

# Graph Algorithms in DROP

**v4.72** 14 March 2020

**Spanning Tree**

**Overview**

A *spanning tree* of an undirected graph is a subgraph that includes all of the vertexes of , with minimum possible number of edges (Wikipedia (2020)). In general, a graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree. If all of the edges of are also edges of a spanning tree of , then is a tree and identical to , that is, a tree has a unique spanning tree and that is itself

**Applications**

1. Use in Path Finding Algorithms: Several path-finding algorithms, including Dijkstra’s algorithm and the A\* search algorithm, internally build a spanning tree as an intermediate step in solving the problem.
2. Use in Cost Minimization Problems: In order to minimize the cost of power networks, wiring connections, piping, automatic speech recognition, etc., people often use algorithms that gradually build a spanning tree – or many such trees – as intermediate steps in the process of finding the minimum spanning tree (Graham and Hell (1985)).
3. Use in Link-State Protocols: The internet and many other telecommunications networks have transmission links that connect nodes together in a mesh topology that includes some loops. In order to avoid *bridge loops* and *routing loops*, many protocols design for such networks – including Spanning Tree Protocol, Open Shortest Path First, Link-State Routing Protocol, Augmented Tree-Based Routing, etc. – require each router to remember a spanning tree.
4. Graph Embeddings with Maximum Genus: A special kind of tree, the Xuong tree, is used in topological graph theory to find graph embeddings with maximum genus. A Xuong tree is a spanning tree such that, in the remaining graph, the number of connected components with an odd number of edges is as small as possible. A Xuong tree and an associated maximum genus embedding can be found in polynomial time (Beineke and Wilson (2009)).

**Definitions**

A tree is a connected, undirected graph with no cycles. It is a spanning tree of a graph if it spans – that is, it includes every vertex of – and is a subgraph of , i.e., every edge in the tree belongs to . A spanning tree of a connected graph can also be defined as a maximal set of edges that contains no cycle, or as a minimal set of edges that connect all vertexes.

**Fundamental Cycles**

Adding just one edge to the spanning tree will create a cycle; such a cycle is called a *fundamental cycle*. There is a distinct fundamental cycle for each edge not in the spanning tree; thus, there is a one-to-one correspondence between fundamental cycles and edges not in the spanning tree. For a connected graph with vertexes, and spanning tree will have edges, and thus, a graph of edges and one of its spanning trees will have fundamental cycles. For any given spanning tree, the set of all fundamental cycles forms a cycle basis, a basis for the cycle space (Kocay and Kreher (2004)).

Fundamental Cut-sets

1. Motivation behind the Fundamental Cut-set: Dual to the notion of a fundamental cycle is the *fundamental cut-set*. By deleting just one edge of the spanning tree, the vertexes are partitioned into two disjoint sets. The fundamental cut-set is defined as the set of edges that must be removed from the graph to achieve the same partition. Thus, each spanning tree defines a set of fundamental cut-sets, one for each edge of the spanning tree (Kocay and Kreher (2004)).
2. Duality between Cut-sets and Cycles: The duality between fundamental cut-sets and fundamental cycles is established by noting that the cycle edges not in the spanning tree can only appear in the cut-sets of the other edges of the cycle; and *vice versa*; edges in a cut-set can only appear in those cycles containing the edge corresponding to the cut-set. This duality can be expressed using the theory of matroids, according to which the spanning tree is the base of a graphic matroid; a fundamental cycle is the unique circuit within the set formed by adding one element to the base, and fundamental cut-sets are defined in the same way as the dual matroid (Oxley (2006)).

**Spanning Forests**

1. Competing Definitions of Spanning Forests: In graphs that are not connected, there can be no spanning tree, and on must consider *spanning forests* instead. Here, there are two competing definitions:
   1. Some authors consider a spanning forest to be the maximal acyclic subgraph of a given graph, or equivalently, a graph consisting of a spanning tree in each connected component of the graph (Bollobas (1998), Mehlhorn (1999)).
   2. For other authors, a spanning forest is a forest that spans all of the vertexes, meaning only that each vertex in the graph is a vertex in the forest. Under this definition, even a connected graph may have a disconnected spanning forest, such as a forest in which each vertex forms a single-vertex tree (Cameron (1994)).
2. Full versus Maximal Spanning Forest: To avoid confusion between these two definitions, Gross and Yellen (2005) suggest the term *full spanning forest* for a spanning forest with the same connectivity as a given graph, while Bondy and Murthy (2008) instead call this kind of forest a maximal spanning forest.

