Minimum Spanning Tree
-
DPHPC

Th. Cambier R. Dang-Nhu Th. Dardinier C. Trassoudaine

ETH Zürich

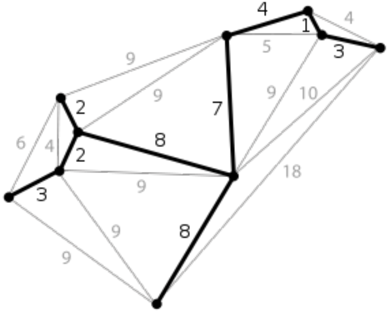
October 2018

1. Problem definition
   - Use cases
2. Algorithms
   - Prim
   - Kruskal
   - Borůvka (Sollin)
   - Others
3. Environment
4. Benchmarking
   - Reference, baseline, tools

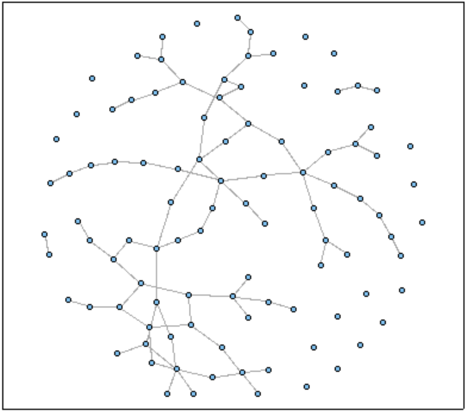
Problem definition

The MST problem

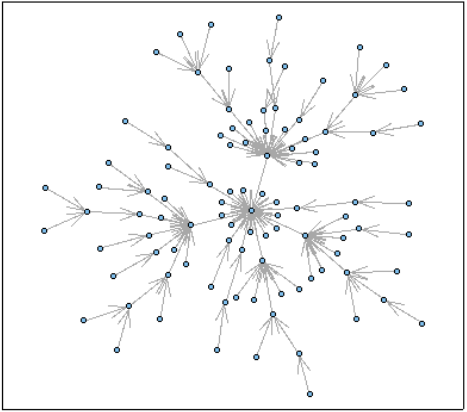
A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted (un)directed graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.



Input sets: G(100, 0.02)



Input sets: PA(100)

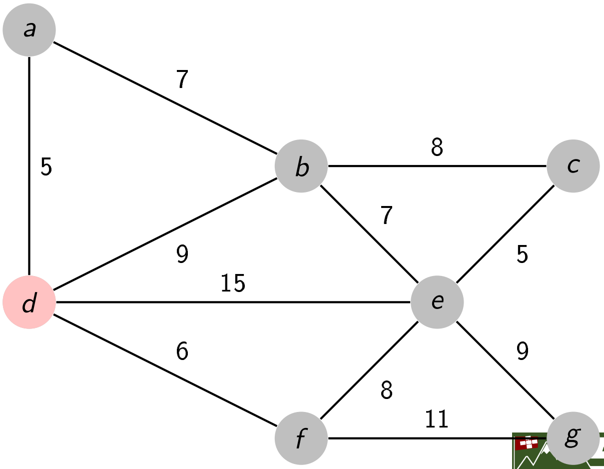


Algorithms

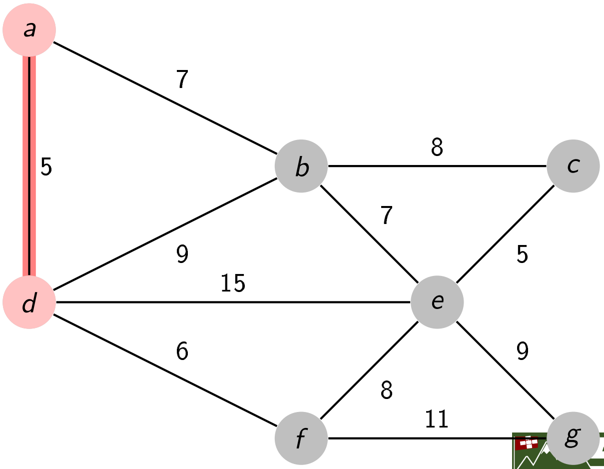
Sequential Prim

- Initialise a tree with a single random vertex.
- Among all the edges connecting the tree to another vertex, find the minimum-weighted one and transfer it to the tree.
- Repeat until all vertices are in the tree.

