Lecture 3
Heuristic Search

Alice Gao

January 10, 2022

Contents

Modified and distributed by Blake VanBerlo, with permission from Alice Gao.

1 Learning Goals

By the end of the lecture, you should be able to

- Describe motivations for applying heuristic search algorithms.
- Trace the execution of and implement the Lowest-cost-first search, Greedy best-first search and A* search algorithm.
- Describe properties of the Lowest-cost-first, Greedy best-first and A* search algorithms.
- Design an admissible heuristic function for a search problem. Describe strategies for choosing among multiple heuristic functions.
- Describes strategies for pruning a search space.

2 Why use Heuristic Search?

So far, I've discussed several uninformed search algorithms: Depth-First Search, Breadth-First Search, and Iterative-Deepening Search. Is there anything unsatisfying about these algorithms? Let's look at the eight-puzzle again. Here are two states of the eight-puzzle. If both states are on the frontier, which one should we expand next?

5	3	
8	7	6
2	4	1

1	2	3
4	5	
7	8	6

For an uninformed search algorithm, the two states appear equivalent. The algorithm will decide on which state to expand based on an arbitrarily pre-defined order. In other words, the algorithm treats the states as black boxes. It doesn't know anything about the internal structure of the state. The only thing the algorithm can do is test whether the state is a goal state or not. That's about it.

What if a human instead of an algorithm is making the decision? A human would most likely expand the state on the right since the state is much closer to a goal. How did we figure out that? Intuitively, we will estimate how many moves it takes to transform this state to a goal state, and we're using this intuition to guide our search. The state on the right is one move away from the goal state. For the state on the left, it is unclear how many moves it requires to transform the left state to the goal state — definitely more than one move. We will use this intuition or domain knowledge to build a heuristic function and use the heuristic function to search more efficiently. This is the main idea behind heuristic search algorithms.

Let's compare the uninformed and heuristic search algorithms at a high level.

An uninformed search algorithm does not actively reason about the goal states. Every state appears equivalent to the algorithm. The algorithm is not guaranteed to find the optimal

solution because it does not consider the costs of the edges.

Heuristic search algorithms can do much better. This is because the algorithm actively reasons about the goal states. The heuristic function estimates the cost of the shortest path from the current state to a goal state by using domain knowledge about the problem. The estimate may not be accurate, but it is the best that we have. The accuracy of the heuristic estimates heavily influences the efficiency of the search algorithm. With the heuristic function, we can find the goal state much faster.

3 Overview

In this section, I will give you an overview of three related algorithms: lowest cost first search, greedy best-first search, and A* search.

Let's consider search problems where we have costs associated with the arcs in the search graph. The arcs represent actions, and every action has an associated cost. Let's assume that the costs are non-negative. Our goal is usually to find the optimal solution which is the path with the minimum cost.

The three algorithms are closely related. They make use of two sources of information: the cost function and the heuristic function.

The cost function for a state n is the actual cost of a path we have found from the start state to the state n. This is an evaluation of what happened in our past. This value is accurate since we already found the path.

In contrast, **the heuristic function** estimates our future. It takes a given state n and makes an educated guess about how far the state n is to the closest goal state.

Formally, given any state n, the heuristic value of the state n is an estimate of the cost of the cheapest/shortest path from this state to any goal state. There could be multiple paths from the current state to a goal state, we only care about estimating the cost of the cheapest one. What the heuristic function is doing reflects our goal as well, which is to find the optimal solution which is the cheapest path from the start node to any goal state.

In general, the heuristic function can be arbitrary. However, a useful heuristic function has the following properties.

- It's usually problem-specific, meaning that it's often based on some domain knowledge we have about a particular problem.
- It should be non-negative. It's an estimate of the total cost of a path, and we do not consider negative costs.
- If we're at the goal state already, the heuristic value should be zero. This makes sense as the cost of the cheapest path from a goal state to any goal state should be zero.
- This final property is important. The heuristic function must be easy to calculate. We should be able to calculate the heuristic value of a state without performing search. Think about it this way. The heuristic function is supposed to help us and make the problem easier, but search is an expensive procedure. If it's difficult to calculate the heuristic function and if we end up having to do search to calculate the heuristic value, what is the point of using the heuristic function in the first place?

Given the cost and the heuristic functions, we can now define the three algorithms. All three algorithms implement the frontier as a priority queue, but they differ by the order in which they remove path from the priority queue.

- Lowest-cost-first search uses the cost function only. It removes the path with the lowest cost from the frontier. In other words, it always selects the cheapest path found so far.

- Greedy best-first search uses the heuristic function only. It removes the path with the lowest heuristic value. In other words, greedy best-first search always selects the node that we believe is closest to a goal.
- A* search uses both the cost and the heuristic functions. A star removes the path with the lowest sum of the two function values.

Strictly speaking, lowest-cost-first search is an uninformed search algorithm since it doesn't use the heuristic function. Greedy best-first search and A* are heuristic search algorithms. I will discuss these algorithms in more detail in the next sections.

4 Lowest-Cost-First Search

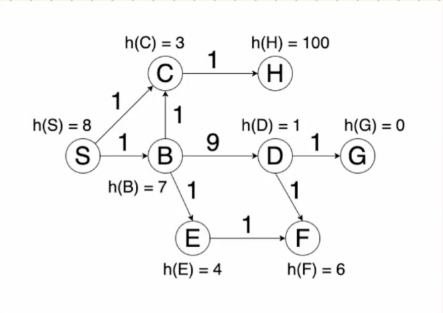
Technically, LCFS is an uninformed search algorithm because it doesn't make use of the heuristic function.

