# Classical and Modern Approaches for Solving the Steiner Tree Problem

Teodora Dobos

Technical University of Munich

June 21, 2022

# Steiner Tree Problem

Given:

- $G = (V, E)$ undirected
- cost function $c : E \to \mathbb{Q}^+$
- terminals set $R \subseteq V$

Objective:

- find tree that connects all terminals and has minimum weight

1. Repetitive Shortest Path Heuristic
2. Kou-Markowsky-Berman Algorithm (Metric Closure)
3. Mehlhorn Algorithm
4. Primal-Dual Method

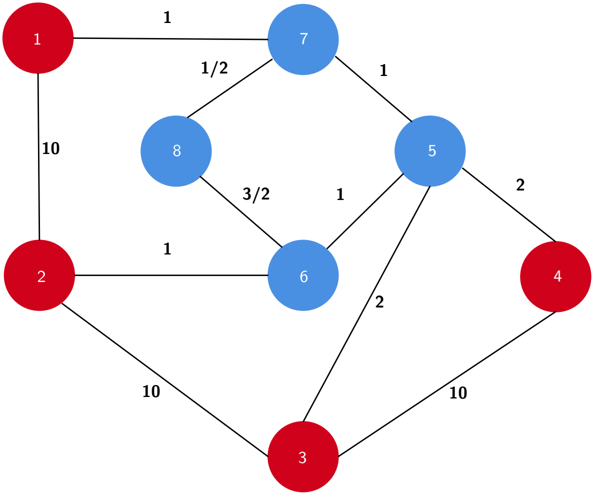
## Classical Approaches Overview

1. Repetitive Shortest Path Heuristic $\rightarrow$ implemented $\checkmark$
2. Kou-Markowsky-Berman Algorithm $\rightarrow$ NetworkX package
3. Mehlhorn Algorithm $\rightarrow$ implemented $\checkmark$
4. Primal-Dual Method $\rightarrow$ implemented $\checkmark$

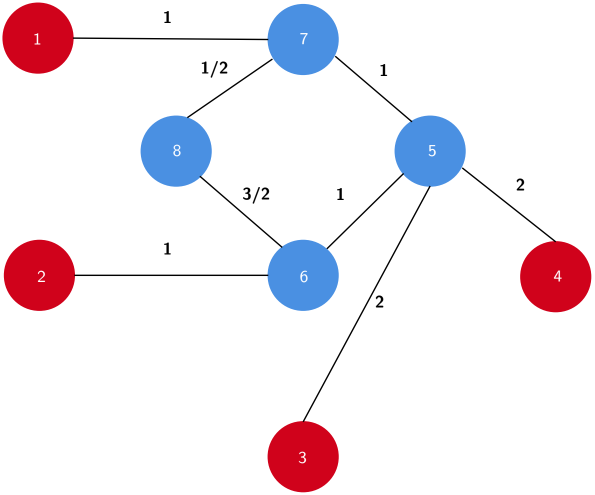
- $2\left(1 - \frac{1}{|R|}\right)$ approximation algorithms

# Repetitive Shortest Path Heuristic

# Input graph

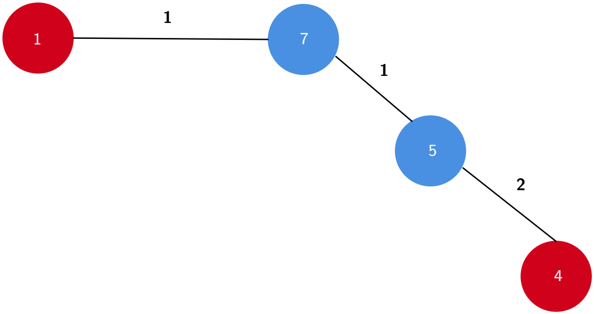


# Compute shortest paths with sources $r \in R$

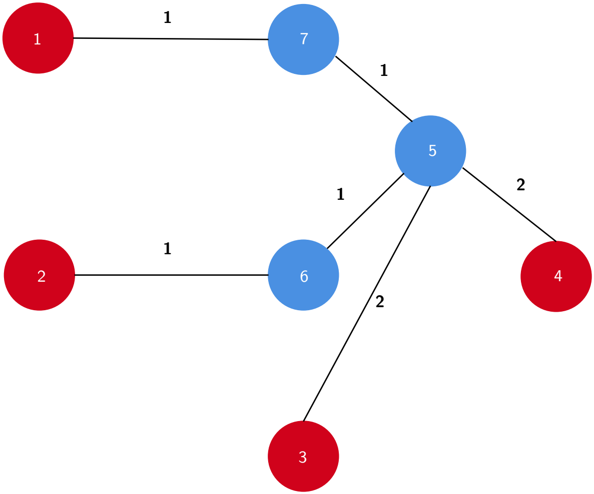


# Choose a random terminal: $T = (\{r\}, \emptyset)$

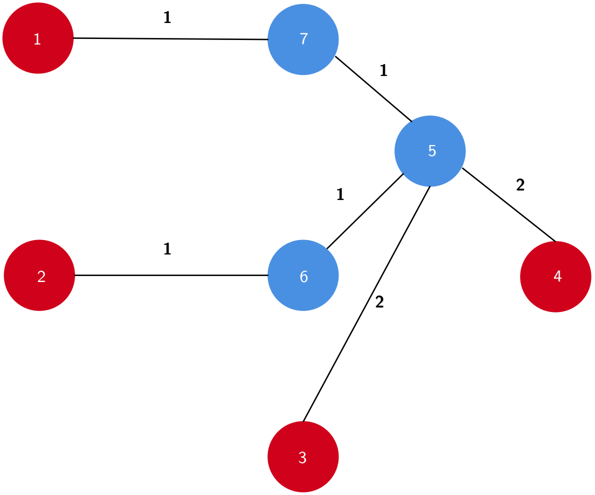
## Choose closest $u \in R$ and connect it to the component via its shortest path



