Minimum Spanning Tree

December 1, 2025

1 Minimum Spanning Tree Final Project

1.1 Matt Eimers

1.1.1 Presentation Example Code

[1]:
```
import itertools
import pulp

# Define Nodes
nodes = ["A", "B", "C", "D", "E"]

# Define Edges with their costs
edges = {
    ("A", "B"): 2,
    ("A", "D"): 1,
    ("D", "E"): 3,
    ("B", "C"): 4,
    ("B", "E"): 5,
    ("C", "E"): 6,
    ("A", "E"): 7
}

# Make edges undirected
edges = {tuple(sorted(k)): v for k, v in edges.items()}
prob = pulp.LpProblem("MST", pulp.LpMinimize)

# Decision variables x_ij in {0,1}
x = pulp.LpVariable.dicts("x", edges, 0, 1, cat="Binary")

# Objective: minimize total cost
prob += pulp.lpSum(edges[e] * x[e] for e in edges)

# Constraint: select n - 1 edges
prob += pulp.lpSum(x[e] for e in edges) == len(nodes) - 1

# Constraint: Subtour elimination constraints
for r in range(1, len(nodes)):
```

```
    for S in itertools.combinations(nodes, r):
        S_edges = [tuple(sorted(e)) for e in edges if e[0] in S and e[1] in S]
        if S_edges:
            prob += pulp.lpSum(x[e] for e in S_edges) <= len(S) - 1

# Solve the problem
prob.solve(pulp.PULP_CBC_CMD(msg=False))
print("Optimal MST edges:")
total_cost = 0
for e in edges:
    if x[e].value() == 1:
        print(f"  {e[0]} - {e[1]}   cost = {edges[e]}")
        total_cost += edges[e]

print("\nTotal MST cost:", total_cost)
```

```
Optimal MST edges:
  A - B   cost = 2
  A - D   cost = 1
  D - E   cost = 3
  B - C   cost = 4

Total MST cost: 10
```

1.1.2 MST with 10 randomized nodes

Below this first cell block will generate a random grpah with 10 nodes. This can be changed to whatever you desire where you can change node_count equalt to however many nodes you want.

[2]:
```
import itertools
import pulp
import random

node_count = 15 # Change this number if you want more nodes
mult = 2 # Can change this multiplier to add more or fewer edges

# Generate node labels like X_{1}, X_{2}, ..., X_{15}
nodes = [f"X_{{{i}}}" for i in range(1, node_count + 1)]

# Genrrate edges ensuring connectivity: first part makes it there is at least␣
 ↪one path between the nodes
edges = {}
for i in range(1, len(nodes)):
    u = nodes[i]
    v = random.choice(nodes[:i])   # Connect to a previous node
    cost = random.randint(1, node_count) # Random cost between 1 and node_count␣
 ↪can be changed
    edges[tuple(sorted((u, v)))] = cost
```

```
# Add extra random edges
extra_edges = int(mult * node_count)
for _ in range(extra_edges):
    u, v = random.sample(nodes, 2)
    u, v = sorted((u, v))
    if (u, v) not in edges:
        edges[(u, v)] = random.randint(1, node_count)

print(f"Generated graph with {len(nodes)} nodes and {len(edges)} edges.")
```

```
Generated graph with 15 nodes and 35 edges.
```

After the code has been generated we now can solve this new graph with nodes and edges.

[3]:
```
prob = pulp.LpProblem("MST10", pulp.LpMinimize)

# Decision variables x_ij in {0,1}
x = pulp.LpVariable.dicts("x", edges, 0, 1, cat="Binary")

# Objective: minimize total cost
prob += pulp.lpSum(edges[e] * x[e] for e in edges)

# Constraint: select n - 1 edges
prob += pulp.lpSum(x[e] for e in edges) == len(nodes) - 1

# Constraint: Subtour elimination constraints
for r in range(1, len(nodes)):
    for S in itertools.combinations(nodes, r):
        S_edges = [tuple(sorted(e)) for e in edges if e[0] in S and e[1] in S]
        if S_edges:
            prob += pulp.lpSum(x[e] for e in S_edges) <= len(S) - 1


# Solve the problem
prob.solve(pulp.PULP_CBC_CMD(msg=False))
print("Optimal MST edges:")
total_cost = 0
for e in edges:
    if x[e].value() == 1:
        print(f"  {e[0]} - {e[1]}   cost = {edges[e]}")
        total_cost += edges[e]

print("\nTotal MST cost:", total_cost)
```

```
Optimal MST edges:
  X_{2} - X_{4}   cost = 5
  X_{5} - X_{6}   cost = 9
  X_{4} - X_{9}   cost = 1
```

```
X_{11} - X_{8}   cost = 7
X_{11} - X_{13}   cost = 10
X_{15} - X_{4}   cost = 2
X_{6} - X_{9}   cost = 4
X_{1} - X_{4}   cost = 5
X_{15} - X_{3}   cost = 4
X_{3} - X_{7}   cost = 6
X_{10} - X_{15}   cost = 12
X_{11} - X_{9}   cost = 6
X_{12} - X_{2}   cost = 1
X_{14} - X_{3}   cost = 3

Total MST cost: 75
```

