Mathematical Tools for Engineering and Management

Lecture 8

7 Dec 2011

- ▷ Models, Data and Algorithms
- ▷ Linear Optimization
- ▷ Mathematical Background: Polyhedra, Simplex-Algorithm
- ▷ Sensitivity Analysis; (Mixed) Integer Programming
- ▷ MIP Modelling
- ▷ MIP Modelling: More Examples; Branch & Bound
- ▷ Cutting Planes; Combinatorial Optimization: Examples, Graphs, Algorithms
- ▷ TSP-Heuristics
- ▷ Complexity Theory
- ▷ Nonlinear Optimization
- ▷ Scheduling
- ▷ Lot Sizing
- ▷ Multicriteria Optimization
- ▷ Oral exam

- Nearest neighbour heuristic (greedy algorithm):

- Nearest neighbour heuristic (greedy algorithm):
  - Start with an arbitrary city

▷ Nearest neighbour heuristic (greedy algorithm):

- Start with an arbitrary city
- Append the city (or one of the cities) closest to the last visited city to the tour...

▷ Nearest neighbour heuristic (greedy algorithm):

- Start with an arbitrary city
- Append the city (or one of the cities) closest to the last visited city to the tour...
- ...until all cities are visited

- ▷ Nearest neighbour heuristic (greedy algorithm):
  - Start with an arbitrary city
  - Append the city (or one of the cities) closest to the last visited city to the tour...
  - ...until all cities are visited

➡ Straightforward and easy to implement

▷ Nearest neighbour heuristic (greedy algorithm):

- Start with an arbitrary city
- Append the city (or one of the cities) closest to the last visited city to the tour...
- ...until all cities are visited

➡ Straightforward and easy to implement

➡ But: might produce arbitrarily bad solutions – or even the worst possible tour!

▷ Nearest neighbour heuristic (greedy algorithm):

- Start with an arbitrary city
- Append the city (or one of the cities) closest to the last visited city to the tour...
- ...until all cities are visited

➡ Straightforward and easy to implement

➡ But: might produce arbitrarily bad solutions – or even the worst possible tour!

▷ Tour expansion heuristic:

▷ Nearest neighbour heuristic (greedy algorithm):

- Start with an arbitrary city
- Append the city (or one of the cities) closest to the last visited city to the tour...
- ...until all cities are visited

➡ Straightforward and easy to implement

➡ But: might produce arbitrarily bad solutions – or even the worst possible tour!

▷ Tour expansion heuristic:

- Order the cities in some way

▷ Nearest neighbour heuristic (greedy algorithm):

- Start with an arbitrary city
- Append the city (or one of the cities) closest to the last visited city to the tour...
- ...until all cities are visited

➡ Straightforward and easy to implement

➡ But: might produce arbitrarily bad solutions – or even the worst possible tour!

▷ Tour expansion heuristic:

- Order the cities in some way
- Start with the tour through the first two cities

▷ Nearest neighbour heuristic (greedy algorithm):

- Start with an arbitrary city
- Append the city (or one of the cities) closest to the last visited city to the tour...
- ...until all cities are visited

➡ Straightforward and easy to implement

➡ But: might produce arbitrarily bad solutions – or even the worst possible tour!

▷ Tour expansion heuristic:

- Order the cities in some way
- Start with the tour through the first two cities
- Insert the next city at the best position...

▷ Nearest neighbour heuristic (greedy algorithm):

- Start with an arbitrary city
- Append the city (or one of the cities) closest to the last visited city to the tour...
- ...until all cities are visited

➡ Straightforward and easy to implement

➡ But: might produce arbitrarily bad solutions – or even the worst possible tour!

▷ Tour expansion heuristic:

- Order the cities in some way
- Start with the tour through the first two cities
- Insert the next city at the best position...
- ...until all cities are inserted

▷ Nearest neighbour heuristic (greedy algorithm):

- Start with an arbitrary city
- Append the city (or one of the cities) closest to the last visited city to the tour...
- ...until all cities are visited

➡ Straightforward and easy to implement

➡ But: might produce arbitrarily bad solutions – or even the worst possible tour!

▷ Tour expansion heuristic:

- Order the cities in some way
- Start with the tour through the first two cities
- Insert the next city at the best position...
- ...until all cities are inserted

➡ Can be improved by smart ordering and definition of "best position"

▷ Nearest neighbour heuristic (greedy algorithm):

- Start with an arbitrary city
- Append the city (or one of the cities) closest to the last visited city to the tour...
- ...until all cities are visited

➡ Straightforward and easy to implement

➡ But: might produce arbitrarily bad solutions – or even the worst possible tour!

▷ Tour expansion heuristic:

- Order the cities in some way
- Start with the tour through the first two cities
- Insert the next city at the best position...
- ...until all cities are inserted

➡ Can be improved by smart ordering and definition of “best position”

➡ But: might also produce bad solutions in general

▷ Nearest neighbour heuristic (greedy algorithm):

- Start with an arbitrary city
- Append the city (or one of the cities) closest to the last visited city to the tour...
- ...until all cities are visited

➡ Straightforward and easy to implement

➡ But: might produce arbitrarily bad solutions – or even the worst possible tour!

▷ Tour expansion heuristic:

- Order the cities in some way
- Start with the tour through the first two cities
- Insert the next city at the best position...
- ...until all cities are inserted

➡ Can be improved by smart ordering and definition of "best position"

➡ But: might also produce bad solutions in general

▷ Try to prove quality of solution ➡ approximation algorithms

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E between the vertices

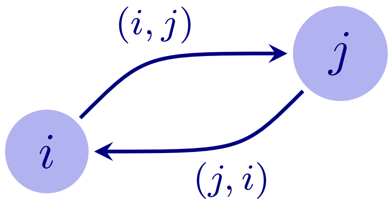
A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E between the vertices

▷ Directed graph: edges (or arcs) have a direction

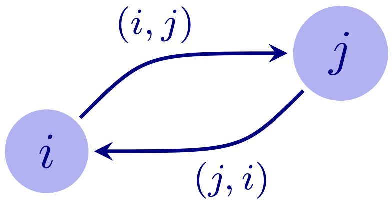
➡ $E \subseteq \{(i,j) \mid i,j \in V,\ i \neq j\}$

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E between the vertices

- Directed graph: edges (or arcs) have a direction
  - ➡ $E \subseteq \{(i,j) \mid i,j \in V,\, i \neq j\}$



A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E between the vertices

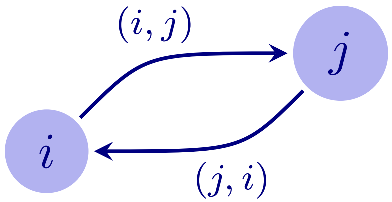


- Directed graph: edges (or arcs) have a direction
  - ➡ $E \subseteq \{(i,j) \mid i,j \in V,\ i \neq j\}$
- Undirected graph: edges have no direction
  - ➡ $E \subseteq \{\{i,j\} \mid i,j \in V,\ i \neq j\}$

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E between the vertices

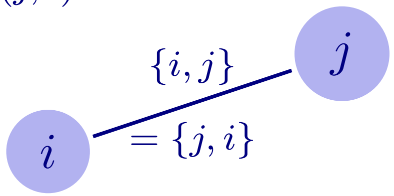
- Directed graph: edges (or arcs) have a direction

  ➡ $E \subseteq \{(i, j) \mid i, j \in V,\ i \neq j\}$



- Undirected graph: edges have no direction

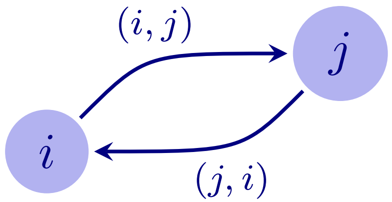
  ➡ $E \subseteq \{\{i, j\} \mid i, j \in V,\ i \neq j\}$



A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E between the vertices

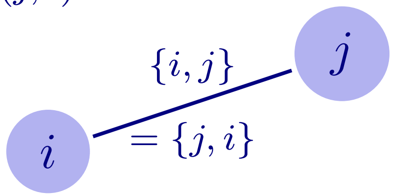
▷ Directed graph: edges (or arcs) have a direction

➡ $E \subseteq \{(i,j) \mid i,j \in V,\, i \neq j\}$



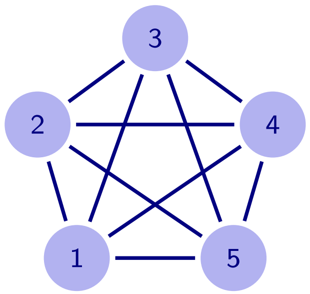
▷ Undirected graph: edges have no direction

➡ $E \subseteq \{\{i,j\} \mid i,j \in V,\, i \neq j\}$



▷ Examples:

complete graph
on n vertices: K_n

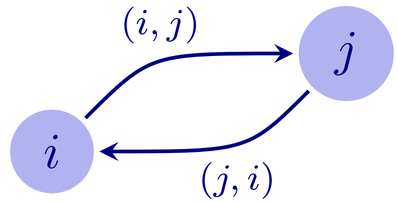


K_5

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E between the vertices

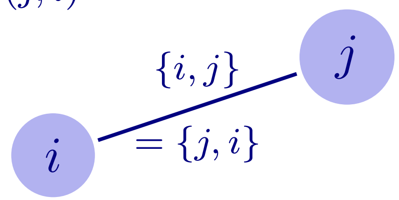
▷ Directed graph: edges (or arcs) have a direction

➡ $E \subseteq \{(i,j) \mid i,j \in V,\ i \neq j\}$



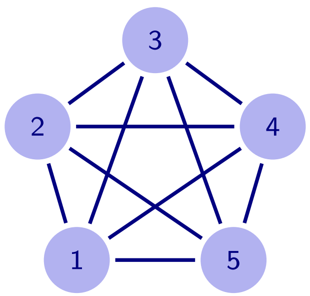
▷ Undirected graph: edges have no direction

➡ $E \subseteq \{\{i,j\} \mid i,j \in V,\ i \neq j\}$



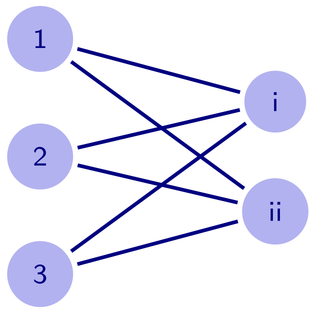
▷ Examples:

complete graph
on n vertices: K_n



K_5

complete bipartite graph:
$K_{m,n}$

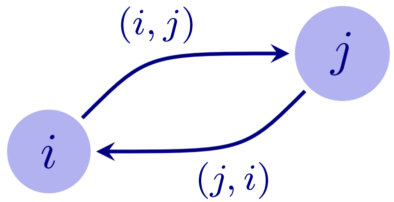


$K_{3,2}$

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E between the vertices

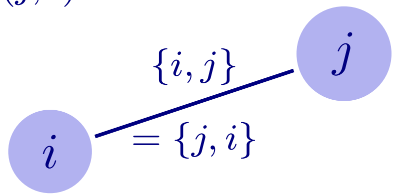
- Directed graph: edges (or arcs) have a direction

  ➡ $E \subseteq \{(i,j) \mid i,j \in V,\, i \neq j\}$



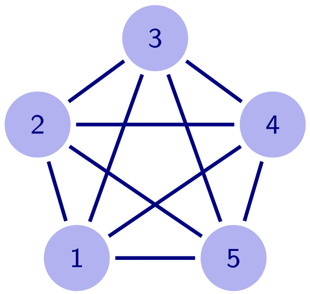
- Undirected graph: edges have no direction

  ➡ $E \subseteq \{\{i,j\} \mid i,j \in V,\, i \neq j\}$



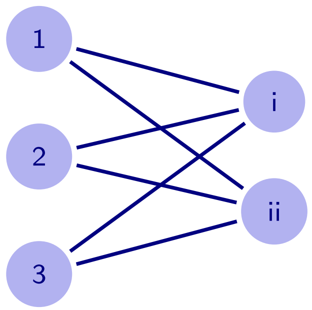
- Examples:

complete graph on n vertices: K_n



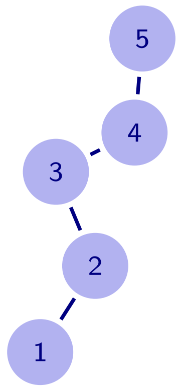
K_5

complete bipartite graph: $K_{m,n}$



$K_{3,2}$

path: P_n

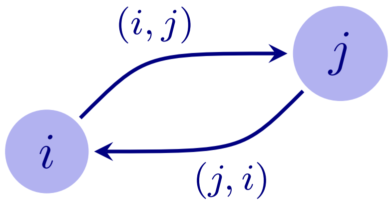


P_5

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E between the vertices

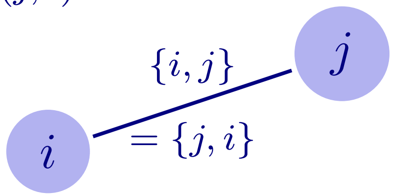
- Directed graph: edges (or arcs) have a direction

