

COMP90054 — AI Planning for Autonomy

2. Search Algorithms

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

Nir Lipovetzky



THE UNIVERSITY OF
MELBOURNE

Semester 2, 2020
Copyright, University of Melbourne

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** $c(a, s)$

→ 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)

→ Different **models** and **controllers** obtained by relaxing assumptions in **blue** ...

Solving the State Model: Path-finding in graphs

Search algorithms for planning exploit the **correspondence** between **(classical) states model** $\mathcal{S}(P)$ and **directed graphs**:

- The **nodes** of the graph represent the **states** s in the model
- The edges (s, s') capture corresponding transition in the model with same cost

In the **planning as heuristic search** formulation, the problem P is solved by **path-finding** algorithms over the **graph** associated with model $\mathcal{S}(P)$

Classification of Search Algorithms

Blind search vs. heuristic (or informed) search:

- **Blind search algorithms:** Only use the basic ingredients for general search algorithms.
 - *e.g., Depth First Search (DFS), Breadth-first search (BrFS), Uniform Cost (Dijkstra), Iterative Deepening (ID)*
- **Heuristic search algorithms:** Additionally use **heuristic functions** which estimate the distance (or remaining cost) to the goal.
 - *e.g., A*, IDA*, Hill Climbing, Best First, WA*, DFS B&B, LRTA*, ...*

Systematic search vs. local search:

- **Systematic search algorithms:** Consider a large number of search nodes simultaneously.
- **Local search algorithms:** Work with one (or a few) candidate solutions (search nodes) at a time.
 - This is not a black-and-white distinction; there are *crossbreeds* (e.g., **enforced hill-climbing**).

What works where in planning?

Blind search vs. heuristic search:

- For **satisficing** planning, heuristic search vastly outperforms blind algorithms pretty much everywhere.
- For **optimal** planning, heuristic search also is better (but the difference is less pronounced).

Systematic search vs. local search:

- For **satisficing** planning, there are successful instances of each.
- For **optimal** planning, systematic algorithms are required.

→ Here, we cover the subset of search algorithms most successful in planning. Only some Blind search algorithms are covered. (refer to Russel & Norvig Chapters 3 and 4 for that).

Search Terminology

Search node n : Contains a *state* reached by the search, plus information about how it was reached.

Path cost $g(n)$: The cost of the path reaching n .

Optimal cost g^* : The cost of an optimal solution path. For a state s , $g^*(s)$ is the cost of a cheapest path reaching s .

Node expansion: Generating all successors of a node, by applying all actions applicable to the node's state s . Afterwards, the *state* s itself is also said to be expanded.

Search strategy: Method for deciding which node is expanded next.

Open list: Set of all *nodes* that currently are candidates for expansion. Also called **frontier**.

Closed list: Set of all *states* that were already expanded. Used only in **graph search**, not in **tree search** (up next). Also called **explored set**.

World States vs. Search States

Reminder: Search Space for Classical Search

A (classical) **search space** is defined by the following three operations:

- **start()**: Generate the start (search) state.
- **is-target(s)**: Test whether a given search state is a target state.
- **succ(s)**: Generates the successor states (a, s') of search state s , along with the actions through which they are reached.

Search states \neq world states:

- Progression:
- Regression:

→ We consider progression in the entire course, unless explicitly stated otherwise.
We use “ s ” to denote world/search states interchangeably.

Search States vs. Search Nodes

- **Search states** s : States (vertices) of the search space.
- **Search nodes** σ : Search states, plus information on where/when/how they are encountered during search.

What is in a search node?

Different search algorithms store different information in a search node σ , but typical information includes:

- $state(\sigma)$: Associated search state.
- $parent(\sigma)$: Pointer to search node from which σ is reached.
- $action(\sigma)$: An action leading from $state(parent(\sigma))$ to $state(\sigma)$.
- $g(\sigma)$: Cost of σ (cost of path from the root node to σ).

