Informed Search

Table of Contents

[Pancake Problem 3](#_Toc138878609)

[Search Heuristics 4](#_Toc138878610)

[Greedy Search 6](#_Toc138878611)

[A\* Search 7](#_Toc138878612)

[Termination 8](#_Toc138878613)

[Optimality 9](#_Toc138878614)

[Proving Optimality 10](#_Toc138878615)

[UCS vs A\* 11](#_Toc138878616)

[A\* Applications 12](#_Toc138878617)

[Comparison of Algorithms 12](#_Toc138878618)

[Creating Admissible Heuristic 14](#_Toc138878619)

[Semi-Lattice of Heuristics 17](#_Toc138878620)

[Graph Search 18](#_Toc138878621)

[A\* Search 18](#_Toc138878622)

[Consistency 19](#_Toc138878623)

[Proof of Optimality 21](#_Toc138878624)

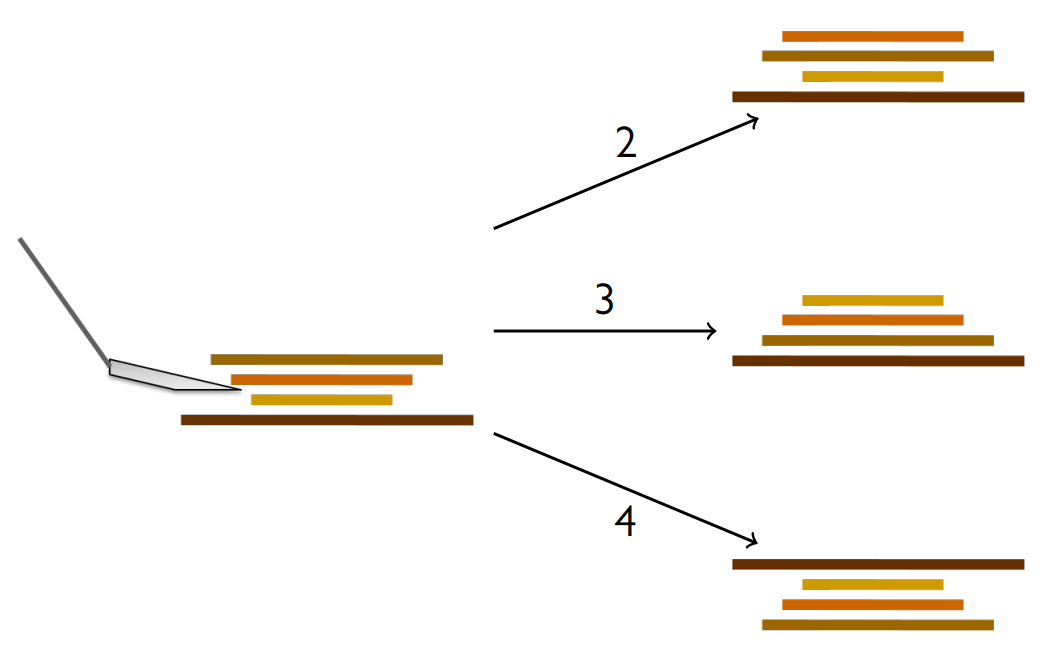
[Pseudocode 22](#_Toc138878625)

[Tree Search 22](#_Toc138878626)

[Graph Search 22](#_Toc138878627)

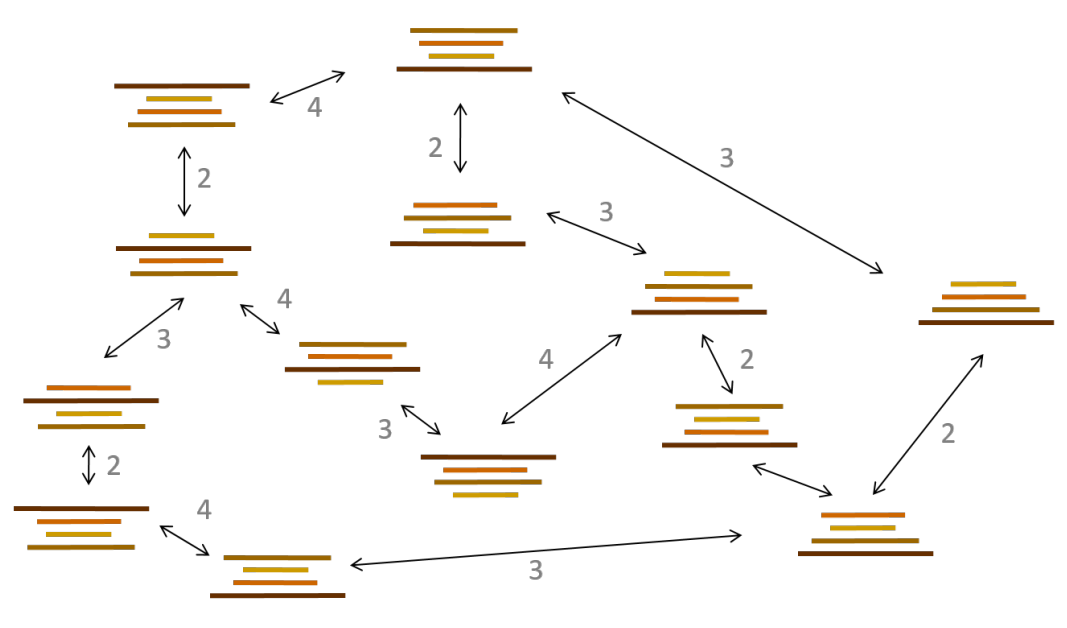
## Pancake Problem

Consider that we have a stack of pancakes of different sizes, and we want to flip them so that they are sorted based on size. The diagram below shows a few possible ways to do this.



