GA3 - Report

CS 325
Baorong Luo, Joy Jin, and Sam Zink
29 Feb 2024

Description of Algorithm(s)

Step 0: The Union-find data structure

This data structure is built based on the modification of Union-Find python implementation from: https://yuminlee2.medium.com/union-find-algorithm-ffa9cd7d2dba

Below is the detailed description of the union-find data structure, its use in our algorithm is discussed in later steps.

The class UnionFind implements the Union-Find (Disjoint Set Union) data structure. This data structure is used to efficiently manage a partition of a set into disjoint subsets.

Constructor (__init__):
Initializes the Union-Find structure with the given number of elements (‘numOfElements’) and an array of elements (‘arrayOfElements’).
Creates a parent array , value array, size array, and count integer to store the identifying parent/root of each element’s set, each element’s coordinate value, the size of any set with that element as the identifying parent/root, and the count of disjoint sets within the structure, respectively.

makeSet Method:
Creates a set for each element, and initializes it with the corresponding values from the array of elements.

find Method:
Implements the Find operation in Union-Find using path compression.
Given a coordinate, it traverses the parent pointers until it finds the root representative of the set to which the coordinate belongs. It applies path compression to flatten the tree structure for future find operations.

union Method:
Implements the Union operation in Union-Find.
Given two coordinates, it finds their root representatives using the find method.
If the roots are the same, the coordinates are already in the same set, and no union is needed. Otherwise, it merges the smaller set into the larger set, updating parent pointers

and sizes accordingly. Decrements the count of disjoint sets and returns True to indicate a successful merge or False if no merge was needed.

Step 1: File read and extract edge data

Read the input file, extract coordinate and edge data into:

- an array of coordinates (ArrayPoints)
- an array of all edges sorted by weight (SortedEdges)
- an array of edges from E Prime. (EPrimeEdges)

Reading Input File:

The code opens the specified input file and reads the first two lines.
It removes spaces and concatenates each line into separate strings (‘ArrayPointstxt’ and ‘ArrayEPrimetxt’). The code then uses regular expressions (re.findall) to extract coordinate pairs from ‘ArrayPointstxt’. The number of vertices (‘VNumVertices’) is then calculated based on the count of commas in ArrayPointstxt. We initialize a 2D array, ‘ArrayPoints’ to store coordinate points and their indices.

‘Edge’ Class:
The class ‘Edge’ includes information about an edge, including coordinates (‘CoordinateA’ and ‘CoordinateB’) and weight (‘Weight’).
The ‘Edge’ class also has rich comparison operators for sort operation utility based on edge weight.

E Prime Edge Data:
If there is E prime edge data (‘ArrayEPrimetxt’ is not "none"), the code processes it similarly to coordinate data. It calculates Manhattan distances($|x_1 - x_2| + |y_1 - y_2|$) between points and creates instances of the Edge class, storing them in a list, ‘EPrimeEdges’.

Number of Possible Edges:
The code calculates the total number of possible edges (‘ENumEdges’) based on the number of vertices, specifically, ENumEdges = (VNumVertices * (VNumVertices - 1)) // 2

Creating Sorted Array of Edges:
The code creates an array of all possible edges (SortedEdges) by iterating over the list of V input vertices in a nested loop. For every vertex in the list, possible edge data is calculated for edges between said vertex and all vertices at later indices of the list. Since the graph is undirected, this method prevents possible edges between two vertices from showing up twice.
For every possible edge calculated, an Edge class object is created encapsulating the coordinates of the edge’s two vertices and the Manhattan distance between the edge’s

coordinates ($|x_1 - x_2| + |y_1 - y_2|$) as the edge's weight; the Edge class objects for each possible edge are inserted into the list of all possible edges. The list of all possible edges is then sorted into non-descending order.

Step 2: Create array of coordinates and UnionFind structure

This step represents every coordinate in the array of coordinates (ArrayPoints) as its own disjoint set. A variable, Forrest, is created to hold the UnionFind structure of all disjoint sets.

This step is done by 'Forrest = UnionFind(VNumVertices, ArrayPoints)', VNumVertices and ArrayPoints being the data we extracted in step 1.

Step 3: Organize coordinates in E' into sets of connected components

This step organizes coordinates reached by E' edges (if any) into disjoint sets of coordinates, such that all coordinates in the same set are connected by edges in E Prime.
To do this, iterate through the array of edges from E Prime ('EPrimeEdges').

- For each edge, check if its two coordinates are in different sets by comparing the root coordinates of their disjoint sets.
- If the edge's two coordinates are in different sets perform a union operation on the two disjoint sets of the edge's coordinates, to reflect that they are connected by an E' input edge.