# Repeat until $T$ contains all terminals



# Remove all non-terminals with degree 1 (no change in this example)

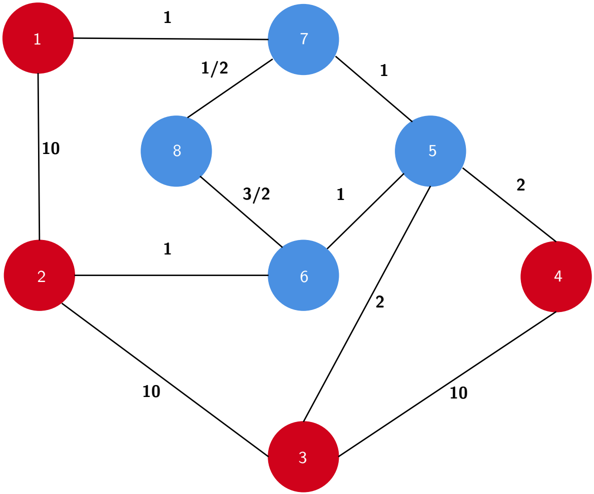


# Repetitive Shortest Path Heuristic Summary

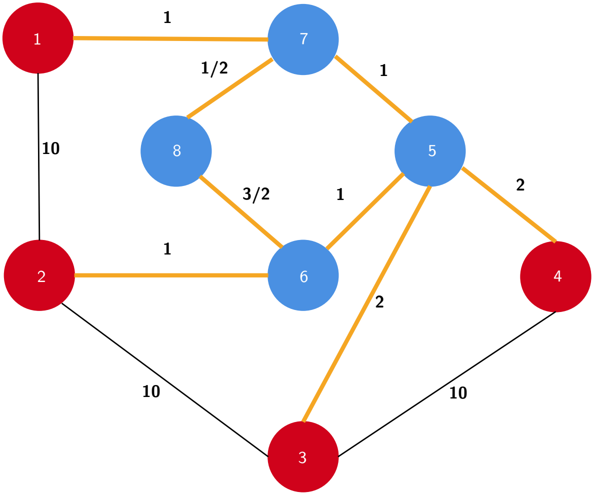
1. compute shortest paths with sources $r \in R$
2. $T = (\{r\}, \emptyset)$ with arbitrary $r \in R$
3. grow $T$ to obtain a feasible solution
   3.1 choose closest $u \in R$
   3.2 add minimum path $u \to T$
4. remove all non-terminals with degree 1

# Kou-Markowsky-Berman Algorithm (Metric Closure)

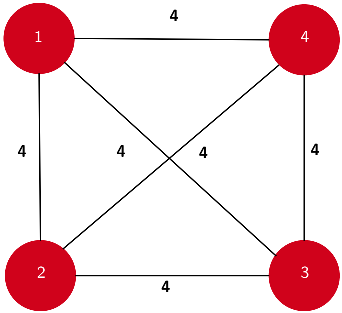
# Input graph $G$



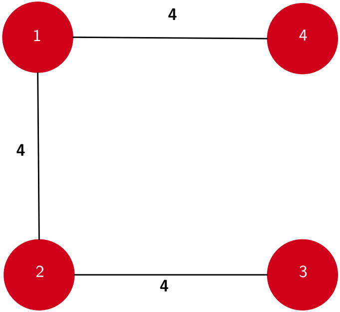
# Compute the metric closure of $G$



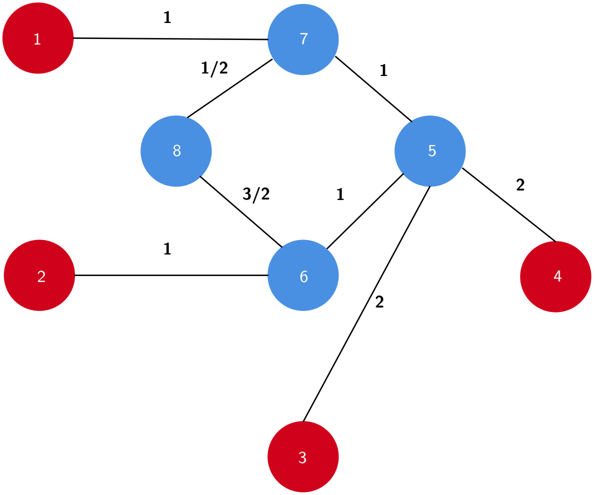
# Metric closure $G_1$



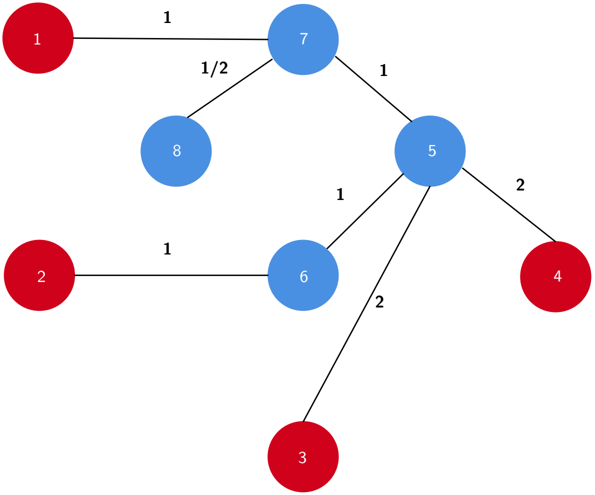
## Compute a minimum spanning tree $G_2 = MST(G_1)$



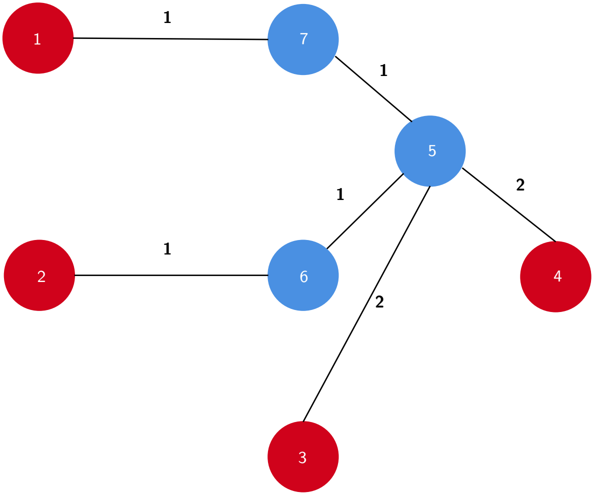
# Replace edges in $G_2$ with corresp. shortest paths in $G \to G_3$