The stack on the left can be considered our **current state**. From this position, based on the number of pancakes we flip at the same time, we can reach various different states.

A part of the total state space graph is shown below.



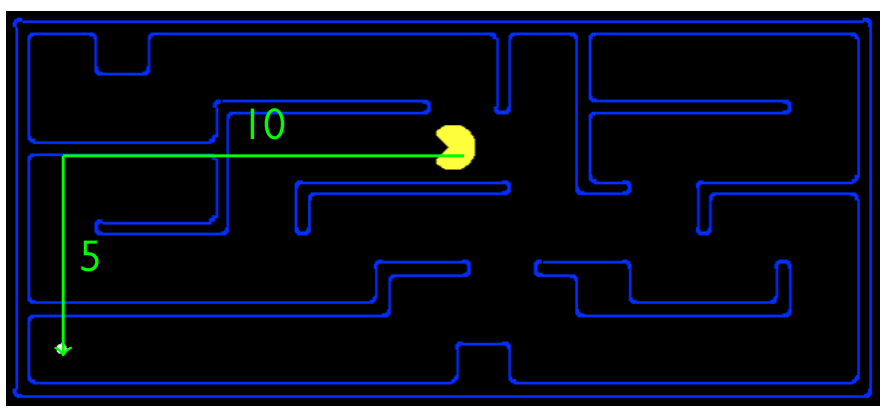
We can solve this problem, as before, using a **unform cost search** algorithm. The steps are:

1. Initialize the search tree using the initial state of the problem.
2. Loop:
   1. If there are no candidates for expansion, return a **failure**
   2. Otherwise, choose a leaf node based on the chosen strategy
   3. If the node contains a **goal state**, return the corresponding solution
   4. Otherwise, expand the node and add the resulting nodes to the search tree

UCS is optimal and complete, but it is a **slow algorithm**. This happens because the algorithm has no clue which direction the final goal state is. We can make it faster using **heuristics**, which essentially nudges UCS in the right direction.

## Search Heuristics

A **heuristic** is an estimate of how close a given state is to the goal state. The heuristic used is specific to the problem we are trying to solve. For example, one possible heuristic for Pac-Man is the **Manhattan Distance**, which is calculated as .



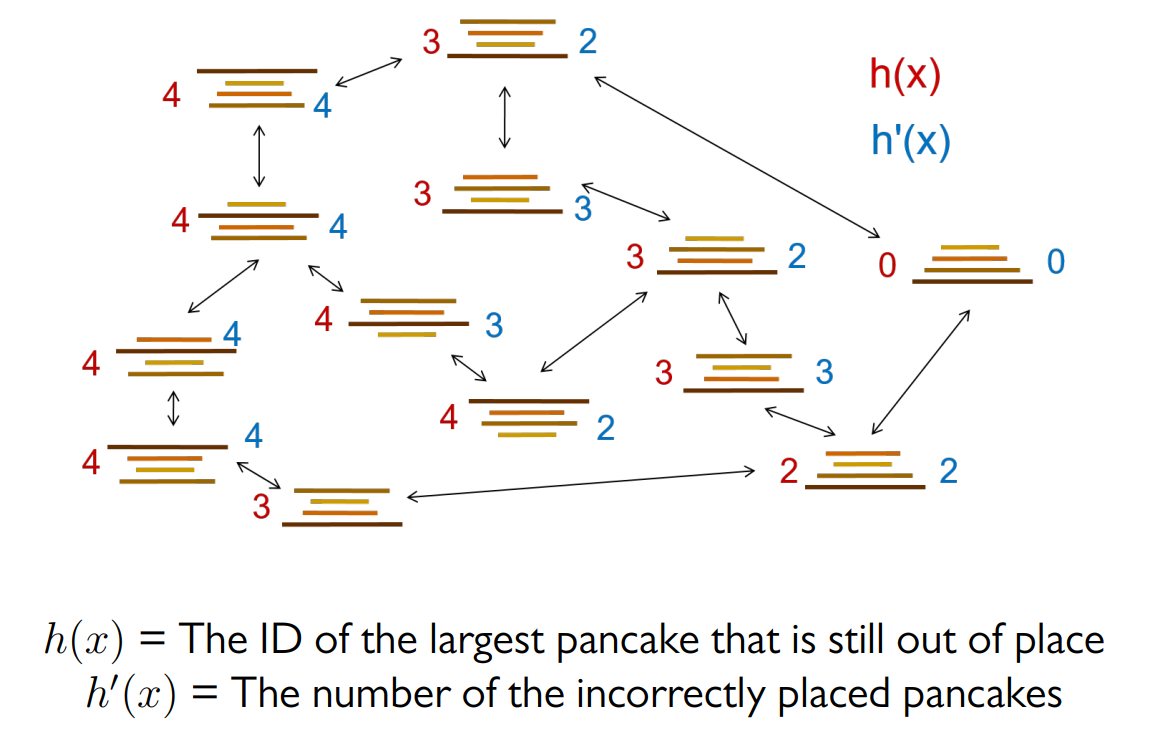
The criterion for a heuristic is that it must be **easy to calculate**. For example, the Manhattan distance heuristic assumes there are no walls, and we can move in a square. We could also have used the Euclidean distance metric, which would be a diagonal line from the current position to the goal. These are easy values to calculate.

