

PATH FINDING STAR POWER

AN INTRODUCTION TO AND ANALYSIS OF THE A^* ALGORITHM
MATH 725 INDEPENDENT PROJECT REPORT

James Hurd
jameshurd.net
jmh@ku.edu

“If I had more time, I would have written a shorter letter.”
~Mark Twain

1. INTRODUCTION

When using graphs to model real-world problems, the shortest path between two vertices is often a problem of great interest. In particular, we are often interested in finding a path of *minimum weight* between vertices in a directed graph with weighted edges. In this report, we will explore “A Formal Basis for the Heuristic Determination of Minimum Cost Paths” by Hart, Nilsson, and Raphael [2] which presents A^* , an algorithm for solving this exact problem. Furthermore, the paper proves the correctness of the algorithm, and that a more optimal algorithm cannot be found. In addition to reviewing the paper, we will also give an implementation of A^* using the Python programming language and the `NetworkX` software package [1]. Now, without further ado, it’s time to find some paths!

2. PAPER SUMMARY

2.1. TERMS AND NOTATIONS

Before we embark on our journey together, it is important to agree up our objects of study and how we will talk about them. Throughout this report, let G be a directed graph on n vertices. We will denote an edge from a vertex i to j as e_{ij} . Though we have seen weighted graphs in class, let us refresh ourselves on the definition since it has been a while since we last encountered one.

Definition 2.1. G is *weighted* if each edge e_{ij} has an associated real number known as the *cost*, denoted c_{ij} .

For our purposes, G will always be weighted. In particular, G will be a δ graph.

Definition 2.2. For a weighted graph G , if there exists a real number $\delta > 0$ such that the cost of each edge is greater than or equal to δ we say G is a δ graph.

In essence, a δ graph is just a graph where there is a positive lower bound on the cost of all edges.

Instead of the usual explicit specification of a G (e.g. $G = (V, E)$) we will be using a rather unorthodox *implicit* specification of G which is composed of a set of source vertices $S \subset V(G)$ and a *successor operator* Γ defined on $V(G)$ which, for a given vertex v , returns a set of pairs (i, c_i) where each i is a neighbor of v reachable via an “in” edge e_{vi} , and c_i is the associated cost of that edge. Next, we can define what it means for a vertex to be *accessible* via another vertex.

Definition 2.3. For a vertex v in G , let the subgraph G_v be implicitly defined with v as the single source vertex and the usual Γ . Any vertex i in $V(G_v)$ is said to be *accessible* from v .

An intuitive way to think of G_v explicitly is simply the connected component of G containing v .

As stated earlier, our goal is to find a path on minimum cost between two vertices in G . We will define the *cost of a path* to be the sum of the costs of each edge in the path. A path from vertex i to vertex j is *optimal* if it minimum cost over all paths from i to j . We will denote this cost $h(i, j)$.

Henceforth, we will only concern ourselves with G_s derived from some *start vertex* s . Within G_s , let the set of vertices we hope to reach (some subset of $V(G_s) \setminus \{s\}$) be denoted as T and referred to as the

goal vertices. For some vertex v in $V(G_s)$ a given goal vertex t is a *preferred goal vertex* of v if $h(v, t)$ has minimum cost among all goal vertices. We denote this cost as simply $h(v)$.

Now we have the language to state our goal a bit more precisely. We would like to create an admissible algorithm that searches G_s to find an optimal path from some start vertex s to one of its preferred goal vertices. We say an algorithm is *admissible* if it is guaranteed to find such a path for any δ graph. Such an algorithm will work by repetitive application of Γ to generate parts of G_s on-demand. If Γ is applied to a vertex, we say the algorithm has *expanded* that vertex.

2.2. INTRODUCING... A^*

With that boatload of notation and setup out of the way, we are finally ready to discuss the star of the show. We would like our algorithm to expand the least number of vertices possible when searching of an optimal path. Ideally, we would *only* expand vertices along an optimal path, otherwise we'd be wasting computation by barking up the wrong tree. To help us with this, we will define an *evaluation function* \hat{f} on the vertices of G_s such that the vertex with the smallest value of \hat{f} is the vertex that should be expanded next. For now, consider \hat{f} as a black box (proof by trust, if you will), as it is easier to discuss the specifics of it after we have defined our algorithm.