# Compute a minimum spanning tree $G_4 = MST(G_3)$



## Delete unnecessary edges

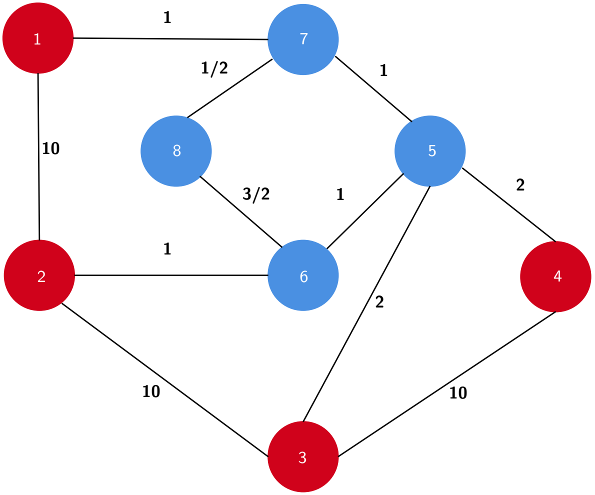


# Kou-Markowsky-Berman Algorithm Summary

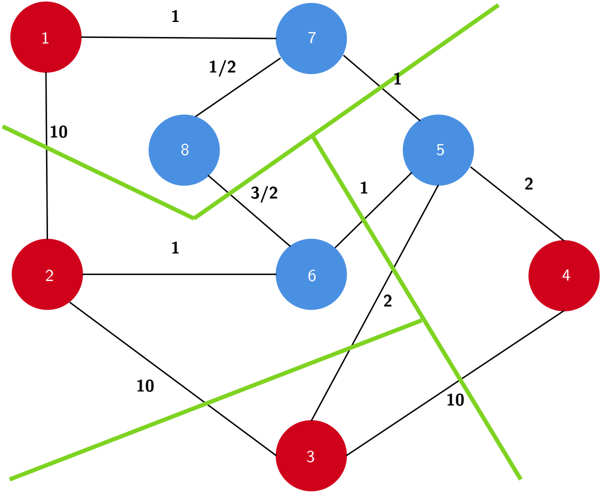
1. compute metric closure $\rightarrow G_1 := (R, E_1, d_1)$
2. compute minimum spanning tree $\rightarrow G_2 := MST(G_1)$
3. replace edges in $G_2$ by their shortest paths in $G$ $\rightarrow G_3$
4. compute minimum spanning tree $\rightarrow G_4 := MST(G_3)$
5. delete unnecessary edges $\rightarrow T$