So far, the uninformed search algorithms focus on finding any solution. There is little guarantee on the quality of the solution found. DFS has no guarantee. BFS and IDS are guaranteed to find the solution with the fewest edges, but this path is not necessarily the one with the least total cost. If all the edges have the same costs, this solution is optimal.

What if our goal is to find the optimal solution? One way to do this is to keep track of the costs of the paths found so far. This idea takes us to the Lowest-Cost-First search algorithm. LCFS maintains a frontier, which is a priority queue ordered by the path costs. It always selects the path with the lowest total cost to remove from the frontier. You may have learned the Dijkstra's shortest path algorithm, which is similar to LCFS.

Example: Let's trace LCFS on a search graph.

If multiple paths have the same cost, we will order the paths by the last node and choose the one that comes first in alphabetical order.



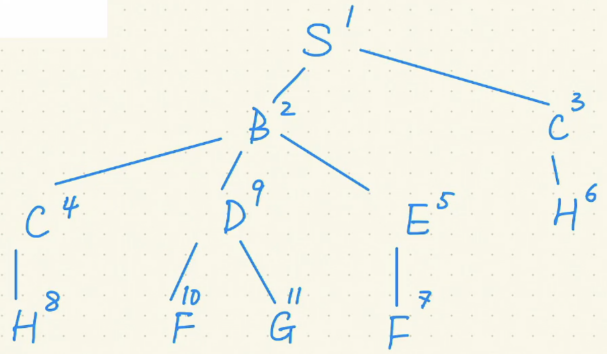
LCFS

frontier : $ SB SC SBC SBD SBE

cost : 0 1 1 2 10 2

SCH SBCH SBEF SBDF SBDG

2 3 3 11 11



4.1 Properties of LCFS

Let's look at the properties of LCFS.

The time and space complexities for LCFS are both exponential. LCFS generates all the

paths whose cost is less than the cost of the optimal solution. The number of such paths could be very large. It is not doing well with space and time because we are doing some intensive optimization to make sure that we find the optimal solution first.

As we start talking about more complex search algorithms using the cost and the heuristic functions, it becomes more challenging to characterize the time and space complexity. It is also less interesting to discuss these complexities because they tend to be exponential in the worst case.

Let's look at completeness and optimality. LCFS is complete and optimal under some mild conditions.

The first condition is that the branching factor must be finite. The algorithm may decide to expand all the children of one node before going deeper into the tree. Having a finite branching factor makes it possible for the algorithm to explore all the children of a node if necessary.

The second condition is that the cost of every edge is bounded below by some positive constant. In other words, the cost of an edge cannot be arbitrarily small. Without such bounds, there could be infinite paths with a finite cost. The Zeno's paradox devised by Aristotle illustrates the idea of this case. Suppose that we have edges with costs 1/2, 1/4, 1/8, 1/16, and etc. We can construct an infinitely many number of paths using these edges and the cost of every path is less than 1. The problem here is that: we would be exploring paths forever and never reach a path of cost of 1.

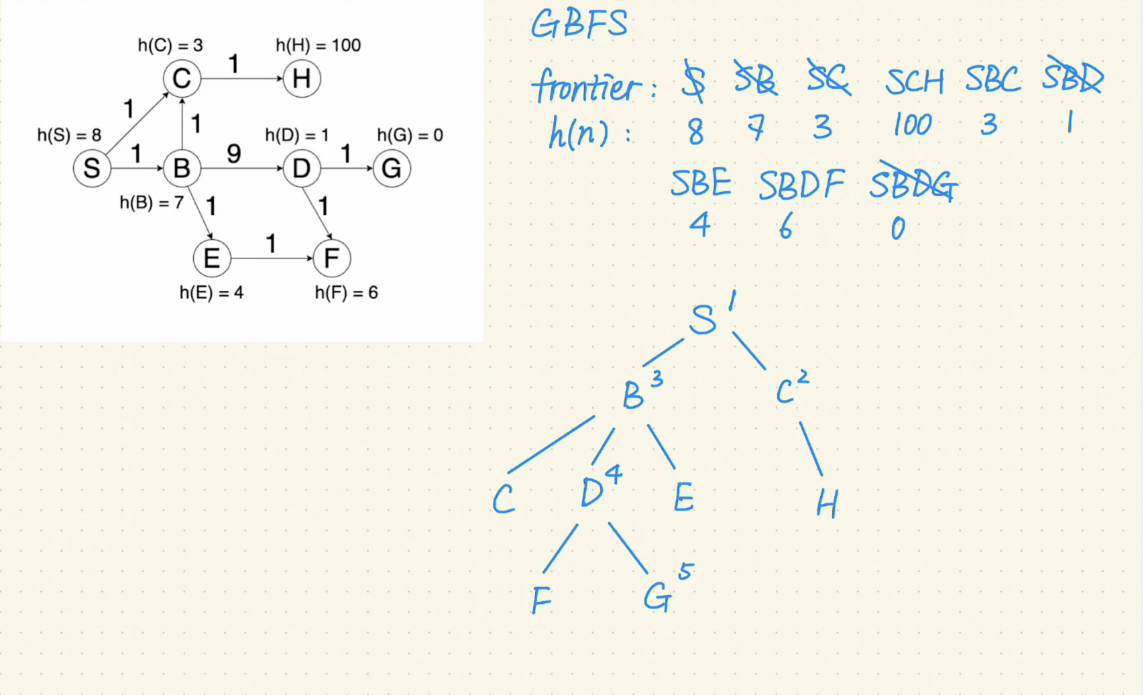
In general, LCFS is cautious and conservative. It looks at all of the potential paths so that it doesn't miss the optimal solution.

5 Greedy Best-First Search

In the previous section, I introduced the Lowest-Cost-First search algorithm. It uses the cost function to determine which state to explore next. Beside the cost function, the heuristic function is another source of information that a search algorithm can use. Greedy BFS makes use of the heuristic function. In fact, it relies on the heuristic function as the only source of information.

