Greedy Algorithms - Notes

Alperen Onur (z5161138)

August 2024

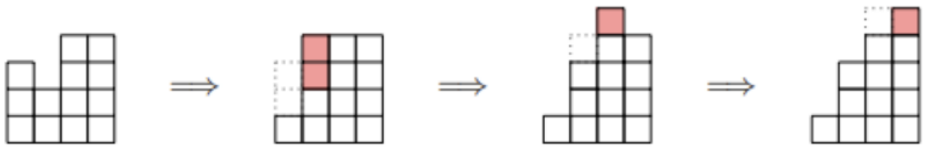
1 Greedy Algorithms

A greedy algorithm is one that solves a problem by dividing it into stages, and rather than exhaustively searching all solutions to a stage (to get from one stage to the next), instead only considers the choice that appears best at that stage of construction of a solution.

This obviously reduces the search space, but it is not always clear whether the locally optimal choice leads to the globally optimal outcome.

2 Stacking Blocks

You are given n stacks of blocks. The ith stack contains $h_i > 0$ identical blocks. For each stack $1 \leq i \leq n$, we can move any number of blocks from stack i to stack $i+1$. Describe an $O(n)$ algorithm to decide if its possible to move blocks between adjacent stacks such that the size of the stack of blocks are in strictly increasing order.



Algorithm.
Let the total number of blocks be $h_1 + .. + h_n = X$. For there to be a sequence of stacks that is increasing, we need $X-(1+2+...+n-1)$ to have an integer positive solution. The sequence is only possible if $h \geq 1, h_1+h_2 \geq 2$ and so on but more generally $H_k = h_1 + ... + h_k \geq 1 + 2 + ... + k = \frac{k(k+1)}{2}$, for each $1 \leq k \leq n$. We can now compute H_k in linear time (for each k) since $H_{k+1} = H_k + h_{k+1}$. Therefore, it is only possible iff $H_k \geq \frac{k(k+1)}{2}$ for all $1 \leq k \leq n$.

3 Fractional Knapsack

There are n flavours of frozen yogurt at your local shop. The ith flavour is dispensed from a machine of capacity c_i litres and contributes $\frac{d_i}{c_i}$ deliciousness per litre; the entire machine's worth contributes d_i.

You have a giant tub of C litres you want to fill the tub so as to maximise the total deliciousness.

The above problem is a classic example of a **fractional knapsack**. We're given a list of n items described by there weights w_i, values v_i and a maximal weight limit W of your knapsack. You can take any fraction between 0 and 1 of each item. Our task is the select a non-negative quantity of each item, with total weight not exceeding W and maximal total value. The solution is bluntly simple, to fill the entire knapsack with the item of highest value per unit weight. If you run out of an item, use the next best and so on.

4 0-1 Knapsack

Sadly, the frozen yogurt shop has now closed down, so you must go to the supermarket instead. There, you can find n flavours of frozen yoghurt in tubs. The ith flavour comes in a tub of capacity c_i litres and contributes d_i deliciousness.

You again want to buy at most C litres, with a maximum total deliciousness.

This is a similar example to the prior except we can only take integer amounts of yogurt (we take per litre). This is now a **0-1 Knapsack** problem. We have a list of n discrete items described by their weights w_i, values v_i and a maximal weight limit W of your knapsack. Our task is to find a subset S of items with total weight not exceeding W and maximal total value. The solution for this problem isn't solved as simply as before, we'll need some extra checks per stage of the problem to ensure it's solved.

5 Activity Selection & Exchange argument

A list of n activities, with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Algorithm.
Among those activities which do not conflict with the previously chosen activities, always choose the activity with the earliest end time. Suppose we are up to activity i, starting at s_i and finishing at f_i, with all earlier finishing activities already processed. If all previously chosen activities finished before s_i, activity i can be chosen without conflict, otherwise we discard activity i. To do this efficiently, we keep track of only the latest finishing time among chosen activities. Since we process activities in increasing order of finishing time, this is just the finishing time of the last activity to be chosen. Every activity is therefore chosen or discarded in constant time so part of this algorithm takes $O(n)$ time. Thus the entire algorithm runs in $O(n \log n)$.

Correctness.
We can use the idea of an exchange argument to show that any other selection of activities can be morphed into our greedy selection. A specific about this problem is that the new selection wont have any conflicts to our greedy solution and selects the same number of activities.