# Mehlhorn Algorithm

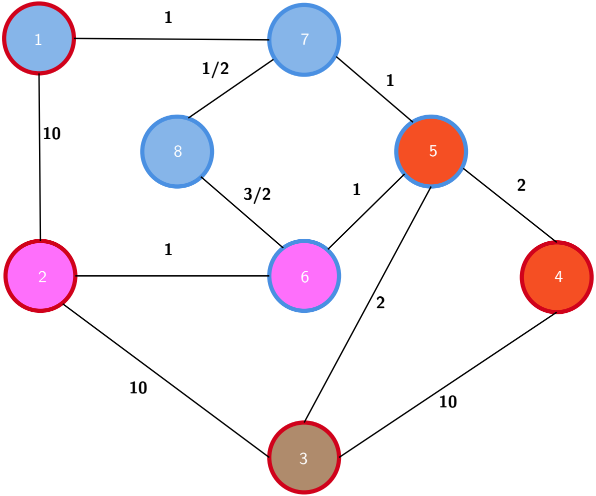
# Input graph



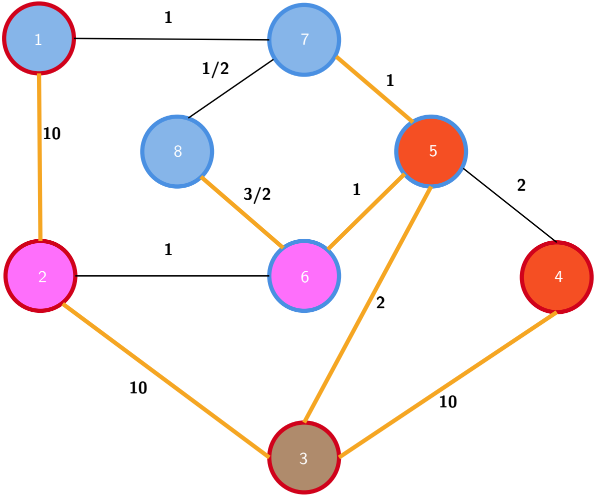
# Partition $V \to$ Voronoi regions



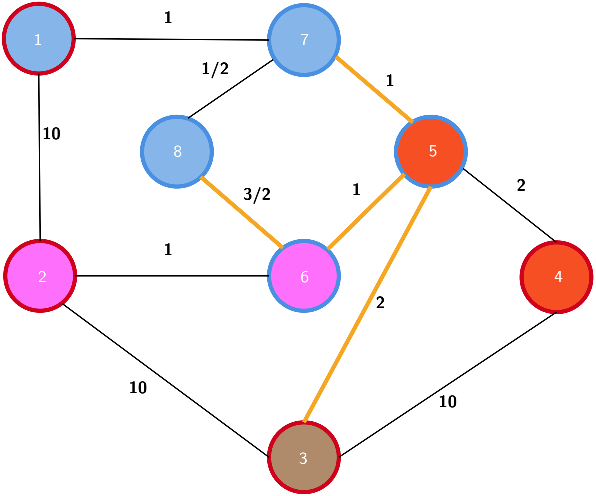
## Partition $V \to$ Voronoi regions



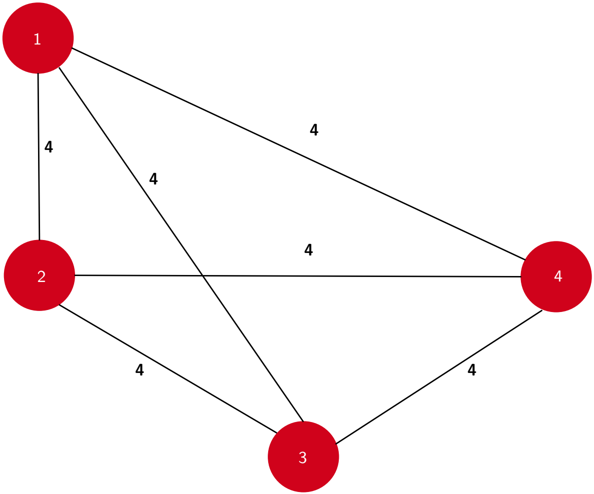
## Consider all interregional edges



# Choose $|R|$ interregional edges that lead to the shortest interterminal paths



# Create a new graph $G'$ induced by terminals



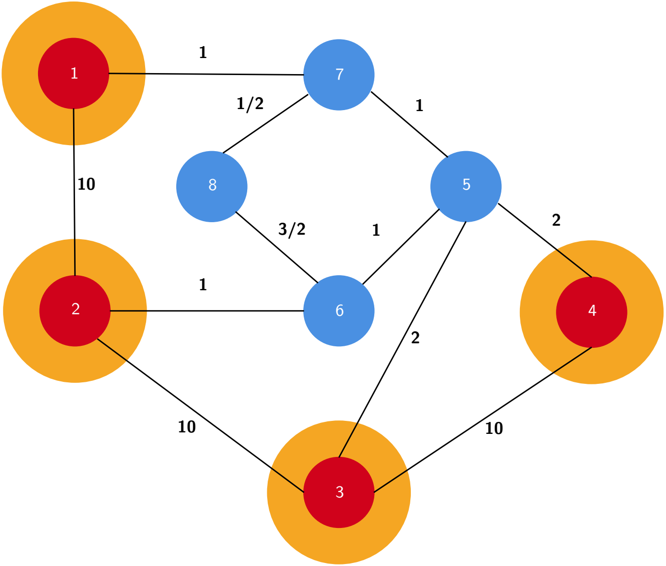
- compute minimum spanning tree $\rightarrow G_2 := MST(G')$
- replace edges in $G_2$ by their shortest paths in $G$ $\rightarrow G_3$
- compute minimum spanning tree $\rightarrow G_4 := MST(G_3)$
- delete unnecessary edges $\rightarrow T$

# Mehlhorn Algorithm Summary

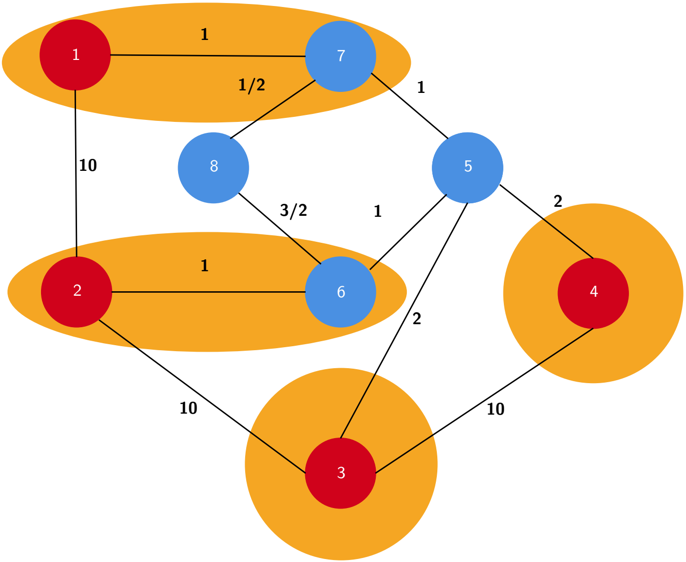
1. construct $G'$ as before $\rightarrow G'$
2. compute minimum spanning tree $\rightarrow G_2 := MST(G')$
3. replace edges in $G_2$ by their shortest paths in $G$ $\rightarrow G_3$
4. compute minimum spanning tree $\rightarrow G_4 := MST(G_3)$
5. delete unnecessary edges $\rightarrow T$

# Primal-Dual Algorithm

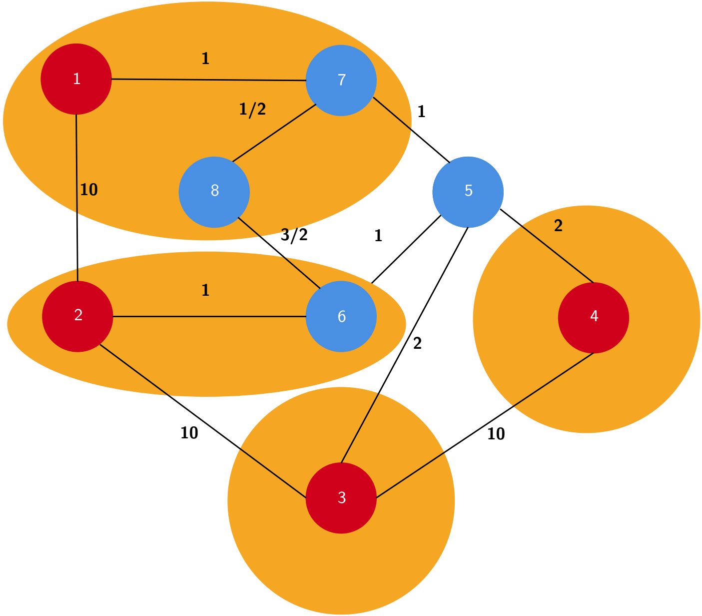
# Active sets $\mathcal{A}_r$ for all terminals $r \in R$



