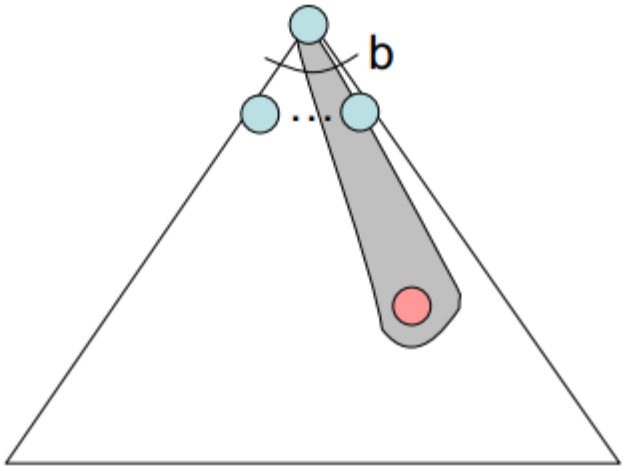
1 Informed Search

A fundamental component of informed search is the notion of a heuristic. A heuristic is a function that estimates how close a given state is to the goal state with a numerical value. The choice of a heuristic function is dependent on the search problem at hand.

1.1 Greedy Search

As the name implies, Greedy Search greedily chooses the node that seems closest to the goal node by using the heuristic function on successor states in the fringe. This often leads to sub-optimal solutions as we ignore overall paths and naively select the next closest node. Therefore, greedy search is not an optimal search algorithm.



1.2 A* Search

A* Search combines two algorithms we have seen already: greedy search and uniform-cost search. This algorithm introduces two costs to consider:
h(n) - the forward cost. This is the estimated value assigned by the heuristic function.
g(n) - the backward cost. The cumulative cost of getting to the current state from the start state.
While greedy search only considers h(n) and ucs only considers g(n), A* Search combines both for a new function f(n) = h(n) + g(n).

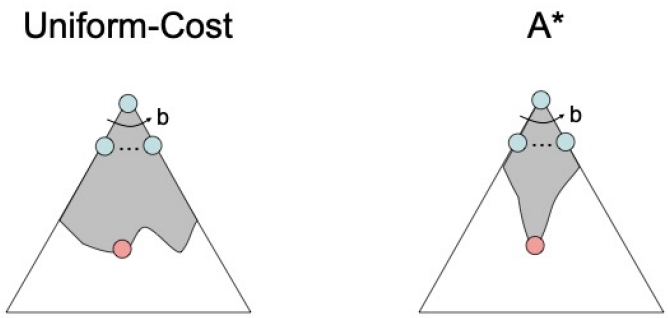
Is this an optimal search algorithm?
It depends on the selected Heuristic! This is where the idea of **admissibility** of heuristics comes in. Inadmissible (pessimistic) heuristics break optimality as they overestimate the actual cost of good plans in the fringe. Admissible (optimistic) heuristics, on the other hand, slow down bad plans but never outweigh true costs. Formally, a heuristic is admissible if for every state n:

$$0 \leq h(n) \leq h^*(n)$$

where h*(n) is the true cost to a nearest goal.

Comparison with UCS and Greedy Search

A* search expands mainly toward the goal, while UCS expands equally in all directions. A* search is not as efficient as greedy search but finds the optimal solution when using admissible heuristics. This is because A* search considers nodes based on both the backward costs and the estimation of forward costs.



Design Principles of Heuristics

Admissible heuristics are usually solutions to **relaxed problems**. For example, in Pacman experiments, we can use Manhattan or Euclidean distance by ignoring all the walls. Inadmissible heuristics can also be useful.

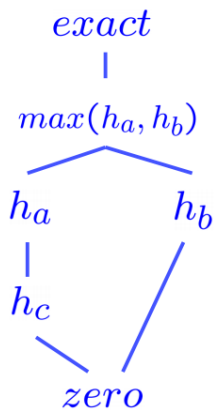
Case Study: 8 Puzzle

- Problem Statement - The 8-puzzle is a sliding puzzle with a 3x3 grid and 8 numbered tiles. The objective is to rearrange the tiles to reach a specific goal configuration by sliding them into an empty space, with the goal typically being an ordered arrangement of numbers.
- Heuristic 1 - Number of misplaced tiles - a rough, yet admissible estimation.
- Heuristic 2 - Total Manhattan distance - closer to the actual cost, more efficient.
- Summary - There is a trade-off between the quality of the estimate and the computational effort per node. As heuristics approach the true cost, you will expand fewer nodes but generally require more computational work per node to compute the heuristic itself.

