Comp3600 Project Proposal - u6938702

1 Description

In the event of a fire in a large populated office space during the year 2020, two primary risk factors are present; the risk of burns from the fire, and the risk of COVID-19 transmission as large crowds exit the building en-masse. The company that owns the building has commissioned an app that is to be installed on each employees phone. In the event of a fire within the building, it will tell employees exactly when and where to move so as to exit the office safely whilst minimizing the likelihood of COVID-19 transmission.

2 Guide to the submitted files

For the sake of completeness we have included all work on the project thus far. This includes work on the development of some data structures, algorithms and testing files that are beyond the scope of this milestone. In the `README.md` we discuss the files `main.cpp` and `test-generator.cpp` which are used to create the user interface and testing binary file respectively.

The files that define the interface for the data structures `graph.h`, `hash-map.h` , `avl.h`, `djikstras.h` and `rdf-types.h`. The file `djikstras.h` contains the interface for Dijkstra's algorithm as well as variants of Dijkstra's algorithm used in this software.

In thus submission we also include the completed (compilable) package `rdf-io` which we have developed in order to perform the read-write operations in this assignment.

Any files that are not mentioned in this section should not be relevant to this deliverable and in some cases may be incomplete.

3 Functionalities

In the following subsections we provide the description of each of our functionalities in addition to discussing the algorithm/data structures required for each functionality. If one must explicitly state the functionalities that they choose to confirm then we select functionalities 1 and 2.

3.1 Functionality 1 - Ingest data about the building, and construct appropriate in-memory data structures

3.1.1 Description

Read the file (consisting of RDF triples serialized in turtle[1] format) which contains the map of all rooms and passageways. Each room can have a unique IRI identifier and the `ex:hasPathTo` predicate indicates that there is a passage leading from room A to room B. The primary part of this functionality is then to construct the data structure that will appropriately store these graphs in memory[2]. Note that in order to implement other functionalities within the application we also perform similar operations to construct the graphs describing the locations of people throughout the building.

The graph can be represented in memory as a hash map. In addition, dynamic metadata such as the *number* of people in each room and *expected* number of infected people in each room could be included; though for ease of marking we choose to not make this an explicit part of our first functionality.

[1] https://www.w3.org/TR/turtle/

[2] Possible addition instead of having 1 fixed time to evacuate the whole building - Read the RDF/XML file which contains the predicted times at which the fire will start burning in each room and determine escape routes based on this information. If this is the case we could use information about the geometry of the building in conjunction with convex hull. If the nature of functionality must be determined to the extent that we have to decide on this now - then we choose not to include this *possible addition*.

3.1.2 Implementation

To perform the IO operations we have created our own package called `rdf-io` in this package we have a `reader` which parses a `.ttl` file and returns a `vector` of triples in `subject predicate object` format. Each triple is is represented as an array of length 3 where each entry is a string. For the sake of efficiency this is a relatively stripped back parser that assumes the syntax within the file contains no errors.

We then convert these triples into a graph structure. There are two separate structs which we use to provide the abstract interface with this graph structure; they are `Graph` and `GraphWithIdInternals`. Both of these structs are located within the `graph.h` file.

The `GraphWithIdInternals` struct is used to implement algorithms such as Dijkstra's algorithm which do not require information such as the `name` (or `url`) of the node. In these algorithms, where possible we use the underlying `Id` of the node (which would either be a row position in a matrix, or the hash used the OOP/adjacency list method). This is because Algorithms such as Dijkstra's are more efficient to implement on the `Id` rather than doing conversions between the `Id` and the string name of the node repeatedly throughout the algorithm.

The `Graph` struct is then used to interface with parts of the software where it makes more sense to work in terms of the name of the node name/urls rather than the internal ids. This includes the points at which the software is reading/writing triples.

We then have two different structures used to store the graph in memory. The choice of which structure to use is determined at run time based on the data.