The idea is that we will use the heuristic as an estimation of which direction to move in. The actual search algorithm will find the exact cost, so the heuristic value will not be exact. However, we will decide which direction to move in based on which move gives us a lower heuristic value.

Not every heuristic makes sense for every problem. The heuristic needs to be able to **closely estimate** the exact cost, since large errors will make the algorithm following the heuristic perform worse. That is why we need to design our heuristic based on the problem. For the above problem, the Manhattan distance heuristic is more accurate than the Euclidean distance heuristic, since Pac-Man cannot move diagonally.

Coming back to the Pancake problem, one possible heuristic is the number of pancakes in the correct position. This is a bad heuristic. Our main problem concerns minimizing the number of pancakes we have to flip, but the heuristic gives us a value which we should be maximizing. Thus, the heuristic will not be a close estimate of our minimum cost. A better heuristic is the number of pancakes in the incorrect position.

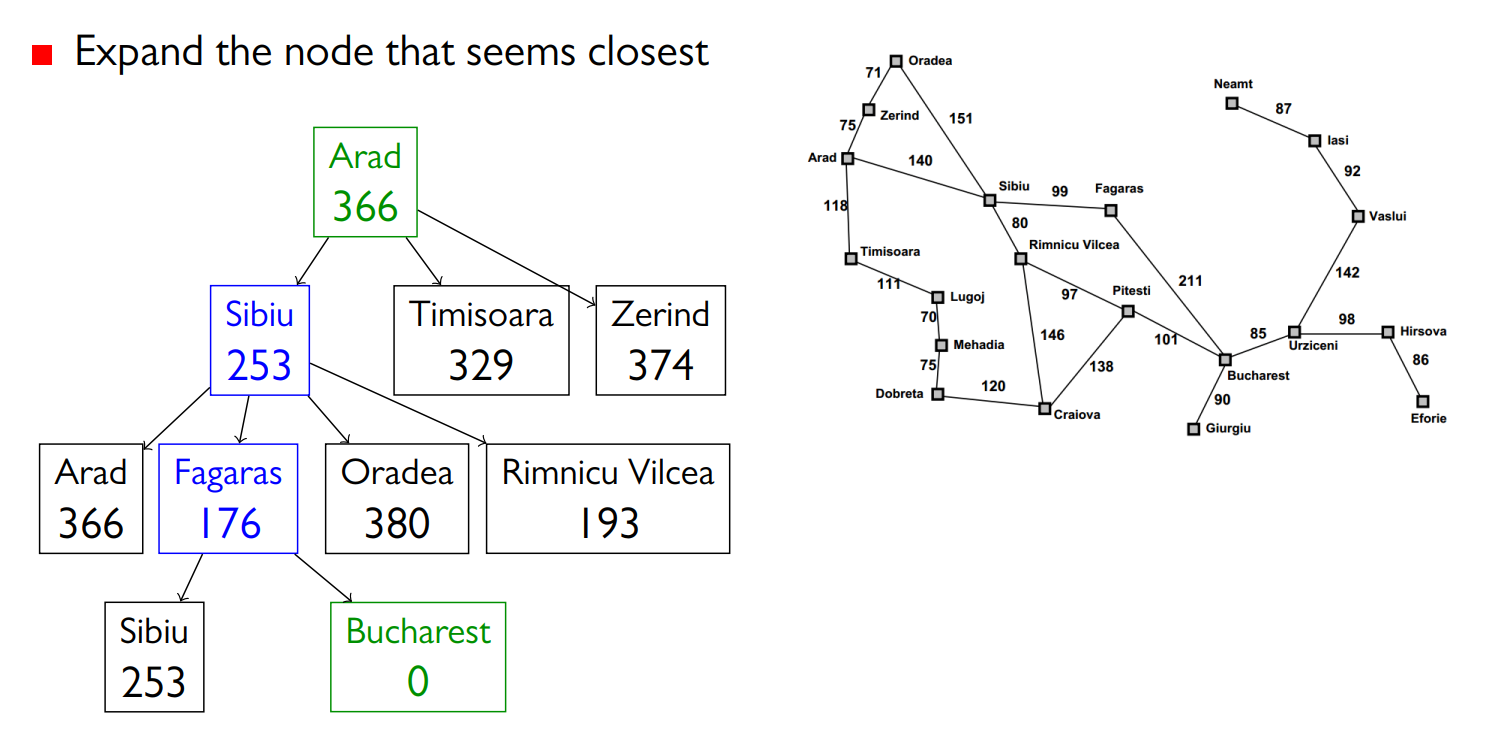
The best heuristic for this problem is the ‘id’ of the largest pancake still displaced. This is because we will have to flip at least that many pancakes. This value will more accurately estimate the cost, as shown below.



When choosing a heuristic, we also need to consider how expensive the heuristic itself is to calculate.

## Greedy Search

The simplest implementation of a heuristic is the **Greedy Search Algorithm**. We choose the next node to expand based on which node gives us the lowest heuristic. However, this algorithm is not always optimal.