**Counting Spanning Trees**

The number of the spanning trees of a connected graph is a well-studied invariant.

**In Specific Graphs**

1. When is a Tree: In some cases, it is easy to calculate directly. If is a tree itself, then
2. is a Cycle Graph: When is a cycle graph with vertexes, then
3. is Complete with Vertexes: For a complete graph with vertexes, Cayley’s formula (Aigner and Ziegler (1998)) gives the number of spanning trees as .
4. is Complete Bipartite: If is a complete bipartite graph then

(Hartsfield and Ringel (2003)).

1. is an n-dimensional Hyper-cube: For an n-dimensional hyper-cube graph, the number of spanning trees is

(Harary, Hayes, and Wu (1988)).

**In Arbitrary Graphs**

1. Arbitrary Graph Spanning Tree Count: More generally, for any graph , the number can be calculated in polynomial time as a determinant of a matrix derived from the graph, using Kirchoff’s matrix-tree theorem.
2. Kirchoff’s Matrix-Tree Theorem Method: Specifically, to compute , one constructs a square matrix in which both the rows and the columns are indexed by vertexes of . The entry in row and column is one of three values:
   1. The degree of vertex if
   2. if vertexes and are adjoint, OR
   3. if vertexes and are different from each other but not adjacent.

The resulting matrix is singular, so its determinant is zero. However, deleting a row and a column for an arbitrarily chosen vertex leads to a smaller matrix whose determinant is exactly .

**Deletion-Contraction**

1. The Deletion-Contraction Recurrence Formula: If is a graph or a multi-graph, and is an arbitrary edge of , then the number of spanning trees of satisfies the *deletion-contraction recurrence*

where is the multi-graph obtained by deleting , and is the contraction of by (Kocay and Kreher (2004)). The term in the formula counts the number of spanning trees of that do not use the edge , and the term counts the spanning trees of that use the edge .

1. Retention of Redundant Graph Edges: In the above formula, is the given graph is a multi-graph, or if a contraction causes two vertexes to be connected to each other by multiple edges, then the redundant edges should not be removed, as that would lead to the wrong total. For instance, a bond graph connecting two edges by edges has different spanning trees, each consisting of one of these edges.

**Tutte Polynomial**

1. Sum over Internal/External Activity: The Tutte polynomial of a graph can be defined as a sum, over the spanning trees of the graph, of terms computed from *internal activity* and *external activity* of the tree. Its value at the arguments is the number of spanning trees, or, in a disconnected graph, the number of maximal spanning forests (Bollobas (1998)).
2. Computational Complexity using Contraction-Deletion Recurrence: The Tutte polynomial can be computed using a deletion-contraction recurrence, but its computational complexity is high: for many values of its arguments, computing it is exactly **#P**-complete, and it is also hard to approximate with a guaranteed approximation ratio. The point at which it can be evaluated using Kirchoff’s theorem, is one of the few exceptions (Jaeger, Vertigan, and Welsh (1990), Goldberg and Jerrum (2008)).

**Algorithms – Construction**

1. Spanning Tree using BFS/DFS: A single spanning tree of a graph can be found in linear time by either depth-first search or breadth-first search. Bothe of these algorithms explore then given graph, starting from an arbitrary vertex , by looping through the neighbors of the vertexes they discover and adding each unexplored neighbor to a data structure to be explored later. They differ in whether the data structure is a stack – in the case of depth-first search – or a queue – in the breadth-first search. In either case, one can form a spanning tree by connecting each vertex, other than the root vertex , to a vertex from which it was discovered. This tree is known as the depth-first search tree or the breadth-first search tree according to the graph exploration algorithm used to construct it (Kozen (1992)). Depth-first search trees are a special case of a class of spanning trees called the Tremaux trees, named after the 19th century discoverer of depth-first search (de Fraysseix and Rosenstiehl (1982)).

