Minimum Spanning Trees and Prim's Algorithm

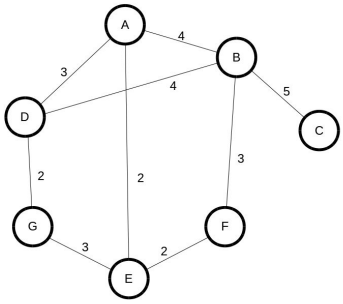
Robin Dawes

February 27, 2021

Gives us a chance to exercise our data structure muscles on a significant problem: finding the minimum weight spanning tree of an edge-weighted graph. We look at a famous algorithm for this problem and compare four - yes four! - implementations.

.

Graphs With Weighted Edges

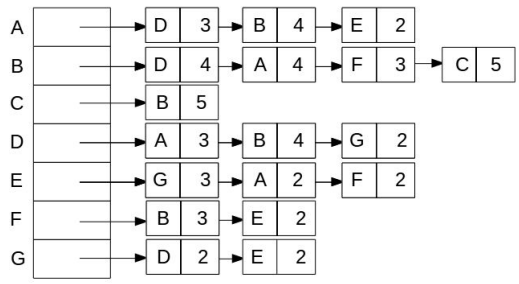


Here is the same graph as before, now with weights added to the edges. We can modify the adjacency matrix and adjacency list representations to incorporate this information.

The adjacency matrix looks like this. Notice that since the graph is undirected the matrix is reflected around its main diagonal.

	A	B	C	D	E	F	G
A	0	4	0	3	2	0	0
B	4	0	5	4	0	3	0
C	0	5	0	0	0	0	0
D	3	4	0	0	0	0	2
E	2	0	0	0	0	2	3
F	0	3	0	0	2	0	0
G	0	0	0	2	3	0	0

Here is the Adjacency List representation of the weighted graph. Observe that the edge-weights are stored in the list elements.



Minimum (Weight) Spanning Trees

DEFINITION: Let G be a graph with weights on the edges. We define the WEIGHT of any subgraph of G to be the sum of the weights of the edges in the subgraph.

One of the classic problems in graph theory can be stated in this way: given a connected graph G with non-negative weights on the edges, find a subgraph T of G such that:

1. T contains all vertices of G
2. T is connected
3. the weight of T is $\leq$ the weight of every subgraph that satisfies requirements 1 and 2

It is not hard to see that T must be a tree - if T contains a cycle we can reduce its weight by eliminating one edge of the cycle. This explains the choice of "T" as the name of the subgraph we are looking for.

So our goal here is to find a minimum weight spanning tree ... and because humans love shortcuts, we always leave out the word "weight" and just call this the Minimum Spanning Tree problem ... which we abbreviate even further to "the MST problem".

We have already seen two algorithms that find spanning trees of a graph: Breadth-First Search and Depth-First Search. These both ignore weights on the edges and so they obviously can't be counted on to find a minimum spanning tree ... but it might seem plausible that we could somehow modify them to solve this problem.

For example, in BFS we add neighbours of the current vertex to the queue in a random order ... what if we added them in order of the weight of the edges that join them to the current vertex? Similarly, in DFS we might decide to always choose the neighbour of the current vertex that has the lowest-weight edge from the current vertex. Try to find graphs where these modifications fail to find a MST.

Those suggestions may have got you thinking about some of the other data structures we have studied. For example, is there some way that BFS or DFS could put vertices into a priority queue so that we we choose the next vertex in such a way that we do get a MST? Well ... hold that thought!

There are many algorithms that solve the MST problem - we will look at two of the most famous ones. We'll spend a lot more time on the first one because once you understand that, the other is very easy to learn.

Both algorithms are based on a completely generic algorithm that we will state here - this algorithm can also be used as a template for other generic algorithms to solve other problems.

In this generic algorithm we will build an MST one edge at a time. We start with an empty set of edges and carefully add edges to it until we end up with an MST.