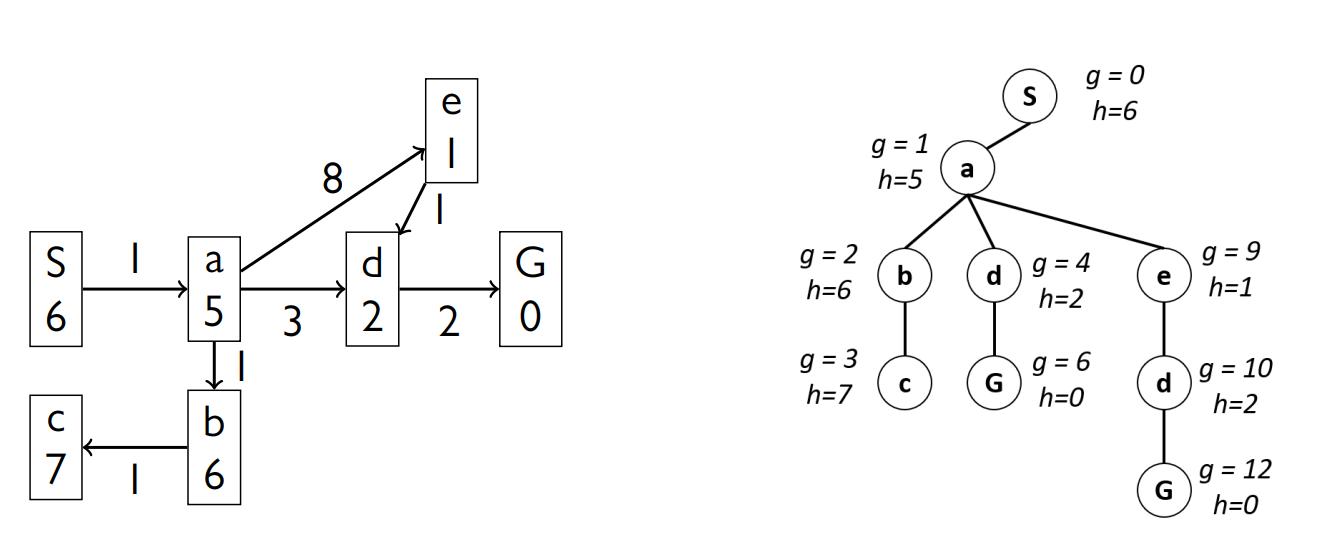


In the example above, the path with the lowest cost is actually Arad Sibiu Rimnicu Vilcea Pitesti Bucharest. Our algorithm follows the heuristic values and thus fails to find this path.

## A\* Search

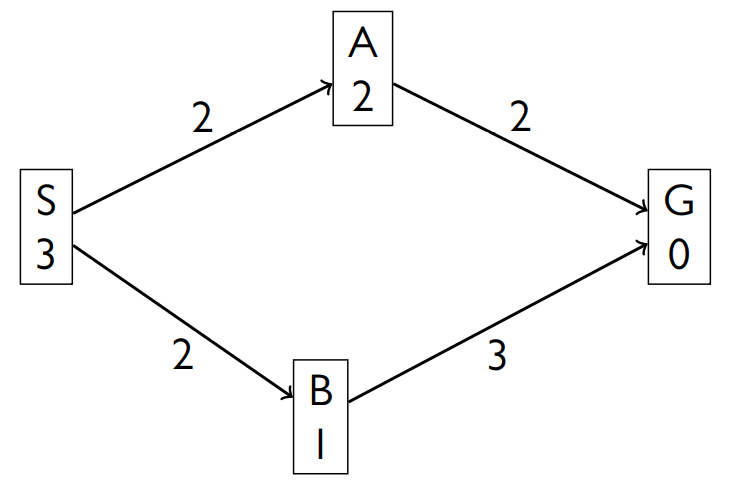
The **A\* Search** algorithm solves the problem seen with the Greedy Search algorithm. It is a mixture of UCS and the greedy search algorithm. It attempts to not expand as many nodes as UCS (thus achieving a faster runtime) while also ensuring optimality (unlike Greedy Search).

The actual cost of reaching the current node from the state node, as used by UCS, is called the **backwards cost**, . The heuristic value at , as used by Greedy Search, is called the **forwards cost**, . The cost used by A\* Search is given by



### Termination

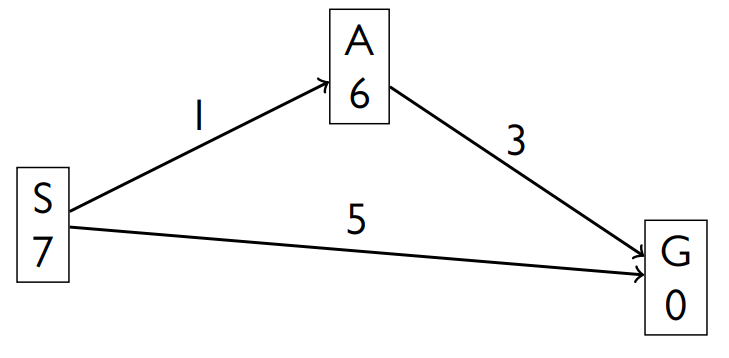
A\* Search, like all other search algorithms, should terminate when we **pop** an item and find that it is the goal state. It should not terminate if we find the goal state when exploring the neighbours of a node.