**References**

* Aigner, M., and G. M. Ziegler (1998): *Proofs from THE BOOK* **Springer-Verlag**
* Beinecke, L. W., and R. J. Wilson (2009): Topics in Topological Graph Theory *Encyclopedia of Mathematics and its Applications* **128** **Cambridge University Press**
* Bollobas, B. (1998): *Modern Graph Theory – Graduate Texts in Mathematics* **184** **Springer**
* Bondy, J. A., and U. S. R. Murty (2008): *Graph Theory – Graduate Texts in Mathematics* **244** **Springer**
* Cameron, P. J. (1994): *Combinatorics; Topics, Techniques, Algorithms* **Cambridge University Press**
* de Fraysseix, H., and P. Rosenstiehl (1982): A Depth-First-Search Characterization of Planarity *Annals of Discrete Mathematics* **13** 75-80
* Goldberg, L. A., and M. Jerrum (2008): Inapproximability of the Tutte Polynomial *Information and Computation* **206 (7)** 908-929
* Graham, R. L., and P. Hell (1985): On the History of the Minimum Spanning Tree Problem *Annals of the History of Computing* **7 (1)** 43-57
* Gross, J. L., and J. Yellen (2005): *Graph Theory and its Applications 2nd Edition* **CRC Press**
* Harary, F., J. P. Hayes, and H. J. Wu (1988): A Survey of the Theory of Hypercube Graphs *Computers and Mathematics with Applications* **15 (4)** 277-289
* Hartsfield, N. and G. Ringel (2003): *Pearls in Graph Theory – A Comprehensive Introduction* **Courier Dover Publications**
* Jaeger, F., D. J. Vertigan, and D. J. A. Welsh (1990): On the Computational Complexity of the Jones and the Tutte Polynomials *Mathematical Proceedings of the Cambridge Philosophical Society* **108 (1)** 35-53
* Kocay, W., and D. L. Kreher (2004): *Discrete Mathematics and its Applications* **CRC Press**
* Kozen, D. (1992): *The Design and Analysis of Algorithms: Monographs in Computer Science* **Springer**
* Mehlhorn, K. (1999): *Leda: A Platform for Combinatorial and Geometric Computing* **Cambridge University Press**
* Oxley, J. G. (2006): *Matroid Theory; Oxford Graduate Texts in Mathematics* **3** **Oxford University Press**
* Wikipedia (2020): [Spanning Tree](https://en.wikipedia.org/wiki/Spanning_tree)

**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:
   1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
   2. 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.
   3. Repeat the previous step until all the vertices are in the tree.

In more detail, it may be implemented following the pseudocode below.

1. Initialization of Edges and Costs: Associate each vertex of the graph with a number - the cheapest cost of connection to - and an edge - the edge providing the cheapest connection. To initialize these values, set all values of to - or to any number larger than the maximum edge weight – and set each to a special flag indicating that there is no edge connecting to earlier vertices.
2. Initializing the Forest and the Vertices: Initialize an empty forest and a set of vertices that have not yet been included in – initially all vertices.
3. Iteration over the Queue Elements: Repeat the following steps until the is empty:
   1. Find and remove a vertex from the having the minimum possible value of
   2. Add to and, if is not the special flag value, also add to
   3. Loop over the edges connecting to other vertices . For each such edge, if still belongs to and has a smaller weight than , perform the following steps:
      1. Set to the cost of edge
      2. Set to point to edge
4. Retrieve the Forest containing the MST’s: Return
5. 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 that all the same weight, and the algorithm will automatically start a new tree in 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 by setting to be a number smaller than other values of – 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.
6. Choices for implementing the Queue: Different variations of the algorithm differ from each other in how the object 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 with minimum cost, but will entail more expensive updates when the value of 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 |  |
| Binary Heap and Adjacency List |  |
| Fibonacci Heap and Adjacency List |  |

1. 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 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.
2. 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 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 is chosen and added to the MST, a decrease-key operation is performed on all vertexes , setting the key to the minimum of its previous value and the edge cost of .
3. Impact of Denseness and Queue Implementation: Using a simple binary heap data structure, Prim’s algorithm can be shown to run in time where is the number of edges and is the number of vertexes. Using a more sophisticated Fibonacci heap, this can be brought down to which is asymptotically faster when the graph is dense enough that is , and linear time when is at least . For graphs of even greater density, having at least edges for some

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

**Proof of Correctness**

1. Basic Thrust of the Algorithm: Let 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 is connected, there will always be a path to every vertex. The output of Prim’s algorithm is a tree, because the edge and the vertex added to are connected.
2. An Alternate Minimum Spanning Tree: Let be a minimum spanning tree of the graph . If

then is a minimum spanning tree. Otherwise, let be the first edge added during the construction of tree that is not in , and let be the set of vertexes connected by edges added before edge . Then one endpoint of edge is in set and the other is not.