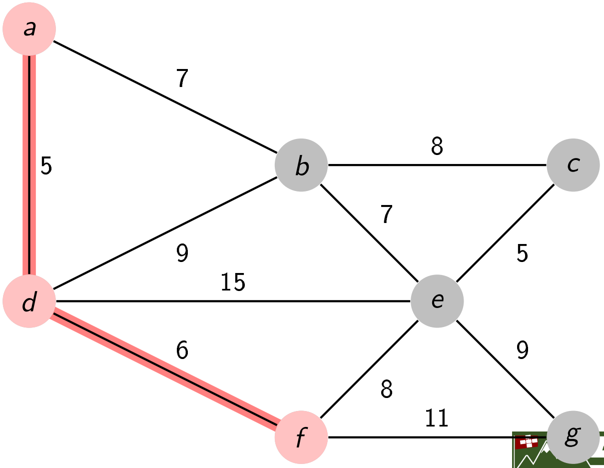
Sequential Prim



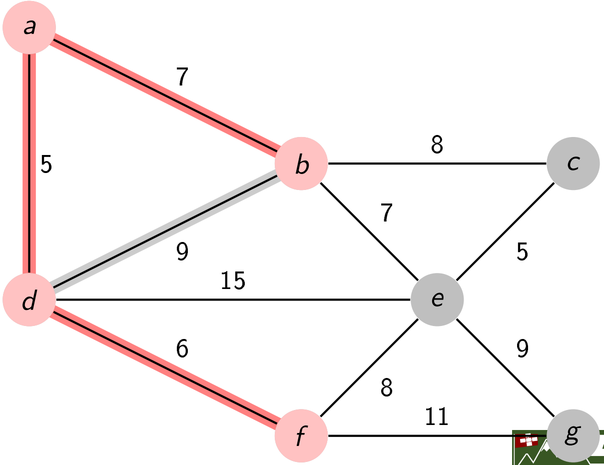
Sequential Prim



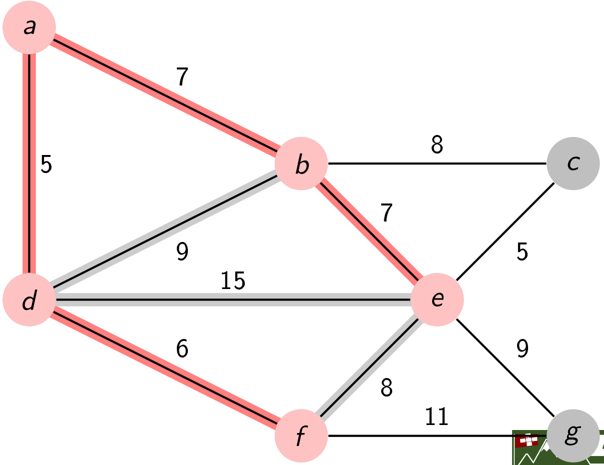
Sequential Prim



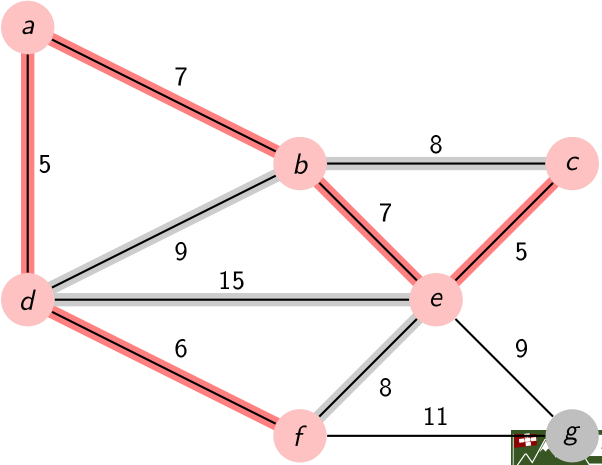
Sequential Prim



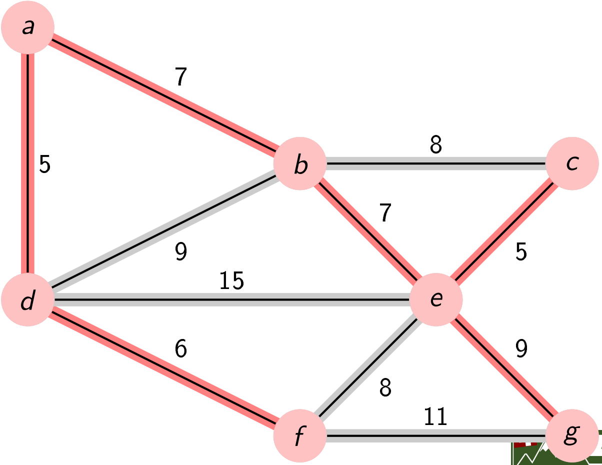
Sequential Prim



Sequential Prim



Sequential Prim



Sequential and parallel Prim

Sequential complexity:

