Graph Algorithms in DROP

v4.71 29 February 2020

Prim's Algorithm

Overview

1. Purpose of the Prim's Algorithm: Prim's algorithm – also known as Jarnik's algorithm – is a greedy algorithm that finds the minimum spanning tree for a weighted, undirected graph. This means that it finds the subset of the edges that form a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex (Wikipedia (2019)).
2. Developers of the Algorithm: The algorithm was developed by the Czech mathematician Jarnik (1930) and later re-discovered and re-published by Prim (1957) and Dijkstra (1959). Therefore, it is also sometimes called the *Jarnik's algorithm* (Sedgewick and Wayne (2011)), *Prim-Dijkstra algorithm* (Cheriton and Tarjan (1976)), *Prim-Jarnik algorithm* (Rosen (2011)), or the *DJP algorithm* (Pettie and Ramachandran (2002)).
3. Comparison with Kruskal's and Boruvska's Algorithms: Other well-known algorithms for this problem include Kruskal's algorithm and Boruvska's algorithm (Tarjan (1983)). These algorithms find a minimum spanning forest in a possibly disconnected graph; in contrast, the most basic form of the Prim's algorithm only finds minimum spanning trees in connected graphs. However, by running Prim's algorithm separately for each connected component of the graph, it can also be used to find the minimum spanning forest (Kepner and Gilbert (2011)). In terms of their asymptotic time complexity, these three algorithms are equally fast for sparse graphs, but slower than other more sophisticated algorithms (Cheriton and Tarjan (1976), Pettie and Ramachandran (2002)). However, for graphs that are sufficiently dense,

Prim's algorithm can be made to run in linear time, meeting or improving the time bounds for other algorithms (Tarjan (1983)).

Description

1. Overview of the Algorithm Steps: The algorithm may be informally described as performing the following steps:
    a. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
    b. Grow the tree by one edge. Of the edges that connect the tree to vertices not yet in the tree, find the minimum weight edge, and transfer it to the tree.
    c. Repeat the previous step until all the vertices are in the tree.

    In more detail, it may be implemented following the pseudocode below.
2. Initialization of Edges and Costs: Associate each vertex v of the graph with a number $C[v]$ - the cheapest cost of connection to v - and an edge $E[v]$ - the edge providing the cheapest connection. To initialize these values, set all values of $C[v]$ to $+\infty$ - or to any number larger than the maximum edge weight – and set each $E[v]$ to a special flag indicating that there is no edge connecting v to earlier vertices.
3. Initializing the Forest and the Vertices: Initialize an empty forest F and a set Q of vertices that have not yet been included in F – initially all vertices.
4. Iteration over the Queue Elements: Repeat the following steps until the Q is empty:
    a. Find and remove a vertex v from the Q having the minimum possible value of $C[v]$
    b. Add v to F and, if $E[v]$ is not the special flag value, also add $E[v]$ to F
    c. Loop over the edges vw connecting v to other vertices w. For each such edge, if w still belongs to Q and vw has a smaller weight than $C[w]$, perform the following steps:
        i. Set $C[w]$ to the cost of edge vw
        ii. Set $C[w]$ to point to edge vw
5. Retrieve the Forest containing the MST's: Return F

6. Starting Vertex for the Algorithm: As described above, the starting vertex for the algorithm will be chosen arbitrarily, because the first iteration of the main loop will have a set of vertices in Q that all the same weight, and the algorithm will automatically start a new tree in $F$when it completes the spanning tree of each connected component of the input graph. The algorithm may be modified to start with any particular vertex s by setting $C[s]$ to be a number smaller than other values of C – for instance, zero – and it may be modified to find only a single spanning tree rather than an entire spanning forest – matching more closest the informal description – by stopping whenever it encounters another vertex flagged as having no associated edge.
7. Choices for implementing the Queue: Different variations of the algorithm differ from each other in how the object Q is implemented: as a simple linked-list, as an array of vertexes, or as a more complicated priority queue data structure. The choices lead to differences in time complexity of the algorithm. In general, a priority queue will be much quicker at finding the vertex v with minimum cost, but will entail more expensive updates when the value of $C[v]$ changes.