Suppose the greedy selects activities $G = g_1, ..., g_r$, and consider some alternative selection $A = a_1, .., a_s$, each in ascending order of end time. We need to show that no other valid selection can have more activities regardless of how A was chosen. This is because we don't get to choose how this alternative selection was made.

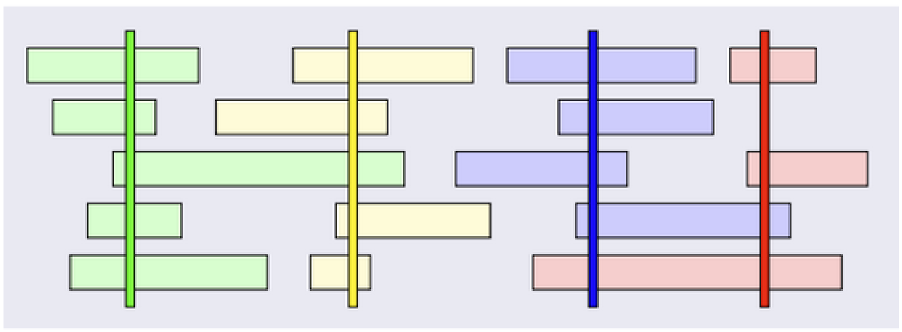
Suppose $g_1 = a_1, g_2 = a_2, .., g_{k-1} = a_{k-1}$ but $g_k \neq a_k$. Then we'll be comparing

$$A = g_1, .., g_{k-1}, a_k, a_{k+1}, ..., a_s$$
$$A' = g_1, .., g_{k-1}, g_k, a_{k+1}, .., a_s$$

All selections of activities in A and A' are valid selections but notice that g_k doesn't conflict with $g_1, .., g_{k-1}$. g_k finishes no later than a_k which in tern finishes before any of $a_{k+1}, .., a_s$ start. In these types of proofs, we want to resolve all disagreements between our greedy solution and alternate solution. Continuing this way, we resolve all disagreements in the alternate solutions until one of time runs out of agreements. However, the greedy solution can not run out first; if a_{r+1} doesn't conflict with $g_1, ..., g_r$ then the greedy wouldn't end after picking g_r. Therefore $s \leq r$; no alternative selection can take more activities than our greedy solution.

6 Interval Stabbing.

You are given a set of n intervals $\chi = \{I_1, ..., I_n\}$. Each interval I_i is defined by two points, $l_i and r_i$. We say that a set P of points stabs χ if every interval in χ contains at least one point in P. Design a $O(n \log n)$ algorithm that returns the smallest set of points that stabs χ.



Algorithm.
Order the intervals in terms of finishing order. Iterate through the intervals and stab all non stabbed intervals that cross the earliest finishing unstabbed interval. Clearly $O(n \log n)$.

Correctness.
We'll conduct an exchange argument to prove our algorithm. This is actually a mult-layered proof where we need to proof that we are indeed choosing the best interval at each step of the algorithm and then use an exchange to show any other solution can't be any better than ours.

Claim. *There exists an optimal solution that only stabs the rightmost section of unstabbed intervals.*
Let x be a non-right endpoint to any interval and consider any optimal solution S that contains x. If $x > r_i$, then for all $1 \leq i \leq n$, $S \setminus \{x\}$ stabs all intervals that S stabs and so S is strictly a more optimal choice. Therefore, we may assume that there always exists some interval I_i, such that $r_{i-1} < x < r_i$; we always should be stabbing within the interval range. In particular, let r_i be the closest right end point larger than x. Then any interval stabbed by x must also be stabbed by r_i and so $S \setminus \{x\} \bigcup \{r_i\}$ stabs the same amount of intervals that S stabs. Now that we have proved our algorithm heuristic is correct, lets complete our proof by assuming the only other possible solution.

Claim. *There exists an optimal solution that stabs the earliest right endpoint.*
Let x denote the overall earliest right endpoint, and consider any optimal solution S that does not contain x. Then the first stabbing point must either come before or after x. If the first stabbing comes after x, then S misses the first interval altogether, which means that S cannot be a valid solution. Therefore, the first stabbing point point must come before or on the x point. Let y denote the first stabbing point in S. But by the previous claim, it follows that $S^* := (S \setminus \{y\}) \bigcup \{x\}$ is a valid solution. Finally since $|S| = |S^*|$, it follows that S^* is also optimal.