For the graph above, for and , so we explore first. From here, we find that is a neighbour. However, the path through from , while it is from . If we terminate as soon as we find when expanding , we will not find the optimal solution. If we wait and expand from first (since that gives a lower ), we will.

### Optimality

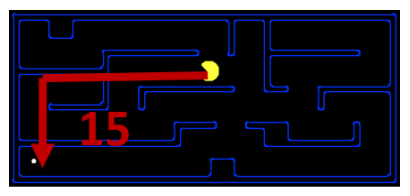
The A\* Search algorithm is **not always optimal**. Consider the graph below.



For this case, the best path is S A G, but the A\* algorithm will choose S G. This is because the heuristic values being used are too large. We need the heuristic to be optimistic or **admissible**, meaning that the value will always be less than or equal to the actual cost. Formally,

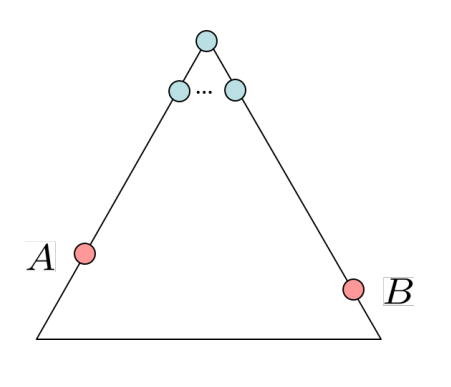
where is the true optimal cost.

For Pac-Man, we can have a situation like the one below.



The Manhattan distance metric is admissible since even if there were no walls, the true cost would be equal to this value. There is no way for the cost to be lower than this.

### Proving Optimality



Suppose we have two goal states, an optimal one , and a suboptimal one . Additionally, suppose we have an admissible heuristic . Our goal is to prove that will exit the fringe before does.

If and are both in the fringe at the same time, then we already know will exit the fringe first.

1. , since (A is a goal state)
2. , since (B is a goal state)

Since we have shown that is always smaller, will be picked first if it is in the fringe.

But what if we have a situation where is already in the fringe but is not. In that case, there will be some ancestor of that is in the fringe. This might even be the root node in the worst case. This situation is not practically possible, but for the sake of this proof, let us assume that we somehow ended up in this worst-case scenario. Suppose the ancestor of that is in the fringe is . Now we will prove that even .

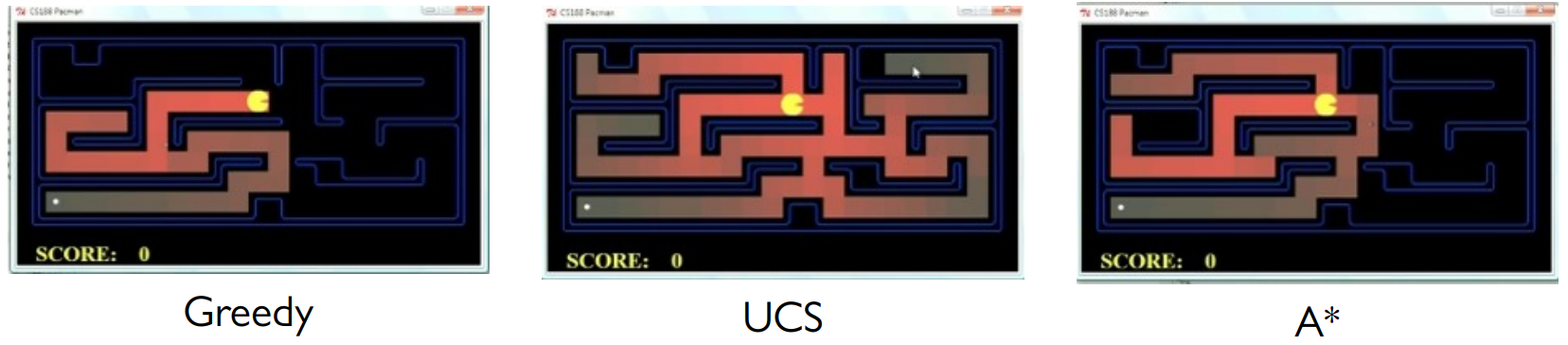
1. , since the sum of the backward cost at at the forward cost from cannot be larger than .

Thus, will be expanded before , since it has a lower cost. Since we did not specify which ancestor is, the above applies to all the ancestors. Thus, all ancestors of will be expanded before , meaning will enter the fringe before is expanded. As show above, once both are in the fringe, is expanded first.

The fun part to this proof is that if we consider , the A\* search algorithm becomes the uniform cost search algorithm. All of the above rules still apply if , which means this proof also works as the proof for the optimality of UCS.

## UCS vs A\*

Both UCS and A\* are optimal, given that the heuristic A\* uses is admissible. However, A\* finds the optimal solution faster.

