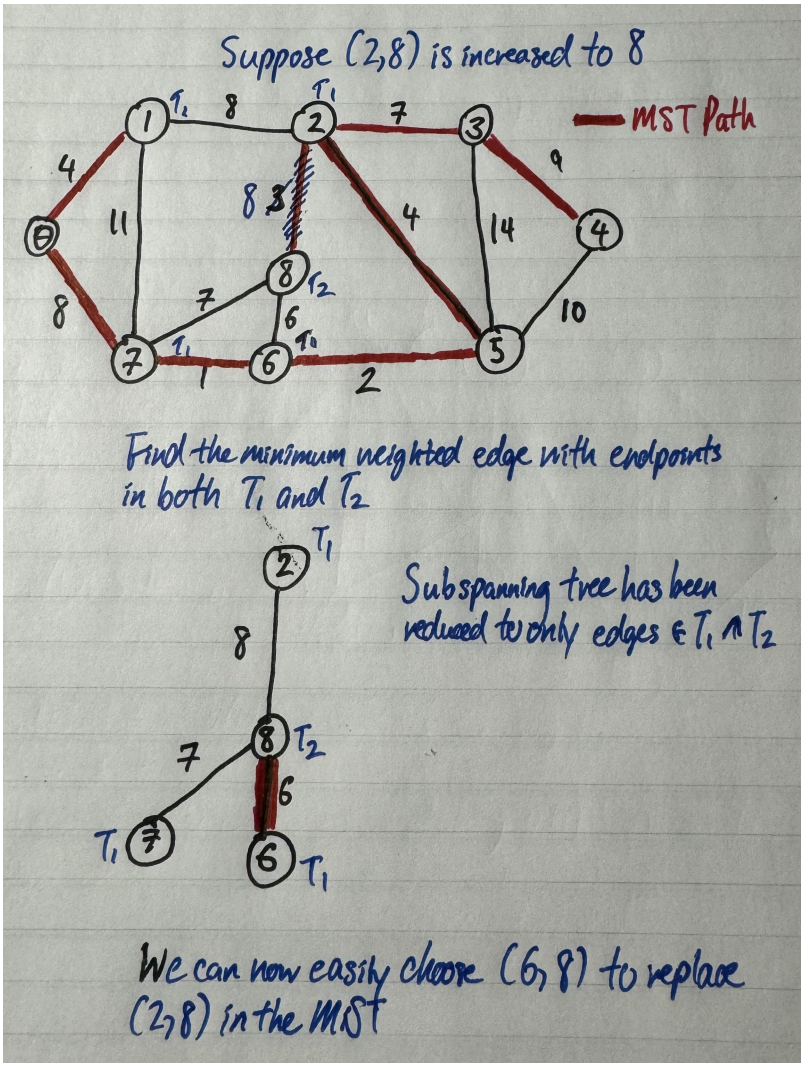
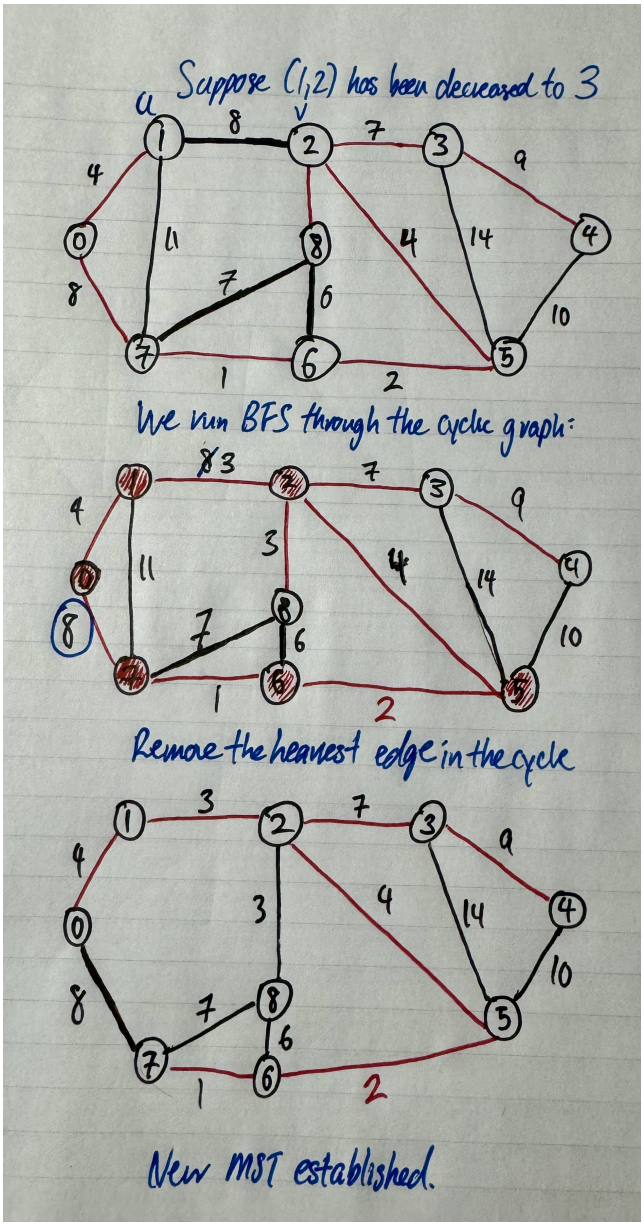
Finding new minimum spanning tree T of weighted undirected graph $G = (V, E)$ after one edge $e \in T$ is increased:

