CS2040 Summary

Week 13, AY 19/20 Sem 2

Wang Zhi Jian
wzhijian@u.nus.edu

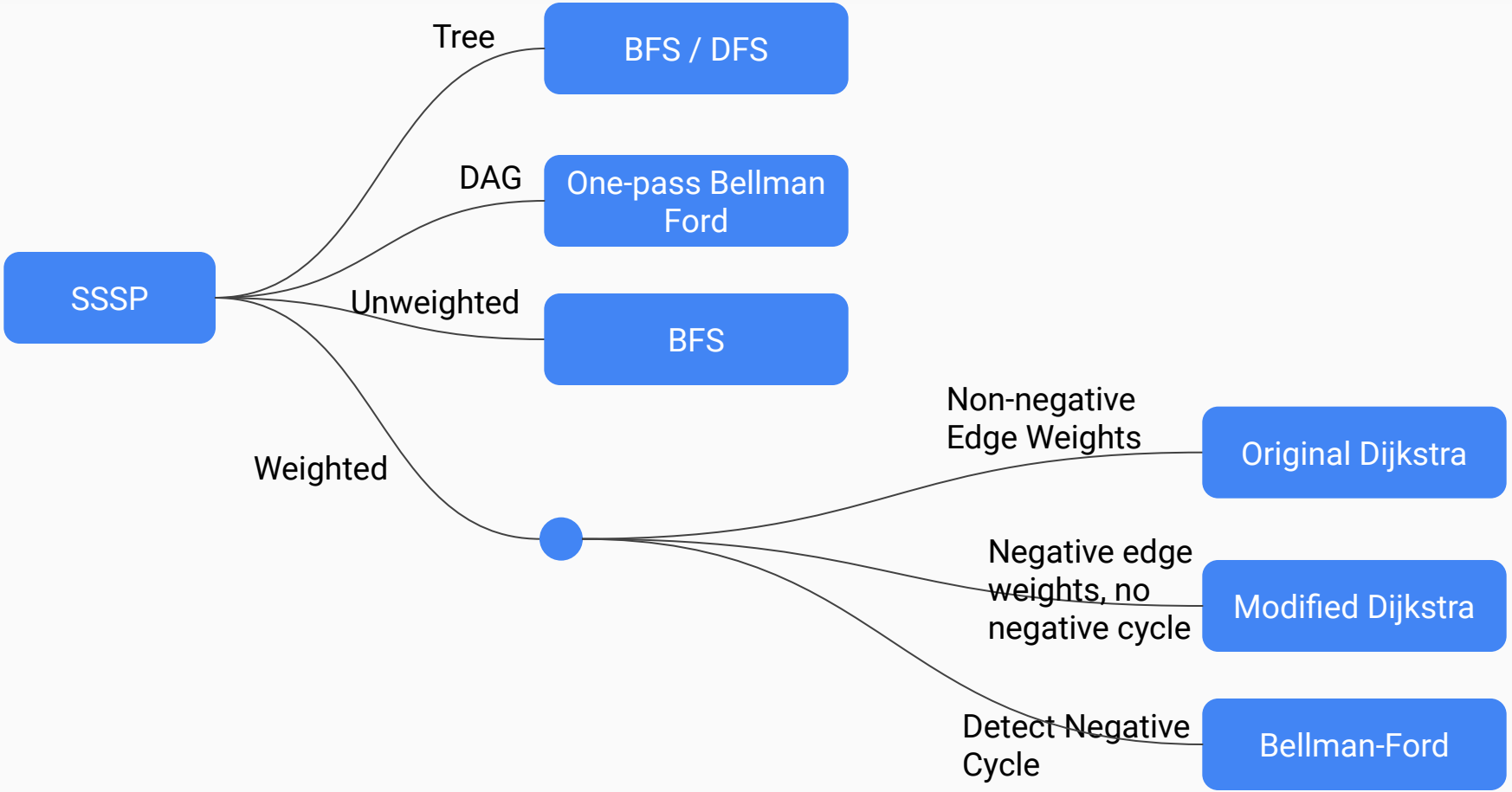
Linear Data Structures

Linked List	**Find**: Locate the position of an element in the list. O(*n*). **Insert**: Given a node, insert an element before/after it. O(1). **Delete**: Given a node, remove it from the list. O(1).
Queue	**Enqueue**: Add new element to the back of the queue. O(1). **Dequeue**: Remove element at the front of the queue. O(1). **Peek**: Returns element at the front of the queue. O(1).
Stack	**Push**: Add new element to the top of the stack. O(1). **Pop**: Remove element from the top of the stack. O(1). **Peek**: Returns element at the top of the stack. O(1).
Hash Table	**Insert**: Add an element to the hash table. Expected O(1). **Delete**: Remove an element from the hash table. Expected O(1). **Find**: Check if an element is in the hash table. Expected O(1).