For Greedy BFS, the frontier is a priority queue ordered by the heuristic value. At every step, the algorithm removes the path with the smallest heuristic value. Intuitively, we are choosing the state that we think is closest to the goal according to our heuristic function.

Example: Let's trace Greedy BFS on a search graph. We will use the same tie-breaking rule as before. Order the paths by their last nodes and choose the one that comes first in alphabetical order.



Greedy BFS did pretty well on this problem because the heuristic values are quite accurate.

5.1 Properties of GBFS

Let me discuss the properties of Greedy BFS.

First, what are its space and time complexities? Unfortunately, both complexities are exponential. Intuitively, the heuristic function does not improve the worst case scenario. In the worst case, the heuristic function may be completely uninformative, for instance, having a

heuristic value of 0 for every state. Then, Greedy BFS is equivalent to an uninformed search algorithm and may have to search the entire space to find a goal.

What about completeness and optimality? Unfortunately, we have some bad news again — Greedy BFS is neither complete nor optimal.

Here is the intuition. Greedy BFS relies on the heuristic function. Unfortunately, we have absolutely no guarantee on the quality of the heuristic function. If the heuristic values are extremely inaccurate, they will cause the algorithm be stuck on paths that do not terminate or return a path that is not optimal.

Let me leave these two examples as practice problems for you. First, construct a search graph where Greedy BFS does not terminate on the search graph. Second, construct a search graph where Greedy BFS terminates but does not return the optimal solution. Basically, I am asking you to prove that Greedy BFS is not complete and nor optimal.

There are many correct examples you can come up with. Let me give you some hints.

For the "not complete" example, construct a graph with two paths. One path ends with a cycle and the other one ends with a goal state. Then, design the heuristic function such that Greedy BFS will go through the cycle forever.

For the "not optimal" example, again, construct a graph with two paths. Both paths start from the same initial state and end at the same goal state. Then, design the heuristic function such that the algorithm will find and return the sub-optimal path first. Try to make your examples as simple as possible. In my opinion, if you can come up with the simplest examples, it shows that you have really understood the algorithm.

6 A* Search

In the previous section, I talked about the greedy best-first search. It relies on the heuristic function to make decisions. There are lots of bad news with greedy best-first search: exponential space and time complexity, not guaranteed to find a solution and not guaranteed to find the optimal solution. A* will do much better than greedy best-first search since A* will take advantage of the cost information as well.

A* search implements the frontier as a priority queue, ordered by $f(n)$. n is the current node. $f(n)$ is the sum of $cost(n)$ and $h(n)$.

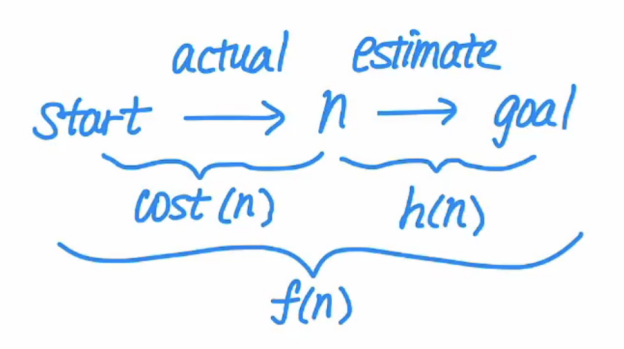
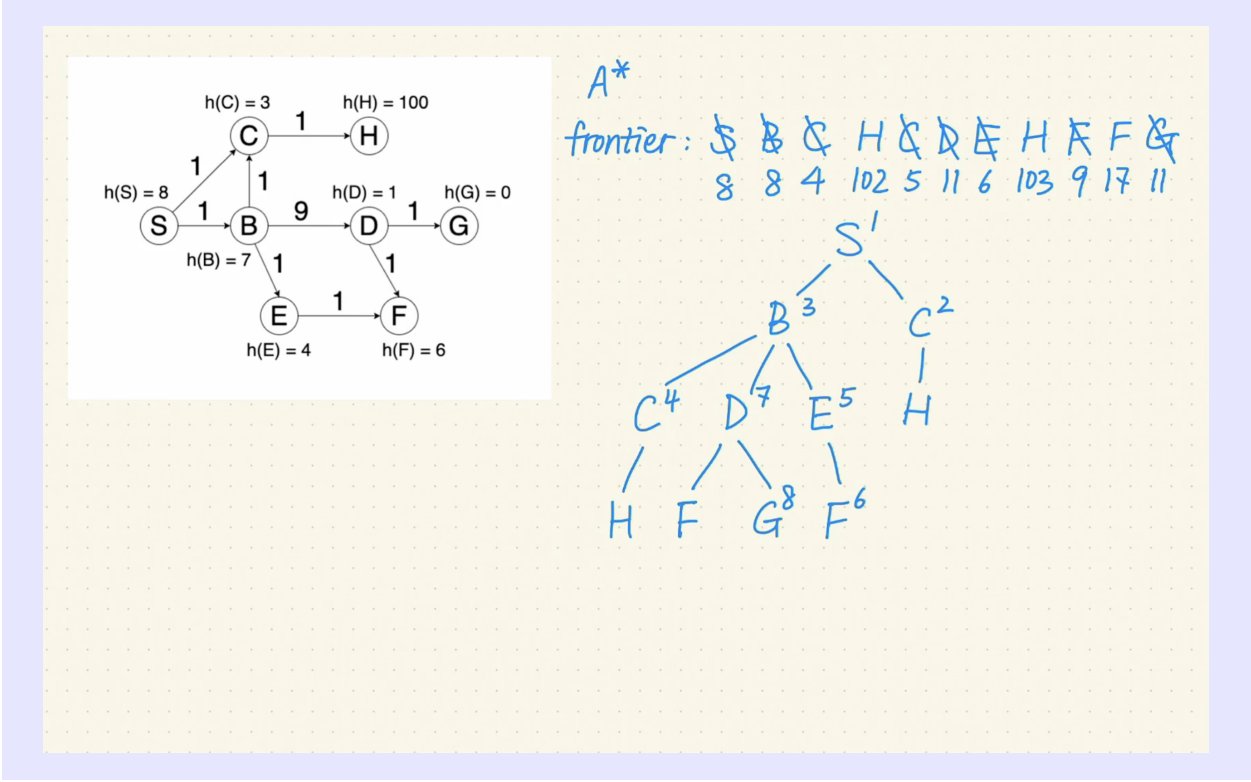