With the evaluation function in mind, we have all the tools to define the algorithm A^* . Note that in the following pseudocode, prefixing a line with $\#$ denotes a comment.

Algorithm 2.1 (Hart, Nilsson, and Raphael).

define $A^*(s, T)$

```

    Mark  $s$  as "open" and calculate  $\hat{f}(s)$ 
    # This is an array indexed by the vertices of the graph that represents the predecessor to each
      vertex in an optimal path.
    let  $pred = []$ 
    # while True will do an infinite loop.
    while True
      let  $n =$  "open" vertex whose value of  $\hat{f}$  is minimal. Resolve ties arbitrarily but always in
        favor of a vertices in  $T$ .
      Mark  $n$  as "closed".
      if  $n \in T$ 
        | Reconstruct an optimal path using  $pred$  starting with  $n$  and terminate.
      let  $succs = \Gamma(n)$ 
      for  $v \in succs$ 
        |  $pred[v] = n$ 
        | Calculate  $\hat{f}(v)$ .
        | if  $v$  is not marked "closed" or  $\hat{f}(v)$  is smaller than before
          | # Since  $\Gamma$  returns a vertex, cost pair, we must access only the first element to get the
            vertex.
          | Mark  $v[0]$  as "open".

```

2.3. THE EVALUATION FUNCTION

That algorithm is great and all, but we are missing one major component which is the heart of A^* , we need to explicitly define the evaluation function \hat{f} . Let $f(v)$ be the *actual* cost of an optimal path constrained to go through a vertex v from s to a preferred goal vertex of v .

We can observe that $f(s) = h(s)$ by definition. It is also easy to see that $f(s) = f(v)$ for every vertex v on an optimal path to a preferred goal vertex of s , and $f(v) > f(s)$ for every v not on such a path. We can decompose $f(v)$ into the sum of two parts:

$$f(v) = g(v) + h(v) \tag{1}$$

where $g(v)$ is the actual cost of an optimal path from s to v , and $h(v)$ is the same as before.

Obviously, computing f for any given vertex is effectively the same as solving the problem at hand, however we can use \hat{f} to *estimate* f . Specifically, we will take $f(\hat{v})$ to be:

$$\hat{f}(v) = \hat{g}(v) + \hat{h}(v) \quad (2)$$

where $\hat{g}(v)$ and $\hat{h}(v)$ estimate $g(v)$ and $h(v)$ respectively. For $\hat{g}(v)$, we can use the smallest cost path from s to v that the algorithm has found so far. This implies $\hat{g}(v) \geq g(v)$, which is exactly what we want for an estimate, as we will never be misled by choosing a path which has a larger cost than we thought. In a sense, this means we will only “overshoot” the total cost of the path.

Choosing a method to compute $\hat{h}(v)$ is a bit trickier, as it is a heuristic function that will depend on the problem we are modeling with our graph. Specifically, we will calculate $\hat{h}(v)$ using information from the problem domain, with the only constraint that $\hat{h}(v) \leq h(v)$ for every vertex v in $V(G_s)$. That way, the heuristic never overestimates the cost of a particular path which would give the potential for A^* to return an incorrect result.

A trivial example for a choice of \hat{h} would be that $\hat{h}(v) = 0$ for all vertices v in $V(G_s)$. Since G is a δ graph, this is a lower bound on $h(v)$ for every v . However, this is not a very helpful choice as it gives no indication of the cost from v to a preferred goal vertex, relying only on information we already know from our search.

To show an example of a better choice of \hat{h} , assume G is modeling cities connected via highways, where the cost of the edge e_{ij} is the distance of the highway connecting city i to city j . In this case, a good choice for $\hat{h}(v)$ would be the distance to travel from city v to our goal via private jet. We can call this the “Taylor Swift distance” (see [4]). Since jets don’t have to worry about following laid out roads and can instead travel in a straight line (the shortest path between two points in Euclidean space) to their destination, the “Taylor Swift distance” will clearly be a lower bound on the actual distance traveled via highways. We shall revisit \hat{h} later, as there is much more we can say about this function.

2.4. A^* IS ADMISSIBLE

We will now prove the admissibility of A^* . Before we dive in however, we will need a lemma to help us out.