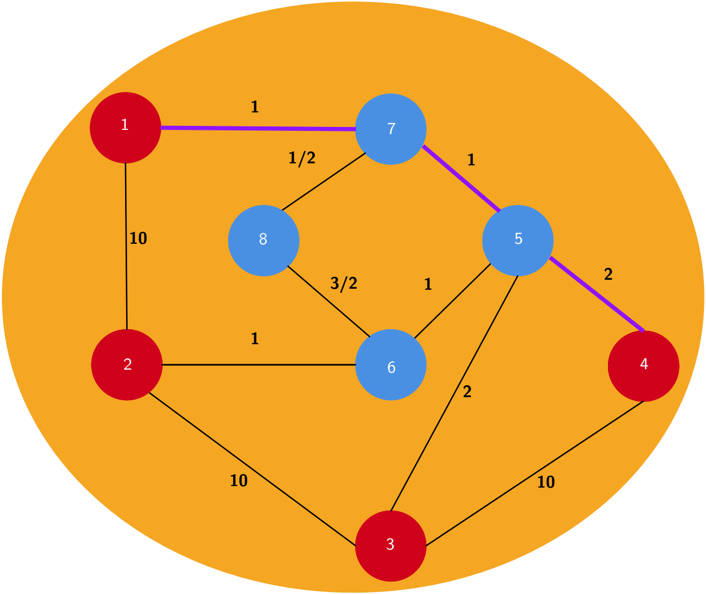
# Tight edge: add node to active set



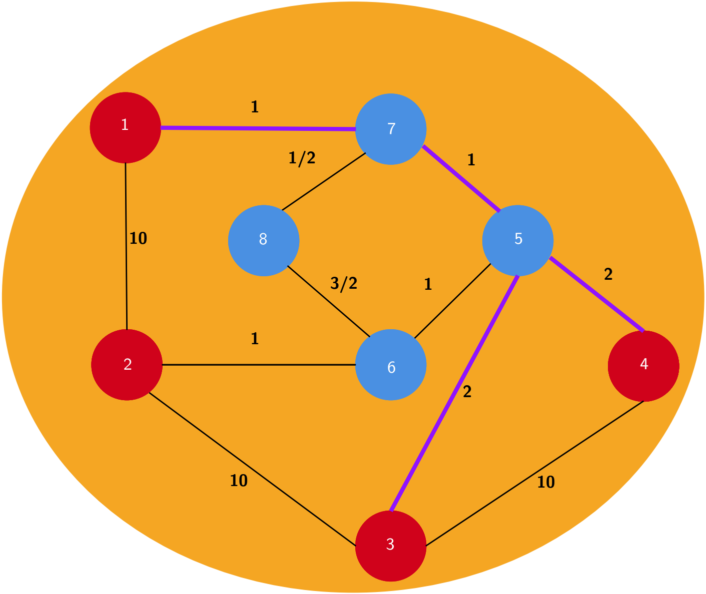
# Tight edge: add node to active set



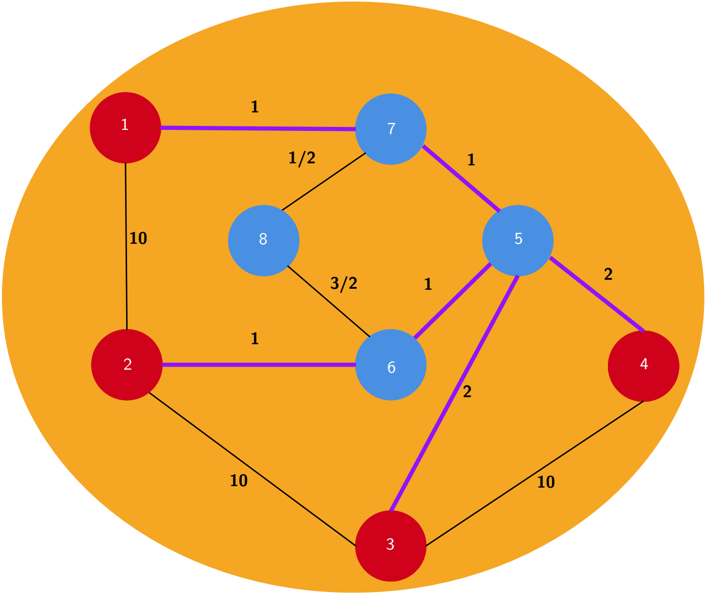
## Merge active sets - find connecting path & add it to the primal solution



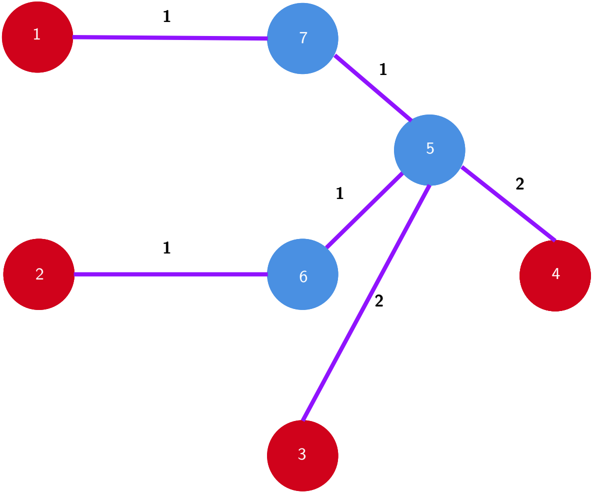
# Merge active sets - find connecting path & add it to the primal solution



# Merge active sets - find connecting path & add it to the primal solution



# Resulting Steiner tree



# Deep Reinforcement Learning & Cherrypick

# From Q-learning...

Q-learning:

- agent learns an action-utility function (Q-function) $\rightarrow$ expected utility of taking an action in a given state: **Q(state, action)**
- create & update a Q-table (fixed size!)

Q-learning:

- agent learns an action-utility function (Q-function) $\rightarrow$ expected utility of taking an action in a given state: **Q(state, action)**

- create & update a Q-table (fixed size!)

Steiner Tree setting:

- **state** $:=$ nodes included in the current solution

- **action** $:=$ add an uncovered node to the state

# From Q-learning...

Q-learning:

- agent learns an action-utility function (Q-function) $\rightarrow$ expected utility of taking an action in a given state: **Q(state, action)**
- create & update a Q-table (fixed size!)