This step is done by iterating over a range of elements (edges) in 'EPrimeEdges'. For each edge (e), it is calling the function union() on coordinate elements of the Forrest union-find structure; the union function is passed the coordinates of the edge's starting point (CoordinateA) and ending point (CoordinateB).

The algorithm is using the union-find data structure to merge sets of coordinates connected by any E' input edges. The union operation combines the sets to which the two coordinates of each E' edge belong. This operation efficiently represents any connected components formed by the E' input edges in the graph.

Step 4: Find edges that belong in E*,

This step of algorithm builds a set of edges E*, such that E* is a minimum weight subset of all edges that, when unioned with E' set of input edges, connects all vertices in the graph G.

Initialization:
The code initializes a variable 'WeightEAsterix' to store the cumulative weight of all edges in E*. This will be used to calculate the total weight of the selected edges.

Per the professor, this report algorithm is to return the set of E* edges, so this algorithm also initializes an empty set to hold E* edges, which will call ContainerE*.

Iteration Over Edges:
The code then iterates over a list of all possible edges (SortedEdges), sorted based on their weights in non descending order, starting with the lowest weight element and incrementing to evaluate each next lowest weight element until a complete set of E* edges are found, that is until all sets in ‘Forest’ are merged into a single set of all vertices.

Union Operation:
For each edge (‘e’), the algorithm checks whether its two coordinates (‘CoordinateA’ and ‘CoordinateB’) are in different sets with the union operation, defined in the UnionFind class and described above, which returns true and merges the sets of the edge’s two coordinates if the vertices are not in the same set, or which returns false when the edge’s two coordinates were already in the same set and did not need to be merged. In the code, this boolean return value is stored in a variable named belongs.

Belongs in E* Check:
If the union operation indicates that the two coordinates were in different sets (i.e., the union operation returns true), the algorithm adds the edge to ContainerE* set which will eventually hold all E* edges. This is because the edge is the lowest weight of the possible edges and because the edge connects points that were in different sets and not yet connected by previously added edges of ContainerE*.

Cumulative Weight:
For edges that belong to E*, their weights are added to the ‘WeightEAsterix’. This variable keeps track of the total weight of all edges in E*.

Result:
When all sets of coordinates are merged into a single set, we know that we have added enough edges to ContainerE* to connect all components of the input graph in conjunction with the E’ input edges.
At this point the algorithm returns ContainerE* as the E* set of edges.

Running Time Analysis:

The first step of the Algorithm is interpreting the input. The algorithm interprets the input in V + E' time, where V is the number of input vertices and E' is the number of E' input edges. This is because each edge and each vertex is interpreted in O(1) time, and there is a V + E' number of edges and vertices to interpret. In mathematical terms, the first step runs in T(V + E').
Since E' is a subset of all possible edges E, the number of edges in E' is upper bounded by the number of all possible edges E.
In mathematical terms, $T(V + E') \leq T(V + E)$, so $T(V + E') = O(V + E)$
Since the number of all possible edges E is equal to V(V-1)/2, the number of vertices V is upper bounded by the number of all possible edges E. So O(V + E) can be represented as O(E) since the lower term is dropped in the O representation of time complexity.
To surmise, the first step runs in T(V + E'), which is O(E).

A next step is calculating the manhattan distance between the coordinates of each edge in E, the set of all possible edges, and encapsulating each edge's coordinates with the manhattan distance between the coordinates as the edge weight. We initialize an empty set to hold encapsulated edge data in O(1) time. We then loop through an E number of all possible edges. For each edge we compute the manhattan distance as the weight, encapsulate this weight with edge's coordinates, and append the encapsulated edge data to the set in O(1) time. So this step runs in T(E +1), which is O(E).
The big O runtime of this step, added with the big O runtime of the first step brings us to O(E) for both steps, since the constant coefficient is dropped in the big O representation.

A next step is calculating the manhattan distance between the coordinates of each edge in E', the set of input edges, and encapsulating each edge's coordinates with the manhattan distance between the coordinates as the edge weight. Since E' is a subset of E, the number of edges in E' is upper bounded by the number of edges in E. So the time complexity to do this step for every edge in E' is upper bounded by the time complexity to do this step for every edge in E, which we have previously established to be O(E). As such, this step runs in O(E).
The big O runtime of this step, added with the big O runtime of the previous steps brings us to O(E) for these steps, since the constant coefficient is dropped in the big O representation.

A next step is initializing a disjoint set with each vertex as its own set, this is O(1) for each of a V number of vertices. As such, this step runs in O(V).

The big O runtime of this step, added with the big O runtime of the previous steps, brings us to O(E) for these steps, since the smaller term of O(V) is dropped in the big O representation.