Time Complexity

1. Determinants of the Time Complexity: The time complexity for the Prim's algorithm depends on the data structures used for the graphs and for ordering the edges by weight, which can be done using a priority queue. The table below shows the typical choices.
2. Table Time Complexity by Algorithm:

Minimum Edge Weight Data Structure	Total Time Complexity
Adjacency Matrix, Searching	$\mathcal{O}(\|V\|^2)$
Binary Heap and Adjacency List	$\mathcal{O}([\|V\| + \|E\|] \log\|V\|) = \mathcal{O}(\|E\| \log\|V\|)$
Fibonacci Heap and Adjacency List	$\mathcal{O}([\|E\| + \|V\|] \log\|V\|)$

3. Implementation using Adjacency Matrix/List: A simple implementation of Prim's, using an adjacency matrix or an adjacency list graph representation and linearly using an array of weights to find the minimum weight edge to add, requires $\mathcal{O}(|V|^2)$ running time. However, this running time can be greatly improved further by using heaps to implement finding minimum weight edges in the algorithm's inner loop.
4. Heap Based Edge Weight Ordering: A first improved version uses a heap to store all edges of the input graph, ordered by their weight. This leads to an $\mathcal{O}(|E|\log|E|)$ worst-case running time. But storing vertexes instead of edges can improve it still further. The heap should order their vertexes by their smallest edge weight that connects them to any vertex in a partially constructed minimum spanning tree (MST) – or ∞ if no such edge exists. Every time a vertex v is chosen and added to the MST, a decrease-key operation is performed on all vertexes w, setting the key to the minimum of its previous value and the edge cost of (v, w).
5. Impact of Denseness and Queue Implementation: Using a simple binary heap data structure, Prim's algorithm can be shown to run in time $\mathcal{O}(|E|\log|V|)$ where $|E|$ is the number of edges and $|V|$ is the number of vertexes. Using a more sophisticated Fibonacci heap, this can be brought down to $\mathcal{O}([|E| + |V|]\log|V|)$ which is asymptotically faster when the graph is dense enough that $|E|$ is $\omega(|V|)$, and linear time when is at least $\mathcal{O}(|V|\log|V|)$. For graphs of even greater density, having at least $|V|^c$ edges for some

$$c > 1$$

Prim's algorithm can be made to run in linear time even more simply, by using a d-ary heap in place of a Fibonacci heap (Johnson (1975), Tarjan (1983)).

Proof of Correctness

1. Basic Thrust of the Algorithm: Let ***P*** be a connected, weighted graph. At every iteration of Prim's algorithm, an edge must be found that connects a vertex in the subgraph to a vertex outside the subgraph. Since ***P*** is connected, there will always be a path to every vertex. The output Y of Prim's algorithm is a tree, because the edge and the vertex added to Y are connected.
2. An Alternate Minimum Spanning Tree: Let Y_1 be a minimum spanning tree of the graph ***P***. If

$$Y_1 = Y$$

   then Y is a minimum spanning tree. Otherwise, let e be the first edge added during the construction of tree Y that is not in Y_1, and let V be the set of vertexes connected by edges added before edge e. Then one endpoint of edge e is in set V and the other is not.
3. Differences between the Current and the Alternate MSTs: Since tree Y_1 is a spanning graph of ***P***, there is a path in tree Y_1 joining the two endpoints. As one travels along the path, one encounters an edge f joining as vertex in set V to one that is not in V. Now, at the iteration where edge e was added to tree Y, edge f could have also been added, and it would have been added instead of edge e if its weight was less than e. Since it was not added, it may be concluded that

$$w(f) \geq w(e)$$

4. Reconstructing Current from Alternate MST: Let tree Y_2 be a graph obtained by removing edge f and adding edge e to the tree Y_1. It is easy to show that tree Y_2 is connected, has the same number of edges as tree Y_1, and the total weight of its edges is not larger than that of tree Y_1, therefore it is also a minimum spanning tree of graph ***P***, and it contains e and all the edges added to before it during the construction of set V.