Steiner Tree setting:

- **state** := nodes included in the current solution
- **action** := add an uncovered node to the state

$\implies$ large # possibilities **(state, action)**

$\implies$ creating & maintaining a Q-table is extremely difficult

Solution: **deep reinforcement learning:**

- use a deep neural network (NN) to approximate the Q-value function
- does **not** compute all **Q(state, action)**
- computes Q(action), i.e. Q(node)

- iteratively grows a connected component $C$ by adding to it a vertex which:
  - is connected to $C$ in $G$
  - has the maximum Q-value

# Cherrypick - Idea

- iteratively grows a connected component $C$ by adding to it a vertex which:
  - is connected to $C$ in $G$
  - has the maximum Q-value
- two main parts:
  1. **Graph Embedding Model** $\implies$ 'manipulable' graph format
  2. **Deep Q Network** $\rightarrow$ learn Q-values

- **similarity matrix** $\rightarrow$ relationship btw. each node and the 'special' $k$ terminals

- **similarity matrix** $\rightarrow$ relationship btw. each node and the 'special' $k$ terminals
- **message passing** $\rightarrow$ obtain **vertex embeddings** for all $v \in V$ considering:
  - similarity matrix
  - neighbors' embeddings
  - $v \in R$?
  - $v$ in the current solution?
- use vertex embeddings to compute **Q-values** for all $v \in V$

# Cherrypick - Graph Embedding Model

- **similarity matrix** $\rightarrow$ relationship btw. each node and the 'special' $k$ terminals
- **message passing** $\rightarrow$ obtain **vertex embeddings** for all $v \in V$ considering:
  - similarity matrix
  - neighbors' embeddings
  - $v \in R$?
  - $v$ in the current solution?
- use vertex embeddings to compute **Q-values** for all $v \in V$

The vertex embeddings and the Q-values are computed using <u>learnable</u> weights!

## Cherrypick - Deep Q Network (DQN)

- learn the Q-values (i.e. optimize the weights) using DQN
- solve a *standard* reinforcement learning task
- adding a node $v$ to the current solution gives a reward that depends on:
  - $v \in R$?
  - row($v$) in the similarity matrix

How do we compute a Steiner tree?

# Cherrypick - Overview

1. Graph embedding model $\implies$ initial Q-values for all nodes
2. While not all terminals are covered:
   2.1 Pick node based on policy & add to the current solution
   2.2 **If learning mode:**
      2.2.1 Optimize loss function $\implies$ update weights used in the graph embedding model & Q-values
      2.2.2 Recompute similarity matrix
      2.2.3 Recalculate Q-values
3. Remove non-terminals with degree 1

1. Graph embedding model $\implies$ initial Q-values for all nodes
2. While not all terminals are covered:
   2.1 Pick node based on policy & add to the current solution
   2.2 **If learning mode:**
      2.2.1 Optimize loss function $\implies$ update weights used in the graph embedding model & Q-values
      2.2.2 Recompute similarity matrix
      2.2.3 Recalculate Q-values
3. Remove non-terminals with degree 1

Special ingredients: 2 DQNs, replay memory (details in the documentation)

# Evaluation

1. Approximation ratio
2. Running time
3. Early stopping mechanism (Primal-Dual Algorithm)
4. Transfer learning (Cherrypick)
5. Weight sharing (Cherrypick)
6. Sequential learning (Cherrypick)

# Evaluation Dataset

1. PACE 2018 Challenge
2. small generated graphs (complete, grid, wheel, ladder)

# Approximation Ratio - all algorithms - less than 400 vertices

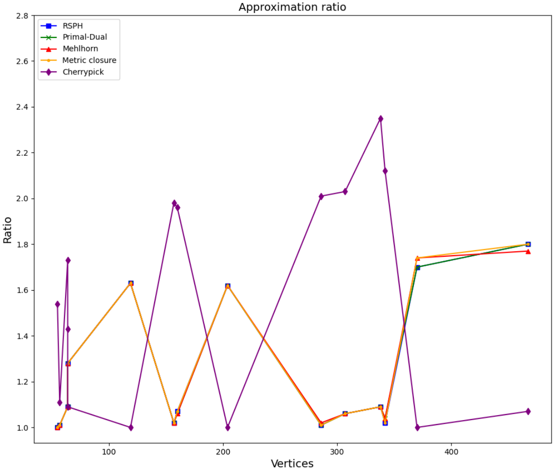



Figure: PACE Challenge Results.

# Approximation Ratio - approximation algorithms - less than 7000 vertices

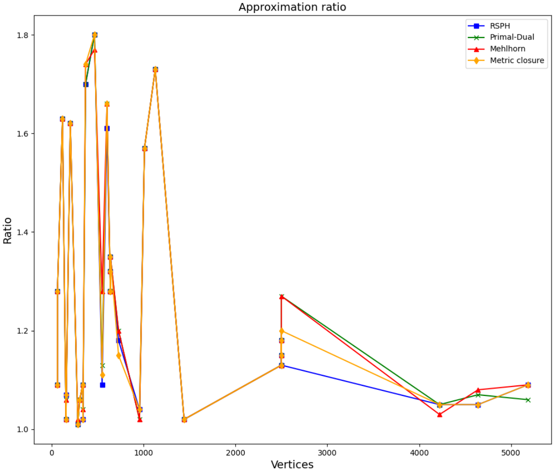



Figure: PACE Challenge Results.

# Approximation Ratio - all algorithms - less than 7000 vertices

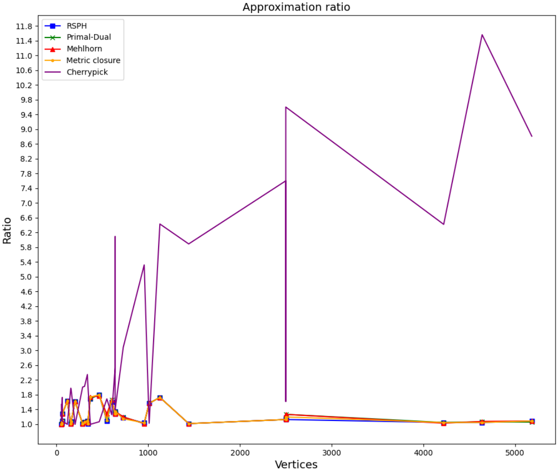



Figure: PACE Challenge Results.

Is it always that bad?