7 Cell towers & Greedy Stays Ahead

Along the long, straight road from Balladonia to Caiguna, houses are scattered quite sparsely, sometimes with long gaps between two consecutive houses. Telstra must provide mobile phone service to people who live alongside the road, and the range of the Telstra's cell tower is 5km. Our task is to design an algorithm for placing the minimal number of cell towers alongside the road such that all houses are covered.

Algorithm.
Process houses from west to east. The first house must be covered by some tower, which we place 5kms to the east of this house. The tower may cover some other houses, but eventually w should reach a house that is out of range of this tower. We should place a second tower 5km east of that house. Continue until all houses are covered. Runs in $O(n)$ time if the houses are ordered, $O(n \log n)$ otherwise.

Correctness.
We'll use a greedy stays ahead argument, similar to fast induction. The key claim is: for all k, the kth tower in the greedy solution is at least as far east as the kth tower in any other placement. If this is true, then it would be impossible to cover all the houses using fewer towers.

Our base case is $k = 1$, where he greedy solution places its first tower five kilometres east of the first house. If an alternative placement were to place it further east, it would not cover this first house, so the claim is true for $k = 1$. Suppose the claim is true for some $k-1$ and consider the kth tower. The greedy placed towers at g_{k-1} and g_k and there is a house h such that $g_{k-1} + 5 < h = g_k - 5$. The alternative placed towers at a_{k-1} and a_k. The induction hypothesis tells us that $a_{k-1} \leq g_{k-1}$, so if $a_k > g_k$ then house h will not be covered by either of the towers either side of it. Therefore $a_k \leq g_k$ as required.

Road Trip You are driving along one road from Sydney to Melbourne, and you can only drive 100km in a day. You have a list of n hotels sorted by its distance from Sydney and you need to ensure that you stop at a hotel every night. Ideally, you'd like to stop fro the fewest nights possible - how should you plan your trip?

Algorithm.
At each night, pick the furthest hotel that is reachable within the 100km range from where you are. Repeat this until the end of your trip. This is clearly an $O(n)$ algorithm.

Correctness.
We'll conduct a **greedy stays ahead proof**. Let g_i be the hotel we stop on night i, G be our greedy solution

that picks $g_1, g_2, ..m, g_m$ hotels and some arbitrary solution O that picks $o_1, o_2, ..., o_m$.

On the first night, g_1 is the farthest hotel we can stop at on night 1. g_1 is at least as far as o_1. It's clear how we can proceed with induction here. Assume the algorithm is correct for the first $k-1$ iterations; g_{k-1} is at least as far as o_{k-1}. That means that every hotel withing 100km of g_{k-1} is also 100kms of o_{k-1} but then since we always choose the furthest away hotel, g_k must be at least as far as o_k. Therefore, G will use at most as many stops as O but since O is arbitrary, G will always win over other solutions.

8 Optimal Ordering

If we identify inversions when solving a problem with a greedy algorithm, we should try to fix these inversions if we can to improve the overall runtime of the algorithm. The result of fixing these inversions in a greedy algorithm is that the solution could not have gotten any worse as a result. We'll often have to make this choice when two heuristics of a problem need to be considered for some problem.

Minimising Job Lateness.
We have a list of n jobs, with duration's d_i and deadlines due_i. All jobs have to be completed, but only one job can be performed at any time. If a job i is completed at a finishing time $finish_i > due_i$, then we sat that it has incurred a lateness $late_i = finish_i - due_i$. Our task is to schedule all the jobs to minimise the largest lateness.

There are two heuristics to consider; jobs of short duration to be scheduled first and jobs due earlier to be scheduled first. There is no compromise to be made here and so we should ignore job duration's and schedule jobs in ascending order of deadlines. Jobs form an inversion if job i is scheduled before job j but due after. If we swap these adjacent jobs when an inversion occurs, we reduce the overall lateness!

9 Prefix Codes

A any encode is any encoding into a binary tree where left branches represent '0' and right branches represent '1'. In a prefix encoding, no ancestor of a codeword is itself a codeword, so all codewords must be located at the leaves.

Given the frequencies of each symbol, design an optimal prefix code which minimises the expected length of an encoded text. The quantity we want to minimise is

$$\begin{aligned} L &= \sum_{i \in S} len(C_i) \times f_i \\ &= \sum_{i \in S} depth(i) \times f_i \end{aligned}$$

The Huffman code.
We can solve this using Huffman encoding.