1. Differences between the Current and the Alternate MSTs: Since tree is a spanning graph of , there is a path in tree joining the two endpoints. As one travels along the path, one encounters an edge joining as vertex in set to one that is not in . Now, at the iteration where edge was added to tree , edge could have also been added, and it would have been added instead of edge if its weight was less than . Since it was not added, it may be concluded that
2. Reconstructing Current from Alternate MST: Let tree be a graph obtained by removing edge and adding edge to the tree . It is easy to show that tree is connected, has the same number of edges as tree , and the total weight of its edges is not larger than that of tree , therefore it is also a minimum spanning tree of graph , and it contains and all the edges added to before it during the construction of set .
3. Metrics Comparison between the MSTs: On repeating the steps above, eventually a minimum spanning tree of graph that is identical to tree is obtained. This shows that is a minimum spanning tree. The minimum spanning allows for the first subset of the first subregion to be expanded into a smaller subset , 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 a set of consecutive vertexes of length .
3. Dividing the Edges among the Processors: Create , , , and as in the sequential algorithm above and divide and as well as the graph between all the processors such that each processor holds the incoming edges to its set of vertexes. Let , denote the parts of and stored on processor .
4. Local Minimum Vertexes and their Eventual Union: Repeat the following steps until is empty:
   1. On every processor, find the vertex having the minimum value in - the local solution.
   2. Min-reduce the local solution to find the vertex having the minimum possible value of – the global solution.
   3. Broadcast the selected node to every processor.
   4. Add to , and if is not special value flag, add to .
   5. On every processor, update and as in the sequential algorithm.
5. Return the Forest containing the MSTs: Return .
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 , assuming that the *reduce* and the *broadcast* operations can be performed in (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](https://hipcor.fatcow.com/hipc2009/documents/HIPCSS09Papers/1569250351.pdf)
* 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](https://en.wikipedia.org/wiki/Prim%27s_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 - a set of trees – where each vertex in the graph is a separate tree.
2. Set of all Graph Edges: Create a set containing all the edges in the graph.
3. Edge-Based Processing and Tree Update: While is not empty and is not yet spanning:
   1. Remove and edge with minimum weight from
   2. If the removed edge connects two different trees, then add it to the forest , 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 /: Kruskal’s algorithm can be shown to run in or equivalently, in , where is the number of edges in the graph and is the number of vertexes, all using simple data structures. These running times are equivalent because:
   1. is at most  and

is

* 1. Each isolated vertex is a separate component of the minimum spanning forest. If one ignores isolated vertexes, one obtains

so is .

1. Rationale behind the Bound Estimate: This bound may be achieved as follows. First, sort the edges by weight using a comparison sort in time; this allows the step that removes an edge with minimum weight from 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 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 operations in time. Thus, the total time is
2. 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 , where 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 be a connected, weighted graph, and be a subgraph produced by the algorithm. cannot have a cycle, being within one subtree and not between two different trees. cannot be disconnected, since the first encountered edge that joins the two components of would have been added by the algorithm. Thus is a spanning tree of .

**Minimality**

1. Proposition: Existence of an MST: It may be shown, using induction, that the following proposition by induction is true; if is the set of edges at any stage in the algorithm, then there is some minimum spanning tree that contains .
2. Induction Proof - Validity at Start: Clearly is tree at the beginning when 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 is true for some non-final edge state and let be a minimum spanning tree that contains .
   1. If the next chosen edge is also in , then is true for .
   2. Otherwise, if is not in , then has a cycle . This cycle contains edges that do not belong to , since does not form a cycle when added to but does in . Let be an edge which is in but not in . Note that also belongs to , and by has not been considered by the algorithm. must therefore haver a weight at least a large as . Then is a tree, and it has the same or less weight as . So is a minimum spanning tree containing and again holds.
4. Completing the Inductive Proof: Therefore, by the principle of induction, holds when has become a spanning tree, which is only possible if 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 on processors (Grama, Gupta, Karypis, and Kumar (2003)), the runtime of the Kruskal’s algorithm can be reduced to , where 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 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](http://tarjomefa.com/wp-content/uploads/2017/10/7793-English-TarjomeFa.pdf)
* 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](http://algo2.iti.kit.edu/documents/fkruskal.pdf)
* Quinn, M. J., and N. Deo (1984): Parallel Graph Algorithms *ACM Computing Surveys* **16 (3)** 319-348
* Wikipedia (2020): [Kruskal's Algorithm](https://en.wikipedia.org/wiki/Kruskal%27s_algorithm)