5. Metrics Comparison between the MSTs: On repeating the steps above, eventually a minimum spanning tree of graph ***P*** that is identical to tree Y is obtained. This shows that Y is a minimum spanning tree. The minimum spanning allows for the first subset of the first subregion to be expanded into a smaller subset X, which is assumed to be minimum.

Parallel Algorithm

1. Parallelizable Component of the Prim's Algorithm: The main loop pf the Prim's algorithm is inherently sequential and thus not parallelizable. However, the inner loop, which determines the next edge of the minimum weight that does not form a cycle, can be parallelized by dividing the vertexes and edges between the available processors (Grama, Gupta, Karypis, and Kumar (2003)). The pseudocode below demonstrates this.
2. Partition the Vertexes among the Processors: Assign each processor P_i a set V_i of consecutive vertexes of length $\frac{|V|}{|P|}$.
3. Dividing the Edges among the Processors: Create C, E, F, and Q as in the sequential algorithm above and divide C and E as well as the graph between all the processors such that each processor holds the incoming edges to its set of vertexes. Let C_i, E_i denote the parts of C and E stored on processor P_i.
4. Local Minimum Vertexes and their Eventual Union: Repeat the following steps until Q is empty:
    a. On every processor, find the vertex v_i having the minimum value in $C_i[v_i]$ - the local solution.
    b. Min-reduce the local solution to find the vertex v having the minimum possible value of $C[v]$ – the global solution.
    c. Broadcast the selected node to every processor.
    d. Add v to F, and if $E[v]$ is not special value flag, add $E[v]$ to F.

e. On every processor, update C_i and E_i as in the sequential algorithm.

5. Return the Forest containing the MSTs: Return F.
6. Performance of the Parallelized Version: This algorithm can generally be implemented on a distributed machine (Grama, Gupta, Karypis, and Kumar (2003)) as well as on shared memory machines (Quinn and Deo (1984)). It has also been implemented in graphical processing units (GPUs) (Wang, Huang, and Guo (2011)). The running time is $\mathcal{O}\left(\frac{|V|^2}{|P|}\right) + \mathcal{O}(|V| \log|P|)$, assuming that the *reduce* and the *broadcast* operations can be performed in $\mathcal{O}(\log|P|)$ (Grama, Gupta, Karypis, and Kumar (2003)). A variant of the Prim's algorithm for shared memory machines, in which Prim's sequential algorithm is being run in parallel, starting from different vertexes, has also been explored (Setia, Nedunchezhian, and Balachandran (2015)). It should, however, be noted that more sophisticated algorithms exist to solve the distributed minimum spanning tree problem in a more efficient manner.

References

- Cheriton, D., and R. E. Tarjan (1976): Finding Minimum Spanning Trees *SIAM Journal on Computing* **5 (4)** 724-742
- Dijkstra, E. W. (1959): A Note on Two Problems in Connexion with Graphs *Numerische Mathematik* **1 (1)** 269-271
- Grama, A., A. Gupta, G. Karypis, and V. Kumar (2003): *Introduction to Parallel Computing 2nd Edition* **Addison Wesley**
- Jarnik, V. (1930): O Jistem Problemu Minimalnim *Prace Moravske Prirodovedecke Spolecnosti* **6 (4)** 57-63
- Johnson, D. (1975): Priority Queues with Updates and Finding Minimum Spanning Trees *Information Processing Letters* **4 (3)** 53-57
- Kepner, J., and J. Gilbert (2011): *Graph Algorithms in the Language of Linear Algebra* **Society for Industrial and Applied Mathematics**

- Pettie, S., and V. Ramachandran (2002): An Optimal Minimum Spanning Tree Algorithm *Journal of the ACM* **49 (1)** 16-34
- Prim, R. C. (1957): Shortest Connection Networks and some Generalizations *Bell System Technical Journal* **36 (6)** 1389-1401
- Quinn, M. J., and N. Deo (1984): Parallel Graph Algorithms *ACM Computing Surveys* **16 (3)** 319-348
- Rosen, K. (2011): *Discrete Mathematics and its Application 7th Edition* **McGraw-Hill Science**
- Sedgewick, R. E., and K. D. Wayne (2011): *Algorithms 4th Edition* **Addison-Wesley**
- Setia, R., A. Nedunchezhian, and S. Balachandran (2015): A New Parallel Algorithm for Minimum Spanning Tree Problem
- Tarjan, R. E. (1983): *Data Structures and Network Algorithms* **Society for Industrial and Applied Mathematics**
- Wang, W., Y. Huang., and S. Guo (2011): Design and Implementation of GPU-Based Prim's Algorithm *International Journal of Modern Education and Computer Science* **4** 55-62
- Wikipedia (2019): Prim's Algorithm