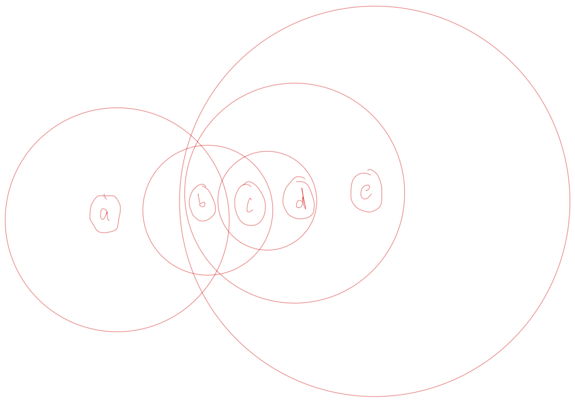
- Make a priority queue of symbols, each weighted by its frequency
- Repeatedly combine the two lowest frequency symbols. Remove the two symbols of lowest frequency, say x and y with frequencies f_x and f_y. Make a new symbol xy with frequency $f_x + f_y$ and insert into a priority queue.
- As we do this, we maintain a forest, initially consisting of n isolated vertices. When symbols x and y are combined, we add a new vertex representing xy whose left and right children are x and y. This records that the codeword for x is to be found by appending a '0' to the codeword for xy and likewise for y by appending a '1'.

10 Tsunami Warning

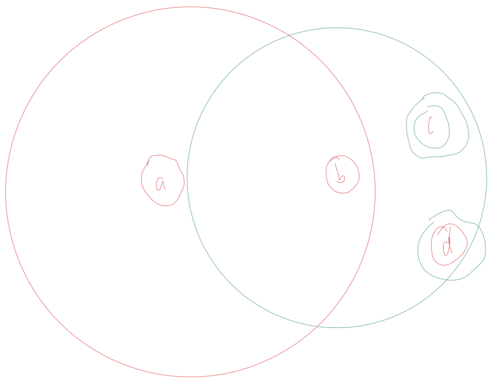
There are n radio towers for broadcasting tsunami warnings. You are given then (x, y) coordinates of each tower and its radius of range. When a tower is activated, all towers within the radius of range will also activate, and those can cause other towers to activate and so on. You need to equip some of these towers with seismic sensors so that when these sensors activate the towers where these sensors are located all towers will eventually get activated and send a tsunami warning.

Attempts to solve problem.
There is a greedy solution but we need o be very mindful of our heuristic. Consider the following two attempts

- Find the unactivated tower with the largest radius (breaking ties arbitrarily), and place a sensor at this tower. Find and remove all towers activated as a result. Repeat.
  Consider the case below.



- Find the unactivated tower the largest number of towers within its range, and place a sensor at this tower. Find and remove all towers activated as a result. Repeat.
  Consider the case below.



We'll use a data structure described in the following section to solve this problem correctly!

Algorithm.
The correct greedy solution is to only place a sensor in each super-tower without incoming edges in the condensation graph (proceeding sections).

Proof.
These super-towers cannot be activated by another super-tower, so they each require a sensor. This shows that there is no solution using fewer sensors. This solution also activates all requires super-towers.
Consider a super-tower with one or more incoming edges. Follow any of these edges backwards, and continue backtracking in this way. Since the condensation graph is acyclic, this path must end at some super-tower without incoming edges. The sensor placed here will then activate all super-towers along our path.

11 Directed Acyclic Graphs (DAG)

Continuing from the previous problem, it is useful to consider the towers as vertices of a directed graph, where an edge from tower a to tower b indicates that the activation of a directly causes the activation of b, that is, b is within the radius of a.

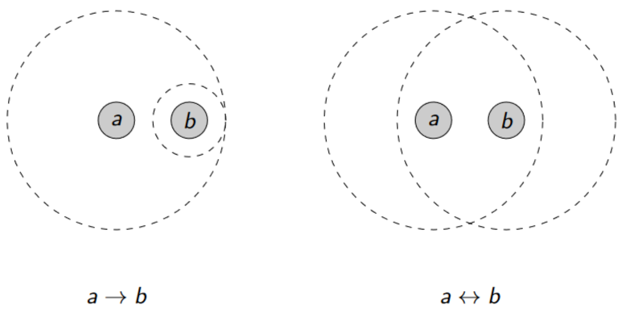
Algorithm.
Let S be a subset of the towers such that activating any tower in S causes the activation of all towers in S. We never want to place more than one sensor in S, and if we place one, then it doesn't matter where we put it. In this way, we can treat all of S as a unit; a super-tower.