Non-Linear Data Structures

UFDS	**Find**: Find the representative of the set an element is in. **Union**: Given two elements, merge the sets they are in. For both of the operations: Normal: O(n). Union by rank/size: O(log n). Union by rank + path compression: O(α(n)).
Priority Queue	**Enqueue**: Insert a new element into the priority queue. O(log n). **Dequeue**: Remove the element with the highest (resp. lowest) value from the priority queue. O(log n). **Peek**: Retrieve the element with the highest (resp. lowest) value from the priority queue. O(1).
AVL	**Insert**: Insert an element into the AVL tree. O(log n). **Find**: Check whether a value exists in the AVL tree. O(log n). **Delete**: Remove a value from the AVL tree. O(log n). **Floor**: Find next value in AVL tree ≤ x. O(log n). **Ceiling**: Find next value in AVL tree ≥ x. O(log n). **Rank**: Find how many values in AVL tree ≤ x. O(log n). **Select**: Find the value of the kth smallest element in the AVL tree. O(log n).

Shortest Path Algorithms



Common Techniques

Lazy Deletion: Delete from data structures where you cannot access internal elements (e.g. stack, queue, priority queue) only when it is convenient (when the element is at the top). (e.g. Tutorial 3 Q3)

Preprocessing: Convert the data you are given into a format that is easier to work with, e.g. for answering queries quickly. (e.g. Tutorial 4 Q3)

Meet in the Middle: Attack the problem from two ends, e.g. run Dijkstra's Algorithm from both source and destination. (e.g. Tutorial 4 Q4, Tutorial 11 Q4)

Vertex/Edge Splitting: Split vertices and edges into two or more vertices / edges.

Graph Duplication: Create multiple copies of the graph to account for additional parameters. (e.g. Tutorial 11 Q3)

Logarithms / Mathematical Manipulation: e.g. convert product into sum of logarithmic terms. Try to make an equation and see if you can manipulate it. (e.g. Tutorial 10 Q3)

Proofs and Finding Counterexamples

Proof by Contradiction: Assume the given statement is not true. Try to find an absurd or impossible situation that will occur as a result.

Proof by Counterexample: Find one example which the statement does not work on.

Data Structures: Consider empty data structure (empty linked list, empty stack, empty AVL), data structure with only one element (linked list with one node, AVL with only the root node, etc), data structures where elements are ordered in a certain way.

Graphs: Consider special graphs (e.g. graph that looks like linked list, star graph, graph where all the nodes are in a single cycle, disconnected graph, graph with only one node, tree).

Stuck? See if some of the questions here can get you going...

- Try working through some small examples or small cases. Do you find any patterns?
- Have you used all the information given in the problem? Each piece of information is given to you for a reason - you should need all the pieces of information most of the time for the best solution.
- Try to consider the problem for a special case, e.g. for a graph problem, consider a tree. Is there a faster algorithm? Can you extend this algorithm from the special case to the general case?
- If you have a sequence of integers, what if you sort the sequence of integers? Does it make it more convenient to work with? If not, is there some other ordering that is easier to work with?
- Can you apply any of the standard procedures to the problem? e.g. If you have two sorted lists, we can use the merge procedure in mergesort to merge them into one big sorted list in $O(n)$.
- If you have a graph, what if you consider the complement of the graph? If it is directed, what if you flip all edges? Can you design a faster algorithm after that? If you don't have a graph, what if you consider some kind of 'inverse' of the problem?
- Can you convert any of the operations in the problem to any of the standard data structure operations in some way?
- Enumerate the data structures and algorithms that you have learnt in your head. Is any one of them applicable to this problem? Try to classify what type of problem this is, and see if you can narrow down what you should use.