Lecture 5: Greedy Algorithms

subject	TDT4121
topics	Djikstra's Algorithm, Huffman Codes, Interval Scheduling Problem, Kruskal's Algorithm, Minimum Spanning Trees, Optimal Caching Problem, Prim's Algorithm, Shortest Path Problem
created	@September 24, 2022 11:17 AM

Chapter 5 - Greedy Algorithms

> An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. One can often design many different greedy algorithms for the same problem, each one locally, incrementally optimizing some different measure on its way to a solution

Goals 🏁

The student should be able to:

- Understand the different *Interval Scheduling* problems ✅
- Understand the *Optimal Caching Problem* ✅
- Understand the *Shortest Path Problem* and how *Dijkstra's algorithm* solves it ✅
- Know what a *Minimum Spanning Tree* is and understand the *Minimum Spanning Tree Problem* ✅
- Understand *Prim's* and *Kruskal's algorithm* ✅
- Understand *Huffman* and *Huffman Codes*

Interval Scheduling Problem

This algorithm always choose an event that ends as early as possible. The algorithms will always choose the event that finished as early as possible for all elements of R. When the algorithm does this, it will always returns an optimal set.

The goal here is to execute as many tasks as possible, that is, to maximise the throughput. It is equivalent to finding a maximum independent set in an interval graph.

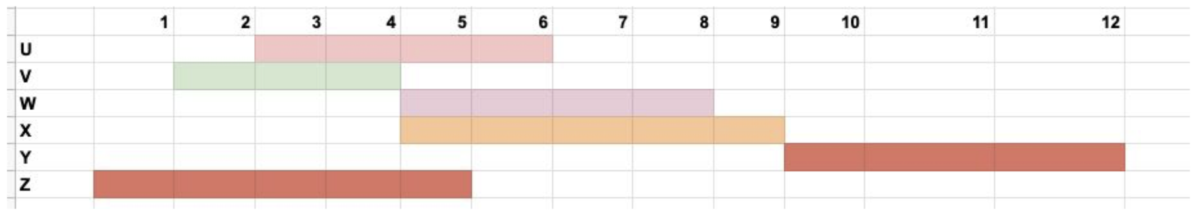
Interval scheduling table of activities

We have the following table of activities:

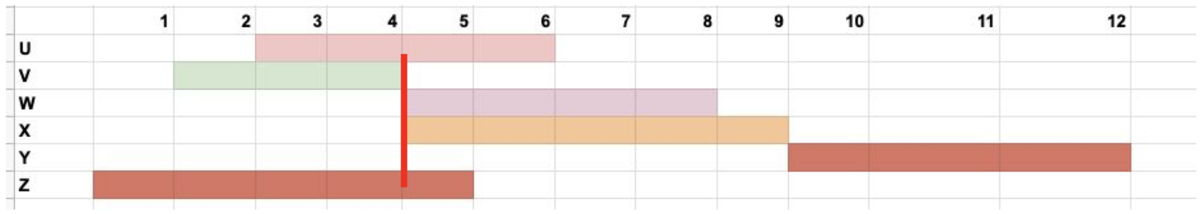
Activity	**U**	**V**	**W**	**X**	**Y**	**Z**
Start time	3	2	5	5	10	1
Finish time	6	4	8	9	12	5

After running the interval scheduling algorithm, it will choose the V element first as it has the smallest finishing time. After that, it will move on to the W element and lastly the Y element. The size of A will then be $A = [V, W, Y]$.

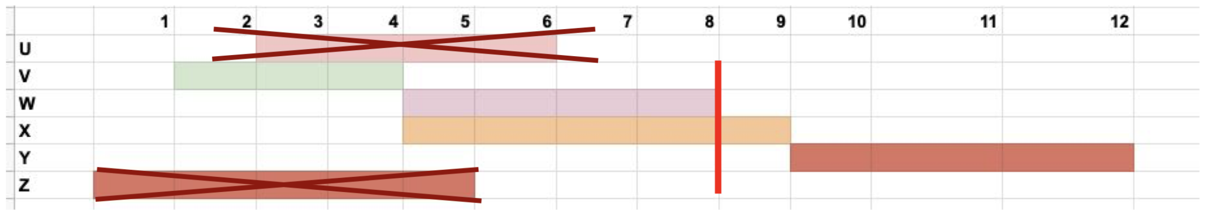
If we take a closer look at why, we see that we have the schedule