  ➡ $E \subseteq \{(i,j) \mid i,j \in V,\, i \neq j\}$



- Undirected graph: edges have no direction

  ➡ $E \subseteq \{\{i,j\} \mid i,j \in V,\, i \neq j\}$



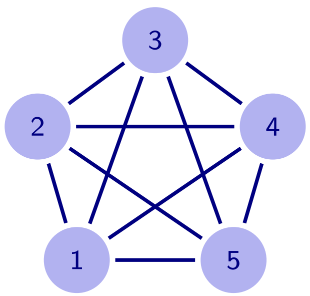
- Examples:

complete graph on n vertices: K_n

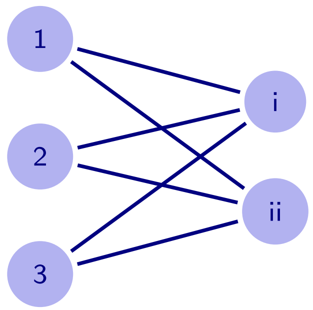
complete bipartite graph: $K_{m,n}$

path: P_n

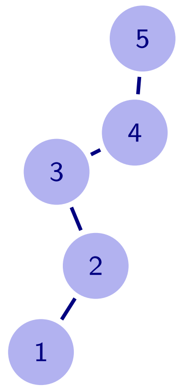
cycle on n vertices: C_n



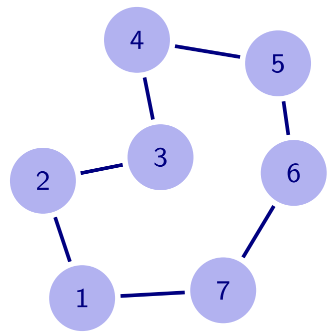
K_5



$K_{3,2}$



P_5



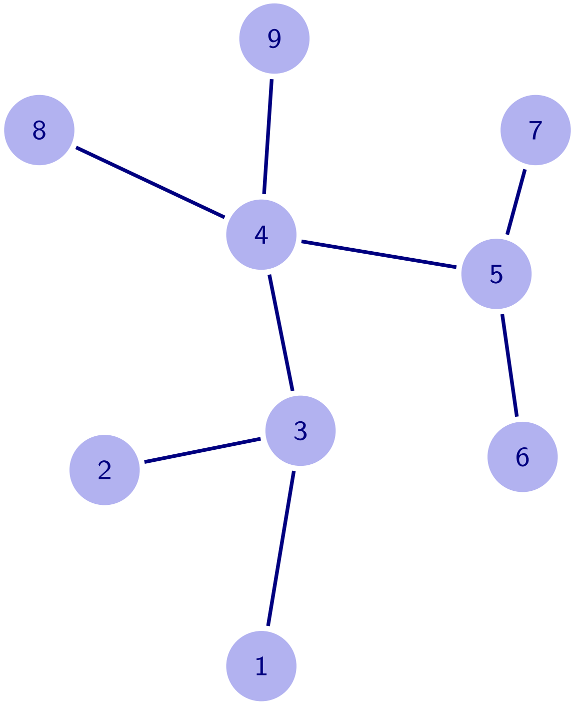
C_7

A tree T is a graph with the following properties:

- T contains no cycles
- T is connected (i.e. every two vertices can be connected by a path in T)
- There is exactly one more vertex than there are edges (i.e. $|V| = |E| + 1$)

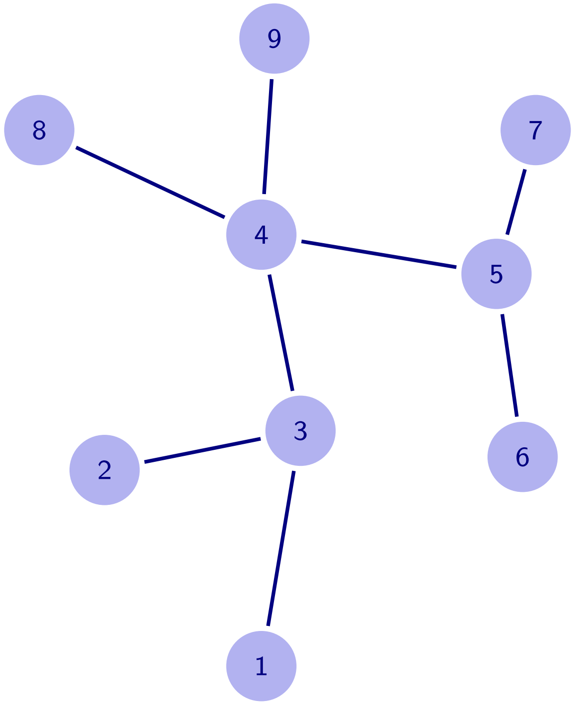
A tree T is a graph with the following properties:

- T contains no cycles
- T is connected (i.e. every two vertices can be connected by a path in T)
- There is exactly one more vertex than there are edges (i.e. $|V| = |E| + 1$)



A tree T is a graph with the following properties:

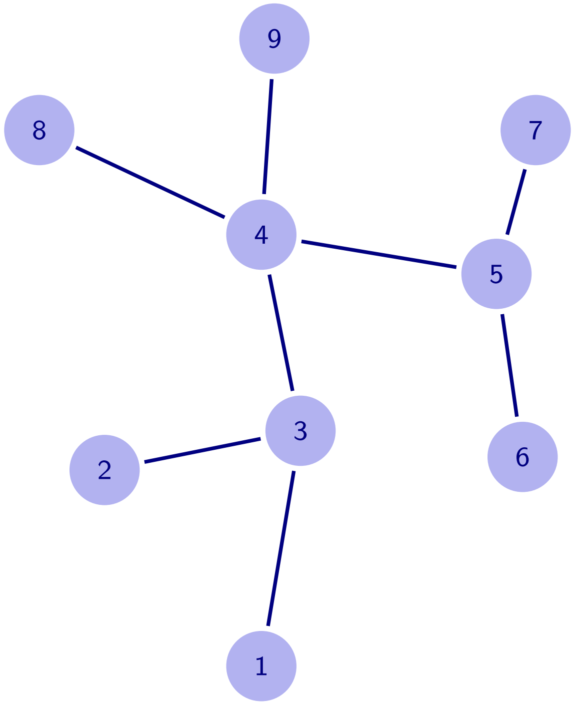
- T contains no cycles
- T is connected (i.e. every two vertices can be connected by a path in T)
- There is exactly one more vertex than there are edges (i.e. $|V| = |E| + 1$)



➡ Removing one edge makes the tree disconnected

A tree T is a graph with the following properties:

- T contains no cycles
- T is connected (i.e. every two vertices can be connected by a path in T)
- There is exactly one more vertex than there are edges (i.e. $|V| = |E| + 1$)

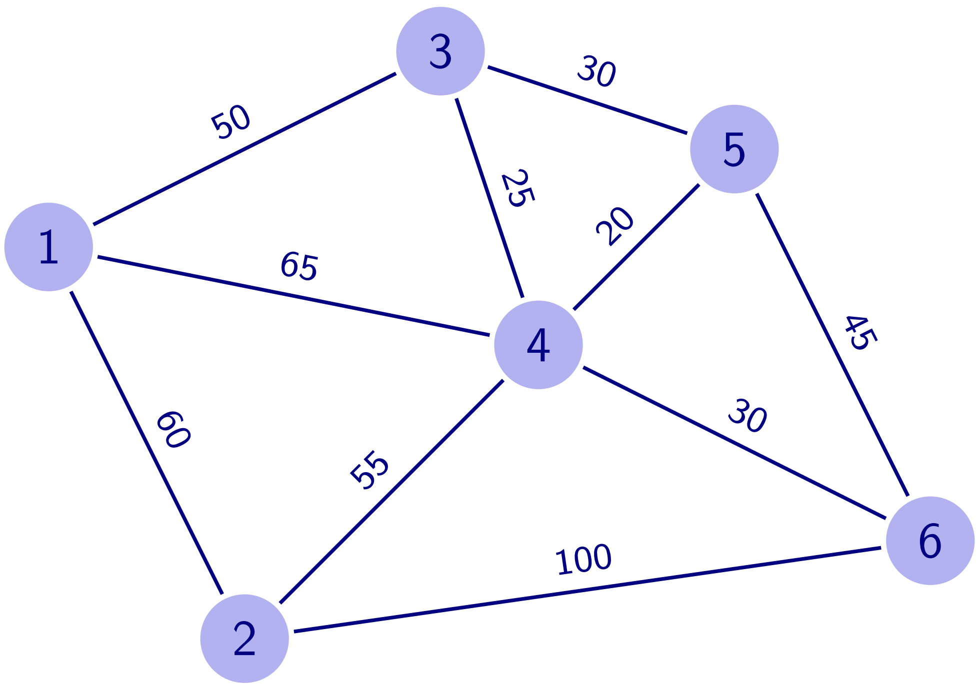


➡ Removing one edge makes the tree disconnected

➡ Adding a new edge creates a cycle

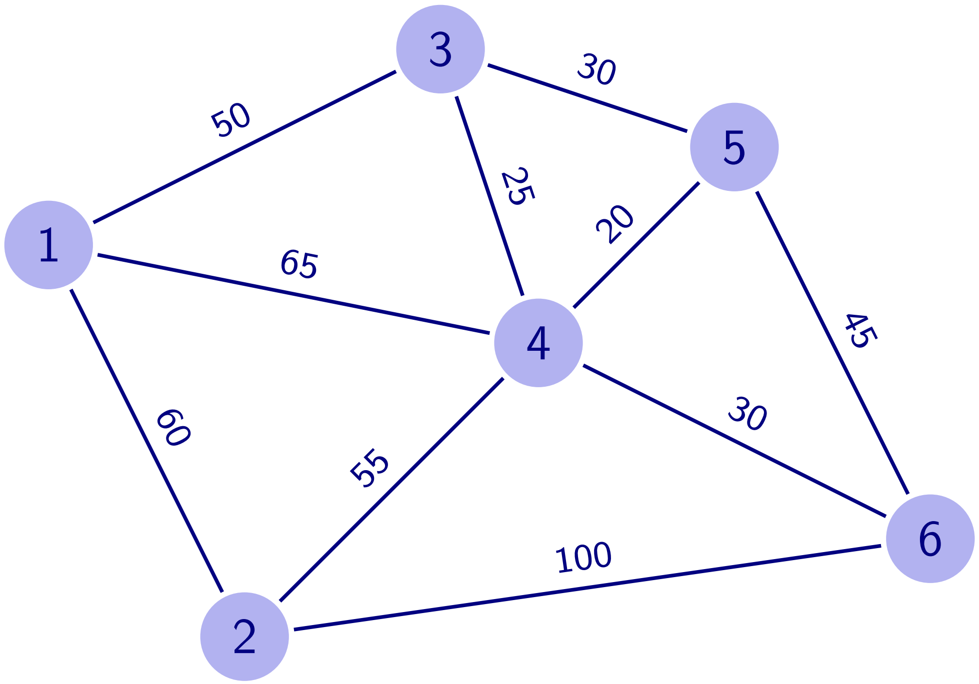
▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...



▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

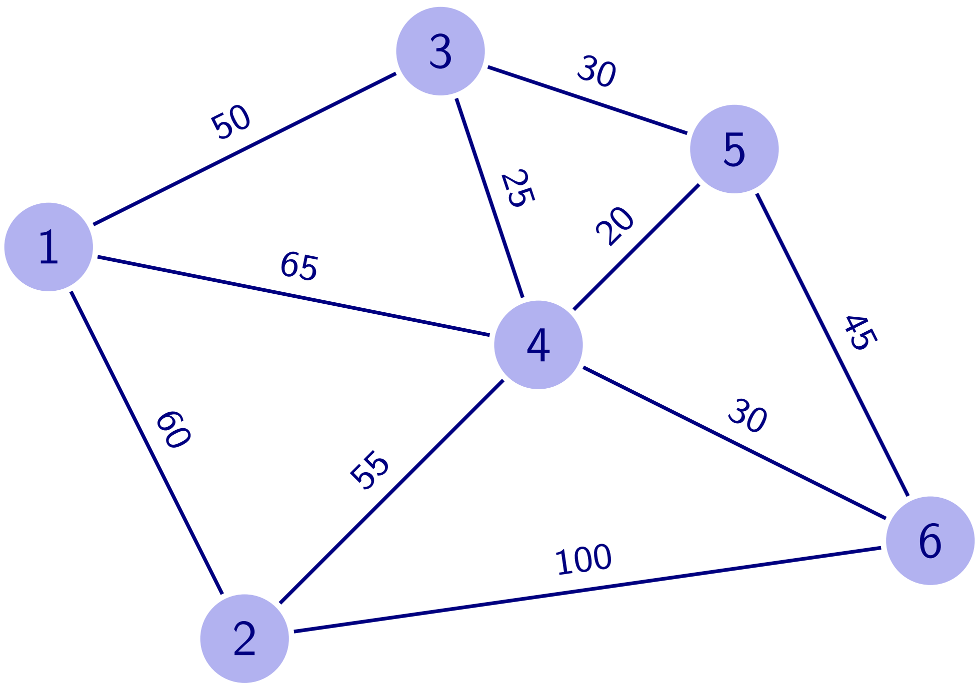
...find a minimum spanning tree for G



▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

...find a minimum spanning tree for G, that is: a subset E' of the edges such that

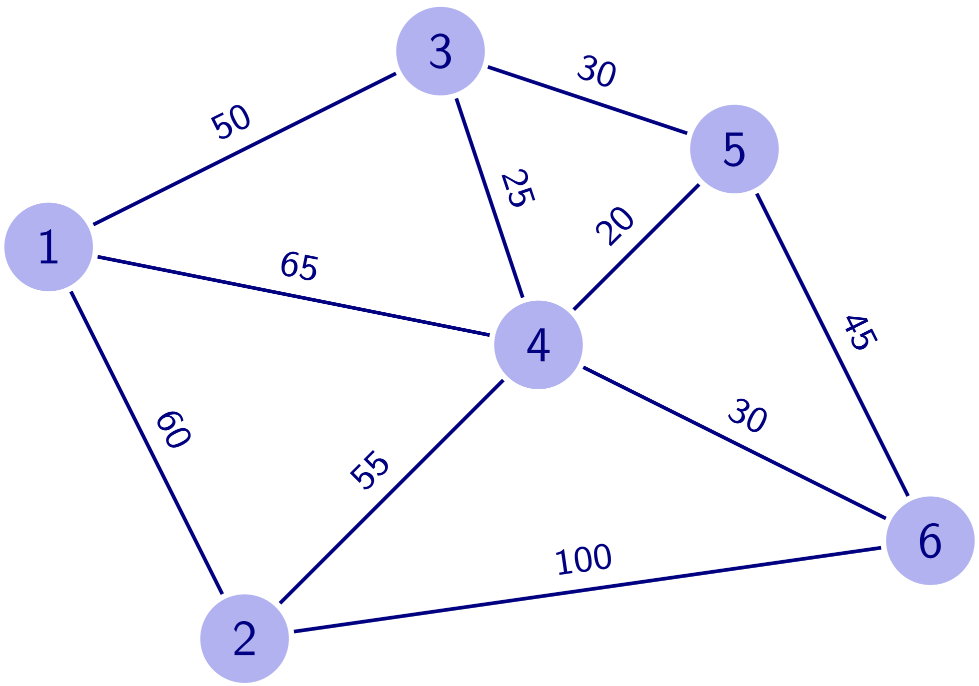
- the edges in E' form a tree



▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

...find a minimum spanning tree for G, that is: a subset E' of the edges such that