# Approximation Ratio - Cherrypick

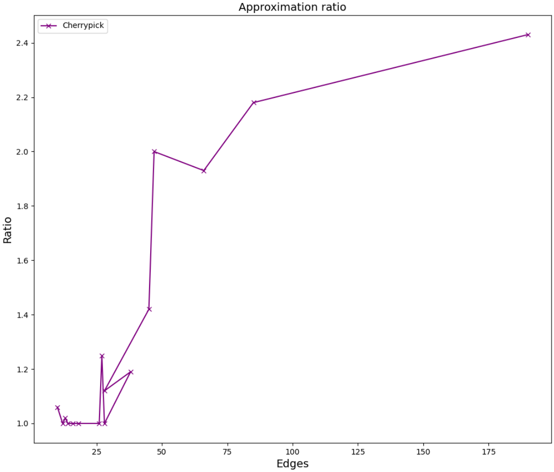



Figure: Results for small graphs.

What would we expect?

| **Algorithm** | **Complexity** |
|---|---|
| Mehlhorn | $O(\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert)$ |
| Repetitive Shortest Path | $O(\lvert R\rvert(\lvert E\rvert + \lvert V\rvert \log \lvert V\rvert))$ |
| KMB (Metric closure) | $O(\lvert R\rvert(\lvert E\rvert + \lvert V\rvert \log \lvert V\rvert))$ |
| Primal-Dual | $O(\lvert V\rvert\lvert E\rvert \log \lvert E\rvert)$ |

Table: Complexities of the approximation algorithms

# Running Time - Algorithmic Complexities

| Algorithm | Complexity |
|---|---|
| Mehlhorn | $O(\|V\|\log\|V\| + \|E\|)$ |
| Repetitive Shortest Path | $O(\|R\|(\|E\| + \|V\|\log\|V\|))$ |
| KMB (Metric closure) | $O(\|R\|(\|E\| + \|V\|\log\|V\|))$ |
| Primal-Dual | $O(\|V\|\|E\|\log\|E\|)$ |

Table: Complexities of the approximation algorithms

Mehlhorn $<$ Repetitive Shortest Path $=$ KMB (Metric closure) $<$ Primal-Dual

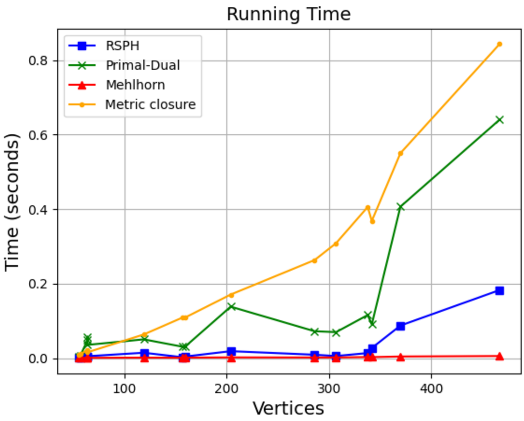



Figure: PACE Challenge Results.

# Running Time - approximation algorithms - less than 7000 nodes

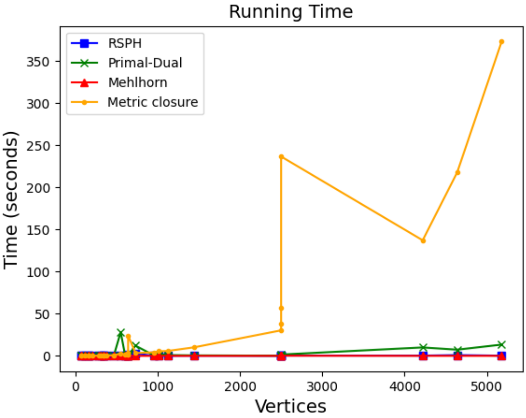



Figure: PACE Challenge Results.

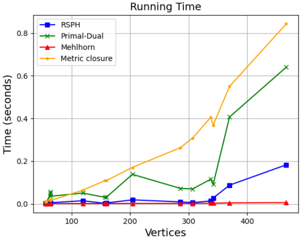
**Theory:**

Mehlhorn $<$ Repetitive Shortest Path $=$ KMB (Metric closure) $<$ Primal-Dual

**Theory:**

Mehlhorn $<$ Repetitive Shortest Path $=$ KMB (Metric closure) $<$ Primal-Dual

**Practice:**



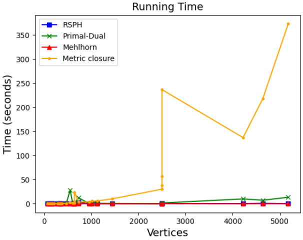
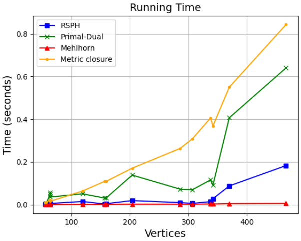




Figure: PACE Challenge Results.

**Theory:**

Mehlhorn $<$ Repetitive Shortest Path $=$ KMB (Metric closure) $<$ Primal-Dual

**Practice:**



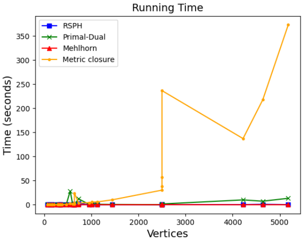




Figure: PACE Challenge Results.

**Primal-Dual faster than KMB (Metric closure)?**

# Running Time - Primal-Dual Algorithm

- edge sorting - most expensive step
- sorting algorithm? $\rightarrow$ Timsort (default in Python)
  - efficient
  - takes advantage of any ordering already present in a collection

## Running Time - Primal-Dual Algorithm

- edge sorting - most expensive step
- sorting algorithm? $\rightarrow$ Timsort (default in Python)
  - efficient
  - takes advantage of any ordering already present in a collection
- Primal-Dual Algorithm:
  - subsets of ordered edges are likely to be present in the set of edges
  - the same ‘amount’ is added to all dual variables in each iteration

## Running Time - Primal-Dual Algorithm

- edge sorting - most expensive step
- sorting algorithm? $\rightarrow$ Timsort (default in Python)
  - efficient
  - takes advantage of any ordering already present in a collection
- Primal-Dual Algorithm:
  - subsets of ordered edges are likely to be present in the set of edges
  - the same ‘amount’ is added to all dual variables in each iteration