Lemma 2.1. *For any nonclosed vertex v and for any optimal path P from s to v , there exists an open vertex v' on P such that $\hat{g}(v') = g(v')$.*

Imagine that A^* is forming an optimal path from s to v . This lemma is essentially saying that *sometime* in forming this path, there will be a vertex v we can choose for which $\hat{g}(v)$ is exactly the value of $g(v)$. This lemma also allows us to state the following,

Corollary 2.1. *Suppose $\hat{h}(v) \leq h(v)$ for all v and suppose A^* has not terminated. Then, for any optimal path P from s to any preferred goal vertex of s , there exists an open vertex v' on P with $\hat{f}(v') \leq f(s)$.*

In less formal terms, if the heuristic function is a lower bound on the actual cost of an optimal path from each vertex to a preferred goal vertex, then $\hat{f}(v)$ serves as a lower bound for $f(v)$ for all v on P . However, $f(v) = f(s)$ for all v on an optimal path, so the corollary follows.

Now, without getting any more sidetracked, we can prove the following:

Theorem 2.1. *If $\hat{h}(v) \leq h(v)$ for all v , then A^* is admissible.*

Proof. The proof will be by contradiction. Assume that A^* does not terminate by finding an optimal path from s to a preferred goal vertex of s . There are three cases we must examine:

- **Case I:** A^* does not terminate at a goal vertex.

The algorithm only terminates when at a goal vertex as can be seen in algorithm 2.1 due to the only return statement being guarded with the condition $n \in T$. Therefore, this case can never occur.

- **Case II:** A^* does not terminate.

Let t be a preferred goal vertex of s accessible within a finite number of steps along an optimal path with cost $f(s)$. Recall that A^* operates on δ graphs, therefore $f(s)$ is at least δM where M is

the number of steps the algorithm takes. Therefore, for any vertex v farther than M steps away from s , we can say

$$\hat{f}(v) \geq \hat{g}(v) \geq g(v) > f(s) \geq \delta M \quad (3)$$

following from the definition of each function.

Now, consider a specific v further than M steps from s . This vertex will never be expanded by A^* , as corollary 2.1 states there will always be some open vertex v' on an optimal path from s to t with $\hat{f}(v') \leq f(s)$. Since $f(s) < \hat{f}(v)$ (as seen in (3)), the algorithm will expand v' instead of v .

With those pesky vertices more than M steps away from s out of the way, we can focus on all vertices accessible within M steps of s . Let the set of said vertices be $\chi(M)$. The only way A^* could not terminate at this point is the continued reopening of vertices in $\chi(M)$ (see the last `if` statement in algorithm 2.1).

Clearly, any vertex v in $\chi(M)$ can only be reopened a finite number of times, as there is a finite number of paths from s to t including only vertices in $\chi(M)$ that pass through v . Let $\rho(M)$ be the maximum number of times a particular vertex in $\chi(M)$ is reopened. Since the size of $\chi(M)$ is finite, the maximum number of times vertices in $\chi(M)$ can be expanded is $|\chi(M)|\rho(M)$, this follows from the multiplication principle of counting.

Since no vertices outside of $\chi(M)$ can be expanded, and the vertices in $\chi(M)$ will only be expanded a finite number of times, A^* must terminate.

- **Case III:** A^* terminates with a path of non-minimum cost.

Suppose A^* terminates at some goal vertex t on a non-minimum cost path. Since this path is non-minimum cost, $\hat{f}(t) > f(s)$. However, by corollary 2.1 just before termination there existed an open vertex v' on an optimal path where

$$\hat{f}(v') \leq f(s) < \hat{f}(t).$$

Since $\hat{f}(v') < \hat{f}(t)$, A^* would have selected v' as opposed to t . Therefore, A^* will never terminate with a non-minimum cost path. □

2.5. A^* IS OPTIMAL

Before discussing the optimality of A^* , we must say a bit more about the about \hat{h} . Specifically, we will place another restriction on \hat{h} called the *consistency assumption*. For any two vertices v and w :

$$h(v, w) + \hat{h}(w) \geq \hat{h}(v) \quad (4)$$

Effectively, this can be interpreted as saying that the estimate of the distance to some vertex v cannot be any better than using the estimate to some other vertex combined w combined with the actual cost of an optimal path from w to v . In other words. . . there is not point in trying to be clever and you should just use $\hat{h}(v)$.