Kruskal's Algorithm

Introduction

1. Principal Idea behind Kruskal's Algorithm: *Kruskal's algorithm* (Kruskal (1956), Wikipedia (2020)) is a minimum spanning tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest (Cormen, Leiserson, Rivest, and Stein (2009)).It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected, weighted graph adding increasing cost arc at each step. This means that it finds the subset of edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* – a minimum spanning tree for each component.
2. Alternate Algorithms for Extracting MSFs: Other algorithms for this problem include Prim's algorithm, reverse-delete algorithm, and Boruvka's algorithm.

The Algorithm

1. Forest with Vertexes Per Tree: Create a forest F - a set of trees – where each vertex in the graph is a separate tree.
2. Set of all Graph Edges: Create a set S containing all the edges in the graph.
3. Edge-Based Processing and Tree Update: While S is not empty and F is not yet spanning:
    a. Remove and edge with minimum weight from S
    b. If the removed edge connects two different trees, then add it to the forest F, combining the two trees into a single tree.

4. Minimum Spanning Tree and Forest: At the termination of the algorithm, the forest forms a set of minimum spanning trees of the graph. If the graph is connected, the forest has a single component, and forms a minimum spanning tree.

Complexity

1. Asymptotic Bounds Using $|E|/|V|$: Kruskal's algorithm can be shown to run in $\mathcal{O}(|E|\log|E|)$ or equivalently, in $\mathcal{O}(|E|\log|V|)$, where $|E|$ is the number of edges in the graph and $|V|$ is the number of vertexes, all using simple data structures. These running times are equivalent because:
    a. $|E|$ is at most $|V|^2$ and

$$\log|V|^2 = 2\log|V|$$

    is $\mathcal{O}(\log|V|)$
    b. Each isolated vertex is a separate component of the minimum spanning forest. If one ignores isolated vertexes, one obtains

$$|V| \leq 2|E|$$

    so $\log|V|$ is $\mathcal{O}(\log|E|)$.
2. Rationale behind the Bound Estimate: This bound may be achieved as follows. First, sort the edges by weight using a comparison sort in $\mathcal{O}(|E|\log|E|)$ time; this allows the step that removes an edge with minimum weight from S to operate in constant time. Next, a disjoint-set data structure is used to keep track of which vertexes are in which components. This needs $\mathcal{O}(|V|)$ operations; since in each iteration where a vertex is connected to a spanning tree, two *find* operations and possibly one union for each edge are needed. Even a simple disjoint-set data structure such as disjoint set

forests with union by rank can perform $\mathcal{O}(|V|)$ operations in $\mathcal{O}(|V|\log|V|)$ time. Thus, the total time is

$$\mathcal{O}(|E|\log|E|) = \mathcal{O}(|E|\log|V|)$$

3. Sophisticated Disjoint Sets and Sorters: Provided that the edges are already sorted or can be sorted in linear time – for example, with counting sort or radix sort, the algorithm can use a more disjoint set data structure to run in $\mathcal{O}\big(|E|\alpha(|V|)\big)$, where α is an extremely slowly growing inverse of the single-valued Ackermann function.

Proof of Correctness

The proof consists of two parts. First, it is proved that the algorithm produces a spanning tree. Second, it is proved that the constructed tree is of minimal weight.

Spanning Tree

Let G be a connected, weighted graph, and Y be a subgraph produced by the algorithm. Y cannot have a cycle, being within one subtree and not between two different trees. Y cannot be disconnected, since the first encountered edge that joins the two components of Y would have been added by the algorithm. Thus Y is a spanning tree of G.