Figure 1: $f(n)$

Above is a picture to illustrate $f(n)$. Think of $f(n)$ as an estimate of the cost of the cheapest path from the start state to a goal state through the current state n. A* makes use of both the cost and the heuristic information. Intuitively, A* combines the ideas of lowest-cost-first search and greedy best-first search.

Example:

Let's trace A* search on this search graph. We will use the same tie breaking rule. Order the paths by their last nodes and choose the one that comes first in alphabetical order.



6.1 Properties of A* Search

Let's look at the properties of A* search. For complexity, A* doesn't do better than the other heuristic search algorithms. Space and time complexity are both exponential.

Why does using the heuristic function not improve the space and time complexities? Using the heuristic function does not improve theoretical guarantees since the guarantees only consider the worst case. The worst case often occurs when the heuristic is uninformative, for example, the heuristic function assigns the same value to every state. In practice, if the heuristic value is accurate, A* performs much better than other heuristic search algorithms.

Here are some good news. A* is complete and optimal as long as the heuristic function satisfies a mild condition — The heuristic function is admissible. Let's look at this condition in more detail.

6.1.1 A* is optimal

The heuristic function is admissible if and only if for any node n, $h(n)$ does not overestimate the cost of the cheapest path from the node n to a goal node. Let $h^*(n)$ be the cost of the cheapest path from n to a goal. Then, we must have that $0 <= h(n) <= h^*(n)$.

An admissible heuristic value for any node n is a lower bound on the cost of the cheapest path from the node n to a goal node. Intuitively, you can think of an admissible heuristic as an optimistic person. The person is optimistic because they think the goal is closer than it actually is. Whenever you ask this person to estimate how far a node n is to the nearest

goal, the person will always tell you a value, which is smaller than the actual cost of the cheapest path.

6.1.2 A* is optimally efficient

In fact, we can say something stronger about A*. In a sense, given a heuristic function, no search algorithm could do better! Formally, A* is optimally efficient. Among all the optimal algorithms that start from the start node and use the same heuristic function, A* expands the minimum number of paths.

7 Designing an Admissible Heuristic

Recall the A* algorithm. It has a nice property: if the heuristic function is admissible, then A* is guaranteed to find an optimal solution. This leads to an important question: how do we come up with an admissible heuristic function?

7.1 Two Heuristics for the 8-Puzzle

Let's look at the 8-puzzle again.

Initial State

5	3	
8	7	6
2	4	1

Goal State

1	2	3
4	5	6
7	8	

There are two well-known heuristics for the 8-puzzle.

The first one is **the Manhattan distance heuristic**. If you are in Manhattan, New York, how do you get from one location to another? You would either travel horizontally or vertically. On a grid, the Manhattan distance between two points is the sum of their horizontal distance and their vertical distance. For example, in the state on the left, the Manhattan distance between 5 and 4 is 3 because their vertical distance is 2, and their horizontal distance is 1.

How do we calculate the Manhattan distance heuristic? Given a particular state, consider every non-empty tile. Calculate the Manhattan distance between the current position of the tile and the goal position of the tile. Add this value for all the non-empty tiles together, and we have the heuristic value. Let's calculate the Manhattan distance heuristic value for the initial state on the left. Calculate this value yourself. Then, keep reading for the answer.

Let's calculate the heuristic value together. I have labeled goal position of each tile. For tile 1, the distance is 1, 2, 3, and 4. For tile 2, the distance is 1, 2, and 3. You can calculate the rest. The sum is 16. So the Manhattan distance heuristic value for this state is 16.

The second heuristic is called **the Misplaced tile heuristic**. For every non-empty tile, if the tile is not in its goal position, then we will add one to the heuristic value. In other words, we'll count the number of non-empty tiles that are not in their goal positions.

Let's calculate the Misplaced tile heuristic value for the state on the left. Calculate this value yourself. Then, keep reading for the answer.

Let's calculate the Misplaced tile heuristic value together. I've already written down the goal positions of the non-empty tiles. How many tiles are not in their goal positions? Six is the only tile in its goal position. Seven out of eight tiles are not in their goal positions. The misplaced tile heuristic value for this state is 7.

Both the Manhattan distance heuristic and the Misplaced tile heuristic are admissible. You will see why soon.

7.2 A Procedure for Constructing an Admissible Heuristic

How can we construct an admissible heuristic function? Here is a general procedure. First, take the original problem and relax it. Usually, the problem has some requirements, expressed as constraints. Take these constraints. Simplify them or remove some of the constraints. Once we relax the problem, we will solve the relaxed problem optimally. The cost of the optimal solution to the relaxed problem is an admissible heuristic function for the original problem. Recall that a heuristic function must be easy to compute. This means that we must relax the problem sufficiently such that the relaxed problem is easy to solve optimally.

