \{\textit{state}(\sigma)\}$ /* close state */
        if is-goal(state(σ)): **return** extract-solution(σ)
        for each $(a, s') \in$ succ($\textit{state}(\sigma)$): /* expand state */
            $\sigma' :=$ make-node(σ, a, s')
            if $h(\textit{state}(\sigma')) < \infty$: *open*.insert($\sigma'$)
return unsolvable

Greedy Best-First Search: Remarks

Properties:

- Complete?

[1] Even for perfect heuristics! E.g., say the start state has two transitions to goal states, one of which costs a million bucks while the other one is free. Nothing keeps Greedy Best-First Search from choosing the bad one.

Greedy Best-First Search: Remarks

Properties:

- Complete? Yes, for safe heuristics. (and duplicate detection to avoid cycles)
- Optimal?

[1] Even for perfect heuristics! E.g., say the start state has two transitions to goal states, one of which costs a million bucks while the other one is free. Nothing keeps Greedy Best-First Search from choosing the bad one.

Greedy Best-First Search: Remarks

Properties:

- Complete? Yes, for safe heuristics. (and duplicate detection to avoid cycles)
- Optimal? No.[1]
- Invariant under all strictly monotonic transformations of h (e.g., scaling with a positive constant or adding a constant).

Implementation:

- Priority queue: e.g., a min heap.
- "Check Duplicates": Could already do in "expand state"; done here after "get best state" *only* to more clearly point out relation to A^*.

[1] Even for perfect heuristics! E.g., say the start state has two transitions to goal states, one of which costs a million bucks while the other one is free. Nothing keeps Greedy Best-First Search from choosing the bad one.

A^*

A^* (with duplicate detection and re-opening)

```
open := new priority queue ordered by ascending g(state(σ)) + h(state(σ))
open.insert(make-root-node(init()))
closed := ∅
best-g := ∅/* maps states to numbers */
while not open.empty():
        σ := open.pop-min()
        if state(σ) ∉ closed or g(σ) < best-g(state(σ)):
          /* re-open if better g; note that all σ′ with same state but worse g
             are behind σ in open, and will be skipped when their turn comes */
          closed := closed ∪ {state(σ)}
          best-g(state(σ)) := g(σ)
          if is-goal(state(σ)): return extract-solution(σ)
          for each (a, s′) ∈ succ(state(σ)):
              σ′ := make-node(σ, a, s′)
              if h(state(σ′)) < ∞: open.insert(σ′)
return unsolvable
```

A^*: Remarks

Properties:

- Complete?

A^*: Remarks

Properties:

- Complete? Yes, for safe heuristics. (Even without duplicate detection.)
- Optimal?

A^*: Remarks

Properties:

- Complete? Yes, for safe heuristics. (Even without duplicate detection.)
- Optimal? Yes, for admissible heuristics. (Even without duplicate detection.)

Implementation:

- Popular method: break ties ($f(s) = f(s')$) by smaller h-value.
- If h is admissible and consistent, then A^* never re-opens a state. So if we know that this is the case, then we can simplify the algorithm.
- Common, hard to spot bug: check duplicates at the wrong point. (Russel & Norvig are way too imprecise about this.)
- Our implementation is optimized for readability not for efficiency!

Question

Question!

If we set $h(n) := 0$ for all n, what does A^* become?

(A): Breadth-first search.
(B): Depth-first search.
(C): Uniform-cost search.
(D): Depth-limited search.

Question

Question!

If we set $h(n) := 0$ for all n, what does A^* become?

(A): Breadth-first search.
(B): Depth-first search.
(C): Uniform-cost search.
(D): Depth-limited search.

$\rightarrow$ (C): Same expansion order. (Details in book-keeping of open/closed states may differ.)