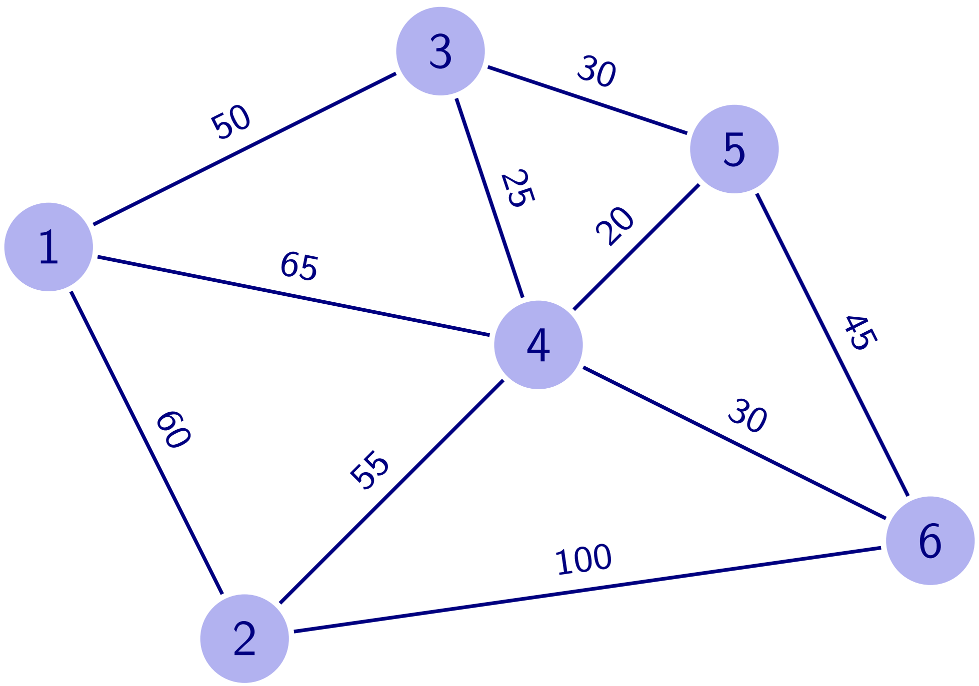
- the edges in E' form a tree
- all vertices of G are in the tree



▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

...find a minimum spanning tree for G, that is: a subset E' of the edges such that

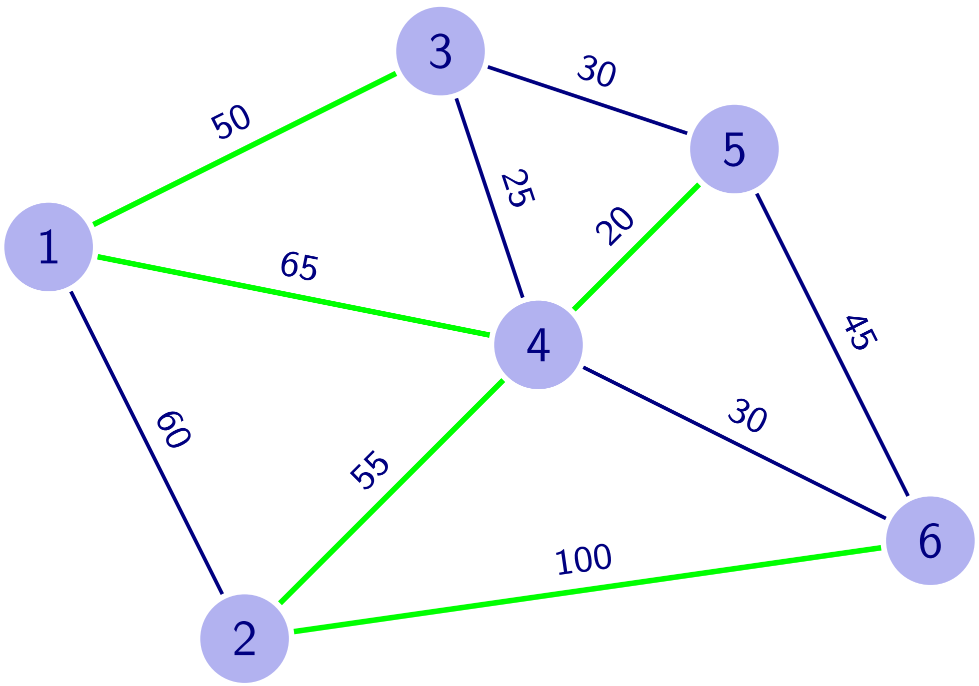
- the edges in E' form a tree
- all vertices of G are in the tree
- the total weight of the tree edges is minimal



▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

...find a minimum spanning tree for G, that is: a subset E' of the edges such that

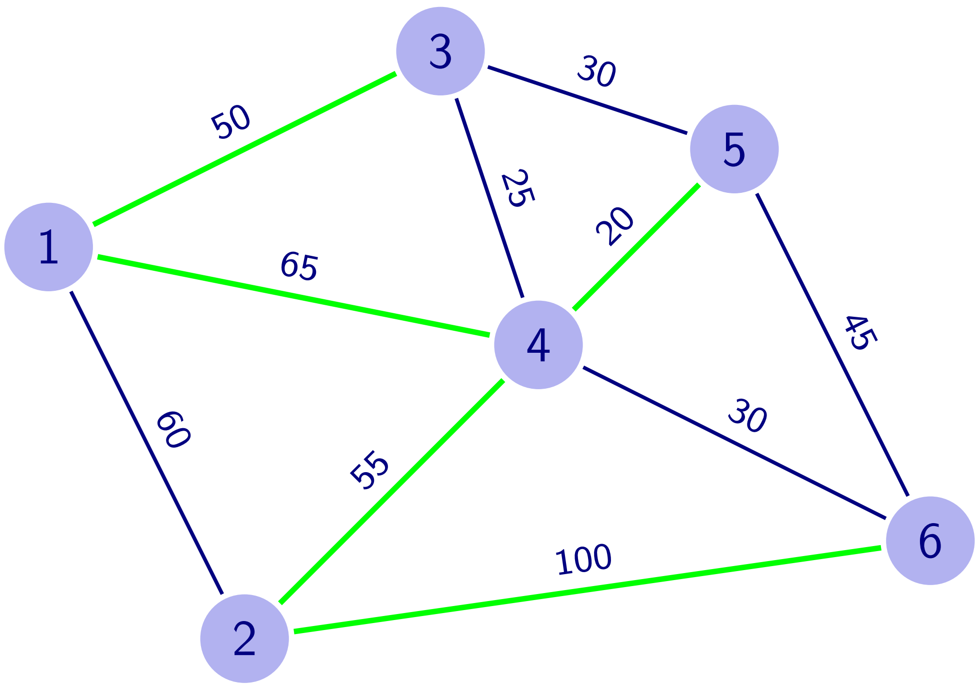
- the edges in E' form a tree
- all vertices of G are in the tree
- the total weight of the tree edges is minimal



▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

...find a minimum spanning tree for G, that is: a subset E' of the edges such that

- the edges in E' form a tree
- all vertices of G are in the tree
- the total weight of the tree edges is minimal

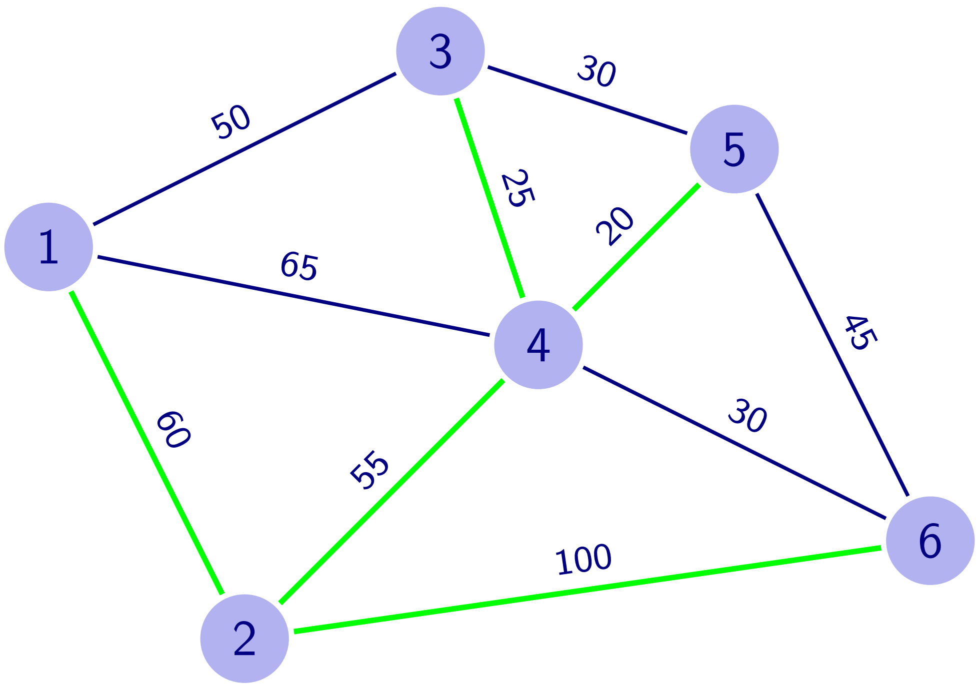


➡ total weight: 290

▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

...find a minimum spanning tree for G, that is: a subset E' of the edges such that

- the edges in E' form a tree
- all vertices of G are in the tree
- the total weight of the tree edges is minimal

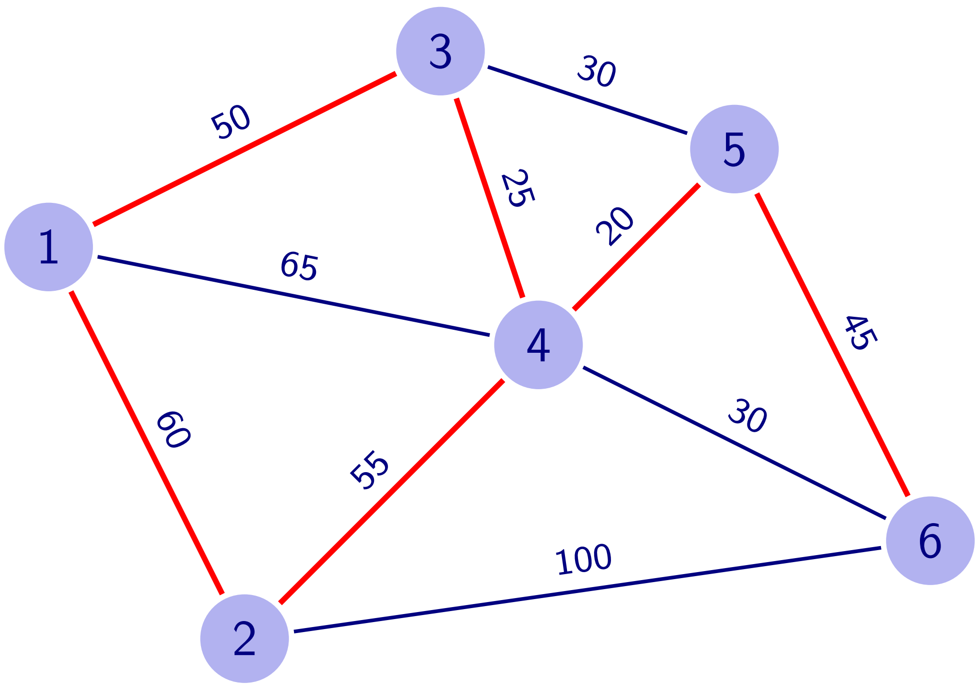


➡ total weight: 260

▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

...find a minimum spanning tree for G, that is: a subset E' of the edges such that

- the edges in E' form a tree
- all vertices of G are in the tree
- the total weight of the tree edges is minimal

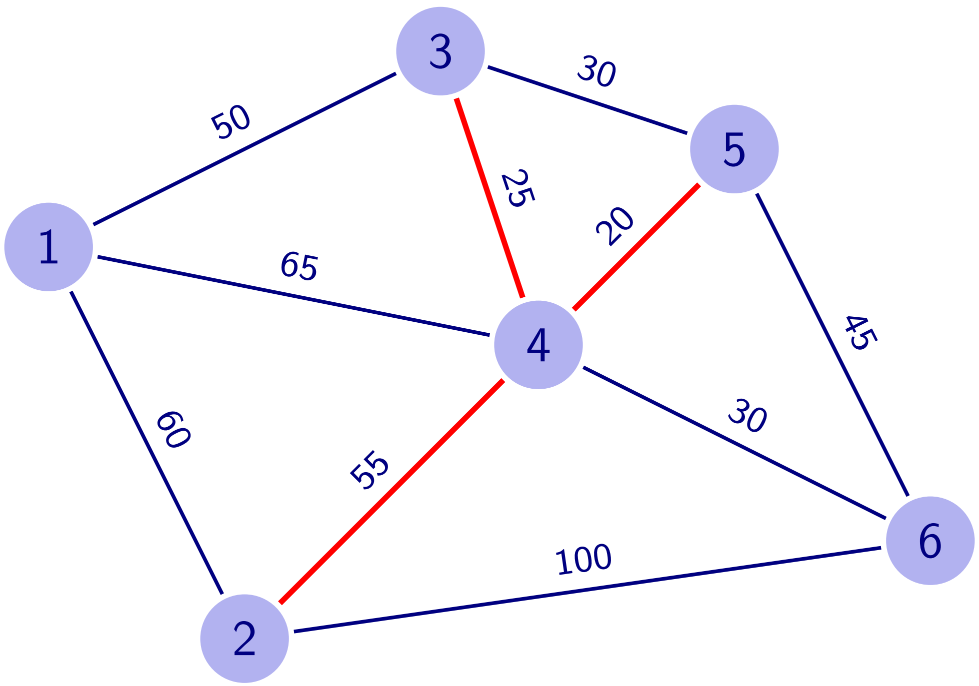


➡ not allowed: not a tree!

▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

...find a minimum spanning tree for G, that is: a subset E' of the edges such that

- the edges in E' form a tree
- all vertices of G are in the tree
- the total weight of the tree edges is minimal

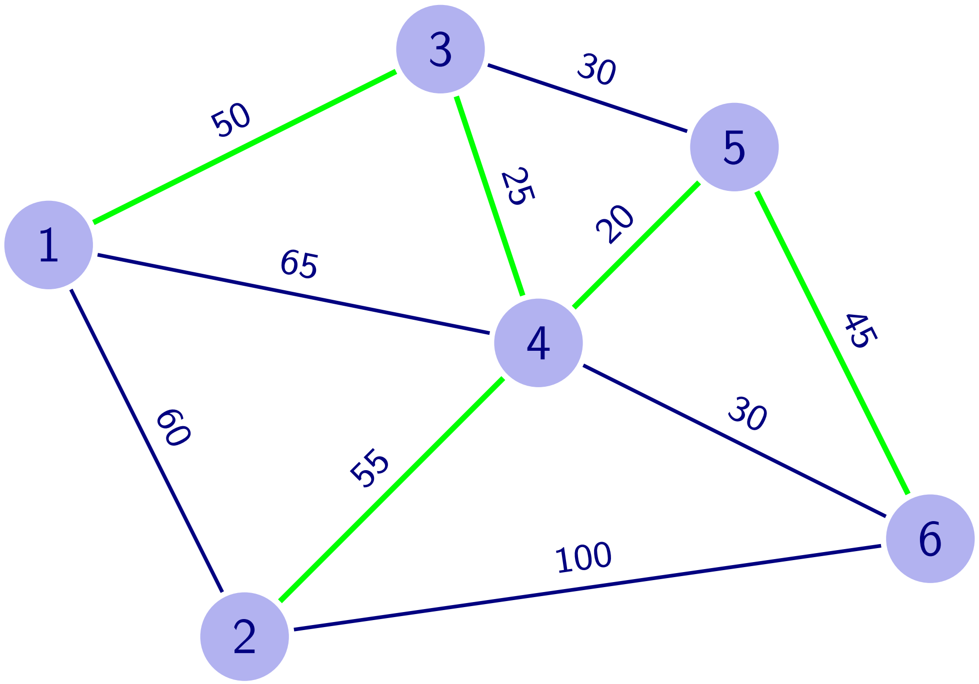


➡ not allowed: misses vertices!

▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

...find a minimum spanning tree for G, that is: a subset E' of the edges such that

- the edges in E' form a tree
- all vertices of G are in the tree
- the total weight of the tree edges is minimal

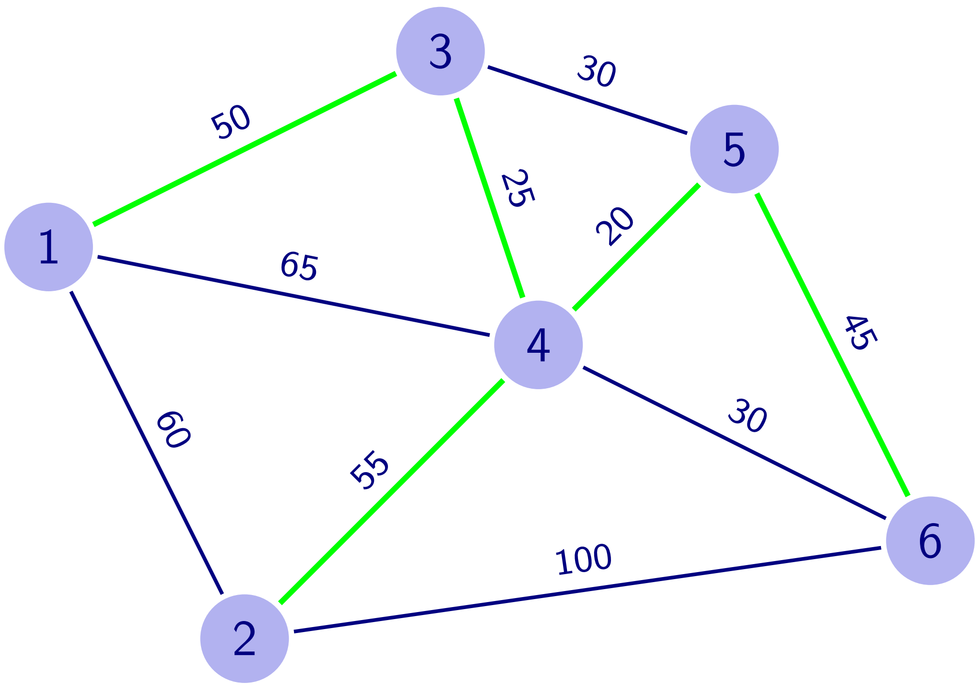


➡ total weight: 195

▷ Given a graph $G = (V, E)$ with non-negative edge-weights w_e for all $e \in E$...

...find a minimum spanning tree for G, that is: a subset E' of the edges such that

- the edges in E' form a tree
- all vertices of G are in the tree
- the total weight of the tree edges is minimal



➡ total weight: 195

▷ Real-world problem:

Connect a set of given computers to form a local network, at minimal cost

- Idea: select cheap edges, as long as they don't result in a cycle (greedy)

▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty

▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...

▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$

▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T

▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

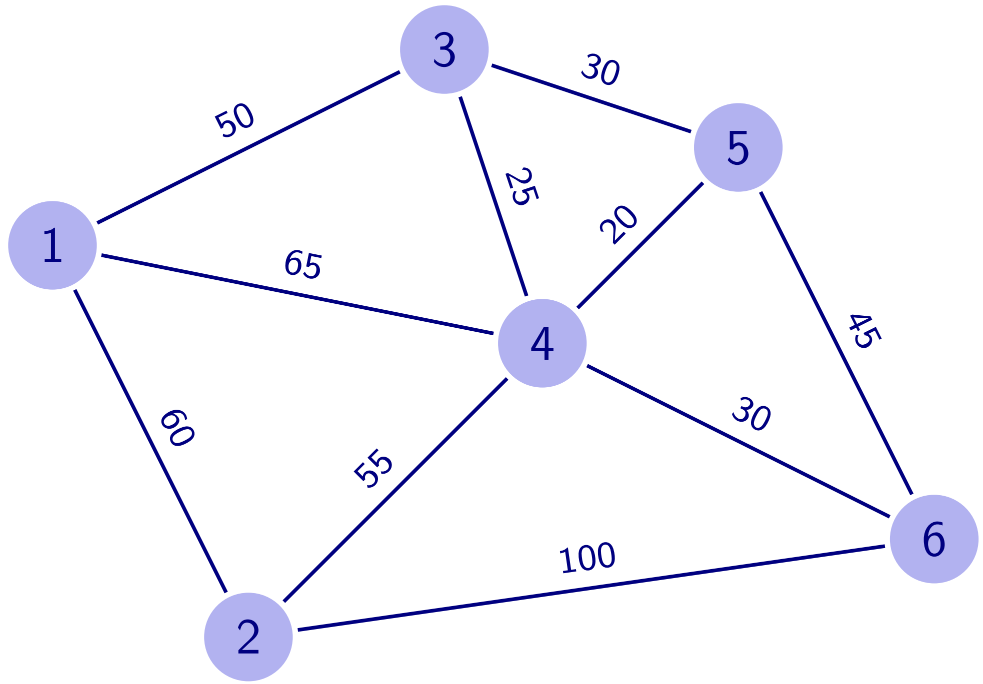
➡ T is a minimum spanning tree

▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges := E, tree T := empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

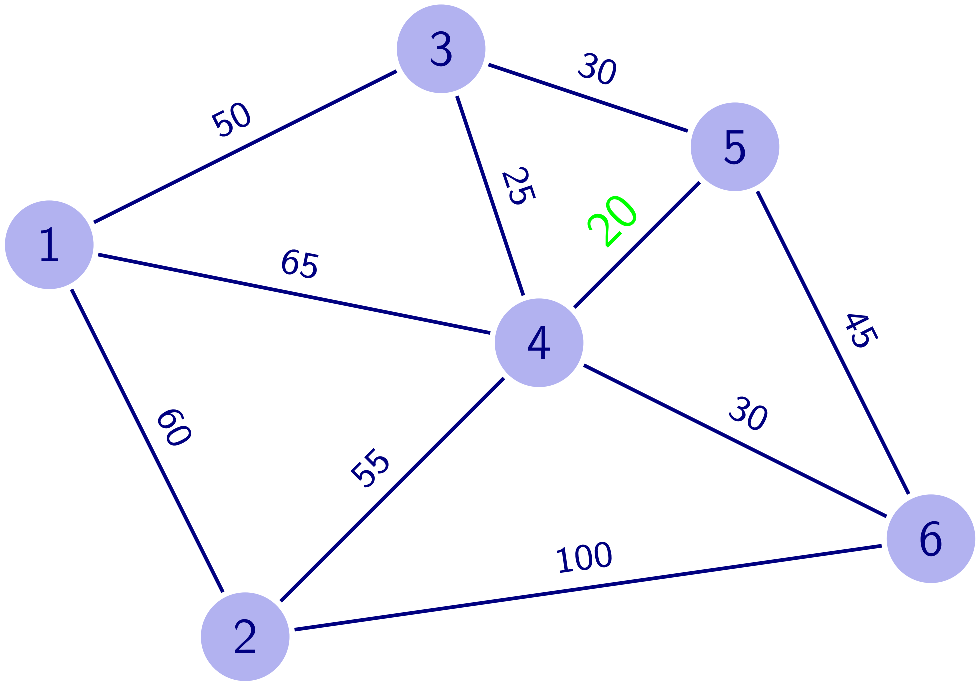


▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree



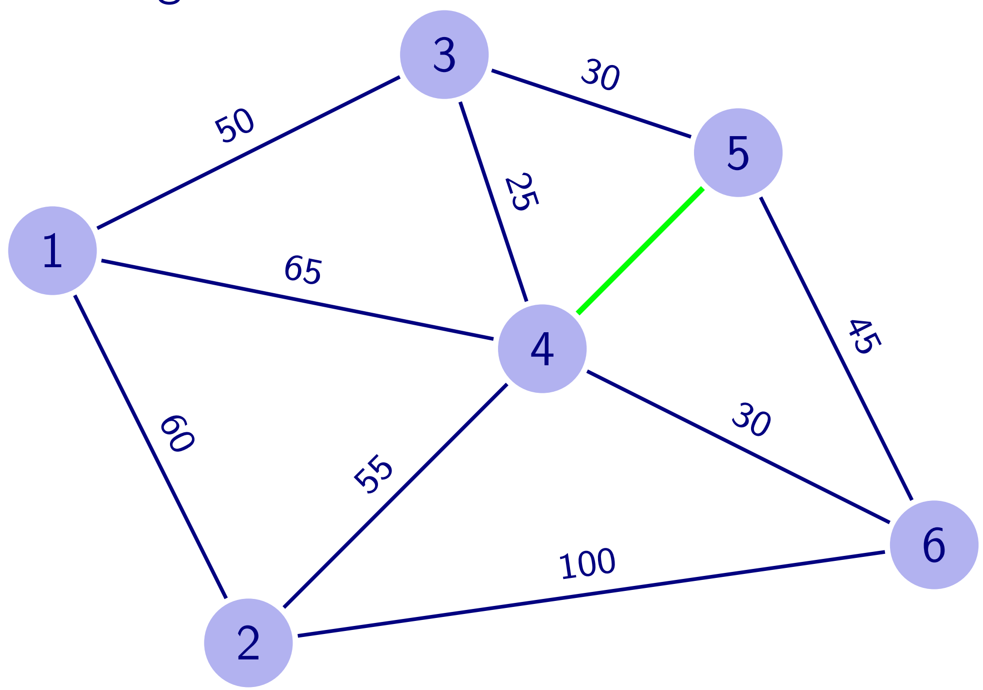
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 20



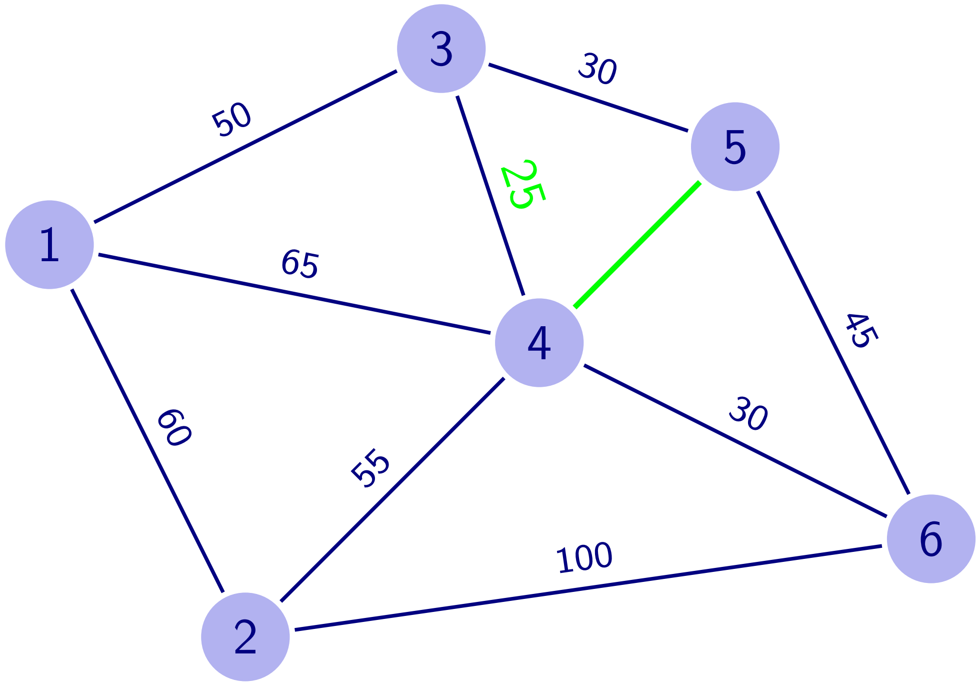
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges := E, tree T := empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 20



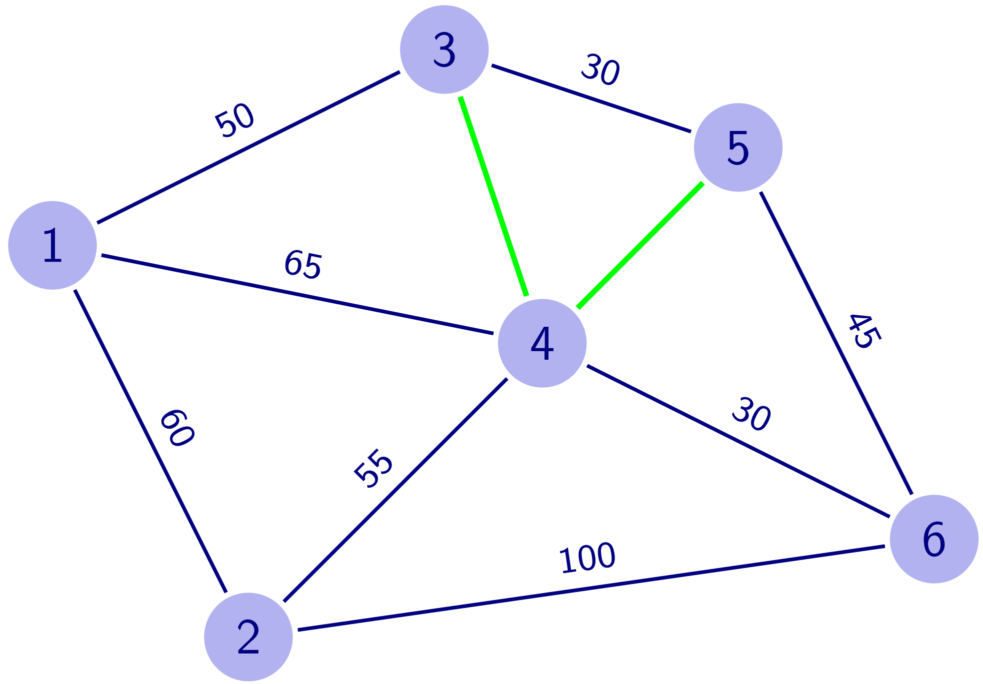
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges := E, tree T := empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 45



