Coursework

Artificial Intelligence

References

Davide Morello

40219838

Artificial Intelligence

References

*This report tries to find the solution to the problem of finding the shortest path in a small network of interconnected caverns. The goal is to identify a programmatical solution to efficiently achieve a method to navigate the network in through the shortest path and in the shortest amount of time.*

# Identifying the problem

As it often happens in informatics, a problem that may appear hard at first can be easily broken down or reduced to a well-known issue. Our specific situation revolves around a small robot trying to navigate a set of caverns, and attempting to reach cavern N, which we will call “target” from its starting point, “source”.

What we know about this scenario is:

* The set of caverns are generally small, and overall the robot will face at worst a network of 20 caverns.
* Not all paths between caves are bidirectional, some may be one-way only.
* The distance between two caves, X and Y is the Euclidean distance of the coordinates of the two caves: √(𝑥2−𝑥1)2+(𝑦2−𝑦1)2

In this situation, it is fairly apparent from the start that what we are facing is small graph of interconnected nodes. Each cavern can easily be represented by a node, while paths between the various caverns, can be represented by weighted edges, where the weight of an edge spanning between two caves is the afore mentioned Euclidean formula.

Once the problem has been identified, obtaining a solution becomes quite clearer, and in this situation, can easily been reduced to finding an efficient goal-based algorithm, capable of finding the shortest path within a weighted Directed Graph, and in other words a graph where the direction of an edge is taken into consideration.

For this specific problem, we are also considering only a situation where we are not dealing with Multigraphs (graphs where two nodes might be linked by multiple edges of different weights), as the weight of each edge is identified by distance between the two nodes it links, making the use of multiple edges redundant.

Part 1: Possible solutions

*Over the years in computing, many problem have been described by simply finding the sequence of actions that lead to the desired goal. Each action changes the state and tries to reach the final state, in which the goal has been achieved, starting from the initial state.*

*In graphs theory, similarly finding the shortest path within a graph is a searching problem, where each iteration attempts to achieve a state defined by the source and target nodes been linked together by a path-able set of edges. Due to the nature of the task, many different algorithms of varying complexity have been formulated, and for the goal of this report I will look into three different known solutions to the problem, that all offer a slightly different way of achieving the goal with slightly different outcomes.*

***For each one of the algorithms, I will try to give a brief overview (the length based on whether the algorithm has been explained in class), and then explain why it can be consider a suitable candidate to solve the problem.***

# Dijkstra's algorithm

## Overview

Dijkstra’s is possibly the most famous algorithm to find the shortest path with a graph. It was found by the Dutch Computer Scientist Edsger W. Dijkstra in 1959 and exists in many different variations.

At its core, Dijkstra’s runs by visiting nodes in the graph starting with source. It continues to repeatedly examine the closest and yet unvisited node and selecting its neighbors to be examined. Dijkstra’s always finds the shortest path from the source to the target, as long as all edges in the graph have a positive weight.

The algorithm is strictly greedy and can be considered a zero heuristics version of the more generalized A\*. It runs in O(N^2) where N is the amount of nodes in the graph. Dijkstra can be fairly slower than a Best-First Heuristics guided algorithm, however unlike some of its alternatives it is guaranteed to find the shortest path, rather than a path.

## A general look at the algorithm pseudocode

For a basic implementation of the algorithm, we can start by initializing three lists:

The three lists are used to properly navigate the graph while keeping an eye on the status of the search, and to iteratively understand whether the path found in the graph is better than the previous ones.

* dist[] with dist(i) ∈ ℝ+ contains the distance between a node and the source
* parent[] with parent(i)∈ V contains a node, the parent of U
* toVisit[] contains the list of nodes to visit

Once the lists are initialized, the algorithm proceeds to interactively navigate each one of the nodes which have not yet been visited, and for each one of them the neighbouring nodes are analysed completing a table of distances. If a node w connects successfully with the source, the distance is compared with the previously found ones (if any) and consequently saved.

The algorithm completes its run when all the

nodes N ∈ toVisit[] have been visited and the list is empty.