7.3 Constructing Admissible Heuristics for the 8-Puzzle

Let's apply this procedure on the 8-puzzle to derive some admissible heuristic functions. How can we relax the 8-puzzle? To relax the problem, we first need to define the constraints formally. What are the rules when we move a tile from its current position A to another position B? There are two requirements. First, A and B must be adjacent. Second, the target position B must be empty. Let's relax one or both of these constraints and see what heuristic function we can derive.

Problem: Which heuristics can we derive from the following relaxed 8-puzzle problem?

A tile can move from square A to square B if A and B are adjacent.

(A) The Manhattan distance heuristic

(B) The Misplaced tile heuristic

(C) Another heuristic not described above

Solution:

In this question, we relaxed the 8-puzzle by removing one constraint. Originally, when we move a tile from square A to B, A and B must be adjacent and B must be empty. In this relaxed problem, the target square B does not need to be empty. The optimal solution to this relaxed problem gives us an admissible heuristic function for the 8-puzzle.

Let's solve this relaxed problem. Given any state of the relaxed problem, we need to determine the smallest number of moves to transform the state to a goal state. This number of moves is the heuristic value for the given state.

5	3	
8	7	6
2	4	1

Let's try an example first. Consider this state of the 8-puzzle above. What is the minimum number of moves we need to take to change it to a goal state? We need to move each tile to its goal position. This is much easier in this relaxed problem since the target square doesn't need to be empty. For example, I am allowed to move tile 1 up to overlap with tile 6.

For tile 1, its goal position is the top left square. We need four steps to move 1 to its goal position, for example, up, up, left, and left. It's fine for 1 to overlap with 6, 3 and 5 along the way since the target square doesn't have to be empty. Similarly, for tile 2, we need to move it up twice and move it to the right once. That's 3 moves in total. You can keep doing this calculation for all the other tiles and calculate the total number of moves that we need.

Let me ask you a question. For each tile that is not in its goal position, how many moves do we need to move it into its goal position? Since we are in a grid, the number of moves is equal to the Manhattan distance between its current position and its goal position.

For example, for tile 1, we need 4 moves, and that is the Manhattan distance between the bottom right square, which the current position of tile 1, and the top left square, which the goal position of tile 1.

For another example, consider tile 2. The current position of tile 2 is bottom left, and the goal position of tile 2 is top middle. So, we need 3 moves, which is the Manhattan distance between the bottom left position and the top middle position.

We are effectively calculating the total Manhattan distance between its current position and its goal position for each tile that is not in its goal position. Therefore, the cost of the optimal solution of the relaxed problem must be equal to the Manhattan distance heuristic.

The correct answer is A.

Problem: Which heuristics can we derive from the following relaxed 8-puzzle problem?

A tile can move from square A to square B.

(A) The Manhattan distance heuristic

(B) The Misplaced tile heuristic

(C) Another heuristic not described above

Solution:

5	3	
8	7	6
2	4	1

Consider one state of the 8-puzzle above. We need to determine the smallest number of moves to transform this state to a goal state. In other words, how many moves does it take to move all the tiles into their goal positions?

We have removed both constraints, which means that we can move a tile from one square to any other square instantly. In other words, a tile can fly from one square to any other square. This is great! This means that, if a tile is not in its goal position, we can move it to its goal position in one step. For this state, we need 7 moves, since all eight tiles except tile 6 are NOT in their goal positions.

What are we really computing here? The number of moves is equal to the number of tiles that are not in their goal positions. This number is equal to the value of the Misplaced Tile heuristic.

The correct answer is B.

Having worked through the two questions, you might be wondering about the third possibility. If we remove the first constraint and keep the second constraint, we can derive yet another admissible heuristic. This third admissible heuristic is called the Gaschnig's heuristic. This heuristic has some very interesting properties. I encourage you to explore its properties on your own.

7.4 Comparing Heuristic Functions

Given a search problem, we can often relax it in multiple ways and derive several different admissible heuristic functions. With multiple heuristic functions, which one should we choose? Let's look at a few ways of comparing heuristic functions.

First, we must require that the heuristic is admissible. This allows A* to find the optimal solution.

Next, we want the heuristic values to be as close to the true costs as possible. The closer the heuristic values are to the true costs, the more accurate the heuristic functions are.

Finally, a nice property is that the heuristic function produces different values for different states. This type of heuristic can help us distinguish different states and decide which state to expand next. On the other hand, if the heuristic function is constant, that is, it has the same value for all of the states, then that's not a very useful heuristic function.

These are intuitive rules we can use to compare heuristic functions. However, for some cases, they can be in conflict. For example, we may have two functions ha and hb where hb is closer to the true costs, but hb assigns very similar values to different states. In this case, should we apply rule 2 or rule 3? Next, I'll introduce a formal definition that we can use to compare different heuristic functions.

7.5 Dominating Heuristic

Let's look at a concept called the dominating heuristic. Given two heuristic functions h_1 and h_2, we can say that h_2 dominates h_1 if two conditions are satisfied.

Condition number one: for every state, h_2 produces a weakly higher value than h_1. Weakly means greater than or equal to, so the two values can be the same.

Condition number two: there exists at least one state where h_2 produces a strictly larger value than h_1. In other words, for every state, h_2 is weakly higher than h_1, and for at least one state, h_2 is strictly higher than h_1.

Definition (Dominating Heuristic)**.** Given heuristics $h_1(n)$ and $h_2(n)$, $h_2(n)$ dominates $h_1(n)$ if

- $(\forall n\ (h_2(n) \geq h_1(n)))$.
- $(\exists n\ (h_2(n) > h_1(n)))$.

This picture below can help you visualize the definition. Let h^* is the true cost. Suppose that h1 and h2 are both admissible. Then for any state, the values should be between zero and h*. If h_2 dominates h_1, then the h_2 value should be to the right of h_1 for every state.