For the root node, $parent(\sigma)$ and $action(\sigma)$ are undefined.

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.

Before We Begin

Blind search vs. informed search:

- **Blind search** does not require any input beyond the problem.

→ Pros and Cons?

- **Informed search** requires as additional input a **heuristic function h** (**Next Chapter**) that maps states to estimates of their **goal distance**.

→ Pros and Cons?

→ Note: In **planning**, h is generated automatically from the declarative problem description

Before We Begin, ctd.

Blind search strategies we'll discuss:

- **Breadth-first search**. Advantage: time complexity.
Variant: **Uniform cost search**.
- **Depth-first search**. Advantage: space complexity.
- **Iterative deepening search**. Combines advantages of breadth-first search and depth-first search. Uses **depth-limited search** as a sub-procedure.

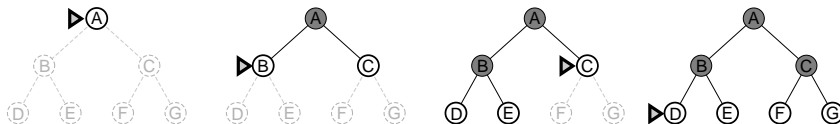
Blind search strategy we won't discuss:

- **Bi-directional search**. Two separate search spaces, one forward from the initial state, the other backward from the goal. Stops when the two search spaces overlap.

Breadth-First Search: Illustration and Guarantees

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

Illustration:



Guarantees:

- Completeness?
- Optimality?

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?
- So the time complexity is
- And if we were to apply the goal test at node-expansion time, rather than node-generation time?

Space Complexity:

Breadth-First Search: Example Data

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

Yields data: (inserting values into previous equations)

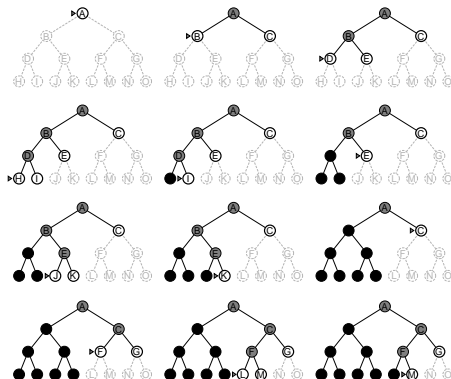
Depth	Nodes	Time		Memory	
2	110	.11	milliseconds	107	kilobytes
4	11110	11	milliseconds	10.6	megabytes
6	10^6	1.1	seconds	1	gigabyte
8	10^8	2	minutes	103	gigabytes
10	10^{10}	3	hours	10	terabytes
12	10^{12}	13	days	1	petabyte
14	10^{14}	3.5	years	99	petabytes

→ So, which is the worse problem, time or memory?

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:

- Optimality?
- Completeness?

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

Iterative Deepening Search: Illustration

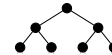
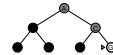
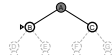
Limit = 0



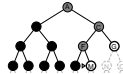
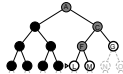
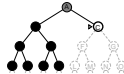
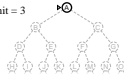
Limit = 1



Limit = 2



Limit = 3



Iterative Deepening Search: Guarantees and Complexity

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

BUT: Optimality? Completeness? Space complexity?

Time complexity:

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

Example: $b = 10, d = 5$

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

→ 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.

Heuristic Search Algorithms: Systematic

→ Heuristic search algorithms are the most common and overall most successful algorithms for classical planning.

Systematic heuristic search algorithms:

- Greedy best-first search.
→ One of 3 most popular algorithms in satisficing planning.
- Weighted A^* .
→ One of 3 most popular algorithms in satisficing planning.
- A^* .
→ Most popular algorithm in optimal planning. (Rarely ever used for satisficing planning.)
- IDA*, depth-first branch-and-bound search, breadth-first heuristic search, ...