1 func Dijkstra(Graph, source):  
2  
3 toVisit = [] // The list of nodes to visit = []  
4  
5 forEach node n in Graph: // Initialization  
6 dist[n] ← ∞ // The starting distance is the to infinite  
7 parent[n] ← -1 // The previous node is instead to non-existent  
8 toVisit.push(n) // All nodes in the graph will need to be visited  
9  
10 dist[source] ← 0 // The distance from source to source is 0  
11   
12 while toVisit is not []:  
13 v ← node in toVisit with min dist[v] // Selects the node with the shortest d  
14 toVisit.pop(v)  
15   
16 for each neighbor w of v: // Checks the neighbors of v  
17 alt ← dist[v] + length(v, w)  
18 if alt < dist[v]: // Selects the new shortest path  
19 dist[w] ← alt   
20 parent[w] ← v   
21  
22 return dist[], parent[]

## Complexity

Dijkstra’s Algorithm can be expressed in function N and E respectively the number of Nodes and Edges in the graph.

A typical implementation of the algorithm such as the one shown in the codes, implement the search through a simple list containing the set of nodes to visit. For this reason, we can consider the complexity of the algorithm a simply O(N^2) as for each node K in the list of nodes to visit, in the worst-case scenario, the given node K will have N-1 neighbors.

The complexity can however be further reduced by the usage of a Fibonacci Heap. Moreover, in case of repeated searches, the overall speed of the algorithm can be further increased by the usage of memoing applied to the distance between nodes which can be pre-calculated or store.

## Correctness

Being a greedy algorithm, Dijkstra focuses on one simple idea: the local best solution, is always formulated by selecting the local best at each iteration. In our situation, the best path to a given node n in a graph, will be explored at each iteration by finding the shortest path from the source, at a given iteration, and attempting to move towards N from that specific point.

Consequently, Dijkstra’s correctness is proved by induction.

In order to do so, we generally consider these invariant lemmas:

* For each node n visited, dist[n] is the shorted distance from the source to the node n if a path is available, otherwise infinite
* For each node v not yet visited, dist[v] is the shortest distance between the source and the node v if a path is available, otherwise infinite

If only the source has been visited, the lemma will hold, as dist[source] is zero, and all nodes are set to an infinite distance.

Working on node N-1 where N is the number of nodes in the source, we can now consider and edge E where the distance dist[v] for the edge n-v is the smallest distance of eny unvisited node and nv is the distance dist[v] is equal to dist[n] + length[n, v]. In this scenario, dist[v] must be the shortest distance from the source as if there was a shortest one, it would have been visited fist. Therefore dist[w] < dist[v] would create a contradiction, just like an existing shorter path to v, different from dist[v] would have been less than dist[n] + length[n, u]. For this reason, even after v has been visited, for each node been visted, the distance distance[w] from w to the source is the shortest distance, using the nodes what have already been visited. Moreover if there is a shorter path using v, this will be updated.

## Conclusion

For our caverns exploration problem, Dijkstra is an optimal candidate.  
The greedy approach offers use a situation, where generally speaking the solution is reached quickly, while the overall complexity of the calculation, given the maximum number of nodes (caverns) being less or equal to twenty, ensures the that the iteration will be completely quickly regardless of input.

Dijkstra’s weaknesses are often related to its inability to obtain correct results when a path contains negative edges, which due to the nature of the iterations could potentially break the logic. This is often solves by the implementation of different algorithms, or variations such as the well known Bellam-Ford that can solve efficiently and correctly graphs with negative edges.

This is however not a problem in our scenario, considering in fact that the distance between two nodes A and B is as simple as the Euclidean distance between Ax, Ay and Bx, By, it is impossible to end up with negative edges and consequently all of our edges will have a positive weight.

# A\* algorithm

## Overview

*Since the algorithm has been throaty explained in class, I will only beefy touch on a few points and focus on why it can be considered one of the optimal candidates to solve the caverns issue.*

In 1968, Peter Hart, Nils Nilsson and Bertram Raphael at Stanford expanded Dijkstra’s algorithm, and creating what is now known as A\*.

A\* is what we can consider as an optimization, and generalization of Dijkstra’s Shortest-Path algorithm, that achieves better performance throughout the usage of heuristics to navigate the research. In other words, the selection of the node to expand will be operated by a function *f(n)* which will focus on evaluating which nodes is the best one to expand.

As an optimization or improvement over Dijkstra, A\* can be classified as informed or Best-Firsts search algorithm which uses heuristics to guide itself in finding the shortest path.

Dijkstra’s vs A\*  
As mentioned a couple of times, Dijkstra’s can be considered a special case for A\* where the heuristics used to determine the which node to expand is zero. In other words, and down to the behavior of the algorithm this determines a fairly different way of expanding nodes.  
While Dijkstra will given a node n ∈ of N, try choose to expand based on the distance between the node, its neighbour and the total distance from the source, A\* will instead use a more refined approach where an estimate of the distance from the node n to the target t is taken in consideration.