Let S be a subset of the edge set E of a weighted graph G. We say S is SAFE if there exists an MST of G that includes all the edges in S. Clearly the empty set is safe.

```
Generic MST():

    S = {}
    while |S| < n-1:
        find an edge e such that S + e is safe
        S = S + e
    return S
```

Every spanning tree must have exactly $n - 1$ edges. This algorithm

preserves the safety of the edge-set it is building, and when it reaches $n-1$ edges the final edge added must complete the MST.

The hard part of course is "find an edge e such that $S+e$ is safe". We have to make sure the algorithm makes good choices ... and we want it to make them quickly.

Prim's MST Algorithm

THE ALGORITHM THAT we are calling Prim's Algorithm was actually first discovered in 1930 by Jarnik. Prim rediscovered it in 1957, and Dijkstra re-rediscovered it in 1959. Because of this history it is sometimes called Jarnik's Algorithm, the Prim-Jarnik Algorithm, the Prim-Dijkstra Algorithm, or the DJP Algorithm. I'll stick with Prim's Algorithm.

Prim's Algorithm effectively grows a MST from scratch: it starts with a single vertex, then chooses an edge that connects another vertex to the first one, then chooses another edge that connects another vertex to one of the two already chosen, and so on. The tree continues to grow until it contains all n vertices. The criterion by which edges are selected is really simple: we always choose the least-weight edge that has one end in the tree so far and one end in the set of vertices we haven't reached yet.

You can see how this fits the generic algorithm stated above. Prim is asserting that by choosing the least weight edge that has one end among the vertices already connected together and the other among those vertices not yet added to the tree, we are guaranteed to always maintain the safety of the edge-set.

You may be asking questions like:

- How do we decide which vertex to start at?
- What do we do if two or more edges are tied for being the least-weight edge from the tree to the rest of the graph?
- How do we know the result we get is an MST?

Here are the tough-guy answers that mad-scientist/evil-genius Prim gives to these questions:

- It doesn't matter where you start. I'll still find an MST. You can't stop me!


You can pick either one! It doesn't make any difference.


You want the truth? You can't handle the truth! I'll never tell.

Fortunately he's kidding about that last answer. We'll prove the correctness of Prim's Algorithm later. For now we will focus on expressing the algorithm clearly, looking at several implementations, and analyzing their complexity.

Here is one way to express the algorithm in a semi-formal way.

```
def Prim(G): # G is a connected graph with weighted edges

    choose any vertex v
    S = {}                      # S is the set of edges we are choosing
    T = {v}                     # T is the tree we are growing
    R = {all vertices except v} # R is the rest of the vertices

    while |S| < n-1:            # keep going until S contains n-1 edges
        let e be the least-weight edge that has one end in T and one end in R
        suppose e = (x,y) with x in T and y in R
        add e to S
        add y to T
        remove y from R

    return S
```

Everything in that algorithm is trivial except for "***let e be the least-weight edge that has one end in T and one end in R***"

We need to think very carefully about how to implement the selection of the appropriate edge.

There are at least four solutions.

Solution 1

We can take the set of all edges and sort them into ascending-weight order. This takes $O(m * \log m)$ time. Now when we choose the least-weight edge that has one end in T and one end in R, we can do that by searching the list of edges from the beginning and stopping as soon as we find an edge that meets the criterion. Note that every search has to start at the beginning of the list because the low-weight edges may

have originally had both ends in R. So we have to do $n-1$ searches (to choose the $n-1$ edges of the MST) and each search takes $O(m)$ time ... this gives $O(m*n)$ as the complexity. That's pretty bad ... let's try again.