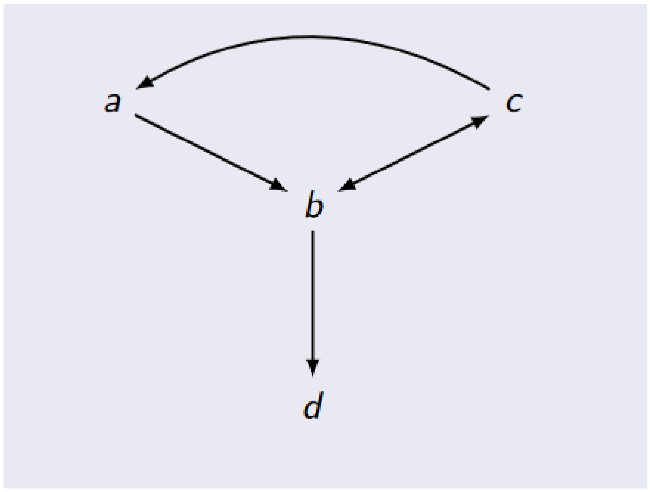
12 Strongly Connected Components (DAG)

Given a directed graph $G = (V, E)$ and a vertex v, the strongly connected component of G containing v consists of all vertices $u \in V$ such that there is a path in G from v to u and a path from u to v. We will denote it b C_v.

We now find strongly connected components $C_v \subseteq V$ containing v by constructing another graph $G_{rev} = (V, E_{rev})$ consisting of the same set of vertices V but with the set of edges E_{rev} obtained by reversing the direction of all edges E of G.

u is in C_v if and only if u is reachable from v and v is reachable from u (use BFS for this). Equivalently, $u$is reachable from v in both G and G_{rev}.





The time complexity for finding these strongly connected components s $O(V(V + E))$ since we conduct a BFS twice.

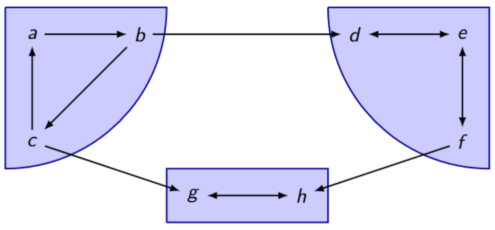
13 Condensation Graphs (DAG)

It should be clear that distinct strongly connected components are disjoint sets, so the strongly connected components form a partition of V. Let C_G be the set of all strongly connected components of a graph G. Define the condensation graph $\sum_G = (G_G, E^*)$

$$E^* = \{(C_{u1}, C_{u2})\ |(u_1, u_2) \in E, C_{u1} \neq C_{u2}\}$$

The vertices of $\sum_G$ are the strongly connected components of G, and the edges of $\sum_G$ correspond to those edges of G that are not within a strongly connected component, with duplicates ignored. A note about strongly connected components are that they are DAGs (contradiction proof; assume there are cycles).

The following is our graph from the Tsunami warning example



The corresponding condensation graph is

$$\begin{tikzcd}
\{a,b,c\} \arrow[rr] \arrow[dr] & & \{d,e,f\} \arrow[dl] \\
& \{g,h\} &
\end{tikzcd}$$

14 Topological Sorting (DAG)

Let $G = (V, E)$ be a directed graph, and let $n = |V|$. A topological sort of G is a linear ordering of vertices $\sigma : V \to \{, ..., n\}$ such that there exists an edge $(v, w) \in E$ then v precedes w in the ordering.A DAG permits a topological sort of its vertices in $O(V + E)$ time. Note that a topological sort is not unique and there may be more than 1 valid ordering.

Algorithm.
Maintain a list L of initially empty vertices, an array D consisting of the in-degrees of the vertices and a set S of vertices with no incoming edges.
While the set S is non-empty, select vertex u in the set.

- Remove it from S and append it to L
- Then,for every outgoing edge $e = (u, v)$ from this vertex remove the edge from the graph, and decrement $D[v]$ accordingly. If $D[v]$ is now zero, insert v into S.

15 Dijkstra's Algorithm

A classic shortest path algorithm. Used to find shortest paths from a source node s to every other vertex.