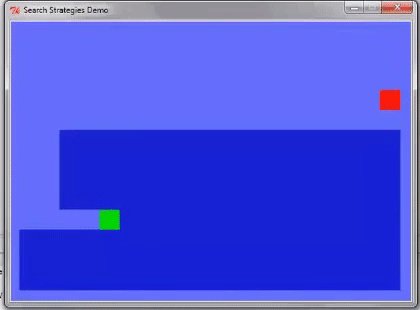


## A\* Applications

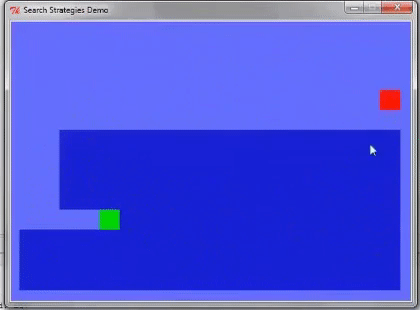
A\* Search is used in a variety of applications including video games, pathing/routing problems, resource planning problems, robot motion planning, language analysis, machine translation, speech recognition, etc.

## Comparison of Algorithms

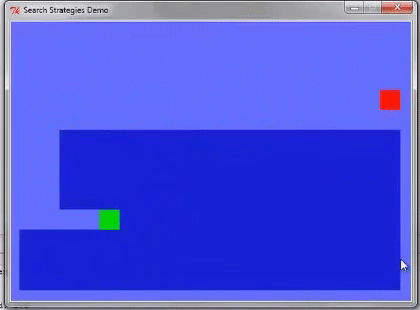
BFS:



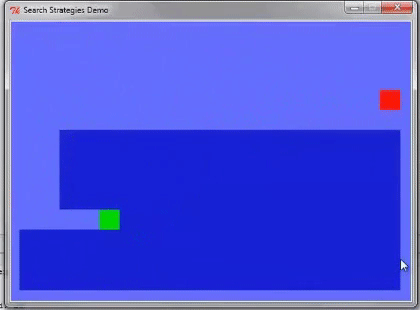
UCS:



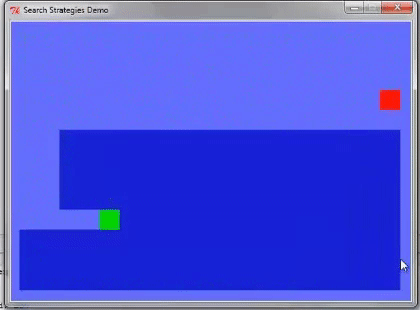
A\*:



DFS:



Greedy:

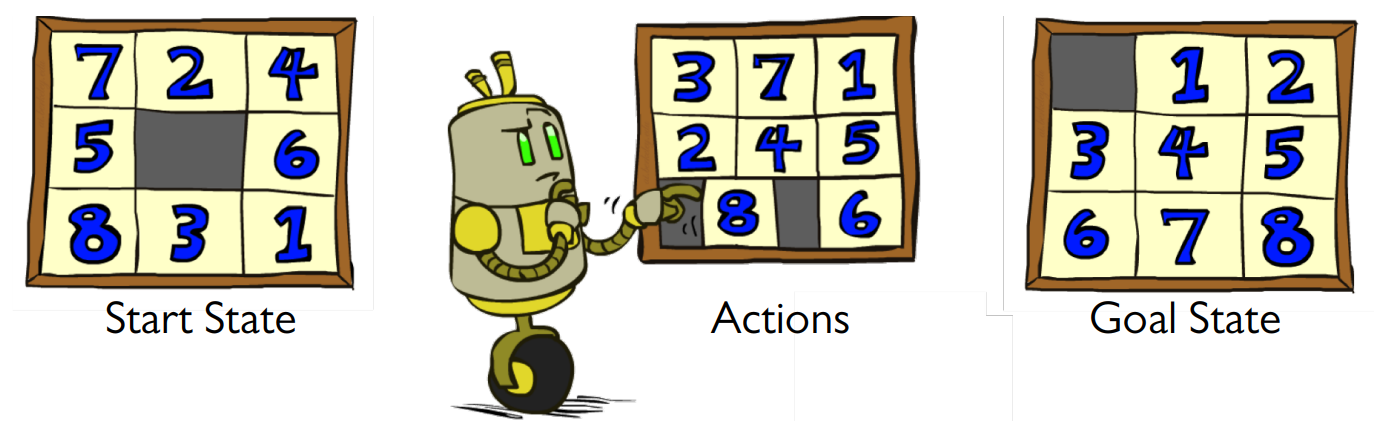


## Creating Admissible Heuristic

Most of the work in solving a hard search problem optimally is in coming up with admissible heuristics. Mostly, the heuristics solve **relaxed problems**, where new actions become available. For example, the Manhattan distance metric in Pac-Man assumes we can go through walls.

Sometimes, even inadmissible heuristics are useful.

Let’s try and create an admissible heuristic for the 8 Puzzle problem.



First, we have to **formalize** the problem:

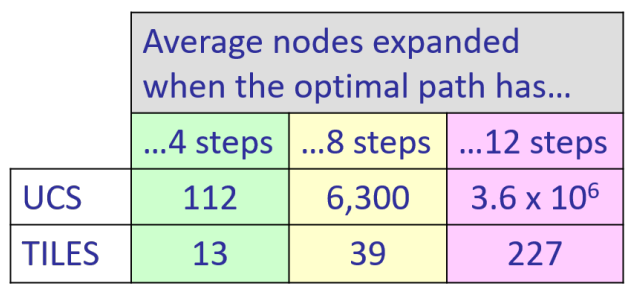
* The **states** are the different puzzle configurations
* The **number of states** is
* The **possible actions** are moving the empty piece in 4 directions. We are considering this instead of moving a number piece, since this is easier.
* The number of **successors** from each state is 4.
* The **cost** is the total number of moves required to reach the goal state.