Solution 2

AT EACH ITERATION we need an edge that has the smallest weight, out of the edges that are available (ie. that have one end in T and the other end in R). We have seen a data structure that makes it easy to repeatedly find the **largest** element in a set, but here we need to find the **smallest** element. So instead of a max-heap, we need a min-heap. Fortunately the structural rules of min-heaps are identical to max-heaps, and the "relational" rule is just reversed: each value is $\leq$ its children rather than $\geq$ its children. This results in the smallest item being at the root of the min-heap.

We could start by putting all the edges into the min-heap but this would be wasteful: the edge at the top of the heap might not be usable due to both of its ends being in R - we would need to remove it and then re-insert it into the heap later when it has become usable. To avoid this we only add edges to the heap when they have one end in T and one end in R. Later we may need to reject some of these edges because both ends are in T when the edge reaches the top of the min-heap, but that's not too much trouble.

We can write this version of Prim's MST Algorithm quite completely. This assumes that there is a function $weight(x, y)$ that returns the weight of the edge joining x and y.

```
def Prim(G): # G is a connected graph with weighted edges

    choose any vertex v
    S = {}                       # S is the set of edges we are choosing
    T = {v}                      # T is the tree we are growing
    R = {all vertices except v}  # R is the rest of the vertices

    create a min-heap called M

    for each neighbour y of v:
        add the item [ (v,y) , weight(v,y) ] to M, using the weight value as the "priority"

    while |T| < n: # keep going until S contains n-1 edges
        e = M.remove_top()
        a = e.a
        b = e.b
                            # e consists of a pair of vertices (a,b) and a weight value
                            # When e was added to the min-heap, a was in T and b was in
                            # R ... we need to see if that is still true ... if not we need
                            # to discard e and try the new top item in M

        while b in T:          # loop until we have a usable edge (ie. an edge with one
                               # end in T and the other end in R)
            e = M.remove_top()
            a = e.a
            b = e.b

        #  now we know e = (a,b) with a in T and b in R
        add e to S
        add b to T
        remove b from R
        for each neighbour y of b:
            if y in R:
                add the item [ (b,y), weight(b,y) ] to M, using the weight value as the "priority"

    return S
```

It's useful to think about what would happen if we tried to run this algorithm on a graph that is not connected. Clearly the algorithm must fail because a disconnected graph cannot have a spanning tree at all. The question to think about is ... where would the error occur? How would you modify the algorithm so that it terminates gracefully on a disconnected graph?

Now we can work out the complexity of this version of the algorithm. Clearly the "remove_top" operations are the dominant steps. In the worst case, every edge will eventually be added to the heap and every edge will eventually be removed from the heap. Since the maximum size of the heap is m, we know adding to the heap and removing the top element from the heap are both in $O(\log m)$. Combining this with the fact that each of these steps could execute m times, we end up

with $m * O(\log m)$ which is equal to $O(m * \log m)$.

But we can simplify that a bit - recall that $m < \frac{n^2}{2}$

$\Rightarrow \log m < \log\left(\frac{n^2}{2}\right) = \log(n^2) - \log 2 = 2 * \log(n) - 1$

So $\log m$ is in $O(\log n)$

This means we can rewrite $O(m * \log m)$ as $O(m * \log n)$ ... which is a lot better than $O(m * n)$, which is the complexity of Solution 1.

At this point we can differentiate between two somewhat nebulous cases: G may be SPARSE - with "few" edges, or G may be DENSE - with "many" edges.

Of course there is a third option: G may be neither sparse nor dense. Tertium datur.

The following definitions are not universally accepted but they will make our discussions much easier so we will adopt them conditionally!

DEFINITION: A graph G is SPARSE if $m \leq k * n$ for some predetermined constant k.

DEFINITION: A graph G is DENSE if $m \geq \frac{n^2}{k}$ for some predetermined constant k.

These definitions are most useful when discussing classes of graphs. For example, trees are sparse since a tree must have exactly $n - 1$ edges. Conversely, the complement of a tree has $\binom{n}{2} - (n - 1)$ edges and is therefore dense.

Recall that the complement of a graph G contains all the edges that G does not contain.