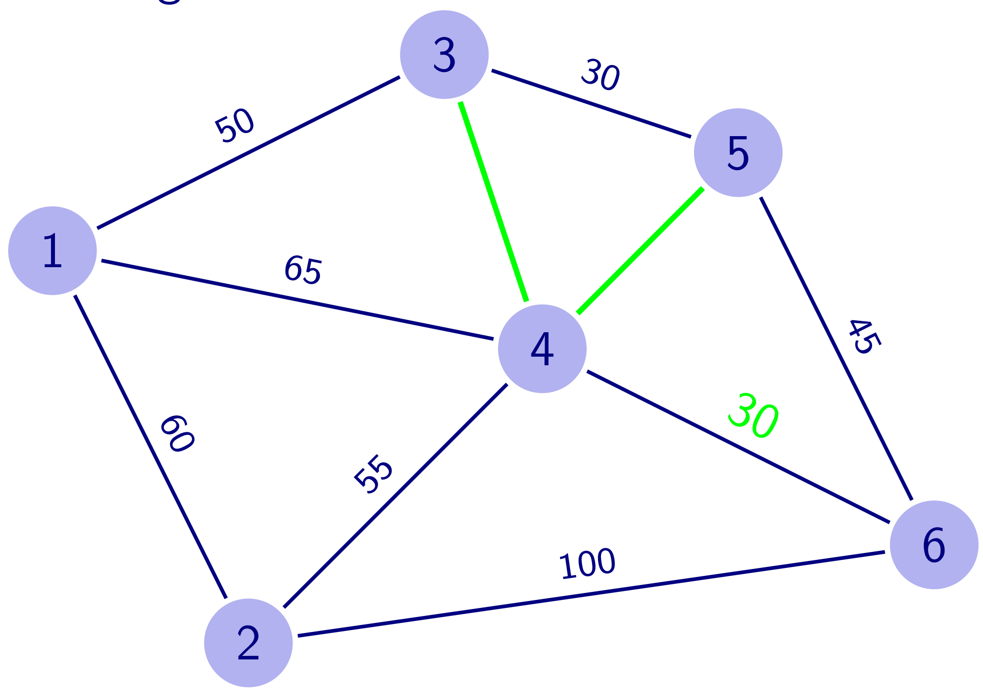
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 45



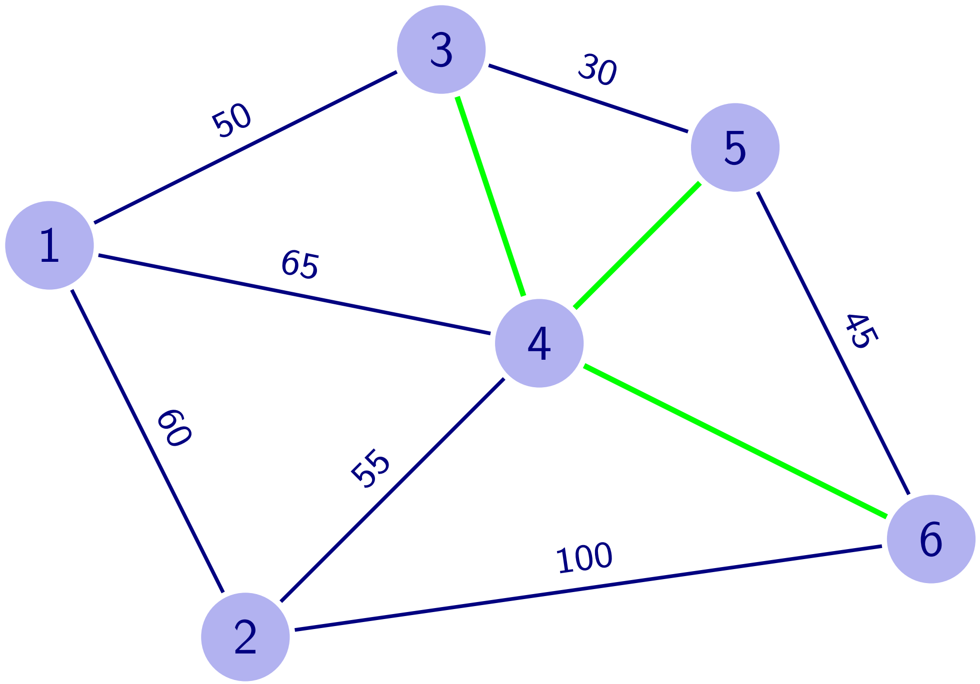
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 75



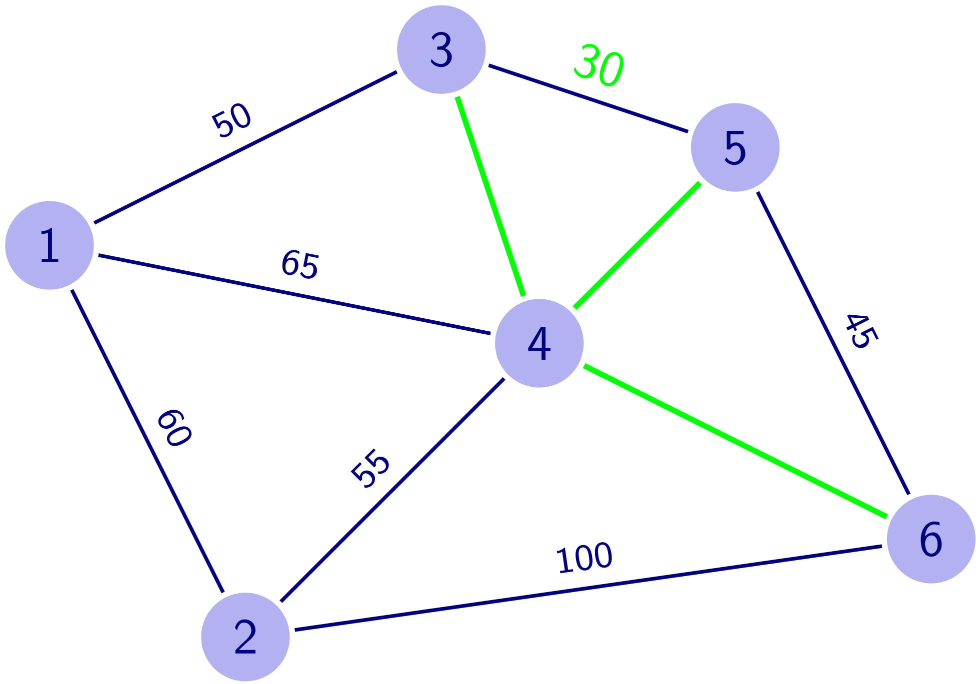
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 75



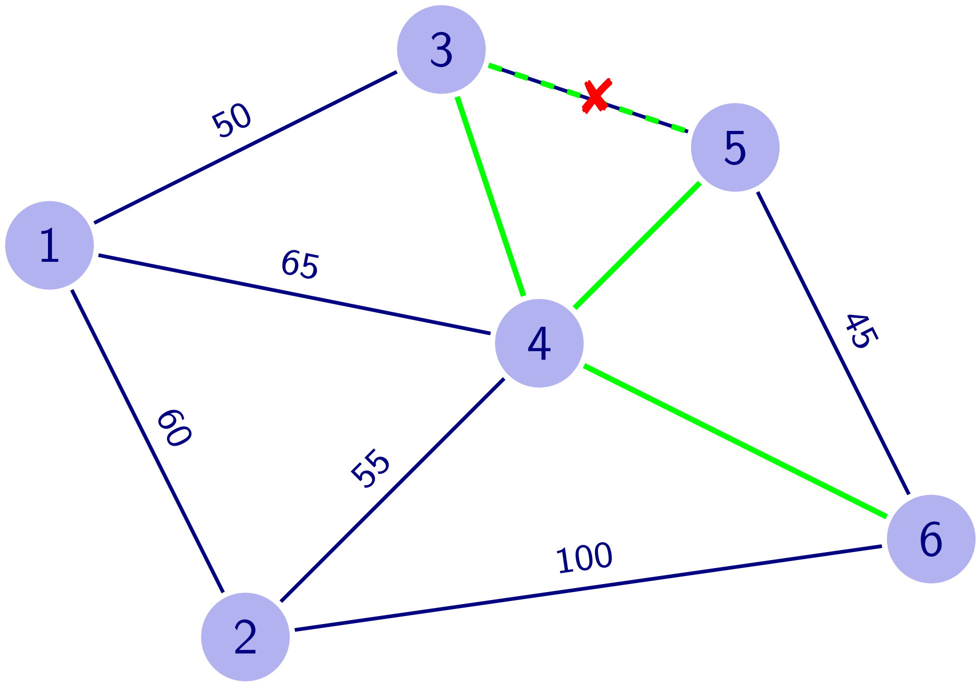
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges := E, tree T := empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 75



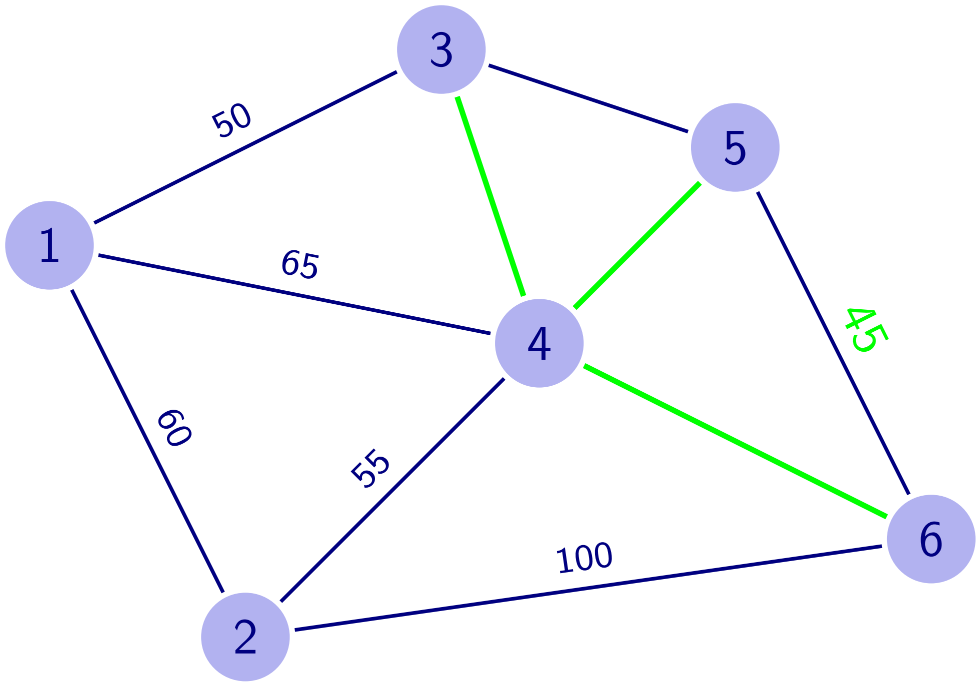
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 75



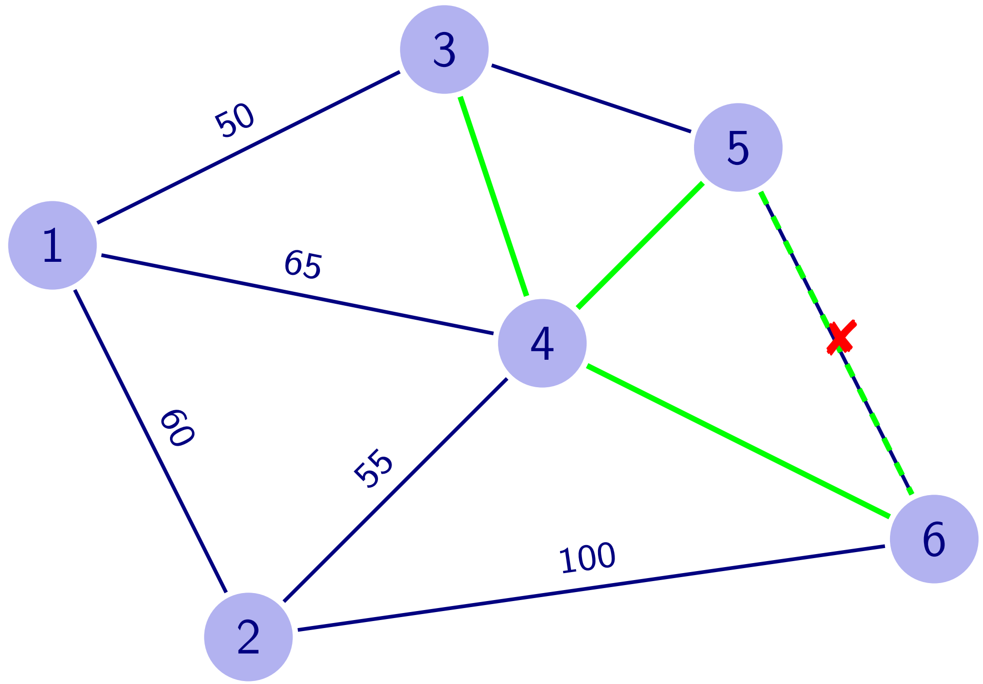
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 75