Case Study: Pancake Problem

- Problem Statement - The Pancake Problem involves arranging a stack of pancakes in a specific order by flipping them, with the goal being to achieve a desired pancake arrangement.
- Heuristic - The number of the largest pancake that is still out of place.

Semi-Lattice of Heuristics
The Semi-Lattice of Heuristics organizes heuristic functions into a partially ordered set (or lattice) where each heuristic is a node, and **dominance** relationships between heuristics are represented as edges in the lattice. Heuristics that dominate others are placed at higher levels in the lattice, indicating their superior performance in guiding the search. Heuristics that are not dominated by any other heuristics form the lower levels of the lattice.



One heuristic, h_a, is said to dominate another heuristic, h_b, if the estimated goal distance for h_a is greater than the estimated goal distance for h_b for every node in the state space graph. Mathematically,

$$\forall n : h_a(n) \geq h_b(n).$$

Dominance intuitively captures the idea of one heuristic being better than another because the dominant one will always more closely estimate the distance to a goal from any given state.
As a general rule, the **maximum** function applied to multiple admissible heuristics will also always be admissible. This is simply a consequence of all values output by the heuristics for any given state being constrained by the admissibility condition, $0 \leq h(n) \leq h^*(n)$. The maximum of numbers in this range must also fall in the same range. It's common practice to generate multiple admissible heuristics for any given search problem and compute the maximum over the values output by them to generate a heuristic that dominates (and hence is better than) all of them individually.
Additionally, the **trivial heuristic** is defined as $h(n) = 0$, and using it reduces A* search to Uniform Cost Search (UCS). All admissible heuristics dominate the trivial heuristic. The trivial heuristic is often incorporated at the base of a semi-lattice for a search problem, forming a dominance hierarchy where it is located at the bottom, while the exact goal distance is placed at the top of the semi-lattice.

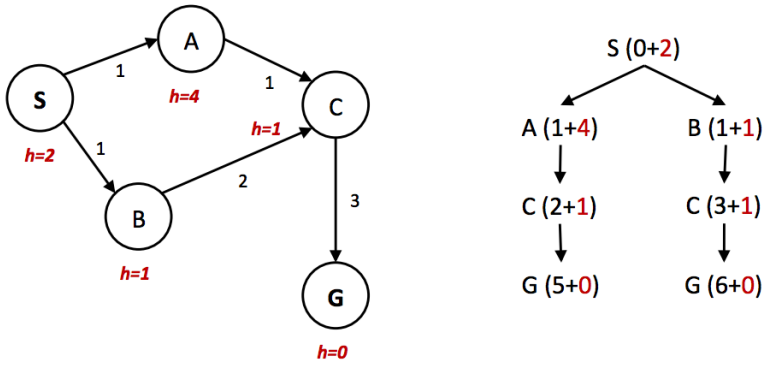
Optimality of A* Tree Search **Theorem.** For a given search problem, if the admissibility constraint is satisfied by a heuristic function h, using A* tree search with h on that search problem will yield an optimal solution.
proof Assume there are two reachable goal states, optimal goal A and suboptimal goal B, in the search tree for a given problem. Let n be an ancestor of A on the fringe. We claim that n is selected for expansion before B based on the following:

1. $g(A) < g(B)$: Optimal A has a lower backward cost to the start state than suboptimal B.
2. $h(A) = h(B) = 0$: Both A and B are goal states, and heuristic satifies the admissiblity constraint.
3. $f(n) \leq f(A)$: By admissibility, $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = g(A) = f(A)$.