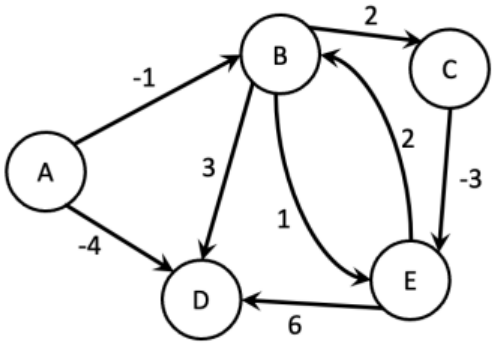
After the weight of edge e is increased, remove edge e from Graph G and MST T. Run a Breadth-First Search in $O(|V| + |E|)$ to determine the vertices in components T_1 and T_2. Since T is now two disconnected components T_1 and T_2, examine every edge in graph G using $O(|E|)$ and determine the minimum weighted edge that has endpoints in both T_1 and T_2. This edge will become the new edge that reconnects T. This algorithm runs in $O(|V| + |E|)$ because of Breadth-First Search and runs in $O(|E|)$ due to observing every edge in G. It returns the new MST because it utilizes the cut property to find the minimum edge weight that crosses T_1 and T_2, updating the original MST.



Finding new minimum spanning tree T of weighted undirected graph $G = (V, E)$ after one edge $e \notin T$ is decreased:

Decrease the weight of the edge not in MST T and let (u, v) describe the vertices of that edge. Run a Breadth-First Search from u to v, revealing a cyclical path. This search algorithm must take $O(V)$ because it is traversing through an MST tree structure. Remove the heaviest edge in the cycle, taking a time of O(1). This will turn T back into an MST because it remains connected while a swapping operation is performed to disconnect the heaviest edge. This algorithm performs graph traversal through the MST in $O(|V|)$ and compares and removes edges in constant time, resulting in $O(|V|)$ runtime.





Dikjstra's:

Vertex	Dist					Prev				
A	0	0	0	0	0	×	×	×	×	×
B	∞	−1	−1	−1	−1	×	A	A	A	A
C	∞	∞	1	1	1	×	×	B	B	B
D	∞	−4	−4	−4	−4	×	A	A	A	A
E	∞	∞	0	0	0	×	×	B	B	B
Visiting	A	D	B	E	C	A	D	B	E	C

Queue
ABDCE
BDCE
BCE
CE
C

`Shortest paths:`
A→A: 0
A→B: −1
A→C: 1
A→D: −4
A→E: 0

Dijkstra's algorithm operates on graphs with only non-negative edge weights. Starting from vertex A, it explores the shortest path to each of its neighbors and updates shortest distances as needed. It is greedy in nature and visits the lightest neighboring edges first, marking vertices as visited using a priority queue. The issue arises when a less optimal edge has been dismissed but has a neighboring edge that could reduce the path weight. Once a vertex has been visited, it cannot be reconsidered. This algorithm neglects the fact that an edge could reduce the current path distance. Each time a shortest distance vertex is extracted from the priority, the algorithm explores its edges and relaxes them if the distance to the connected vertex is less than previously known. Dijkstra's does not operate correctly on graphs with negative edge weights.

Bellman-Ford:

Current Vertex	A	B	C	D	E
	0	∞	∞	∞	∞
A	0	-1	∞	-4	∞
B	0	-1	1	-4	0
C	0	-1	1	-4	-2
D	0	-1	1	-4	-2
E	0	-1	1	-4	-2

`Shortest paths:`
A$\rightarrow$A: 0
A$\rightarrow$B: -1
A$\rightarrow$C: 1
A$\rightarrow$D: -4
A$\rightarrow$E: -2
Bellman Ford's algorithm is capable finding shortest paths with both positive and negative edge weights, as long as there are no negative cycles. Vertices are visited in lexicographical order and their edges are relaxed if a shorter path is found. The shortest path can have at most $V - 1$ edges, so the algorithm iteratively performs relaxation $V - 1$ times, where v is total number of vertices. Here, the optimal path was found in the first round of edge relaxations.

Finding contiguous subsequence with the maximum sum:

Pseudocode:

1. Initialize a `currSum` variable to 0
2. Initialize a `maxSum` variable to 0
3. Initialize `tempStart, startIndx, endIndx` to 0
4. For i from index 0 to seq.length - 1:

   `currSum` += seq[i]

   If `currSum` > `maxSum`:

   `maxSum` = `currSum`

   `startIndx` = `tempStart`

   `endIndx` = i

   If `maxSum` < 0:

   `currSum` = 0

   `tempStart` = i + 1
5. Return the subsequence from `startIndx` to `endIndx` of input sequence

`Recurence: M[i] = max(seq[i], M(i-1) + seq[i])`
`M[i]: sum of the max contiguous subsequence ending at position i`
This algorithm traverses the sequence while maintaining two sum variables of the ongoing and global subsequences. In each iteration, the algorithm considers updating the global maximum if the current subsequence has a sum greater. If the current subsequence becomes negative, the current sum is no longer advantageous to consider and is reset. The overall runtime is proportional to the length of the sequence as elements in the sequence are visited only once, having linear complexity.