Minimality

1. Proposition: Existence of an MST: It may be shown, using induction, that the following proposition ***P*** by induction is true; if F is the set of edges at any stage in the algorithm, then there is some minimum spanning tree that contains F.

2. Induction Proof - Validity at Start: Clearly ***P*** is tree at the beginning when F is empty; any spanning tree will do, and there exists one, because a connected weighted graph always has a minimum spanning tree.
3. Inductive Proof - Intermediate Stage Validity: Now assume that ***P*** is true for some non-final edge state F and let T be a minimum spanning tree that contains F.
    a. If the next chosen edge e is also in T, then ***P*** is true for $F + e$.
    b. Otherwise, if e is not in T, then $T + e$ has a cycle C. This cycle contains edges that do not belong to F, since E does not form a cycle when added to F but does in T. Let f be an edge which is in C but not in $F + e$. Note that F also belongs to T, and by ***P*** has not been considered by the algorithm. f must therefore haver a weight at least a large as e. Then $T - f + e$ is a tree, and it has the same or less weight as T. So $T - f + e$ is a minimum spanning tree containing $F + e$ and again ***P*** holds.
4. Completing the Inductive Proof: Therefore, by the principle of induction, ***P*** holds when F has become a spanning tree, which is only possible if F is a minimum spanning tree itself.

Parallel Algorithm

1. Strategies for Parallelizing the Algorithm: Kruskal's algorithm is inherently sequential and hard to parallelize. It is, however, possible to perform the initial sorting of the edges in parallel, or alternatively, to use a parallel implementation of the binary heap to extract the minimum-weight edge in every iteration (Quinn and Deo (1984)). As parallel sorting is possible in time $\mathcal{O}(n)$ on $\mathcal{O}(\log n)$ processors (Grama, Gupta, Karypis, and Kumar (2003)), the runtime of the Kruskal's algorithm can be reduced to $\mathcal{O}\big(|E|\alpha(|V|)\big)$, where α is again the inverse of the single-valued Ackermann function.
2. The Filter-Kruskal Parallel Version: The variant of Kruskal's algorithm, named Filter-Kruskal, has been described by Osipov, Sanders, and Singler (2009) and is

better suited for parallelization. The basic idea behind Filter-Kruskal is to partition the edges in a way similar to quicksort and to filter out the edges that connect the vertexes of the same tree to reduce the cost of sorting.

3. Advantages of the Filter-Kruskal Scheme: Filter-Kruskal lends itself better for parallelization as sorting, filtering, and partitioning can be performed easily by distributing the edges between the processors (Osipov, Sanders, and Singler (2009)).
4. Other Approaches to Kruskal Parallelization: Finally, other variants of a parallel implementation of Kruskal's algorithm have been explored. Examples include a scheme to that uses helper threads to remove edges that are definitely not part of the MST in the background (Katsigiannis, Anastopoulos, Konstantinos, and Koziris (2012)), and a variant that runs the sequential algorithm in p subgraphs, and then merges those subgraphs until only one, the final MST, remains (Loncar, Skrbic, and Balaz (2014)).

References

- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- Grama, A., A. Gupta, G. Karypis, and V. Kumar (2003): *Introduction to Parallel Computing 2nd Edition* **Addison Wesley**
- Katsigiannis, A., N. Anastopoulos, K. Nikas, and N. Koziris (2012): An Approach to Parallelize Kruskal's Algorithm using Helper Threads
- Kruskal, J. B. (1956): On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem *Proceedings of the American Mathematical Society* **7 (1)** 48-50
- Loncar, V., S. Skrbic, and A. Balaz (2014): Parallelization of Minimum Spanning tree Algorithms using Distributed Memory Architectures *Transactions on Engineering Technologies* 543-554
- Osipov, V., P. Sanders, and J. Singler (2009): The Filter-Kruskal Minimum Spanning Tree Algorithm

- Quinn, M. J., and N. Deo (1984): Parallel Graph Algorithms *ACM Computing Surveys* **16 (3)** 319-348
- Wikipedia (2020): Kruskal's Algorithm