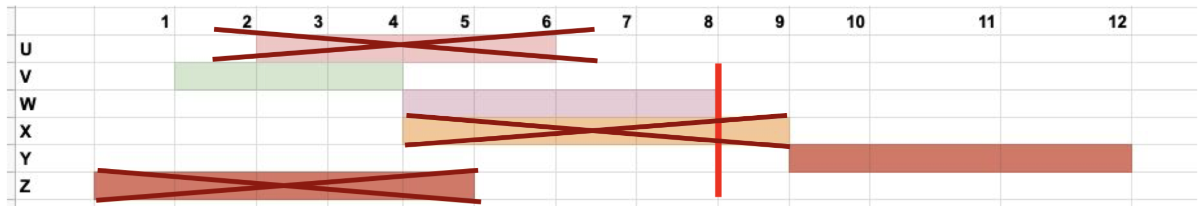
We take a look at the earliest finish and find that V finishes earliest.



Here we see that with the earliest finish V, the algorithm is incompatible with U and Z, meaning the next earliest finish is W.



With W, it is now incompatible with X. Our next earliest finish is Y.



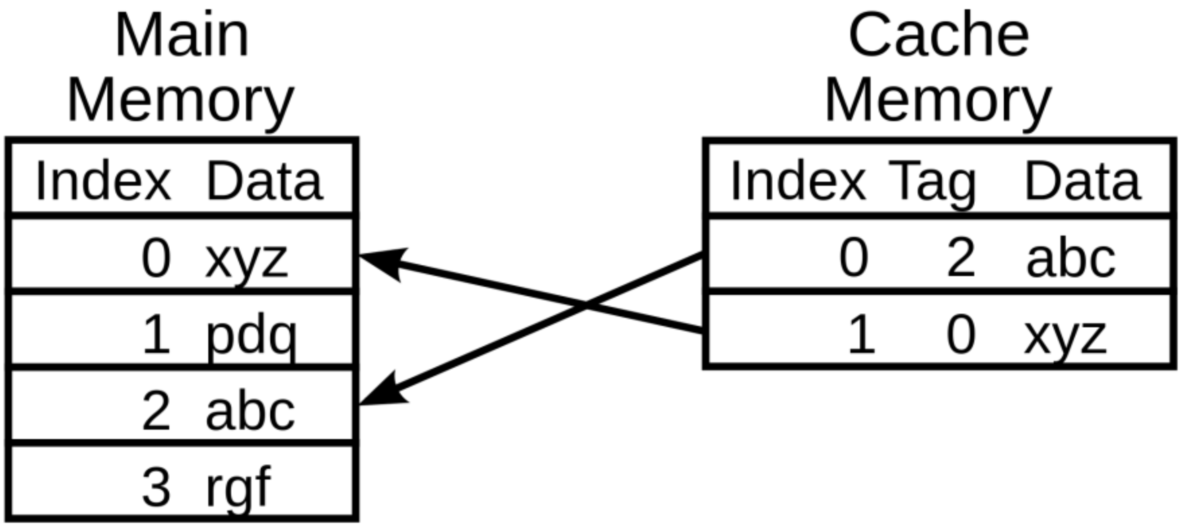
Our solution is now $A = [V, W, Y]$.

Optimal Caching Problem

Optimal caching is a technique that reduces the number of cache misses over any way you might think about managing the cache. The farthest in the future algorithm is optimal.

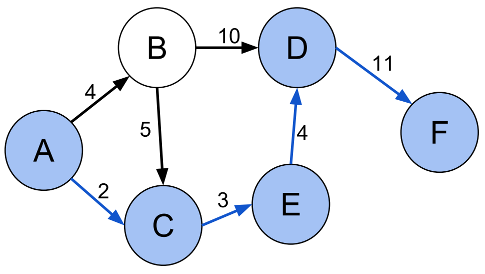
For example, if we want to Bring E in the Cache we have to look for the data which is requested farthest from this point, if A is requested 70 steps from now and C is requested in 3 steps from now, then we definitely have to evict A to bring in E. But we don't know the future, we therefore need an **LRU** (least recently used) algorithm. It checks for some past data accesses - things that has been recently requested will be requested again soon.