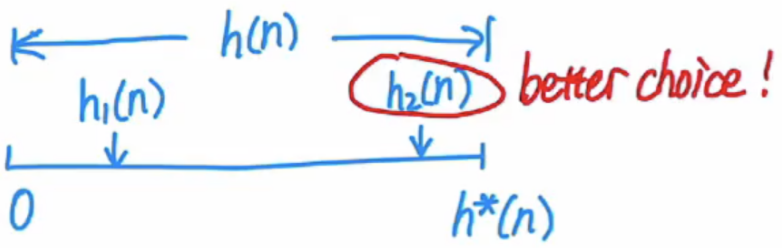


Figure 2: Dominating Heuristic

One consequence of this relationship is the following theorem: if h_2 dominates h_1, then running A* with h_2 will not expand more nodes than running A* with h_1. Therefore, if

we use h_2, A* will spend less time exploring the search graph and find an optimal solution faster.

If $h_2(n)$ dominates $h_1(n)$, A* using h_2 will never expand more nodes than A* using h_1.

7.6 Comparing Heuristics for 8-Puzzle

Let's apply the definition of dominating heuristic to the two heuristics we derived for the 8-puzzle: the Manhattan distance heuristic and the Misplaced tile heuristic. Based on dominating heuristic, which one is better? Does one heuristic dominate the other? Or neither dominates the other one?

Problem: Which of the two heuristics of the 8-puzzle is better?

(A) The Manhattan distance heuristic dominates the Misplaced tile heuristic.

(B) The Misplaced tile heuristic dominates the Manhattan distance heuristic.

(C) I don't know....

Solution:

The question asks us to compare two heuristic functions for the 8-puzzle using the concept of dominating heuristic. Does one dominate the other or does neither dominate the other one?

The correct answer is A. The Manhattan distance heuristic dominates the Misplaced tile heuristic.

Initial State

5	3	
8	7	6
2	4	1

To verify this relationship, we need to verify the two conditions in the definition of dominating heuristic.

Condition one: we need to show that, for every state, the Manhattan distance heuristic value is greater than or equal to the Misplaced Tile heuristic value. Let's consider the

state on the right as an example. Consider a tile that is not in its goal position.

Since the tile is not in its goal position, the Misplaced tile heuristic will add 1 to the heuristic value for this tile.

What about the Manhattan distance heuristic? The Manhattan distance heuristic will add at least 1 to the heuristic value for this tile. If its current position and its goal position are adjacent, The Manhattan distance heuristic will add exactly 1 to the heuristic value. Otherwise, the Manhattan distance heuristic will add more than 1 to the heuristic value. For example, for tile 3, the Manhattan distance heuristic adds 1 to the heuristic value. For tile 1, the Manhattan distance heuristic adds 4 to the heuristic value.

Therefore, for every state, the Manhattan distance heuristic value has to be greater than or equal to the Misplaced tile heuristic value.

Remember that it's possible for the two heuristic values to be the same. Consider the state on the left. Only one tile, tile 8, is out of place, and its current position and its goal position are adjacent. For this state, the values of both heuristic functions are one.

Next, we need to verify the second condition. We need to find a state such that the Manhattan distance heuristic produces a strictly higher value than the Misplaced tile heuristic.

We already have an example: the state above. The Manhattan distance heuristic value is 16, whereas the Misplaced tile heuristic value is 7. Therefore, there exists one state such that the Manhattan distance heuristic produces a strictly higher value than the Misplaced tile heuristic.

The correct answer is A. The Manhattan distance heuristic dominates the Misplaced tile heuristic.

8 Pruning the Search Space

8.1 Cycle Pruning

Let's look at cycle pruning.

What is cycle pruning? Cycle pruning means that, whenever we detect that we are following a cycle, we stop following the path and discard it.

There are several reasons for performing cycle pruning. First, a cycle may cause an algorithm to not terminate. For instance, DFS will get trapped in a cycle forever. Moreover, exploring a cycle is a waste of time since a cycle cannot be part of a solution.

How do we perform cycle pruning?

Let's take a look at the pseudo-code. The important part is in blue. When we generate a current node's successor, we will check whether the successor is on the current path. If the successor is on the current path, it is part of a cycle and we do not add the successor to the frontier.

Algorithm 1 Search w/ Cycle Pruning

```
1:  procedure SEARCH(Graph, Start node s, Goal test goal(n))
2:      frontier := {⟨s⟩};
3:      while frontier is not empty do
4:          select and remove path ⟨n_0, ..., n_k⟩ from frontier;
5:          if goal(n_k) then
6:              return ⟨n_0, ..., n_k⟩;
7:          for every neighbour n of n_k do
8:              if n ∉ ⟨n_0, ..., n_k⟩ then
9:                  add ⟨n_0, ..., n_k, n⟩ to frontier;
10:     return no solution
```

What is the complexity of cycle pruning?

To check whether the new node is in a cycle, we can check whether it's on the current path. In the worst case, we need to go through all the nodes on the path. This is going to take time that is linear in the length of the path.

Most search algorithms need to store multiple paths on the frontier. So linear time is the best that we can achieve for these algorithms. Breadth-first search is one example of such an algorithm.

However, for DFS, we can do better. DFS only remembers one path at a time. We can store all the nodes on the current path in a hash map. Checking whether a node is in the hash map requires constant time.

Another way to implement cycle pruning for DFS is to add a Boolean flag to each node. The flag is true if the node is on the current path and false otherwise. When we generate a new

node, the flag tells us whether the node is on the current path or not.

8.2 Multiple-Path Pruning

Next, let's look at the second pruning strategy: multi-path pruning.

Why do we want to use multi-path pruning? When solving a search problem, we only need to find one path to any node. Once we found a path to a node, we can discard all other paths to the same node — this is called multi-path pruning.