Combining 1 and 2 yields $f(A) < f(B)$. Combining this inequality with 3 results in $f(n) < f(B)$. Therefore, n is expanded before B, and this holds true for all ancestors of A, including A itself. □

Graph Search One failure we can observe in a search tree is to detect repeated states, which can cause exponentially more work. Even in situations where our search technique doesn't involve an infinite loop, it's often the case that we revisit the same node multiple times because there are multiple ways to reach that same node. A natural solution is to simply keep track of which states you've already expanded and never expand them again. More explicitly, maintain a 'reached' **closed set** of expanded nodes while conducting the search. Then, ensure that each node isn't already in the set before expansion and add it to the set after expansion if it's not. Tree search with this added optimization is known as **graph search**.

Consistency of Heuristics An additional caveat of graph search is that it tends to ruin the optimality of A*, even under admissible heuristics. Consider the following simple state space graph and corresponding search tree, annotated with weights and heuristic values:



In the above example, it's clear that the optimal route is to follow $S \to A \to C \to G$, yielding a total path cost of $1+1+3 = 5$. The only other path to the goal, $S \to B \to C \to G$ has a path cost of

$1 + 2 + 3 = 6$. However, because the heuristic value of node A is so much larger than the heuristic value of node B, node C is first expanded along the second, suboptimal path as a child of node B. It's then placed into the "reached" set, and so A* graph search fails to expand it when it visits it as a child of A, so it never finds the optimal solution. Hence, to maintain optimality under A* graph search, we need an even **stronger** property than admissibility, **consistency**. The central idea of consistency is that we enforce not only that a heuristic underestimates the *total* distance to a goal from any given node, but also the cost/weight of each edge in the graph. The cost of an edge as measured by the heuristic function is simply the difference in heuristic values for two connected nodes. Mathematically, the consistency constraint can be expressed as follows:

$$\forall A, C \quad h(A) - h(C) \leq cost(A, C)$$

Optimality of A* Graph Search

Theorem. For a given search problem, if the consistency constraint is satisfied by a heuristic function h, using A* graph search with h on that search problem will yield an optimal solution.

Proof. In order to prove the above theorem, we first prove that when running A* graph search with a consistent heuristic, whenever we remove a node for expansion, we've found the optimal path to that node.

Using the consistency constraint, we can show that the values of $f(n)$ for nodes along any plan are nondecreasing. Define two nodes, n and n', where n' is a child of n. Then:

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

If for every parent-child pair (n, n') along a path, $f(n') \geq f(n)$, then it must be the case that the values of $f(n)$ are nondecreasing along that path. We can check that the above graph violates this rule between $f(A)$ and $f(C)$. With this information, we can now show that whenever a node n is removed for expansion, its optimal path has been found. Assume towards a contradiction that this is false - that when n is removed from the fringe, the path found to n is suboptimal. This means that there must be some ancestor of n, n'', on the fringe that was never expanded but is on the optimal path to n. Contradiction! We've already shown that values of f along a path are nondecreasing, and so n'' would have been removed for expansion before n.

All we have left to show to complete our proof is that an optimal goal A will always be removed for expansion and returned before any suboptimal goal B. This is trivial, since $h(A) = h(B) = 0$, so

$$f(A) = g(A) < g(B) = f(B)$$

just as in our proof of optimality of A* tree search under the admissibility constraint. Hence, we can conclude that A* graph search is optimal under a consistent heuristic. □

1.3 Summary

In this note, we discussed search problems and their components: a state space, a set of actions, a transition function, an action cost, a start state and a goal state.
Regarding the search problems, they can be solved using a variety of search techniques, including but not limited to the five we study:

- Breadth-first Search
- Depth-first Search
- Uniform Cost Search
- Greedy Search
- A* Search

The first three search techniques listed above are examples of uninformed search, while the latter two are examples of informed search which use heuristics to estimate goal distance and optimize performance. We additionally made a distinction between tree search and graph search algorithms for the above techniques.

- With a admissible heuristic, Tree A* is optimal.
- With a consistent heuristic, Graph A* is optimal.
- With h=0, the same proof shows that UCS is optimal.

A couple of important highlights from the discussion above: for heuristics that are either admissible/consistent to be valid, it must by definition be the case that $h(G) = 0$ for any goal state G. Additionally, consistency is not just a **stronger** constraint than admissibility, consistency *implies* admissibility. This stems simply from the fact that if no edge costs are overestimates (as guaranteed by consistency), the total estimated cost from any node to a goal will also fail to be an overestimate.