The *cache* is a small and fast memory. Cache process the sequence of “**page requests**”. Page requests are if a client wants to access something in memory and it is in big slow memory, and if it is not in small fast memory then you have to bring it in. A **cache miss** is when you request some data from the cache but it is not stored there.



Shortest Path Problem

The shortest path problem involves finding the shortest path between two nodes in a graph. The shortest path such that the sum of weights of the graphs edges is the lowest possible. Shortest path algorithms are applied to automatically find directions between physical locations, such as driving directions on web mapping websites like MapQuest or Google Maps. For this application fast specialised algorithms are available.



The shortest path problem can be defined for graphs, whether **undirected**, **directed** or **mixed**. It is defined here for undirected graphs; for directed graphs the definition of path requires that consecutive vertices be connected by an appropriate directed edge.

Djikstra’s Algorithm

Djikstra’s algorithm finds the shortest path between between nodes. It starts at a given source node and picks unvisited nodes with the lowest distance, and calculates the distance through it to each unvisited neighbour and then updates the neighbours distance if it is smaller.

Although the algorithm is meant to be used on a directed graph, we can make a graph directed by adding edges in each direction between every node that has an edge.

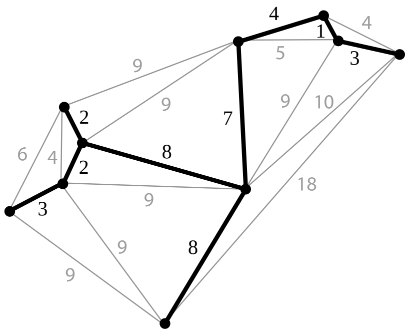
An *undirected* graph is basically the same as a *directed* graph with **bidirectional** connections (= two connections in
opposite directions) between the connected nodes.

So you don't really have to do anything to make it work for an undirected graph. You only need to know all of the nodes that can be reached from every given node through e.g. an **adjacency list**.

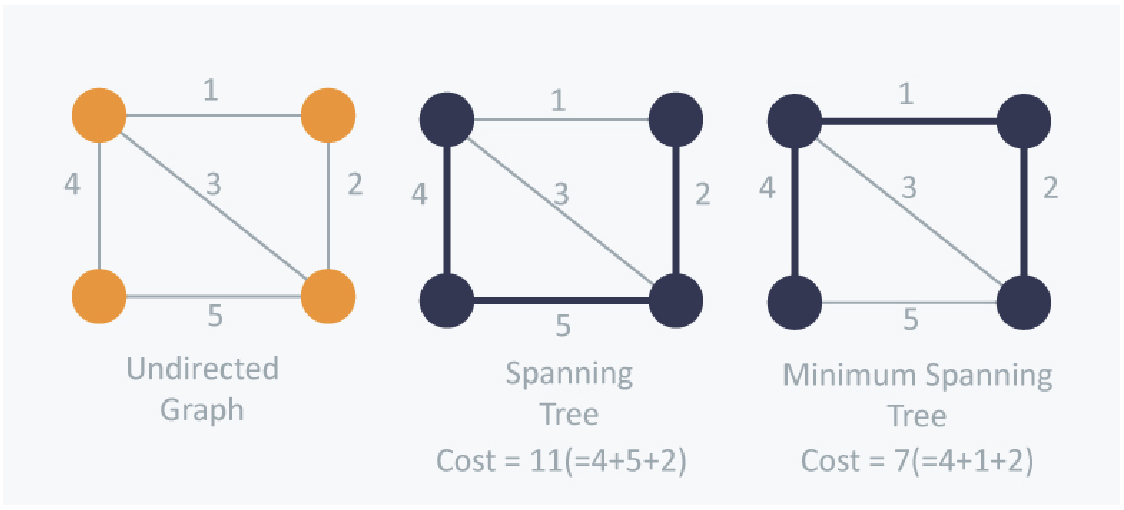
Djikstra's algorithm works because all edge weights are non-negative values, and the node with the lowest weight is always chosen. Because the path with the lowest weight is always chosen, it will always compute the shortest path.