The origins: Shakey Video

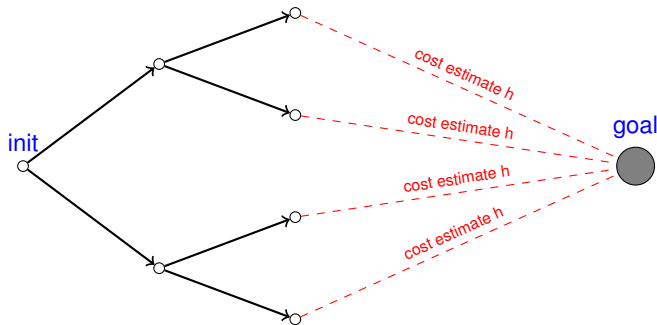
Heuristic Search Algorithms: Local

→ Heuristic search algorithms are the most common and overall most successful algorithms for classical planning.

Local heuristic search algorithms:

- Hill-climbing.
- Enforced hill-climbing.
 - One of 3 most popular algorithms in satisficing planning.
- Beam search, tabu search, genetic algorithms, simulated annealing, ...

Heuristic Search: Basic Idea



→ Heuristic function h estimates the cost of an optimal path to the goal; search gives a preference to explore states with small h .

Heuristic Functions

Heuristic searches require a heuristic function to estimate remaining cost:

Definition (Heuristic Function). Let Π be a planning task with state space Θ_{Π} . A *heuristic function*, short *heuristic*, for Π is a function $h : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$. Its value $h(s)$ for a state s is referred to as the state's *heuristic value*, or *h-value*.

Definition (Remaining Cost, h^*). Let Π be a planning task with state space Θ_{Π} . For a state $s \in S$, the state's *remaining cost* is the cost of an optimal plan for s , or ∞ if there exists no plan for s . The *perfect heuristic* for Π , written h^* , assigns every $s \in S$ its remaining cost as the heuristic value.

Heuristic Functions: Discussion

What does it mean to “estimate remaining cost”?

- For many heuristic search algorithms, h does not need to have any properties for the algorithm to “work” (= be correct and complete).
→ h is *any* function from states to numbers . . .
- Search **performance** depends crucially on “how well h reflects h^* ”!!
→ This is informally called the **informedness** or **quality** of h .
- For some search algorithms, like A^* , we can *prove* relationships between formal quality properties of h and search efficiency (mainly the number of expanded nodes).
- For other search algorithms, “it works well in practice” is often as good an analysis as one gets.

→ We will analyze in detail approximations to one particularly important heuristic function in planning: h^+ .

Heuristic Functions: Discussion, ctd.

“Search performance depends crucially on the informedness of h . . . ”

Any other property of h that search performance crucially depends on?

Properties of Heuristic Functions

Definition (Safe/Goal-Aware/Admissible/Consistent). Let Π be a planning task with state space $\Theta_{\Pi} = (S, L, c, T, I, S^G)$, and let h be a heuristic for Π . The heuristic is called:

- **safe** if $h^*(s) = \infty$ for all $s \in S$ with $h(s) = \infty$;
- **goal-aware** if $h(s) = 0$ for all goal states $s \in S^G$;
- **admissible** if $h(s) \leq h^*(s)$ for all $s \in s$;
- **consistent** if $h(s) \leq h(s') + c(a)$ for all transitions $s \xrightarrow{a} s'$.