In cases where we have a high ratio of edges to nodes (which in realistic building architectures are only going to occur in buildings with a small number of rooms/nodes) we use and adjacency matrix (see file `graph-matrix.cpp`) to store the weighted graph representing rooms on the building and the weight of their connection. In order to minimise the escape time - this weight is the hallway length; in order to minimise COVID transmission, this weight becomes the COVID *risk* associated with going to the next room[3]. Hence, for our application this will likely be a weighted combination of both factors. In addition we use a hash map/table (`hash-map.h`)[4] to store the relation between node names (`strings`/`urls`) and the `int`/`Id` representing the row/column which represents the node within the adjacency matrix. Whilst this technique is effective when we have a high ratio of edges to nodes, it becomes ineffective as this ratio decreases due to the fact that the adjacency matrix becomes sparse and uses a large amount of memory to store zero-values that provide no information.

In the case where the ratio of edges to nodes is lower, we instead use and adjacency-list (linked-list) to represent the graph structure in memory. This is because this particular data structure scales primarily with respect to the number of edges rather than with respect to the number of nodes. This data structure also requires the use a hash map/table (`hash-map.h`) in order to store a mapping between the `string`/`url` that represents the name of the node, and the `int` which is used for the ID of the node within the linked list. Depending on the results of empirical tests it may be the case that the entire adjacency-list is indeed stored as a hash table (rather than just storing the mapping between node ID's and node names within a hash table).

Note that we have not yet determined the exact nature heuristic which will be used to select between these two graph data structures at run-time. This is because we need to perform empirical testing with both data structures in order to come to an informed decision.

A note on the rdf-io parser - The writer tries to balance a trade off between efficiency in the writing process and producing a compact, well-structured document. In order to do so we first place the triples into a nested structure (see the `SubjectData` and `PredicateData` structs in the `rdf-types.h` header) and then writes out the triples in the compact syntax turtle allows for triples with shared `subjects` or shared `subject predicate` pairs. However, for the sake of efficiency we skip creating the shortened syntax for triples with shared `objects` or shared `predicate object` relationships.

[3] This makes the graph a directed graph - however this is not an issue as Djikstra's algorithm (and the variants therein that we describe); which we run over this graph in Functionality 2; works for both directed *and* undirected graphs

[4] In early development stages of this software we have been using the `map` package to provide this functionality.

3.2 Functionality 2 - Calculating the escape routes

3.2.1 Description

Read the file (consisting of RDF triples serialized in turtle[5] format) that contains the location of each employee and write (dynamically modify) annotations indicating where they need to move at each timestamp such that the expected number COVID-19 transmissions is minimized whilst all employees are still safely evacuated from the fire. The annotations indicating movement are to be stored in a separate turtle file.

3.2.2 Implementation

Again we can use `rdf-io` to do any reading/writing of triples to/from relevant documents.

In the first functionality where we create the `Graph` and `GraphWithInternalIds` we provide an abstracted interface for accessing the underlying graph structure in a uniform manner (which we recall can be either linked-list/hash-table or an adjacency matrix depending on the ratio of edges/nodes). This makes it easy implement standard graph search algorithms such as Dijkstra's algorithm or searches to find the MST.

To develop this functionality, we first chose to implement Dijkstra's algorithm (see the function `djikstras` in the file `djikstras.cpp`). This allows us to sequentially take each possible pairing of person and exit in the building and determine the optimal path (according to the *weight* encoded into edges of the graph in functionality 1) by which that person can escape from the building. We then improved upon this functionality by noting that we could change the ending condition in Djikstras algorithm to be when the current node is *any* exist node rather than a particular exit node[6]. This is in the process of being implemented in `djikstrasMultiEndpoint`. We intend to further improve upon this algorithm in `multiStartMultiEnd` wherein we shall cache previously found optimal paths from the `djikstrasMultiEndpoint` algorithm so as to reduce repeated calculation. If time allows we may investigate ways to optimize the distance calculations using dynamic programming techniques or investigate solutions using the Floyd-Marshall algorithm.