Minimum Spanning Trees

The minimum spanning tree is a subset of the edges in a connected undirected graph. This is the spanning tree whose sum of edge weights is the lowest possible weight. If we therefore have edge (u, v) that has a lower weight than all other edges in the graph, it will therefore be included in the minimum spanning tree.



Even simple graphs like the following, has several spanning trees, but only one *minimum spanning tree*.



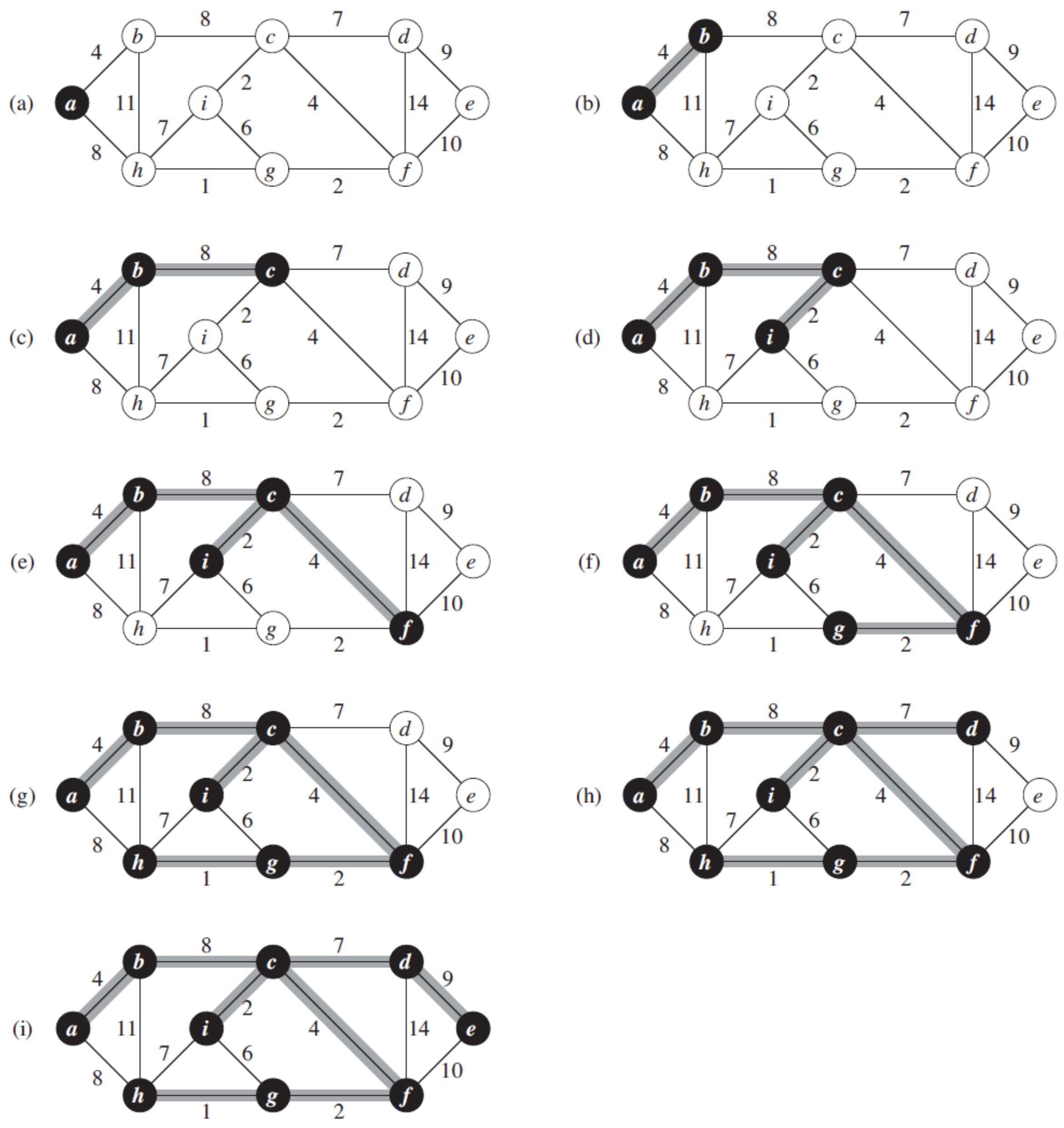
Minimum Spanning Tree Problem

The minimum labeling spanning tree problem is **to find a spanning tree with least types of labels if each edge in a graph is associated with a label from a finite label set instead of a weight**. A bottleneck edge is the highest weighted edge in a spanning tree.