These distinctions really only make sense when n is large.

Clearly if G is sparse then this algorithm runs in $O(n * \log n)$ time.

However if the graph is dense, then this algorithm runs in $O(n^2 * \log n)$ time.

Maybe we can beat that complexity ... we have two more possible solutions to look at.

Solution 3

In Solution 2 we spent a lot of time putting edges in and out of a big min-heap, and there was always a possibility that we would have to remove many edges from the top of the heap before finding one we could use. We can attempt to avoid this by using a heap containing vertices instead of edges.

In this solution we use a heap that contains one node for each vertex in R. Every time we add a vertex to T, we see if it gives us a new, less costly connection to any of the vertices in R. If it does, we move the corresponding node upwards in the heap.

Aha! This is why we learned about changing priorities of items in a heap!

This heap will initially contain $n-1$ nodes (the start vertex never needs to be in the heap). The node at the top will correspond to the vertex in R with the least cost edge to T, so we know we can always use the top node - there will never be a need to discard the top node and get another one.

This is looking good ... let's look at it closely. There are no insert operations after the initial build since we build the heap with all $n-1$ vertices of R, and after that we only remove things. Each **`remove_top`** operation takes $O(\log n)$ time and there are only $n-1$ vertices to remove, so it looks like we are heading towards $O(n * \log n)$ complexity. But ... we also have to consider the updates.

Doing an individual update on the heap takes $O(\log n)$ time ... so if we do the updates individually, the total amount of work spent on heap updates is $O(m * \log n)$ because each edge might cause an update. This means that for sparse graphs the algorithm is in $O(n * \log n)$. However, if the graph is very dense (complete or nearly complete) then it is possible that every time we add a vertex to T, it could give us a reduced cost to every vertex in R. Doing the heap updates individually will be $O(n^2 * \log n)$. This is just the same as Solution 2.

However, we know from our study of heaps that if we need to update the priority of every item in the heap, the best thing to do is just rebuild the heap from scratch - and that is an $O(n)$ operation. If we rebuild the heap on each iteration of the loop, we end up with $O(n^2)$ for the whole algorithm. This is better than Solution 2 for dense graphs, but worse on sparse graphs.

The challenge with this version of the algorithm is deciding when we should rebuild the heap, and when we should just do the updates

one by one. One option would be to rebuild the heap whenever the number of updates caused by a single vertex is greater than some threshold value, but do the updates one at a time when the number is below this threshold.

Determining an appropriate threshold value is a very difficult problem - it helps if you have a priori knowledge about the size and density of the graphs you will be dealing with. If you do have this information you can simply have two versions of the algorithm - one for sparse graphs and one for dense graphs.

This difficulty comes from the fact that for a specific graph G with n vertices and m edges we can certainly say that G is sparse if m is close to n, and G is dense if m is close to $\binom{n}{2}$ ... but values of m that are near the middle of the range of possibilities are harder to classify.

You can work out the pseudo-code for this version of Prim's Algorithm - it is a lot like Solution 2, with added operations to do the heap updates/rebuilds.

Solution 4

I liked the heap, but there are other options.

WE CAN SIMPLIFY Solution 3 by ditching the heap. Instead of the heap, we can keep an array of all the vertices in R, together with their minimum cost to connect to T, and the vertex they would connect to (this allows us to keep track of the edges we are using). Every time we add a vertex to T, we look at its neighbours and if necessary we update their "best connection" info. To find the next vertex to connect, we just do a linear search of the array. This array would look something like this (I have added row labels).

Vertex	1	2	3	4	...	n
Still in R?	N	Y	Y	N	...	Y
Connector	-	7	1	13	...	6
Cost	-	8	3	-	...	9

EXPLANATION: Of the vertices we are shown, 1 and 4 have already been added to the tree. Vertex 1 has no connector so it must have been the starting vertex. Vertex 4's connector is Vertex 13 so we can deduce that Vertex 13 was added to the tree before Vertex 4. Similarly the connector information for Vertices 2 and n tell us that Vertices 7 and 6 are also already in the tree.