A next step is iterating through E' to merge the sets of each E' edge's coordinates, if the coordinates are not already in the same set. The Kruskal's algorithm iterates through an E number of all edges, to merge the sets of each edge's coordinates, if the coordinates are not already in the same set; it is well proven that Kruskal's algorithm does this in O(ElogE) time.
Since E' is a subset of E, the number of edges in E' is upper bounded by the number of edges in E.
As such, the time complexity to run these operations for E' subset of edges is upper bounded by the time complexity Kruskal's algorithm takes to run these same operations for the E set of all edges, so this step runs in O(ElogE).
The big O runtime of this step, added with the big O runtime of the previous steps, brings us to O(ElogE) for these steps, since the smaller terms are dropped in the big O representation.

A last step is to run a modified Kruskal's Algorithm, starting from a disjoint set consisting of connected components of the input graph with E' edges as sets, instead of starting from a disjoint set consisting of each coordinate as its own set. Kruskal's Algorithm has a well proven runtime of O(ElgE) (proof of correctness and a detailed version of analysis for Kruskal's can be found here).
The number of connected components of the input graph with E' edges, is upper bounded by the number of vertices, which Kruskal's algorithm classically starts with. As such the runtime of this step is upper bounded by the runtime of Kruskal's algorithm, so this step runs in O(ElogE).
The big O runtime of this step, added with the big O runtime of the previous steps, brings us to O(ElogE) for the entire algorithm, since the smaller terms are dropped in the big O representation.

This satisfies the time complexity requirements for this project, as the big O representation of our algorithm's time complexity is the same as the big O time complexity of Kruskal's algorithm.

Proof of Correctness:

Overview:

One way to think about our problem is that we are finding an MST between connected components of the input graph with E' edges. Components being a subset of G, they can be either independent vertices or subgraphs. This means we can run a MST algorithm just modified for connected input components instead of vertices. We have chosen to use Kruskals for our solution.

Our algorithm returns the weight of E*, so the main proof relies in that E* is calculated correctly.

Restatement of Assignment Definition:

Proposed algorithm takes an input of all vertices V and a subset of Edges E denoted as E' such that $E' \subseteq E$. This algorithm seeks to find E*, where E* is the minimum weight subset of E that connects the graph G with E' preexisting edges.

Mechanisms of Referened Algorithm:

Our algorithm is based on the well proved Kruskals' algorithm. This algorithm starts by treating every vertice as its own connected component and seeks to create larger connected components from existing connected components by joining independent components with an edge. Kruskals' algorithm constructs a set of minimum spanning tree edges by sorting the set of all edges, E, by weight in ascending order and iterating through E to include edges that would connect components of the graph not yet connected by the previously included edges. This is a broad overview of Kruskals skipping key processes of execution, for more information on the specifics of our implementation of Kruskal's read our *Description of Algorithm(s)* section defined above or consult the chapter 7 of our classroom textbook linked here. While Kruskal's algorithm treats every vertice as its own component and connects them to form a MST, in our algorithm we are using Kruskals to form a "MST" so to speak between components. Instead of strict vertices we are also starting with already made trees. So our modified Kruskal's algorithm seeks to join every connected component where a component can either be a vertice or a connected subgraph (can either be cyclic or acyclic).

Reasoning for data representation:

It is correct to represent any E' connected components as sets in the disjoint sets to build E* from because, by the assignment definition, the set of E' input edges will be a part of the E' U E* set of edges that will span all V vertices, and E' can represent zero or more connected components which may or may not have cycles per the illustrated example in the assignment definition.

It is correct to represent each vertex not reached by the E' edges as its own set in the disjoint sets to build E* from because, E* must span all vertices by definition.

These sets of E' connected components and sets of each other vertex must be disjoint and not connected to each other in the input graph of E' edges and Vertices, no contradictions are possible because:

- <u>For the vertices not reached by E'</u>
  The edges in E' are the only input edges, so if vertices are not connected by any edges in E', there are no other edges they could be connected by.
- <u>For connected components of E'</u>
  If any sets of vertices connected by E' had any shared vertices, they would then be connected at those shared vertices and in the same set.

Intuitive Proof:

As defined above our algorithm works with disjoint components, being either sub graphs or vertices. Our algorithm first tries to find the edge with the minimum weight that connects two disjoint components.

Claim: No cycles can appear

The goal of E* is to connect G using the minimum weight possible.

If a cycles exists in E* then we can remove an edge and E* will still work to connect G, as per definition of a cycle. When removing this edge we are subtracting the removed edge's weight from the total weight of E*. In this algorithm the manhattan weight is used, $|x_{v1} - x_{v2}| + |y_{v1} - y_{v2}|$ The manhattan weight will always return a real positive number, as $\forall v \in V, v \in R^2$ and absolute value always returns a positive number and any addition of positive real numbers is positive. This means when we remove an edge from E*, subtracting a positive number from a positive, E* will always be less. So when E* connects G but contains a cycle, E* will always be less when remove the cycle, and by definition E* MUST be the lightest subset of E that connects G, and therefore can't contain a cycle.