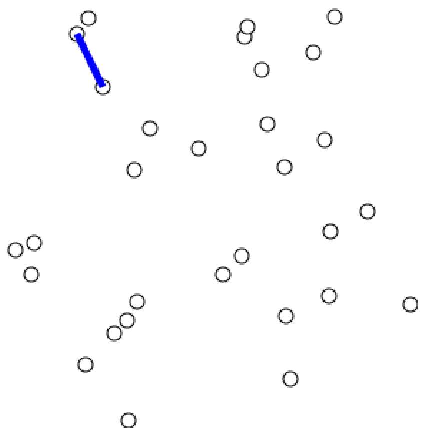
Prim's Algorithm

This is one algorithm to find a minimum spanning tree. Step by step:

1. Choose a random node and highlight it
2. Find all the edges that go to unhighlighted nodes. Highlight the edge with the lowest weight.
3. Highlight the node you just reached.
4. Look at all the nodes highlighted so dar. Highlight the edge with the lowest weight
5. Highlight the node you just reached
6. Highlight the edge with lowest weight. Choose from all the edges that
    a. Come from all of the highlighted nodes.
    b. Reach a node that you haven't highlighted yet
7. Repeat steps 5 and 6 until there are no more un-highlighted nodes.



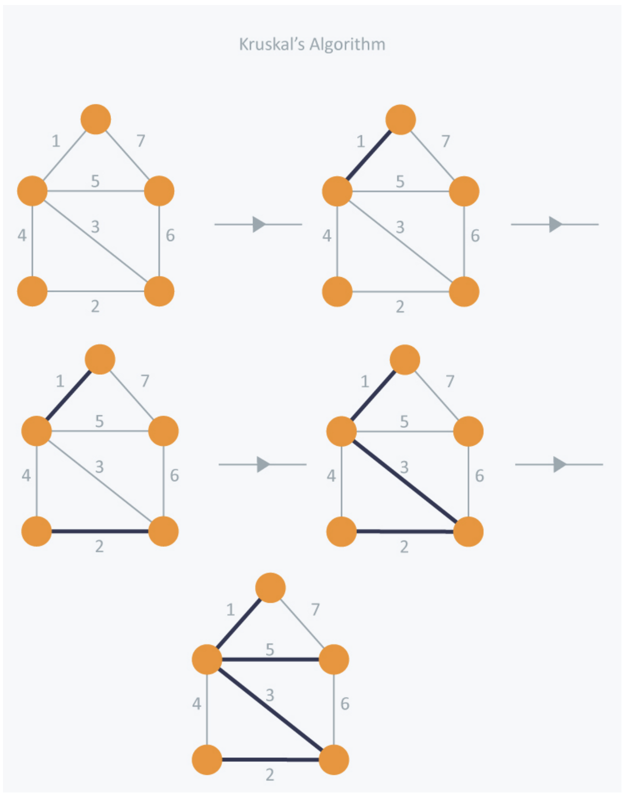
Animated:



Kruskal's Algorithm

This is an algorithm to find the minimum spanning tree. It works step by step:

1. Find the edge with the least weight and highlight it
2. Find the next edge with the lowest weight and highlight it
3. Continue selecting the lowest edges until all nodes are in the same tree
    a. If you have more than one edge with the same weight, choose an edge with the lowest weight.
    b. Be careful not to complete a cycle (route one node back to itself). If your choice completes a cycle, discard your choice and move onto the next largest weight.



Considering a statement about Minimum Spanning Trees

We have a weighted undirected graph $G = (V, E)$ where all of the edges are weighted differently. The nodes V are divided into two disjoint sets X and Y. We consider Kruskal's algorithm to find the minimum spanning tree. This algorithm maintains a forest, initially consisting of unconnected individual vertices and disjointed sets of data. To compute the minimum spanning tree, the algorithm will find the lowest weighted union between the disjointed data sets, meaning the minimum spanning tree will include the edge between X and Y with the lowest weight.

Huffman Coding

Goal: Reduce the code tree