→ **Relationships?**

Greedy Best-First Search

Greedy Best-First Search (with duplicate detection)

```

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

Greedy Best-First Search: Remarks

Properties:

- Complete?
- Optimal?
- Invariant under all strictly monotonic transformations of h (e.g., scaling with a positive constant or adding a constant).

Implementation:

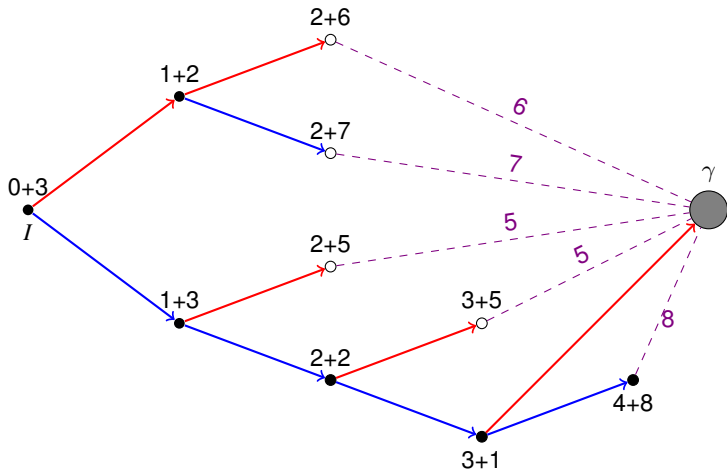
- Priority queue: e.g., a [min heap](#).
- “Check Duplicates”: Could already do in “expand state”; done here after “get best state” *only* to more clearly point out relation to A^* .

A*

A* (with duplicate detection and re-opening)

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

A*: Example



A*: Terminology

- **f -value** of a state: defined by $f(s) := g(s) + h(s)$.
- **Generated nodes**: Nodes inserted into *open* at some point.
- **Expanded nodes**: Nodes σ popped from *open* for which the test against *closed* and *distance* succeeds.
- **Re-expanded nodes**: Expanded nodes for which $state(\sigma) \in closed$ upon expansion (also called **re-opened** nodes).

A*: Remarks

Properties:

- Complete?
- Optimal?

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!

Quizz!

Weighted A*

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

```
open := new priority queue ordered by ascending  $g(\text{state}(\sigma)) + W * h(\text{state}(\sigma))$ 
open.insert(make-root-node(init()))
closed :=  $\emptyset$ 
best-g :=  $\emptyset$ 
while not open.empty():
     $\sigma := \text{open.pop-min}()$ 
    if  $\text{state}(\sigma) \notin \text{closed}$  or  $g(\sigma) < \text{best-g}(\text{state}(\sigma))$ :
        closed := closed  $\cup \{\text{state}(\sigma)\}$ 
        best-g( $\text{state}(\sigma)$ ) :=  $g(\sigma)$ 
        if is-goal( $\text{state}(\sigma)$ ): return extract-solution( $\sigma$ )
        for each  $(a, s') \in \text{succ}(\text{state}(\sigma))$ :
             $\sigma' := \text{make-node}(\sigma, a, s')$ 
            if  $h(\text{state}(\sigma')) < \infty$ : open.insert( $\sigma'$ )
return unsolvable
```

Weighted A*: Remarks

The **weight** $W \in \mathbb{R}_0^+$ is an **algorithm parameter**:

- For $W = 0$, weighted A* behaves like
- For $W = 1$, weighted A* behaves like
- For $W \rightarrow \infty$, weighted A* behaves like

Properties:

- For $W > 1$, weighted A* is **bounded suboptimal**: if h is admissible, then the solutions returned are at most a factor W more costly than the optimal ones.

Hill-Climbing

Hill-Climbing

```
 $\sigma := \text{make-root-node}(\text{init}())$   
forever:  
  if  $\text{is-goal}(\text{state}(\sigma))$ :  
    return  $\text{extract-solution}(\sigma)$   
   $\Sigma' := \{ \text{make-node}(\sigma, a, s') \mid (a, s') \in \text{succ}(\text{state}(\sigma)) \}$   
   $\sigma := \text{an element of } \Sigma' \text{ minimizing } h$  /* (random tie breaking) */
```

Remarks:

- Makes sense only if $h(s) > 0$ for $s \notin S^G$.
- Is this complete or optimal?
- Can easily get stuck in local minima where immediate improvements of $h(\sigma)$ are not possible.
- Many variations: tie-breaking strategies, restarts, ...

Enforced Hill-Climbing

Enforced Hill-Climbing: Procedure *improve*