Suppose that on the next iteration we find that Vertex 3 has the lowest connection cost. We add the edge $(1,3)$ to our set S of Tree edges, we add Vertex 3 to T, and we update Vertex 3's neighbours. Suppose there is an edge from Vertex 3 to Vertex n with cost 7. This is an improvement over the current cost (9) to connect Vertex n, so we update this information. Suppose there is also an edge from Vertex 3 to Vertex 4 - we ignore this because Vertex 4 is not in R - it has already been added to T. Suppose there is an edge from Vertex 3 to Vertex 2 with cost 15. This is greater than the current cost to connect to Vertex 2 (8), so we ignore it.

The array would now look like this:

Vertex	1	2	3	4	...	n
Still in R?	N	Y	N	N	...	Y
Connector	-	7	1	13	...	6
Cost	-	8	-	-	...	7

We would start the next iteration by searching the array for the vertex in R with the lowest connection cost, and repeat the update process.

The complexity of this version of Prim's Algorithm is very easy to determine. We need $n-1$ iterations, each of which involves a linear search of the array - this is in $O(n^2)$. The total number of updates is $\leq m$, and each update takes constant time. Thus the entire algorithm takes $O(n^2+m)$ time ... but since $m < n^2$, this simplifies to $O(n^2)$. Note that this version is not sensitive to the density of the graph - it is in $O(n^2)$ whether the graph is sparse or dense.

Pseudo-code for this algorithm looks something like this:

```
def Prim(G):

    Create array A with three rows (1,2,3) and n columns (1, ... n)
        # row 1 is "Still in R"
        # row 2 is the connector vertex
        # row 3 is the cost
        # note that I have not bothered with a row to store the vertex
        #   numbers since they are just the column numbers

    # we will start our tree with vertex 1 - why not?
    A[1][1] = N # vertex 1 is not in R
    for x = 2 .. n:
        A[1][x] = Y
        A[2][x] = -
        A[3][x] = infinity
    for each neighbour y of vertex 1:
        A[2][y] = 1
        A[3][y] = weight(1,y)

    T = {1}
    S = {} # S is the set of edges we are choosing
    while |S| < n-1:
        search A for the vertex still in R with the smallest value in the third row
        let this be Vertex x
        add x to T
        add the edge (x,A[2][x]) to S
        A[1][x] = N
        A[3][x] = -       # makes it easier to see which vertices are still available
        for each neighbour y of x:
            if y is still in R and weight(x,y) < A[3][y]:
                A[2][y] = x
                A[3][y] = weight(x,y)

    return S
```

Concluding Observations

SOLUTION 1 IS just bad.

Solution 2 is efficient on sparse graphs, but inefficient on dense graphs.

Solution 3 can have exactly the same complexity as Solution 2 (good on sparse, not so good on dense), or exactly the same complexity as Solution 4 (good on dense, not so good on sparse). The complexity of Solution 3 depends on whether we do the heap updates individually or by rebuilding the heap on each iteration. It is difficult to achieve the best of both worlds.

Solution 4 has the same complexity on all graphs - which makes it good for dense graphs but not so good for sparse graphs.

	Sparse Graphs	Dense Graphs
Solution 1	$O(n^2)$	$O(n^3)$
Solution 2	$O(n * \log n)$	$O(n^2 * \log n)$
Solution 3	$O(n * \log n)$ or $O(n^2)$	$O(n^2 * \log n)$ or $O(n^2)$
Solution 4	$O(n^2)$	$O(n^2)$

Is this the end of the Prim implementation story? Not at all! Using data structures called Fibonacci Heaps it is possible to reduce the complexity to $O(m + n * \log n)$ for all graphs ... and the search continues for further improvements.