1.1.3 Traveling Salesman Problem using Christofides' Algorithm

Case 1: When solving a Traveling Salesman problem we need a grpah with edges that go to every other node from a single node. So for this if we have n number of nodes then we should have edges $= \frac{n(n-1)}{2}$. Now in this first case we will be running the Christofides Algorithm on a completed graph to see for the TSP what is the Christofides tour cost.

[4]:
```python
import itertools
import pulp
import random

node_count = 100  # Change this number if you want more nodes

# Generate node labels like X_{1}, X_{2}, ..., X_{100}
nodes = [f"X_{{{i}}}" for i in range(1, node_count + 1)]

# Generate edges ensuring connectivity
edges = {}
for i in range(1, len(nodes)):
    u = nodes[i]
    v = random.choice(nodes[:i])   # Connect to a previous node
    cost = random.randint(1, node_count)  # Random cost between 1 and node_count
    edges[tuple(sorted((u, v)))] = cost

# Add extra random edges to complete the graph
total_edges = node_count * (node_count - 1) // 2  # Total edges in a complete␣
 ↪graph
while len(edges) < total_edges:
    u, v = sorted(random.sample(nodes, 2))
    if (u, v) not in edges:
        edges[(u, v)] = random.randint(1, node_count)

print(f"Generated graph with {len(nodes)} nodes and {len(edges)} edges.")
```

```
Generated graph with 100 nodes and 4950 edges.
```

[5]:
```python
from collections import defaultdict

# Build distance matrix directly from the completed graph
dist = {u: {} for u in nodes}
for u in nodes:
    for v in nodes:
        if u == v:
            dist[u][v] = 0
        else:
            a, b = sorted((u, v))
            dist[u][v] = edges[(a, b)]

# Prim's MST on the complete metric graph
def prim_mst(nodes, dist):
    n = len(nodes)
    start = nodes[0]
    in_mst = {v: False for v in nodes}
    key = {v: float("inf") for v in nodes}
    parent = {v: None for v in nodes}
    key[start] = 0.0
    for _ in range(n):
        u = min((v for v in nodes if not in_mst[v]), key=lambda v: key[v])
        in_mst[u] = True
        for v in nodes:
            if not in_mst[v] and dist[u][v] < key[v]:
                key[v] = dist[u][v]
                parent[v] = u
    mst_edges = []
    degrees = {v: 0 for v in nodes}
    total_cost = 0.0
    for v in nodes:
        if parent[v] is not None:
            u = parent[v]
            mst_edges.append((u, v))
            degrees[u] += 1
            degrees[v] += 1
            total_cost += dist[u][v]
    return mst_edges, degrees, total_cost

mst_edges, degrees, mst_cost = prim_mst(nodes, dist)

# Find odd degree vertices in the MST
odd_vertices = [v for v, deg in degrees.items() if deg % 2 == 1]
unmatched = set(odd_vertices)
matching_edges = []
```

```
while len(unmatched) > 1:
    i = min(unmatched)
    best_j = None
    best_w = float("inf")
    for j in unmatched:
        if j == i:
            continue
        w = dist[i][j]
        if w < best_w:
            best_w = w
            best_j = j
    matching_edges.append((i, best_j))
    unmatched.remove(i)
    unmatched.remove(best_j)


# Building a multigraph which combines MST edges + matching edges
multi_adj = defaultdict(list)
for u, v in mst_edges:
    multi_adj[u].append(v)
    multi_adj[v].append(u)
for u, v in matching_edges:
    multi_adj[u].append(v)
    multi_adj[v].append(u)

# Eulerian tour using Hierholzers algorithm
def eulerian_tour(adj, start):
    local_adj = {u: list(vs) for u, vs in adj.items()}
    stack = [start]
    path = []

    while stack:
        u = stack[-1]
        if local_adj[u]:
            v = local_adj[u].pop()
            local_adj[v].remove(u)
            stack.append(v)
        else:
            path.append(stack.pop())

    return path[::-1]

start_node = nodes[0]
euler_tour = eulerian_tour(multi_adj, start_node)

# Shortcut Eulerian tour to get TSP Christofides Algorithm tour
```

```python
visited = set()
tsp_tour = []
for v in euler_tour:
    if v not in visited:
        visited.add(v)
        tsp_tour.append(v)

# Make it a cycle
tsp_tour.append(tsp_tour[0])

# Compute tour cost using metric distances
tsp_cost = sum(dist[tsp_tour[i]][tsp_tour[i + 1]] for i in range(len(tsp_tour)␣
 ↪- 1))

print("\nChristofides Algorithm TSP tour:")
chunk_size = 10
for i in range(0, len(tsp_tour), chunk_size):
    chunk = tsp_tour[i:i+chunk_size]
    print(" -> ".join(chunk))
print(f"\nChristofides tour cost: {tsp_cost}")
```

```
Christofides Algorithm TSP tour:
X_{1} -> X_{56} -> X_{83} -> X_{73} -> X_{3} -> X_{45} -> X_{62} -> X_{64} ->
X_{88} -> X_{72}
X_{75} -> X_{42} -> X_{2} -> X_{84} -> X_{21} -> X_{44} -> X_{90} -> X_{80} ->
X_{9} -> X_{37}
X_{12} -> X_{92} -> X_{25} -> X_{20} -> X_{28} -> X_{17} -> X_{74} -> X_{48} ->
X_{7} -> X_{57}
X_{53} -> X_{70} -> X_{65} -> X_{19} -> X_{34} -> X_{63} -> X_{52} -> X_{23} ->
X_{69} -> X_{54}
X_{97} -> X_{41} -> X_{99} -> X_{81} -> X_{39} -> X_{31} -> X_{27} -> X_{33} ->
X_{36} -> X_{50}
X_{24} -> X_{61} -> X_{8} -> X_{98} -> X_{4} -> X_{5} -> X_{22} -> X_{68} ->
X_{93} -> X_{18}
X_{16} -> X_{78} -> X_{49} -> X_{26} -> X_{43} -> X_{51} -> X_{87} -> X_{58} ->
X_{86} -> X_{60}
X_{95} -> X_{71} -> X_{55} -> X_{94} -> X_{38} -> X_{47} -> X_{59} -> X_{10} ->
X_{77} -> X_{29}
X_{96} -> X_{13} -> X_{100} -> X_{40} -> X_{91} -> X_{6} -> X_{76} -> X_{85} ->
X_{35} -> X_{32}
X_{30} -> X_{46} -> X_{89} -> X_{15} -> X_{82} -> X_{66} -> X_{67} -> X_{79} ->
X_{11} -> X_{14}
X_{1}

Christofides tour cost: 1907
```