```
def improve( $\sigma_0$ ):  
    queue := new fifo queue  
    queue.push-back( $\sigma_0$ )  
    closed :=  $\emptyset$   
    while not queue.empty():  
         $\sigma$  = queue.pop-front()  
        if state( $\sigma$ )  $\notin$  closed:  
            closed := closed  $\cup$  {state( $\sigma$ )}  
            if  $h(\text{state}(\sigma)) < h(\text{state}(\sigma_0))$ : return  $\sigma$   
            for each ( $a, s'$ )  $\in$  succ(state( $\sigma$ )):  
                 $\sigma'$  := make-node( $\sigma, a, s'$ )  
                queue.push-back( $\sigma'$ )  
  
    fail
```

↪ Breadth-first search for state with strictly smaller h -value.

Enforced Hill-Climbing, ctd.

Enforced Hill-Climbing

```
 $\sigma := \text{make-root-node}(\text{init}())$   
while not  $\text{is-goal}(\text{state}(\sigma))$ :  
     $\sigma := \text{improve}(\sigma)$   
return  $\text{extract-solution}(\sigma)$ 
```

Remarks:

- Makes sense only if $h(s) > 0$ for $s \notin S^G$.
- Is this optimal?
- Is this complete?

Properties of Search Algorithms

	DFS	BrFS	ID	A*	HC	IDA*
Complete	No	Yes	Yes	Yes	No	Yes
Optimal	No	Yes*	Yes	Yes	No	Yes
Time	∞	b^d	b^d	b^d	∞	b^d
Space	$b \cdot d$	b^d	$b \cdot d$	b^d	b	$b \cdot d$

- Parameters: d is solution depth; b is branching factor
- Breadth First Search (BrFS) optimal when costs are uniform
- A*/IDA* optimal when h is **admissible**; $h \leq h^*$

Quiz!

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!

If we set $h(n) := 0$ for all n , what can greedy best-first search become?

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

Quiz!, ctd.

Question!

Is informed search always better than blind search?

(A): Yes.

(B): No.

Summary

Distinguish: World states, search states, search nodes.

- **World state:** Situation in the world modelled by the planning task.
- **Search state:** Subproblem remaining to be solved.
 - In **progression**, world states and search states are identical.
 - In **regression**, search states are sub-goals describing sets of world states.
- **Search node:** Search state + info on “how we got there”.

Search algorithms mainly differ in **order of node expansion**:

- **Blind** vs. **heuristic** (or **informed**) search.
- **Systematic** vs. **local** search.

Summary (ctd.)

- **Search strategies** differ (amongst others) in the order in which they **expand search nodes**, and in the way they use **duplicate elimination**. Criteria for evaluating them are **completeness**, **optimality**, **time complexity**, and **space complexity**.
- **Breadth-first search** is optimal but uses exponential space; **depth-first search** uses linear space but is not optimal. **Iterative deepening search** combines the virtues of both.

Summary (ctd.)

Heuristic Functions: Estimators for **remaining cost**.

- Usually: The more **informed**, the better performance.
- Desiderata: **Safe**, **goal-aware**, **admissible**, **consistent**.
- The ideal: **Perfect heuristic h^*** .

Heuristic Search Algorithms:

- Most common algorithms for **satisficing planning**:
 - **Greedy best-first search**.
 - **Weighted A^*** .
 - **Enforced hill-climbing**.
- Most common algorithm for **optimal planning**:
 - **A^*** .

Reading

- *Artificial Intelligence: A Modern Approach (Third Edition)* , Chapter 3 “Solving Problems by Searching” and the first half of Chapter 4 “Beyond Classical Search”.

Content: An overview of various search algorithms, including blind searches as well as greedy best-first search and A*.

- Search Tutorial in the context of path-finding <http://www.redblobgames.com/pathfinding/a-star/introduction.html>