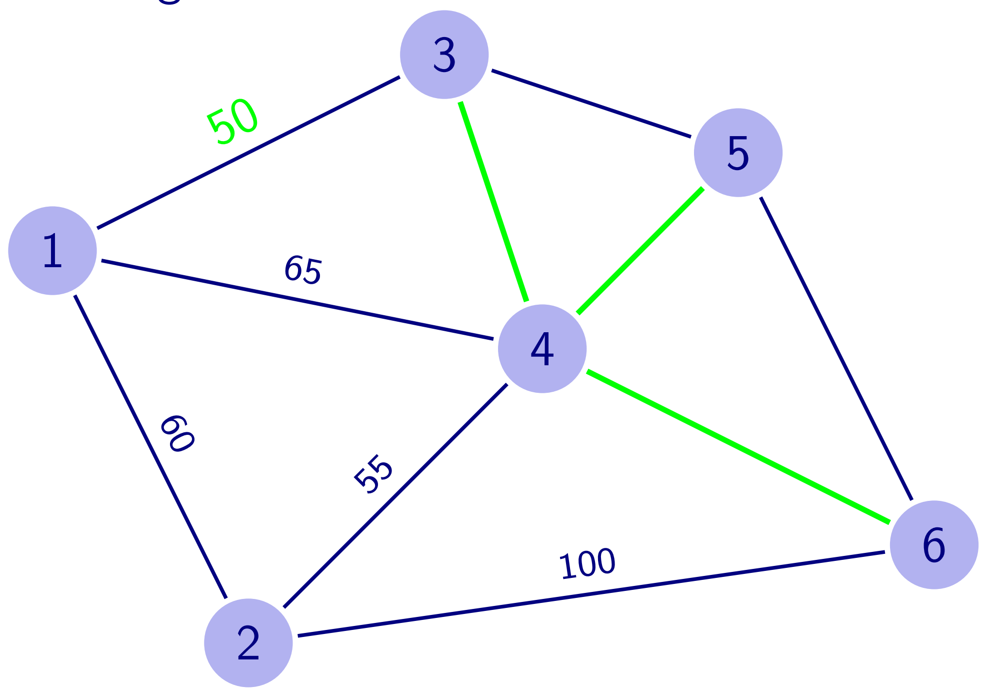
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 75



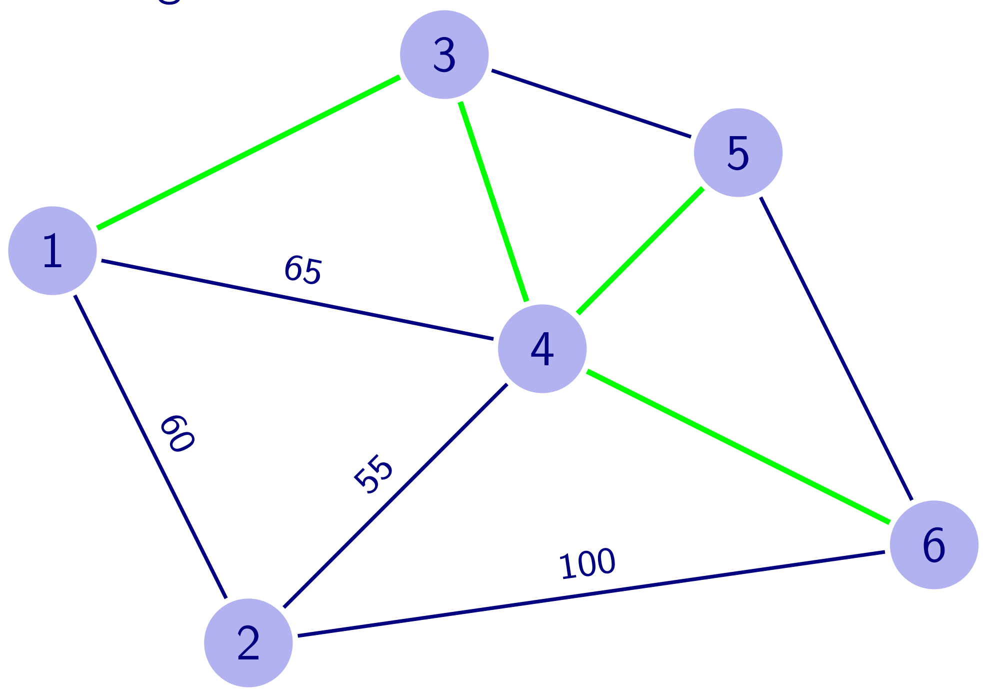
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges := E, tree T := empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 125



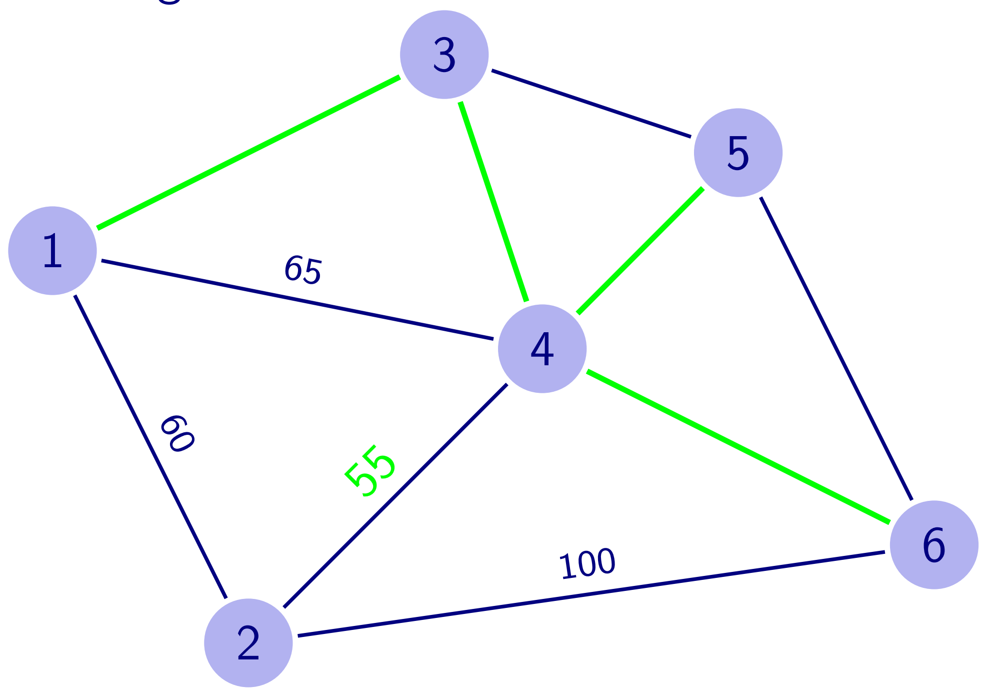
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges := E, tree T := empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 125



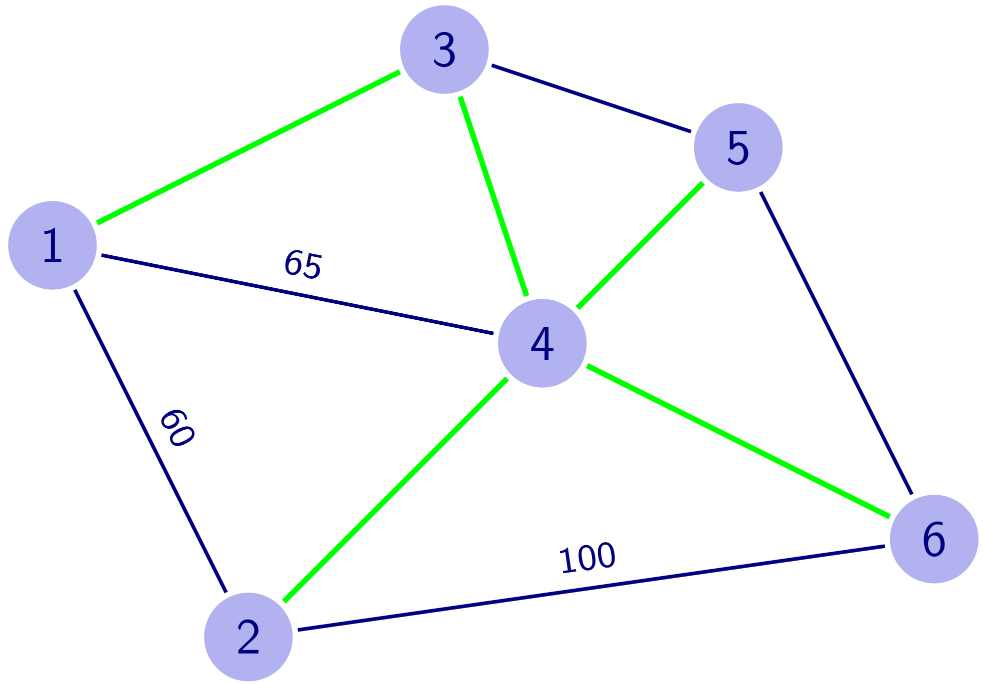
▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges $:= E$, tree $T :=$ empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree

total weight: 180

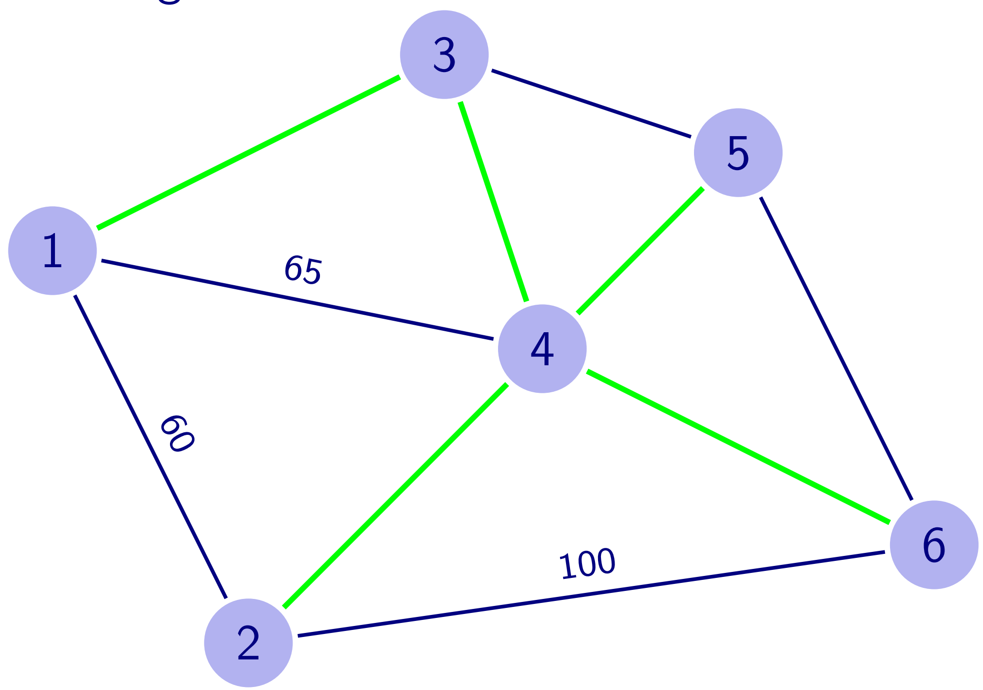


▷ Idea: select cheap edges, as long as they don't result in a cycle (greedy)

- Set of potential edges := E, tree T := empty
- Until all vertices are in the tree T...
- ...determine cheapest remaining potential edge: $\rightarrow\ e$
- ...if adding edge e to the tree T does not result in a cycle:

  add e to T
- ...remove e from the set of potential edges

➡ T is a minimum spanning tree



total weight: 180

➡ T contains all vertices

➡ done!

- Kruskal's algorithm is fast (polynomial runtime) and relatively easy to implement (greedy algorithm)

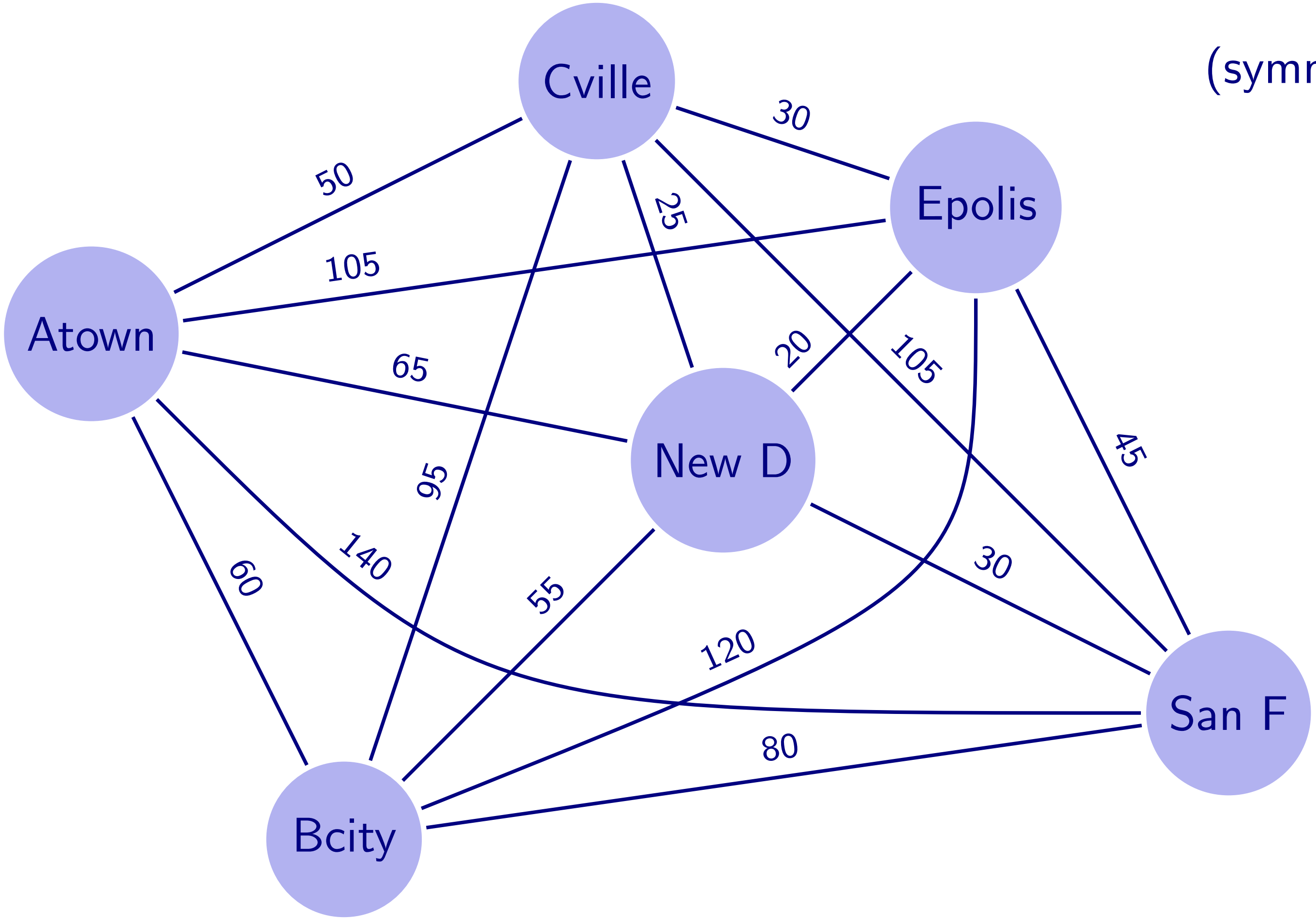
- Kruskal's algorithm is fast (polynomial runtime) and relatively easy to implement (greedy algorithm)
- Still it always computes an optimal tree! (Proof by contradiction)

- Kruskal's algorithm is fast (polynomial runtime) and relatively easy to implement (greedy algorithm)
- Still it always computes an optimal tree! (Proof by contradiction)
- Published by Joseph B. Kruskal in 1956

Joseph B. Kruskal (1928–2010)

Problem formulation: Given a set of cities together with travel times to travel *between every two cities*, find a tour leading through every city such that the total travel time is minimized.