Generally, the consistency assumption will be satisfied for most choices of \hat{h} . The main places we need to be on the look out for it not being satisfied is when \hat{h} has some parameter that varies independently between vertices (e.g. a random variable).

The consistency assumption allows us to say something interesting about \hat{f} as a whole.

Lemma 2.2. *If the consistency assumption is satisfied, then $\hat{g}(v) = g(v)$ for any vertex v closed by A^* .*

As a consequence of this, A^* never needs to reopen a closed vertex. This is because if we combine the facts that $\hat{g}(v) = g(v)$, $h(v, w) + \hat{h}(w) \geq \hat{h}(v)$ and $\hat{h}(v) \leq h(v)$ we can conclude that if v is expanded, the optimal path to it has already been found. This means we can get rid of part of that pesky final `if` statement provided the consistency assumption is satisfied.

Now, onwards to optimality! Let us denote the set of all subgraphs generated from repeated applications of Γ starting with some vertex v as $\{G_{v,\theta}\}$ where θ indexes each subgraph and is in some index set Θ .



Definition 2.4. Let Θ_v^A be the index set used by some algorithm A at vertex v , and let $\Theta_n^{A^*}$ be the corresponding set for A^* . If $\Theta_n^{A^*} \subset \Theta_n^A$ we say that A is *no more informed* than A^*

In a sense, this gives us a quantitative way to say that A uses no more information about the problem than A^* . Using this, we can state a theorem about *how* exactly A^* is optimal compared to other algorithms.

Theorem 2.2. *Let \mathcal{A}^* be the set of all algorithms for which the consistency assumption is satisfied that act identically to A^* except that they resolve ties differently. Let A be an admissible algorithm no more informed than the algorithms in \mathcal{A}^* . Then for any δ graph G_s , there exists A^* in \mathcal{A}^* such that every vertex expanded by A is also expanded by A^* .*

To be formal and precise, it is necessary to generalize A^* to a *set* of algorithms for stating theorems and proving things where ties matter, but for the purposes of our discussion we can choose our favorite A^* in \mathcal{A}^* .

The above theorem states that A^* is optimal in the sense that it expands the fewest number of vertices required to find an optimal path among algorithms no more informed than it.

3. IMPLEMENTATION

As alluded to earlier, we can give an implementation of A^* using the Python programming language and the `NetworkX` software package. All of the source code for the implementation along with the source files for this paper can be found at [3].

`NetworkX` has a class for directed graphs that we can use. We can attach a weight to each edge, and a `heuristic` field that we can use to attach a value of \hat{h} to each vertex. In terms of choosing a vertex for which \hat{f} is minimal, we can sort the open vertices using a priority queue data structure where vertices with smaller values of \hat{f} are further forward in the queue. For storing the predecessor of each vertex, a dictionary data structure can be used.

Minimal concern is given to tie breaking in this implementation as we are only operating on graphs which will expand few vertices in the case of a tie. If we wanted to implement the algorithm exactly, a custom implementation of a priority queue which breaks ties correctly may be required.

The implementation at [3] also has the capability of producing animations of the algorithm being run, but we will exclude that piece from our discussion here for the sake of brevity.

4. CONCLUSION

With that, we have reached the end our journey together through [2]! Just to recap, we sought to develop an algorithm that would find a minimum-cost path through a directed, weighted graph. Using information from the problem domain known as the *heuristic*, the A^* algorithm is not only correct but the best algorithm possible unless you use more information than the chosen heuristic exudes. We also reviewed some of the design choices that went into an implementation of A^* using the Python programming language and `NetworkX` software package.

REFERENCES

- [1] Aric Hagberg, Pieter J. Swart, and Daniel A. Schult. “Exploring network structure, dynamics, and function using `NetworkX`”. In: (Jan. 2008). URL: <https://www.osti.gov/biblio/960616>.
- [2] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.
- [3] James Hurd. *Star Power*. URL: <https://github.com/alexacallmebaka/star-power>.
- [4] Isabella O’Malley. “Why Taylor Swifts globe-trotting in private jets is getting scrutinized”. In: *The Associated Press* (Feb. 2024). URL: <https://apnews.com/article/taylor-swift-climate-jet-carbon-emissions-kelce-chiefs-02ac425d24281bd26d73bfdf4590bc82>.