- $O(V^2)$: Adjacency matrix representation
- $O(E \log V)$: Adjacency list representation with the use of a binary heap

Sequential and parallel Prim

Sequential complexity:

- $O(V^2)$: Adjacency matrix representation
- $O(E \log V)$: Adjacency list representation with the use of a binary heap

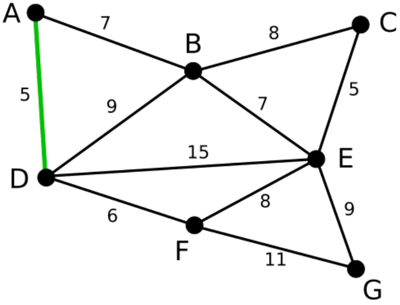
We can parallelize the search of the edge of minimum weight by dividing the vertices and edges between processors to compute local minima.

Sequential Kruskal

- Sort all edges by growing weight.
- For each edge: Add it to the MST if it doesn't create a cycle.

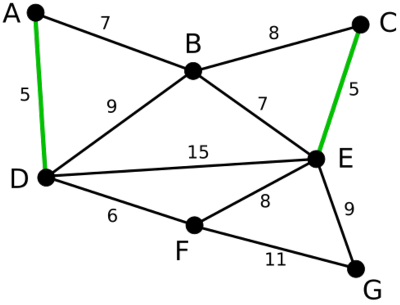
Sequential Kruskal

- Sort all edges by growing weight.
- For each edge: Add it to the MST if it doesn't create a cycle.



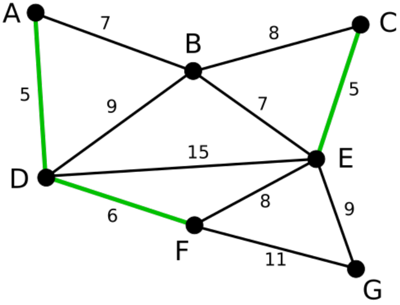
Sequential Kruskal

- Sort all edges by growing weight.
- For each edge: Add it to the MST if it doesn't create a cycle.



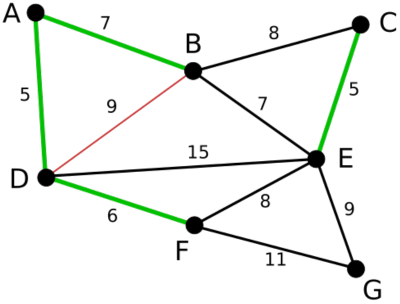
Sequential Kruskal

- Sort all edges by growing weight.
- For each edge: Add it to the MST if it doesn't create a cycle.



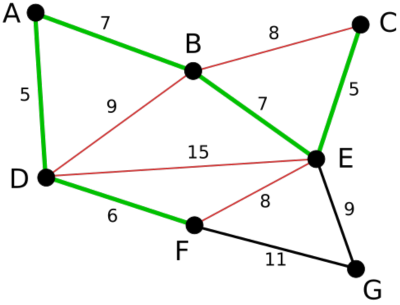
Sequential Kruskal

- Sort all edges by growing weight.
- For each edge: Add it to the MST if it doesn't create a cycle.



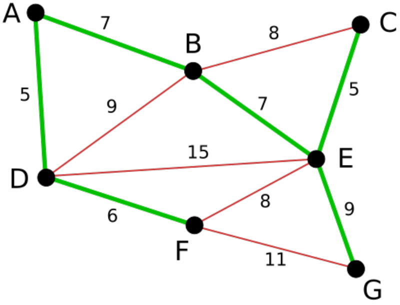
Sequential Kruskal

- Sort all edges by growing weight.
- For each edge: Add it to the MST if it doesn't create a cycle.



Sequential Kruskal

- Sort all edges by growing weight.
- For each edge: Add it to the MST if it doesn't create a cycle.



Data structure: Union-find

Represents the connected components of the graph given our MST.
Implementation with an array *parent* and 3 operations:

- Create: Every vertex is in its own component
  1. $parent[x] = x$.

Data structure: Union-find

Represents the connected components of the graph given our MST.

Implementation with an array *parent* and 3 operations:

- Create: Every vertex is in its own component
  1. $parent[x] = x$.
- Find(x): Find the component of this vertex.
  1. If $parent[x] \neq x$, then $parent[x] = find(parent[x])$
  2. Return $parent[x]$.

Data structure: Union-find

Represents the connected components of the graph given our MST.
Implementation with an array *parent* and 3 operations:

- Create: Every vertex is in its own component
  1. $parent[x] = x$.
- Find(x): Find the component of this vertex.
  1. If $parent[x] \neq x$, then $parent[x] = find(parent[x])$
  2. Return $parent[x]$.
- Union(x, y): Unite two components.
  1. $parent[find(x)] = find(y)$.