Problem formulation: Given a set of cities together with travel times to travel *between every two cities*, find a tour leading through every city such that the total travel time is minimized.



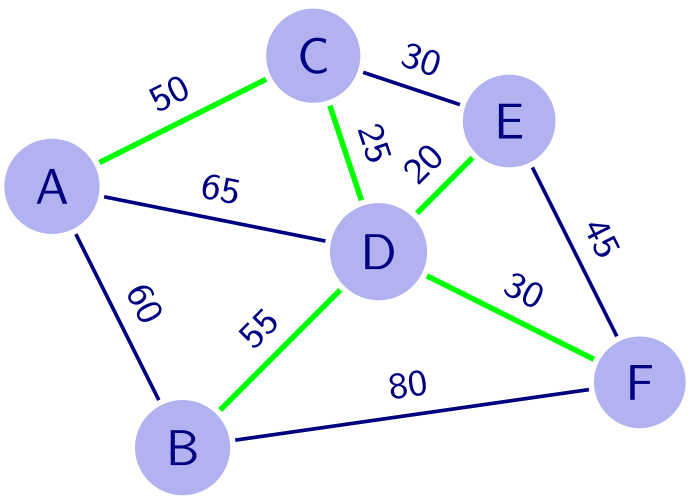
(symmetric TSP)

- MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

▷ MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

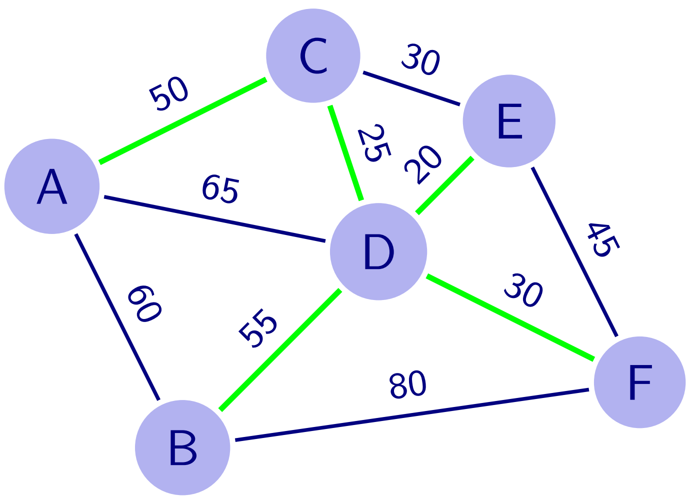
- Compute an MST for the graph, using distances as edge weights

- MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:
  - Compute an MST for the graph, using distances as edge weights



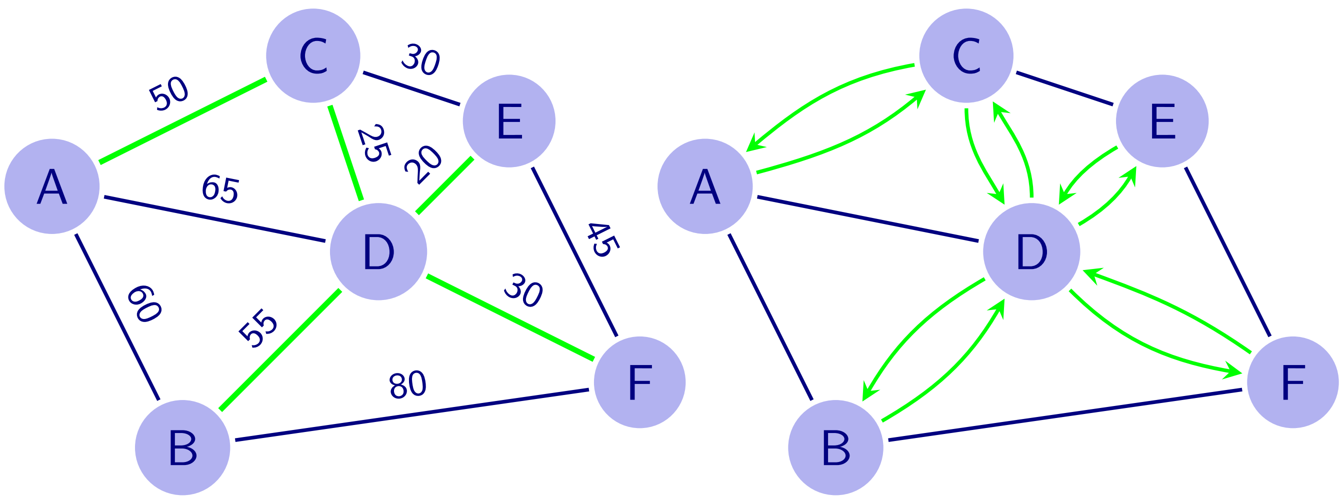
▷ MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

- Compute an MST for the graph, using distances as edge weights
- Create a "fake tour" by going to and back for every edge of the tree



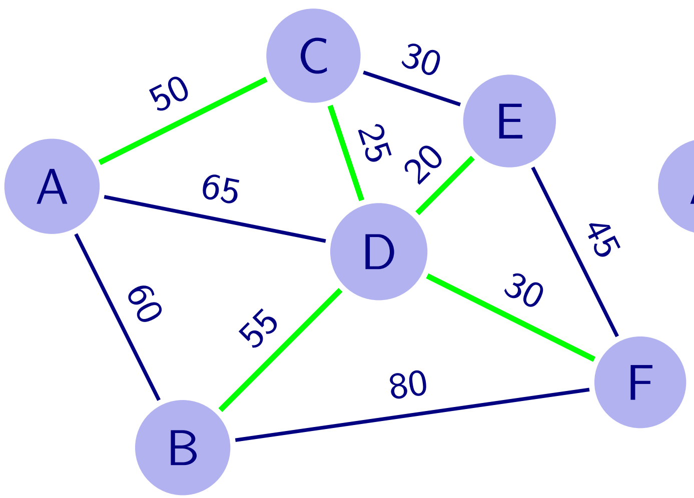
▷ MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

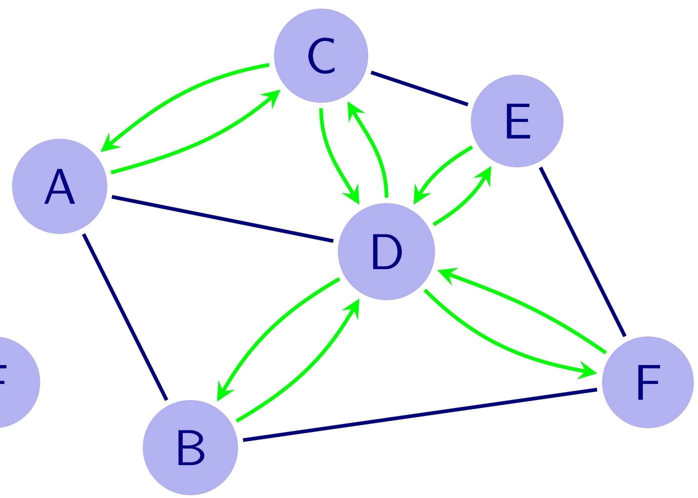
- Compute an MST for the graph, using distances as edge weights
- Create a "fake tour" by going to and back for every edge of the tree



▷ MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

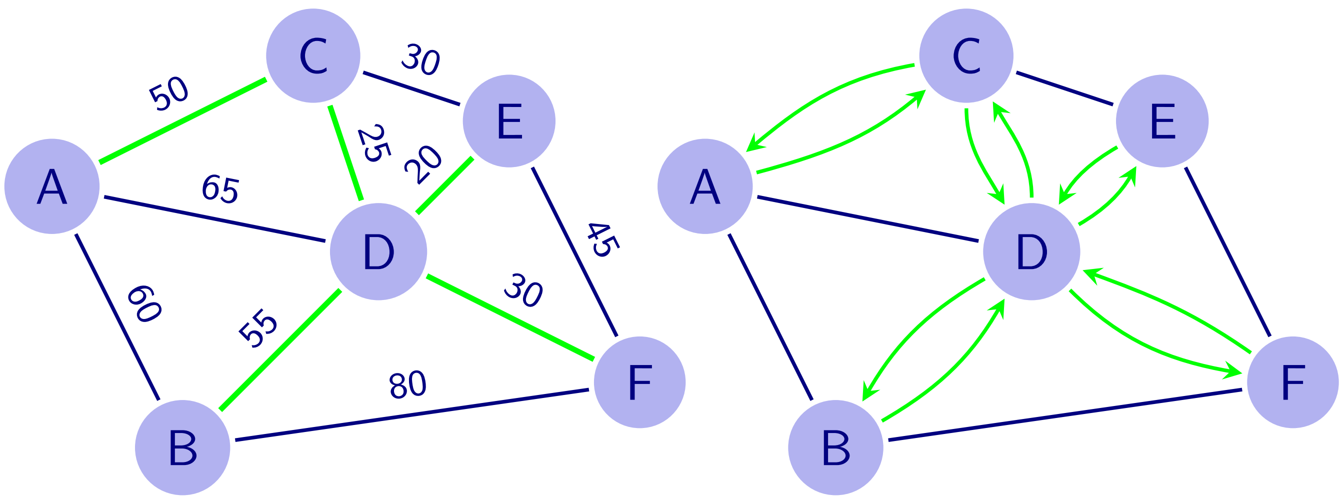
- Compute an MST for the graph, using distances as edge weights
- Create a “fake tour” by going to and back for every edge of the tree
- Start traversing the “tour” at some city





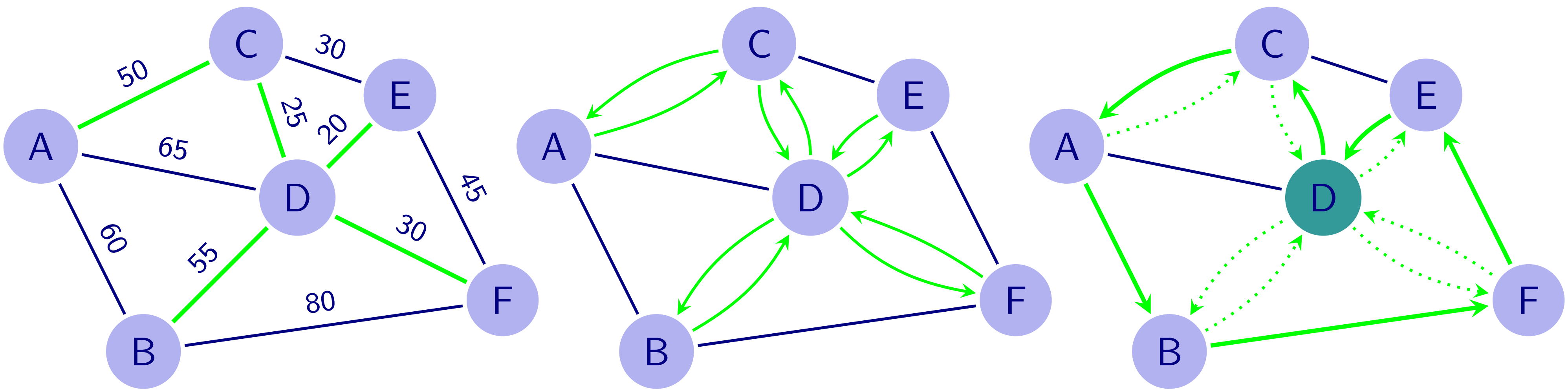
▷ MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

- Compute an MST for the graph, using distances as edge weights
- Create a "fake tour" by going to and back for every edge of the tree
- Start traversing the "tour" at some city
- Whenever the tour returns to an already visited city, replace the edge by the shortcut to the following city



▷ MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

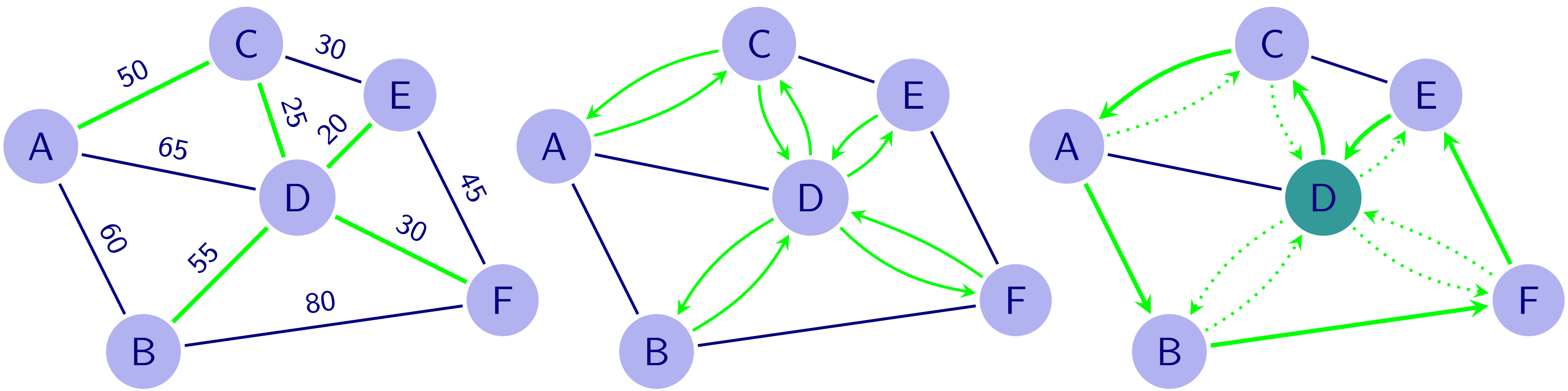
- Compute an MST for the graph, using distances as edge weights
- Create a “fake tour” by going to and back for every edge of the tree
- Start traversing the “tour” at some city
- Whenever the tour returns to an already visited city, replace the edge by the shortcut to the following city