What is the relationship between cycle pruning and multi-path pruning? This is an interesting question. Think about this for a minute. Then, keep reading.

You may have realized that cycle pruning is a special case of multi-path pruning. Following a cycle is one way to have multiple paths to the same node. Pruning a cycle is one example of pruning multiple paths to the same node.

How do we perform multi-path pruning? Let's take a look at the pseudo-code. Our goal is to keep only one path to a given node at any time. We will create an explored set to store all the visited nodes.

Algorithm 2 Search w/ Multi-Path Pruning

```
1:  procedure SEARCH(Graph, Start node s, Goal test goal(n))
2:      frontier := {⟨s⟩};
3:      explored := {};
4:      while frontier is not empty do
5:          select and remove path ⟨n_0, ..., n_k⟩ from frontier;
6:          if n_k ∉ explored then
7:              add n_k to explored
8:              if goal(n_k) then
9:                  return ⟨n_0, ..., n_k⟩;
10:             for every neighbour n of n_k do
11:                 add ⟨n_0, ..., n_k, n⟩ to frontier;
12:     return no solution
```

After removing a path from the frontier, we will take n_k the last node on the path and check whether n_k is in the explored set or not. If n_k was visited, we will do nothing. Otherwise, we will add n_k to the explored set and expand the node — performing the goal test and adding its successors to the frontier.

In practice, the "explored" set can become extremely large. It often contains an exponential number of nodes. Because of this, it is crucial to ensure that the data structure allows for efficient look-ups. For example, a hashmap is a great choice.

8.3 A Problem with Multi-Path Pruning

So far, multi-path pruning seems like a great idea. It can make a search algorithm more efficient by allowing it to explore fewer states and find a goal more quickly. Unfortunately, multi-path pruning could cause a problem.

Multi-path pruning specifies that we should keep the first path found to any node and discard all other paths. What if the first path is not the one with the least cost? If this happens, multi-path pruning could cause the search algorithm to discard the optimal solution.

Let's consider two search algorithms: LCFS and A*, and answer two questions for each algorithm.

First, could multi-path pruning cause the algorithm to discard the optimal solution? If the answer is yes, we need to show an example. If the answer is no, we need to prove it.

Second, if multi-path pruning causes the algorithm to discard the optimal solution, what can we do about it? Can we modify the algorithm to ensure that it still finds the optimal solution even with multi-path pruning?

8.4 LCFS with multi-path pruning

Let's consider LCFS first. Can LCFS with multi-path pruning discard the optimal solution?

Problem:

Can LCFS with multi-path pruning discard the optimal solution?

If your answer is no, you need to explain why. If your answer is yes, you need to come up with an example of when this happens.

Solution:

The answer is: No. This is not possible. For LCFS, multi-path pruning will not discard the optimal solution.

Here is the intuition. LCFS is a cautious search strategy. It does not use heuristics. It only considers the actual cost of each path found. The algorithm always explores the path with the smallest cost first. For example, it will first explore all of the paths with a total cost of 1. After that, it will move on to all the paths with a total cost 2 and 3 and 4 and so on. If the algorithm always explores the paths in order of increasing costs, then it should never run into a case where it finds a longer path to a node first and then a shorter path to a node later on.

I have also included an informal proof by contradiction below. It starts by assuming that LCFS finds a longer path before finding a shorter path. Next, I show that this is

impossible based on how the algorithm selects which path to expand at each step.

Proof. Assume that LCFS first found path p to node n and then found path p' to node n. Assume that $cost(p) > cost(p')$.

When LCFS expanded path p, a portion of p' must be on the frontier. Let's call this portion p''. It must be that

$$cost(p'') \leq cost(p') < cost(p).$$

LCFS chose to expand p before p''. This contradicts its definition that LCFS always expands the path with the smallest cost first. □

8.5 A* search with multi-path pruning

Next, let's consider A* search.

Assume that we are running A* with an admissible heuristic function. Can A* with multi-path pruning discard the optimal solution?

Problem: Can A* search with multi-path pruning discard the optimal solution?

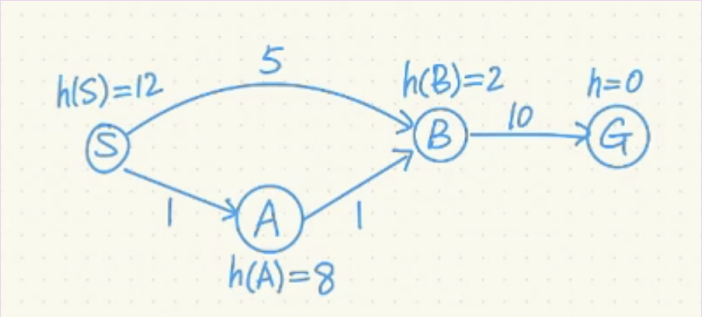
If your answer is no, please explain why. Give an informal proof. If your answer is yes, please come up with an example of when this happens. Try to make your search graph as simple as possible. If you can come up with a simple example, it means that you have understood the essence of this problem.

Solution:

Unfortunately, the answer is yes. We can construct a search graph for which A* with multi-path pruning will discard the optimal solution.

The correct answer is yes. To prove this, we need to come up with an example to show that this happens.

Let's look at an example. A student came up with this example and I modified it for this question.



S is a start node, and G is the goal node. There are two paths from S to G. The top path is SBG with a total cost of 15. The bottom path is SABG with a total cost of 12. The bottom path is the optimal solution. You can verify that the given heuristic is admissible.

Trace A* search on the search graph

Let's trace A* search on this search graph. You should expect that the algorithm finds and returns the top path, which is not optimal.