Sequential and parallel Kruskal

Sequential complexity:

- $O(E \log E)$: Sort all edges by growing weight.
- $O(E)$ (in practice): For each edge: Add it to the MST if it doesn't create a cycle.

Sequential and parallel Kruskal

Sequential complexity:

- $O(E \log E)$: Sort all edges by growing weight.
- $O(E)$ (in practice): For each edge: Add it to the MST if it doesn't create a cycle.

We can parallelize the sort on $O(\log E)$ processors: $O(E)$ (in practice).

A better parallel approach: Filter-Kruskal

Similar to a quick sort[1]:

1. If $E <$ *threshold*, solve using classical Kruskal
2. Choose a pivot (edge)
3. Partition in two sets $E_{\leq}, E_{>}$ (weight)
4. Recursive call to solve problem with $E_{\leq}$
5. Filter out the edges of $E_{>}$ that connect two vertices of the same component
6. Recursive call to solve problem with $E_{>}$

[1] Osipov, Sanders, and Singler 2009.

A better parallel approach: Filter-Kruskal

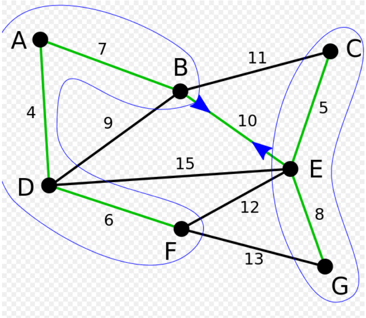
Similar to a quick sort[1]:

1. If $E <$ *threshold*, solve using classical Kruskal
2. Choose a pivot (edge)
3. Partition in two sets $E_{\leq}$, $E_{>}$ (weight)
4. Recursive call to solve problem with $E_{\leq}$
5. Filter out the edges of $E_{>}$ that connect two vertices of the same component
6. Recursive call to solve problem with $E_{>}$

Better for parallelization since we can distribute the edges for filtering and partitioning.

[1] Osipov, Sanders, and Singler 2009.

Borůvka (Sollin)



[2,3]

```
1) Init V as independant sets.
2) Initialize MST as empty.
3) While #sets > 1, do:
   a)  Find closest E
       from this set to another.
   b)  Add this E to MST
       if not already added.
4) Return MST.
```

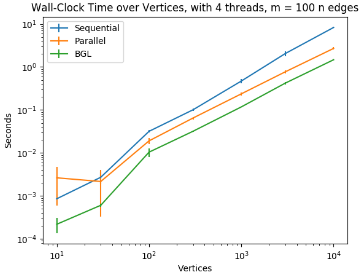
[2] Chung and Condon 1996.

[3] Bader and Cong 2006.

- Prim (Parallel & Seq)
- Kruskal (Parallel & Seq), Kruskal filter
- Sollin (Parallel & Seq)

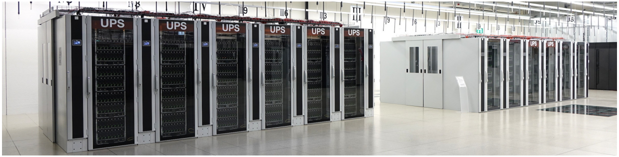
- Randomization

- Correctness



Environment

Architecture



EULER Cluster

Xeon E$x, x \in \{3, 5, 7\}$
x86_64 architecture

Source : `https://scicomp.ethz.ch/wiki/Euler`

Tools

- CMake
  v3.3+
- C++11
  GCC v4.9.2+
- OpenMPI (shared memory)
  v1.6.5+

Benchmarking

Tools

- **Measures :** LibSciBench library
- **Interpretation :**
  - LibSciBench's R scripts
  - (Custom python scripts)

Ref : `https://spcl.inf.ethz.ch/Research/Performance/LibLSB/`

Baseline

`https://en.wikipedia.org/wiki/Otakar_Bor%C5%AFvka`

Borůvka's serial algorithm

$O(E \cdot log(V))$

Bader, D. A. and G. Cong (2006). "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs". In: *Journal of Parallel and Distributed Computing* 66.11, pp. 1366–1378.

Chung, S. and A. Condon (1996). "Parallel Implementation of\Boruvka". In: *ipps*. IEEE, p. 302.

Osipov, V., P. Sanders, and J. Singler (2009). "The Filter-Kruskal Minimum Spanning Tree Algorithm". In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. New York, New York: Society for Industrial and Applied Mathematics, pp. 52–61.