▷ MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

- Compute an MST for the graph, using distances as edge weights
- Create a "fake tour" by going to and back for every edge of the tree
- Start traversing the "tour" at some city
- Whenever the tour returns to an already visited city, replace the edge by the shortcut to the following city

➡ The found solution misses the optimum by a factor of at most 2 (approximation factor)

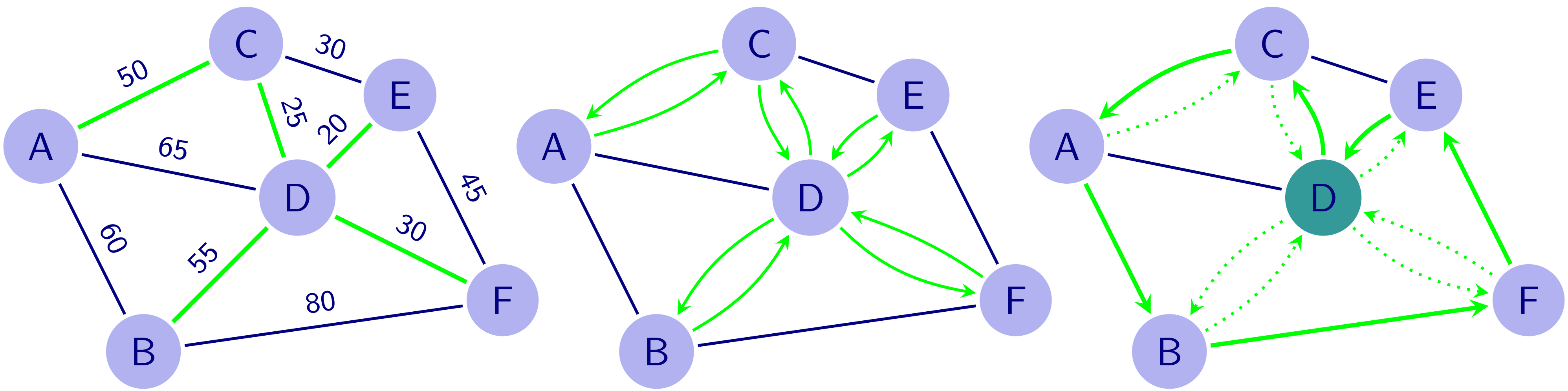


▷ MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

- Compute an MST for the graph, using distances as edge weights
- Create a "fake tour" by going to and back for every edge of the tree
- Start traversing the "tour" at some city
- Whenever the tour returns to an already visited city, replace the edge by the shortcut to the following city

➡ The found solution misses the optimum by a factor of at most 2 (approximation factor)

Proof: $L_{\text{sol}} \leq L_{\text{“fake tour”}}$

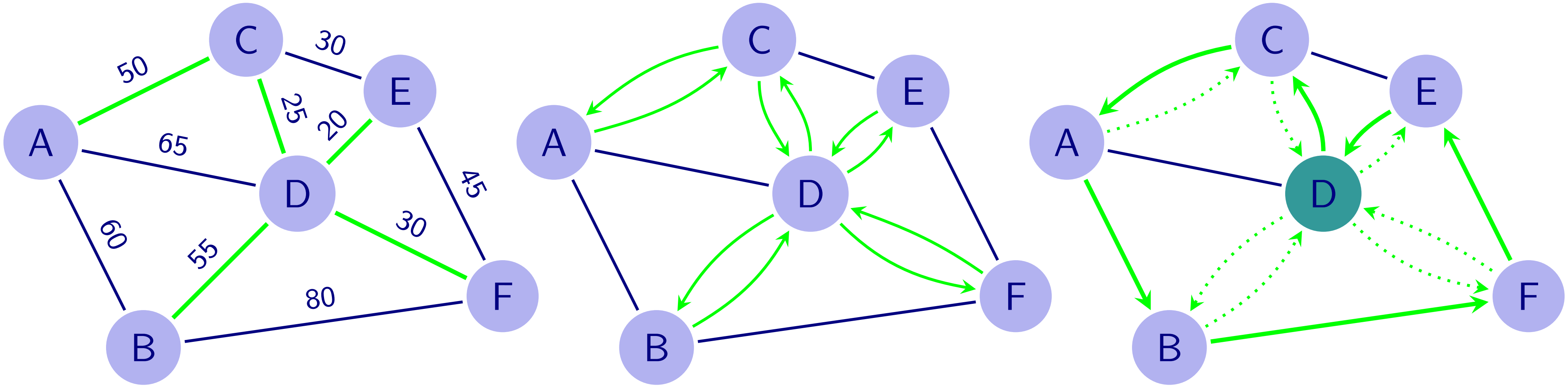


▷ MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

- Compute an MST for the graph, using distances as edge weights
- Create a “fake tour” by going to and back for every edge of the tree
- Start traversing the “tour” at some city
- Whenever the tour returns to an already visited city, replace the edge by the shortcut to the following city

➡ The found solution misses the optimum by a factor of at most 2 (approximation factor)

Proof: $L_{\text{sol}} \leq L_{\text{“fake tour”}} \leq 2 \cdot L_{\text{MST}}$

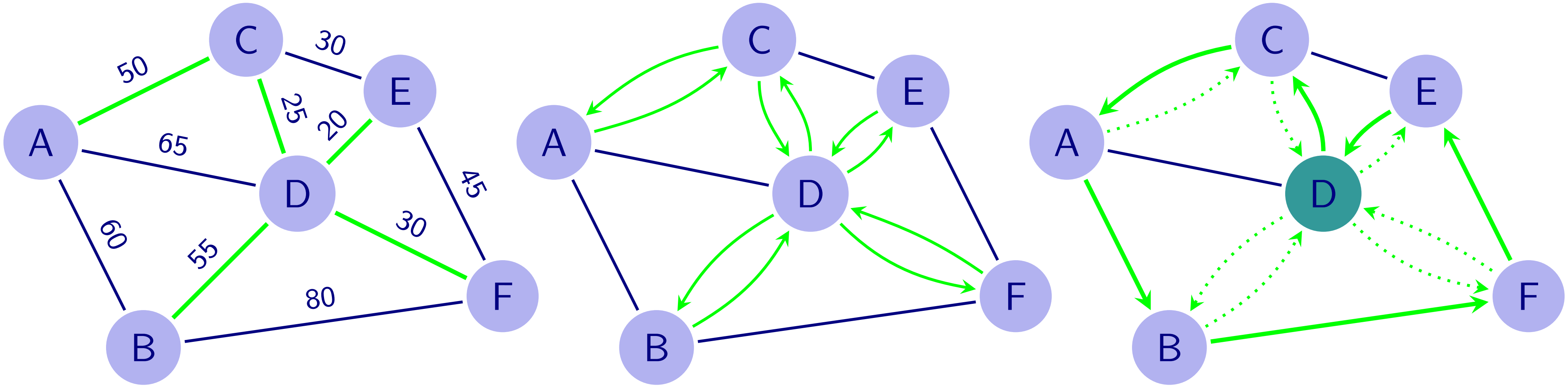


▷ MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

- Compute an MST for the graph, using distances as edge weights
- Create a "fake tour" by going to and back for every edge of the tree
- Start traversing the "tour" at some city
- Whenever the tour returns to an already visited city, replace the edge by the shortcut to the following city

➡ The found solution misses the optimum by a factor of at most 2 (approximation factor)

Proof: $L_{\text{sol}} \leq L_{\text{“fake tour”}} \leq 2 \cdot L_{\text{MST}} \leq 2 \cdot L_{\text{opt}\setminus\text{edge}}$

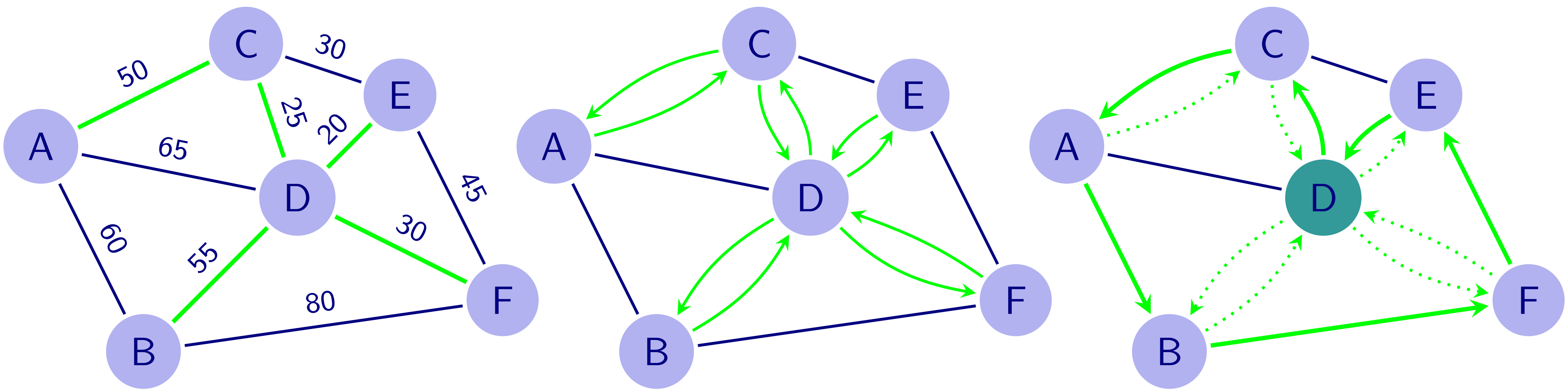


▷ MST can be used for an approximation algorithm for the (symmetric, euclidean) TSP:

- Compute an MST for the graph, using distances as edge weights
- Create a "fake tour" by going to and back for every edge of the tree
- Start traversing the "tour" at some city
- Whenever the tour returns to an already visited city, replace the edge by the shortcut to the following city

➡ The found solution misses the optimum by a factor of at most 2 (approximation factor)

Proof: $L_{\text{sol}} \leq L_{\text{“fake tour”}} \leq 2 \cdot L_{\text{MST}} \leq 2 \cdot L_{\text{opt}\setminus\text{edge}} \leq 2 \cdot L_{\text{opt}}$



- Great flexibility in formulating real-life problems

▷ Great flexibility in formulating real-life problems

▷ Usually integer-programming formulation is possible, but inefficient

➡ Specially designed algorithms

- ▷ Great flexibility in formulating real-life problems
- ▷ Usually integer-programming formulation is possible, but inefficient
  - ➡ Specially designed algorithms
- ▷ Wide variety of algorithms:

- Great flexibility in formulating real-life problems
- Usually integer-programming formulation is possible, but inefficient
  - ➡ Specially designed algorithms
- Wide variety of algorithms:
  - ➡ Primal algorithms (heuristics):

    Provide feasible solutions, but without guarantee of optimality

- Great flexibility in formulating real-life problems
- Usually integer-programming formulation is possible, but inefficient
  - ➡ Specially designed algorithms
- Wide variety of algorithms:
  - ➡ Primal algorithms (heuristics):

    Provide feasible solutions, but without guarantee of optimality
  - ➡ Dual algorithms (branch & bound, approximation algorithms):

    Provide upper/lower bounds on the optimal solution, but without explicitly giving one

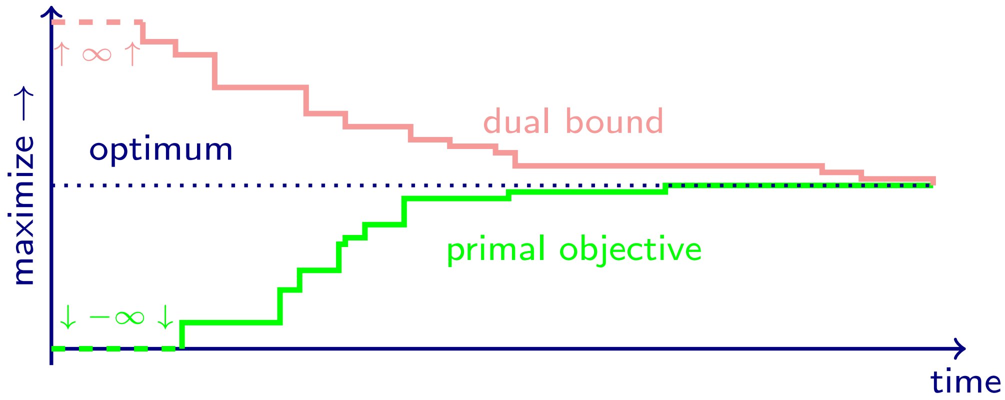
- ▷ Great flexibility in formulating real-life problems
- ▷ Usually integer-programming formulation is possible, but inefficient
  - ➡ Specially designed algorithms
- ▷ Wide variety of algorithms:
  - ➡ Primal algorithms (heuristics):

    Provide feasible solutions, but without guarantee of optimality
  - ➡ Dual algorithms (branch & bound, approximation algorithms):

    Provide upper/lower bounds on the optimal solution, but without explicitly giving one

➡ Combination: Primal-dual algorithms

- ▷ Great flexibility in formulating real-life problems
- ▷ Usually integer-programming formulation is possible, but inefficient
  - ➡ Specially designed algorithms
- ▷ Wide variety of algorithms:
  - ➡ Primal algorithms (heuristics):
    Provide feasible solutions, but without guarantee of optimality
  - ➡ Dual algorithms (branch & bound, approximation algorithms):
    Provide upper/lower bounds on the optimal solution, but without explicitly giving one
- ➡ Combination: Primal-dual algorithms



- Models, Data and Algorithms
- Linear Optimization
- Mathematical Background: Polyhedra, Simplex-Algorithm
- Sensitivity Analysis; (Mixed) Integer Programming
- MIP Modelling
- MIP Modelling: More Examples; Branch & Bound
- Cutting Planes; Combinatorial Optimization: Examples, Graphs, Algorithms
- TSP-Heuristics
- Network Flows, Complexity Theory
- Nonlinear Optimization
- Scheduling
- Lot Sizing
- Multicriteria Optimization
- Oral exam