$\rightarrow$ efficient sorting

Early Stopping Mechanism for Primal-Dual Algorithm

# Early Stopping Mechanism - Primal-Dual Algorithm

Idea:

- stop each iteration with probability $p = 0.3$ if at least two active components were already merged
- make solution feasible:
  - merge, iteratively, the largest two components $C_1$, $C_2$ $\rightarrow$ add shortest path $r_1 \rightarrow r_2$ with $r_i \in C_i, i \in \{1, 2\}$
- combination Primal-Dual Algorithm & Repetitive Shortest Path Heuristic

# Early Stopping Mechanism - Primal-Dual Algorithm - less than 400 vertices

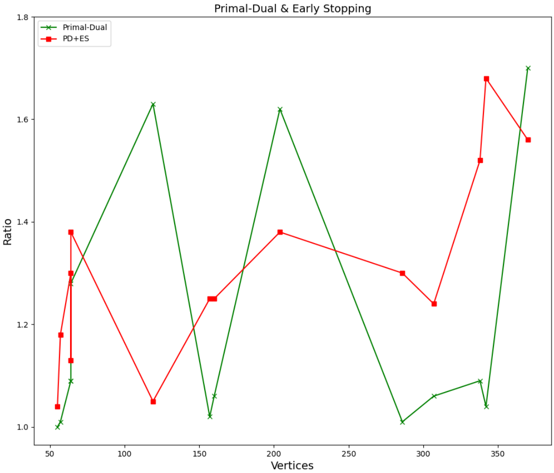



Figure: PACE Challenge Results.

# Early Stopping Mechanism - Primal-Dual Algorithm - less than 7000 vertices

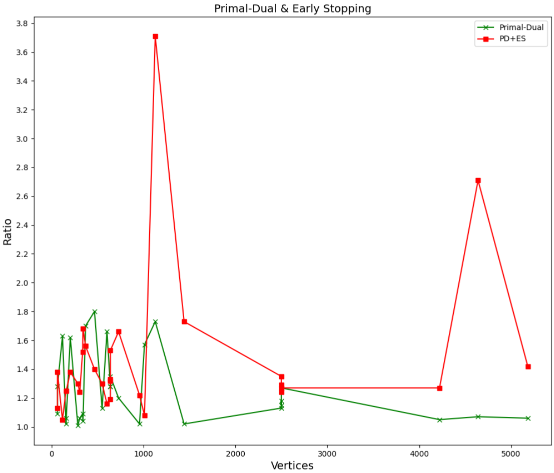



Figure: PACE Challenge Results.

# Early Stopping Mechanism - Primal-Dual Algorithm - Comments

- good results for small graphs, but no guarantees...
- drawbacks:
  - does not take into consideration the ‘progress’ made between the last iteration in which two components were merged and the iteration that is stopped early
  - shortest path computation

# Cherrypick evaluation scenarios

Consider a graph $G$ and a subgraph $G' \subset G$.

- randomly initialize weights to solve instance given by $G \to$ Q-values $Q$
- set $Q$ as the initial Q-values when solving instance given by $G'$
- apply Cherrypick

## Transfer Learning - Cherrypick

Consider a graph $G$ and a subgraph $G' \subset G$.

- randomly initialize weights to solve instance given by $G \to$ Q-values $Q$
- set $Q$ as the initial Q-values when solving instance given by $G'$
- apply Cherrypick

Idea:

- use learned global knowledge locally

| Instance | $\|V(G)\|$ | $\|V(G')\|$ | $\|E(G)\|$ | $\|E(G')\|$ | **Tr.L.** | **No Tr.L.** |
|---|---|---|---|---|---|---|
| 1 | 17 | 13 | 24 | 16 | 2.5 | 2.8 |
| 2 | 17 | 13 | 24 | 13 | 1 | 1.6 |
| 3 | 19 | 14 | 32 | 20 | 1.71 | 3.23 |
| 4 | 19 | 14 | 32 | 20 | 3.29 | 2.52 |
| 5 | 28 | 28 | 63 | 51 | 2.46 | 1.82 |
| 6 | 28 | 20 | 63 | 37 | 2.24 | 2.71 |
| 7 | 12 | 8 | 26 | 13 | 1 | 3 |
| 8 | 12 | 10 | 23 | 15 | 1 | 1 |
| 9 | 21 | 17 | 27 | 21 | 1 | 1 |
| 10 | 12 | 10 | 26 | 17 | 1 | 3 |

Table: **Subgraph** results obtained with & without the transfer learning approach.

Idea:

- improve the initial solution computed for an instance in multiple, independent algorithm executions
- 1st run: randomly initialize weights and solve the instance $\rightarrow$ Q-values $Q$
- next run: use $Q$ as the initial Q-values and solve the same instance
- repeat for a fixed number of times (4)

# Weight Sharing - Cherrypick

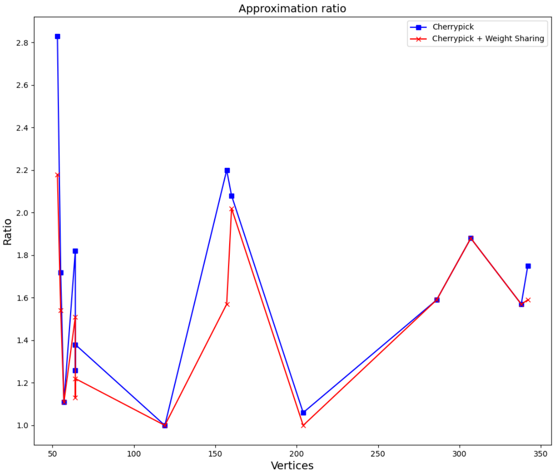



Figure: PACE Challenge Results.

# Summary (1)

- Classical approaches achieve similar approximation ratios in practice
- Mehlhorn is the fastest algorithm in theory & practice
- KMB (Metric closure) is the slowest in practice (among the classical approaches)
- Primal-Dual is faster than KMB (Metric closure) in practice due to Timsort

## Summary (2)

- Cherrypick approximation ratios grow with the number of edges
- Primal-Dual - Early stopping mechanism works well for small graphs
- Cherrypick - Transfer learning approach leads to good results
- Cherrypick - Weight sharing approach works well, but is computationally expensive