Case 2: With the way our graph is generated in the part with the MST model we only generated 2 times the node count extra edges. Now for this we will be with using the same graph code in that section and we see that not all the nodes are connected together. Here we will have a sparse graph where some nodes only have a few edges. In this case when we run the Christofides Algorithm we will need to compute the shortest path method. In this case we will be using the Dijkstra Method to do so. After this we will run the Chrisofides Algorithm like we did previously did to calculate the Christofides tour cost.

[6]:
```
import itertools
import pulp
import random

node_count = 100  # Change this number if you want more nodes

# Generate node labels like X_{1}, X_{2}, ..., X_{100}
nodes = [f"X_{{{i}}}" for i in range(1, node_count + 1)]

# Generate edges ensuring connectivity
edges = {}
for i in range(1, len(nodes)):
    u = nodes[i]
    v = random.choice(nodes[:i])   # Connect to a previous node
    cost = random.randint(1, node_count)  # Random cost between 1 and node_count
    edges[tuple(sorted((u, v)))] = cost

# Add extra random edges
extra_edges = int(5 * node_count) # Can change this multiplier to add more or␣
 ↪fewer edges like in MST
for _ in range(extra_edges):
    u, v = random.sample(nodes, 2)
    u, v = sorted((u, v))
    if (u, v) not in edges:
        edges[(u, v)] = random.randint(1, node_count)

print(f"Generated graph with {len(nodes)} nodes and {len(edges)} edges.")
```

```
Generated graph with 100 nodes and 556 edges.
```