Now let’s attempt a few heuristics.

Attempt 1:

One heuristic could be the **number of misplaced tiles**. Let this number be . This heuristic is admissible, since at least tiles must be moved to reach the goal state. For the given start state, .

Using this heuristic, the number of nodes that must be expanded by the UCS and A\* algorithms respectively for the cases where the optimal number of steps is 4, 8 and 12 respectively is provided below.



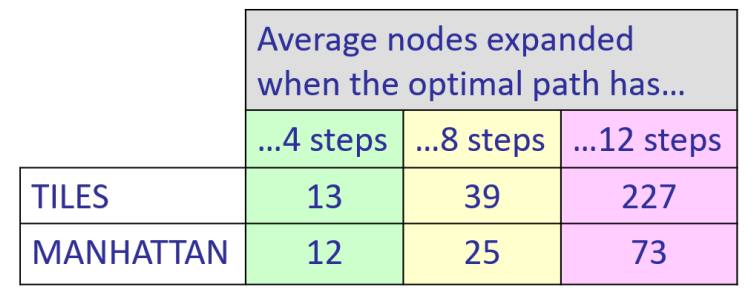
We can see that the A\* algorithm does a much better job, exponentially better as the number of optimal steps increases.

The heuristic relaxes the main problem by assuming that the tiles can be picked up and swapped with any location.

Attempt 2:

A second heuristic can be created by assuming that we can move in any direction, even if there are other tiles in the way. To get to the start state, we will thus need to calculate the sum of the **Manhattan distance** for every tile. For the given start state, . The movement restrictions we face in the real problem makes this heuristic admissible, since we will not be able to move in as few steps in reality.

The number of nodes expanded in this case by UCS and A\* are provided below.



As can be seen, this gives us better performance. This is because the heuristic is closer to the actual cost. Thus, if we can create a heuristic that is closer to the actual cost while still being admissible, we will achieve better performance.

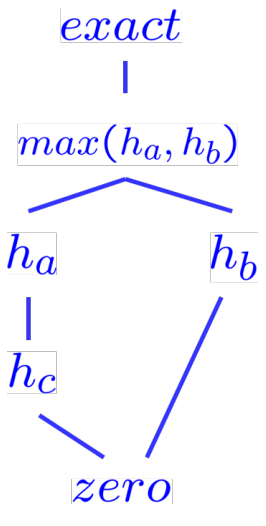
Attempt 3:

A third heuristic could be the actual cost. This is admissible and will actually give us better performance. The problem is if we want to calculate the heuristic, we have to actually solve the problem.

From this we can reach one more conclusion. When designing a heuristic, there is a **trade-off** between the quality of the heuristic and the work required to calculate the heuristic. The better the heuristic, the more work we need to do to calculate it.

### Semi-Lattice of Heuristics

Heuristics can be arranged based on how good they are.



At the bottom, we have the zero heuristic. This is technically a heuristic, but it is basically worthless since we end up with UCS. At the top, we have the exact heuristic, which we saw is not a usable heuristic. In between lies all the other heuristics.

If , , meaning the heuristic always does a better job than the heuristic , is said to **dominate** . On the other hand, if the heuristic does better in some states but the heuristic does better in some states, both are considered to be equally good. In this case, we can consider the maximum value of the two, , thus getting the best of both.

## Graph Search

In tree search, we repeat ourselves a lot by expanding the same notes more than once. **Graph Search** requires that we not expand the same node more than once, thus saving us this work.