Add S to the frontier. Next, remove S from the frontier and add S to the explored set. S is not a goal. Let's expand it. S has two successors, A and B. Add SA and SB to the frontier. SA has a f value of 9. SB has a f value of 7.

Next, SB has the lower f value. Remove B from the frontier and add B to the explored set. B is not a goal. Let's expand it. B has one successor G. We will add SBG to the frontier. SBG has a f value of 15.

Next, SA has the smaller f value of 9. Remove A from the frontier and add A to the explored set. A is not a goal. Let's expand it. A has one successor B. Add SAB to the frontier with a f value of 4.

Next, SAB has the smaller f value of 4. Remove B from the frontier. B is already in the explored set. Multi-path pruning happens here and we will not add B's successors to the frontier. Let's continue.

Next, SBG is the only path on the frontier. Remove G from the frontier and add G to the explored set. G is a goal. Stop and return the solution SBG with a cost of 15.

This example shows you that A* search with multi-path pruning may return a sub-optimal solution first.

Reflecting on this example

Let's reflect on this example, what went wrong? After expanding S, we chose to expand B before expanding A since SB has a smaller f value. This causes us to find the top

path to B first, but this is a longer path to B. Later on, when we found a shorter path to B through A, we had to discard it due to multi-path pruning.

The real culprit is the heuristic value of A. Suppose that we use the heuristic values to estimate the cost of the path from A to B. This can be estimated as: $h(A) - h(B) = 8 - 2 = 6$. 6 is a gross over-estimate of the actual cost, which is 1. Because of this over-estimate, We assumed that the bottom path is longer and decided to explore the top path first.

I invite you to take this example and think about which components of the example are necessary for it to work? If you want to tweak this example, which components could you change and which components must stay the same?

8.6 Fixing A* Search with Multi-Path Pruning

We just learned some bad news: Multi-path pruning can cause A* to discard the optimal solution. How can we fix this? There are several strategies.

First strategy: We can find and discard all the longer paths to the node on the frontier.

However, discarding all the longer paths seems wasteful. Some of the longer paths may be quite close to a goal node. A better choice would be to replace the longer paths with the shorter ones that we just found. This is strategy #2.

So far, both strategies are quite computationally intensive. If the frontier is large, searching for a path on the frontier may take a long time. Can we avoid this work altogether? For the third strategy, we want to ensure that the algorithms find the shortest path to a node before finding any longer path. This is possible since we have control over one thing — the heuristic function. If we choose the heuristic function intelligently, then A* with multi-path pruning can be optimal.

Let's look at how we should choose the heuristic function to ensure that A* with multi-path pruning is optimal.

Recall that, if we don't perform multi-path pruning, the optimality of A* search requires the heuristic function to be admissible. We can define an admissible heuristic using this inequality. Consider any node m and any goal node g. The LHS is the heuristic estimate of the cost of the cheapest path from m to g. The extra term h(g) is not a problem since the heuristic value of a goal node should be zero. The RHS is the actual cost of the cheapest path from m to g. In other words, with an admissible heuristic, for every path to a goal node, the estimated cost of the path should be less than or equal to the actual cost of the path.

Definition. The heuristic function $h(S)$ is admissible if and only if for any node m and any goal node g,

$$h(m) - h(g) \leq cost(m, g).$$

Now, if we perform multi-path pruning, we need the heuristic function to satisfy a stronger condition. The heuristic needs to be consistent. A consistent heuristic requires that, for any two nodes m and n, the estimated cost of the cheapest path from m to n is less than or equal to the actual cost of the cheapest path from m to n. In other words, take the previous inequality for an admissible heuristic function and replace the second goal node g with any node n. The inequality must hold, not only when the second node is the goal node, but when the second node can be any node.

Definition. The heuristic function $h(S)$ is consistent if and only if for any two nodes m and n,

$$h(m) - h(n) \leq cost(m, n).$$

In practice, it is challenging to verify the definition of a consistent heuristic function directly, since we need to check every pair of nodes in the search graph. Fortunately, every consistent heuristic satisfies the monotone restriction, which is a simpler condition and easier to verify.

Definition. A heuristic function $h(S)$ satisfies the monotone restriction if and only if for any edge from m to n,

$$h(m) - h(n) \leq cost(m, n).$$

The monotone restriction requires that: For any edge from m to n, the same inequality must be satisfied.

The monotone restriction is simpler since we do not need to consider every pair of nodes. We only need to consider any pair of nodes that are directly connected by an edge.

If the heuristic is consistent, then A* search with multi-path pruning is optimal. The Poole and Mackworth book provides a proof of this. Check it out if you are interested.

8.7 Constructing a consistent heuristic

After this discussion, you might be wondering about a question: How can we come up with a consistent heuristic?

I am not aware of a general procedure to do this. Fortunately, most admissible heuristic functions are consistent. In fact, it's challenging to come up with a heuristic that is admissible but not consistent.

Therefore, it's often sufficient to construct an admissible heuristic and verify that it is also consistent using the monotone restriction.

9 Practice Problems

1. Construct a search graph and a heuristic function such that the heuristic function is not admissible and A* does not find the optimal solution on this search graph.
2. Prove that A* is optimal if the heuristic function is admissible.
3. Prove that A* is optimally efficient if the heuristic function is admissible.
4. Construct a search graph such that a solution exists at a finite depth in the search tree but GBFS does not terminate.
5. Construct a search graph such that a solution exists at a finite depth in the search tree, GBFS terminates but the first solution that GBFS returns is not optimal.
6. There are potentially two ways of performing pruning. (1) We can prune a state before adding it to the frontier. (2) We can prune a state after removing it from the frontier. If we use approach 1, we may discard the optimal solution when pruning. Construct a search graph and show that using approach 1 will cause us to discard the optimal solution.