[7]:
```
import heapq
from collections import defaultdict

# Build adjacency list from your undirected edges
adj = {v: [] for v in nodes}
for (u, v), w in edges.items():
    adj[u].append((v, w))
    adj[v].append((u, w))

# Dijkstra method for single source shortest paths
```

```python
def dijkstra(start, nodes, adj):
    dist = {v: float("inf") for v in nodes}
    dist[start] = 0.0
    heap = [(0.0, start)]
    while heap:
        d, u = heapq.heappop(heap)
        if d > dist[u]:
            continue
        for v, w in adj[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                heapq.heappush(heap, (nd, v))
    return dist

# Metric closure for shortest path distance between every pair
dist = {u: dijkstra(u, nodes, adj) for u in nodes}

# Prim's MST on the complete metric graph
def prim_mst(nodes, dist):
    n = len(nodes)
    start = nodes[0]
    in_mst = {v: False for v in nodes}
    key = {v: float("inf") for v in nodes}
    parent = {v: None for v in nodes}
    key[start] = 0.0
    for _ in range(n):
        u = min((v for v in nodes if not in_mst[v]), key=lambda v: key[v])
        in_mst[u] = True
        for v in nodes:
            if not in_mst[v] and dist[u][v] < key[v]:
                key[v] = dist[u][v]
                parent[v] = u
    mst_edges = []
    degrees = {v: 0 for v in nodes}
    total_cost = 0.0
    for v in nodes:
        if parent[v] is not None:
            u = parent[v]
            mst_edges.append((u, v))
            degrees[u] += 1
            degrees[v] += 1
            total_cost += dist[u][v]
    return mst_edges, degrees, total_cost

mst_edges, degrees, mst_cost = prim_mst(nodes, dist)
```

```
# Find odd-degree vertices in the MST
odd_vertices = [v for v, deg in degrees.items() if deg % 2 == 1]
unmatched = set(odd_vertices)
matching_edges = []

while len(unmatched) > 1:
    i = min(unmatched)
    best_j = None
    best_w = float("inf")
    for j in unmatched:
        if j == i:
            continue
        w = dist[i][j]
        if w < best_w:
            best_w = w
            best_j = j
    matching_edges.append((i, best_j))
    unmatched.remove(i)
    unmatched.remove(best_j)


# Building a multigraph which combines MST edges + matching edges
multi_adj = defaultdict(list)
for u, v in mst_edges:
    multi_adj[u].append(v)
    multi_adj[v].append(u)
for u, v in matching_edges:
    multi_adj[u].append(v)
    multi_adj[v].append(u)

# Eulerian tour using Hierholzers algorithm
def eulerian_tour(adj, start):
    local_adj = {u: list(vs) for u, vs in adj.items()}
    stack = [start]
    path = []

    while stack:
        u = stack[-1]
        if local_adj[u]:
            v = local_adj[u].pop()
            local_adj[v].remove(u)
            stack.append(v)
        else:
            path.append(stack.pop())

    return path[::-1]
```

```
start_node = nodes[0]
euler_tour = eulerian_tour(multi_adj, start_node)

# Shortcut Eulerian tour to get TSP Christofides Algorithm tour
visited = set()
tsp_tour = []
for v in euler_tour:
    if v not in visited:
        visited.add(v)
        tsp_tour.append(v)

# Make it a cycle
tsp_tour.append(tsp_tour[0])

# Compute tour cost using metric distances
tsp_cost = sum(dist[tsp_tour[i]][tsp_tour[i + 1]] for i in range(len(tsp_tour)␣
 ↪- 1))


print("\nChristofides Algorithm TSP tour:")
chunk_size = 10
for i in range(0, len(tsp_tour), chunk_size):
    chunk = tsp_tour[i:i+chunk_size]
    print(" -> ".join(chunk))

print(f"\nChristofides tour cost: {tsp_cost}")
```

```
Christofides Algorithm TSP tour:
X_{1} -> X_{33} -> X_{12} -> X_{48} -> X_{8} -> X_{9} -> X_{32} -> X_{36} ->
X_{34} -> X_{97}
X_{2} -> X_{90} -> X_{94} -> X_{44} -> X_{23} -> X_{66} -> X_{14} -> X_{98} ->
X_{75} -> X_{16}
X_{77} -> X_{40} -> X_{47} -> X_{55} -> X_{6} -> X_{7} -> X_{83} -> X_{52} ->
X_{91} -> X_{46}
X_{45} -> X_{56} -> X_{100} -> X_{28} -> X_{10} -> X_{3} -> X_{43} -> X_{54} ->
X_{57} -> X_{27}
X_{89} -> X_{21} -> X_{67} -> X_{18} -> X_{61} -> X_{64} -> X_{39} -> X_{79} ->
X_{82} -> X_{63}
X_{80} -> X_{99} -> X_{24} -> X_{15} -> X_{65} -> X_{59} -> X_{25} -> X_{72} ->
X_{74} -> X_{81}
X_{37} -> X_{71} -> X_{84} -> X_{30} -> X_{58} -> X_{76} -> X_{60} -> X_{78} ->
X_{95} -> X_{85}
X_{93} -> X_{51} -> X_{42} -> X_{87} -> X_{19} -> X_{17} -> X_{49} -> X_{38} ->
X_{29} -> X_{22}
X_{41} -> X_{5} -> X_{26} -> X_{69} -> X_{88} -> X_{35} -> X_{62} -> X_{86} ->
X_{50} -> X_{68}
X_{73} -> X_{53} -> X_{4} -> X_{96} -> X_{70} -> X_{92} -> X_{13} -> X_{20} ->
```

```
X_{31} -> X_{11}
X_{1}

Christofides tour cost: 2205.0
```