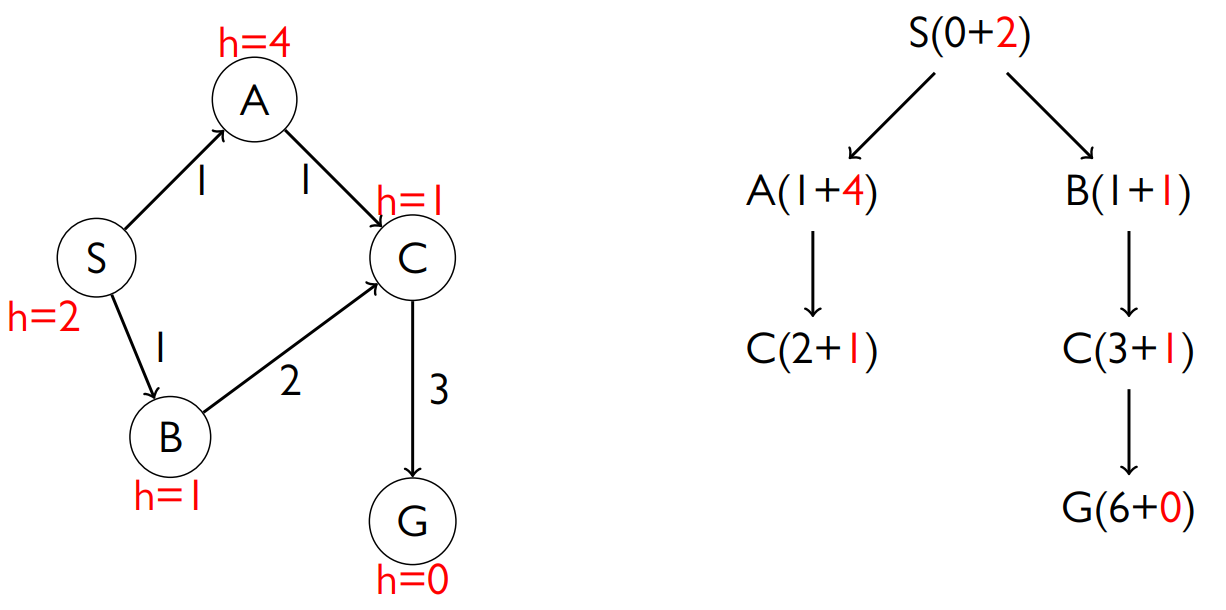


We implement this internally as though it were a tree search. The only change is that we also check if a node has been expanded earlier before expanding it. We keep track of previously expanded nodes using a set.

Graph search is **complete** since we check if a node is the goal node when we expand it. No node remains unexpanded, so we will definitely find a solution. However, depending on the algorithm, graph search may not be **optimal**.

### A\* Search

A\* Search is the best tree search algorithm we have seen so far. However, it does not guarantee optimality in graph search. Consider the diagram below.

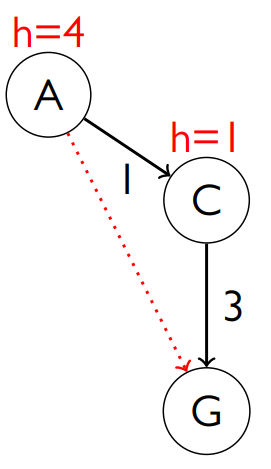


The heuristic used in this case is admissible, but it still fails to find the optimal path, which is . This is because the node is expanded once and not a second time.

If we only consider the **backward cost**, which would give us UCS, this will still be optimal. This is because UCS expands nodes from lowest to highest based on their real cost, which results in the optimal solution. However, A\* Search does not expand nodes in the lowest to highest order. This is what breaks the optimality in this case. Since the backward cost is working for UCS, the issue must lie in the heuristic we are using, and indeed it does.

### Consistency

**Admissibility** only requires that the heuristic from a node be less than the actual cost from that node to the goal state. This can create some absurd situations, such as the one below.



The heuristic values used implies that the cost from to is , even though the cost from to , a much larger distance, is . This is breaking a criteria called **consistency**. It requires that the **heuristic arc cost** be less than the **actual arc cost**, i.e., .

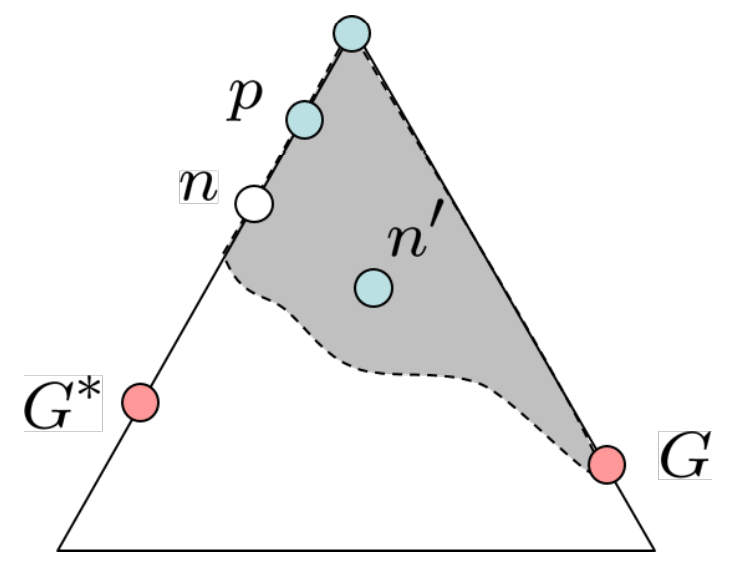
Based on this, if we can redefine the heuristic, the A\* search algorithm will become optimal. The consequence of using consistency is that the value along a path **never decreases**.

Since A\* search expands nodes based on increasing values, by incorporating consistency, this means nodes that reach some state optimally are expanded before nodes that reach suboptimally. Consistency also **implies admissibility**.

In general, most heuristics we make that are admissible also end up being consistent, especially when working with a relaxed version of the original problem.

### Proof of Optimality

Our main concern in this case is that we have an optimal solution and a suboptimal solution which are both reachable via some state . If we expand while going down the path to reaching , the suboptimal path, we will be unable to re-expand while doing down the path to reaching , the optimal path, when dealing with a graph search. We claim that consistency makes it impossible to use the suboptimal path, meaning that that will not be expanded for the first time when going down the path to . To make the equations less complicated, is denoted as when using the suboptimal path.



Suppose and have some ancestor . Thus,

[Due to consistency]

[Since and are the same state]

[Since is reached sub optimally]

Thus, is expanded before and . Once is expanded, we choose the descendent path which has a lower cost, which is .

## Pseudocode

### 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

for child-node in EXPAND(STATE[node], problem) do

fringe ← INSERT(child-node, fringe)

end

end

### Graph Search

function GRAPH-SEARCH(problem, fringe) return 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

for child-node in EXPAND(STATE[node], problem) do

fringe ← INSERT(child-node, fringe)

end

end