Since the assignment specification asks for an *exact* algorithm/data-structure - we nominate the `multiStartmultiEnd` algorithm. Whilst it would be best to run empirical tests over methods for caching paths; we note that one may wish a method to be specified in this deliverable. In this case we nominate a hash map where keys are hashes of nodes for which the optimal path is known, and where the values are a tuple of the optimal exit path and the total *weight* (`int`) of that path. Note that once we have a determined the optimal path between two points we can infer the optimal paths of all nodes in that path by taking sub-paths of the original path.

We chose Dijkstra's algorithm initially as (to the best of our knowledge) it has the best average case running time for finding he shortest paths between two randomly selected nodes; we then add our modifications to reduce repeated computations (we will formalize this concept of *reduced repeated computations* in terms of dynamic programming within our final report).

3.3 Functionality 3 - Identify those most likely to have COVID following the evacuation

3.3.1 Description

There are n rooms available in the nearby hotel quarantine facility. Find the n people most likely to be infected following the evacuation so that they can be quarantined[7].

3.3.2 Implementation

This could be best solved using dynamic programming methods to first identify the *routes* that would result in the greatest likelihoods of transmission (this part may involve some modified version of Dijkstra's algorithm) and then match these with residents that traversed these routes (may involve use of hash maps). A basic (fallback)

[5] https://www.w3.org/TR/turtle/

[6] An exit node corresponds to Room in the building from which one can directly exit the building

[7] Possible extension - additionally find all individuals with $x\%$ probability of infection and alert them via the app that they need to self isolate if they are not already in hotel quarantine.

method to do this would be to calculate the likelihood of infection for each resident being infected and add this value along with an identifier for each resident to an Red Black tree. We can then efficiently gather the n ids with the highest estimated infection value from the tree. The fallback could still use dynamic programming to recursively calculate the change in expected infection levels across each timestamp.

We have created the header file `RB-tree.h` in case we choose to use this particular fallback approach. We choose a red-black tree over an AVL tree for this purpose as this application has a high ratio of insert/search operations; and insertions over red-black trees are more efficient than insertions in AVL trees.

4 Assumptions

- The rooms and passageways are discrete regions. Time is broken into discrete (integer) units and the time taken to move through a given hallway/room is the same for all employees.
- The area of each room, and length of each passageway are provided as inputs. The the location of each individual is provided as input.
- If a group of individuals are in a room, then every uninfected person has an equal chance of being infected at that point in a time. This probability is a (given) function of the area of the room, number of people in the room and, if known, the expected number of infected individuals within the room.
- Escaping the fire in (within a given time) *always* takes precedence over minimizing transmission levels.
- Optimisation of time taken to produce the escape file takes precedence over all other functionalities (as this is the time critical procedure). This assumption is required as otherwise one could optimize the net time taken to calculate the escape path and risk of COVID transmission by doing the two operations simultaneously.
- The URI's in the input files are well formed and begin with the charaters 'http://'. This allows us to reduce the size of the domain of keys that we are working with in the hash function for the hash map.
- The total distance of any non-repeating/non-overlapping path through the graph is less than the maximum value for int on the given OS.

5 Potential Issues

- Running this algorithm with a large number of entities (e.g. > 10000) may result in additional memory requirements (i.e. > 2GB of RAM as given in the usage section).
- Given that the second functionality is trying to optimize paths for a group of entities rather than for an individual entity, it may not be possible to find the optimal solution using the algorithms taught in this class. If that is the case, we could add the constraint that the app acts as an 'agent' for each person which uses Dijkstra's algorithm to determine the best path through the building on a per-entity basis.

6 Usage

This software is being developed on Linux machine (SMP Debian 4.19.132-1) in C++11. The 64-bit machine being used has a Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz and has 8GB of RAM. We intend for this software to be run on a Linux machine with at least 2GB of RAM.