Maintain a set S of vertices for which the shortest path weigh has been found, initially empty. S is represented by a boolean array. For every vertex v, maintain a value d_v which is the weight of the shortest 'known' path from s to v. Initially $d_s = 0$ and $d_v = \infty$ for all other vertices. At each stage, we greedily add to S the vertex $v \in V \setminus S$ which has the smallest d_v value. Record this value as the length of the shortest path from s to v, and update other d_z values as necessary.

Dijkstra's Updates.
If here is an edge from v to z with weight $w(v, z)$, the shortest known path to z may be improved by taking the shortest path to v followed by this edge. Therefore we check whether

$$d_z > d_v + w(v, z),$$

and if so we update d_z to the value $d_v + w(v, z)$.

Correctness.
The proof that follows is an **exchange argument**. Assume there is the path P returned by Dijkstra's that uses only intermediate vertices in S and another supposedly short path P'. Now suppose v is the next vertex to be added to S. Portioning the path P' using only intermediate vertices in S will allow us find the weight of this path, d_y. But v was chosen to have the smallest d-value among all vertices outside of S, so the path P must have a weight d_v such that $d_v \leq d_y$. Therefore, d_v is indeed the weight of the shortest path from s to v.

Dijkstra's Implementation & Time Complexity.
Dijkstra's can use either an array or augmented heap approach to store d_v values. Storing the distance values in an array will result in an $O(n^2 + m)$ time complexity since at each of the n vertices, we need to perform a linear scan on an array of length n (to find the smallest $d[v]$ not selected) then check every outgoing edge from v (check for some $z \in V \setminus S$ to update $d[v]$).

The augmented heap improves on Dijkstra's by realising that we can delete unneeded values of d_v all together from the data structure. An augmented heap is represented by an array $A[1...n]$ where the left child of $A[j]$ is stored in $A[2j]$ and the right child in $A[2j+1]$ with every element of A of the form $A[j] = (i, d_i)$ represents vertex i. A min-heap property will be maintained with respect to the d-values only. Another array $P[1...n]$ stores the position of elements in the heap such hat $A[P[i]] = (i, d_i)$.

Changing a d-value of vertex i is now an $O(\log n)$ operation since accessing the heap is done in $O(1)$ time and popping from the heap takes $O(\log n)$ (fix-up/fix-down). The overall time complexity of this new Dijkstra's is now $O((n+m)\log n)$ since each n stage requires a deletion from the heap and each edge causes at most one update of a key in the heap (also taking $O(\log n)$ time). However, since there is a path from v to every other vertex, we know $m \geq n-1$, we can simply to $O(m \log n)$.

16 Kruskal's Algorithm

A minimal spanning tree (MST) T is a connected graph G is a sub graph of G which is a tree, and among all such trees it minimised the total length of all edges in T. Kruskal's algorithm is a algorithm that produces a MST, and if all weights are distinct, then such a MST is unique.

Algorithm.
Sort edges E in increasing order by weight. An edge e is added if its inclusion does not introduce a cycle in the graph constructed thus far, or discard otherwise. The process terminates when the forest is connected ($n-1$ edges added).

Efficient implementation.
We need to determine if an edge $e = (u, v)$ will introduce a cycle in the forest F; there is already a path between u and v. To achieve this, we can use the **Union Find** data structure to store our vertices which has the following operations

- $Union(a, b)$ which changes the data structure by merging he sets containing a and b into a single set $A \bigcup B$.
- $Find(a)$ which returns the representative set containing a. $Find(a) = Find(b)$ iff a and b are in the same set. $Find(a)$ can change over time, as a result of $Union$ operations.

Maintain a forest, where each tree represents one of our disjoint sets. Begin with n vertices and no edges s initially each vertex belongs in a different set. $Find$ should return the root of a respective vertices set and an array is used to record whether a vertex has a parent or not.

- $Find(a)$ should traverse he edge to the parent to find the root of the tree containing a. This takes $O(h)$ time.
- $Union(a, b)$ should do nothing if a and b are already in the same tree or add en edge otherwise to merge the two trees into one. This takes $O(h)$ time too.

The union find data structure is guarantees that $h \leq \log_2 n$ due to each tree guaranteed to have a height of at mot $\log_2 n$ so both $O(h)$ operations are now $O(log(n)$. The time complexity of Kruskal's using this data structure is $O(m \log_2 n)$ since we have to sort m edges of a graph G which takes $O(m \log m)$ time but $m \leq n^2$ so $O(m \log n)$ is the final time complexity.