Claim: The lightest edge must be an element of E*
The edge e, where e is an edge with minimum weight that connects two independent components, e must be included in E*; for the graph to be connected each component needs to map to any other component and this edge connects the individual components with the least Manhattan weight, per assignment description.

Removing e and replacing it with other edge will result in larger E* value
If we remove e from the set of E*, then our graph is not connected. From here we can pick any other vertice to connect our disjoint components. However any component we add will be larger than e as by definition e is the least weighted edge. This is the same for every other edge that connects two components. Our algorithm by definition uses the lightest edge to join components, if we don't do this and take another edge, then we can always reduce E* by just using our original edge.

Consider D, the minimum weighted subset of E that connects G where G = G(V, D U E')

(I am assuming D is the minimum spanning tree for components, and E* is the set of lightest edges that connect G, this proves that E* is the correct answer)
If E* = D, then there is no more work to be done.

Else if E* != D, then $\exists e \in E^*, e \notin D$ (Our algorithm returns e as the edge with the least weight that connects two individual components.) D must be an acyclic connection of the graph G, in union with E'. (From here we prove that D isn't the minimum weighted spanning tree because we can always reduce D)

If we assume this and we have a connected graph not including e, we need to prove that the span D including e instead of some other edge that makes the graph connected is less than D. However, if we add e right away to the connected set of D a cycle appears. This violates are criteria for G. From this we loop through the cycle and find the redundant edge, h (this is the edge that when paired with e makes the cycle). Since e is less than said edge we remove h and our graph D is a connected graph that equals E* (This process can be recursed for any edge in the set E*, strictly described in the Physical Proof section below). However D could be any subgraph as long as it connects G and doesn't include e. This means that the claim D = E* is only true in a few cases. It may be the case that when we add e and consequently delete h, that h's neighbors are not in the set of lightest edges that connect the graphs. From here we rerun this step with f's neighbors, using the the element from E* that connects those two components. Further and further D will eventually morph into E* because we are exchanging an element from D for an element in E* because when we do we maintain the property of a connected graph G but with each exchange decreasing the weight until eventually we can't and we are only left with E* (because how are algorithm is run E* will contain the lightest edges) (Mathematical version found below).

This proof is slightly different for the intuitive proof instead updating E* to get to E^*_x where the intuitive proof built E^*_x, denoted as D, to E*.

NOTE: This proof below is largely based on the proof found here (Queens College) and here (Stanford University). These proofs prove that Kruskals returns a MST. Our proof takes these premade proofs and alters them to fit our assignment, which is just an alternate Kruskals.

Theorem:

After running our algorithm we will return the a set of edges' weight, E*, that is a minimum Manhattan weight subset of all edges that, when unioned with E' set of input edges, connects all vertices in the graph G.

Physical Proof:

First:

1) G is connected.
   This is because $\forall v \in V$, there exists a path P that connects v to every other element in the set of V.
2) The **Components** of G are together acyclic. (Cycles may form in individual components based on are input E', but not between components, as per are algorithm)

Second:

E* is the minimum set of edges that connects G in union with E' this will be proven through the use of induction. Let E^*_x be the minimum weight of edges that connects G. If E* = E^*_x then E* is the value that joins G with the minimum expended weight. If E* != E^*_x then there exists an edge e ∈ E^*_x of minimum weight that is not in E*. As well E* ∪ e contains a cycle $C_{1 \text{ see end of proof}}$ such that:

a) Every edge in C has a weight less than the weight of e. (Else our algorithm would have picked e)
b) There is some edge f that is not in E^*_x
   Consider the set E^*_2 = E* \ {e} ∪ {f} the subset of E with e and f

a) E^*_2 is a subset of E that connects G
b) E^*_2 has more edges in common with E^*_x than E* did
c) Total manhattan weight $E^*_2 \geq$ E*

This process can be repeated for finding E^*_3 which has even more edges in common with E^*_x. By induction we repeat this process until we reach E^*_x.

$\text{weight}(E^*) \leq \text{weight}(E^*_2) \leq \text{weight}(E^*_3) \leq \ldots \leq \text{weight}(E^*_x)$
Since E^*_x is the minimum manhattan weight set that connects G these inequalities must be equalities and we can conclude that E* is the minimum subset of edges that connect G.

C_1 with the way out algorithm is run cycles can develop. This specific cycle is a cycle between individual components described in the “Mechanics” section of *Proof of Correctness.*