This means, that given a node n, the last node being visited, A\* will instead choose how to proceed based on the formula: ***f(n) = g(n) + h(n),*** where g(n) is the cost so far to reach node n, and more importantly h(n) is a heuristic used to estimate the cost of the shortest path admissible to reach the target node t from the current node n.

In order to work however, A\* requires the function that calculates the path to always underestimate the final cost (or distance).

## Complexity

In general, we can say that the time complexity for the A\*, just like it’s Space complexity can be very different based on the implementation.

The time complexity, can vary based on the heuristics used to evaluate the node to expand.

In an optimal scenario, such as search tree the complexity can be polynomial or linear, however it can grow to be exponential to the length of the solution, such as when the calculation revolves around the shortest path. Generally speaking, a good heuristic function will give A\* a great edge over Dijkstra’s and other more basic algorithms, reducing the branching factoring (number of nodes expanded) and reaching an optimal solution much quicker.

Space on the other hand, can represent a slightly worse obstacle, in order to operate quickly, and based on the implementation A\* similarly to Dijkstra’s, does keep information for all the expanded nodes in memory.

## Conclusion

The main application of A\*, the one for which the algorithm was designed is to be a replacement and improvement over Dijkstra’s as a general traversal algorithm for graphs.

In its current application, A\* is nowadays used not only for academic and scientific problems, but also daily for pathfinding issues within games in various different flavors such as the D\* dynamic variant.

For our speleology problem, A\* himself is definitively a prime candidate, for reasons very similar to the ones that also make its special case Dijkstra great option.

The graph, or caverns networks, that we are analyzing do not involve more than 20 nodes. In this situation, A\* which is often quite faster than the already ideal Dijkstra can find a solution incredibly quickly, without having to restrict our acceptance criteria to something which is not the shortest possible path.

Moreover, just like for Dijkstra’s, we do not have any possible issue related to negative paths. As we are currently attempting to find path in a weighted graph with only positive edges (and thus absolutely no negative cycles), A\* will also find the optimal path.

Part2: Pseudocode – bidirectional dikstra

*In order to solve the problem, I have decided to implement a Dijkstra’s Algorithm bi-bidirectionally, searching from the source and from the target.*

*The chosen search method is implemented with Python3, and relies on a few different libraries in order to provide a GUI and plotting functionality.*

*In this part, I will provide the pseudocode for the language agnostic part of the program which are of interest for the coursework. And in other words, the implementation of bidirectional Dijkstra that allows stepping.  
More Software Engineering related information regarding the implementation or technologies will be found in part3.*

## Overview and general information

Before providing the pseudocode, I felt like providing some information regarding the implementation of the search that might make things a bit more understandable.

In my design, the GUI program is meant to take in different algorithm and allow the user to switch between them comfortably. In order to achieve this kind of design, and allow the user to step through the execution of the algorithm some choices have been made:

* The solving function of the algorithm is “wrapped” into a class, the class keeps the state at each iteration of the search. This allows a very clean implementation of an Undo/Redo stuck without “contaminating the code” responsible for solving the problem.
* The algorithm function, along with its wrapping class makes use of two functions: a logger, and a renderer. This two simple arguments and are separate from the general shortest-path logic.

The pseudocode will therefore focus on providing information on how the program parses the data, and uses it to solve the graph rather than trying to provide software engineering information the actual Python3 implementation.

## Choice of algorithm

As mentioned in Part 1 of this report, one of the best choices for solving the problem along with A\* is the Dijkstra’s algorithm. In order to achieve the best result however, Dijkstra’s like many other pathfinding algorithms can be run bidirectionally, alternating between searching for the shortest path:

* Forward: starting from the source, and towards the target
* Backward: starting from the target, and towards the source, however always considering nodes that can directed from towards the goal

This different, which can seem a small flavour improvement does however create a huge performance boost, often cutting down the speed by much more than just half. This is due to the way in which the algorithms expand nodes: cutting down the overall shortest path in two halves, means that the overall radius of research for each is also halved, this will geometrically reduce the volume of the sphere of research by much more than half, making the bidirectional version of the algorithm much faster than its basic alternative.

## Bidirectional Dijkstra, initialization

class Dijkstra extends Algorithm{  
 constructor(graph, source, target, logger, renderer){  
  
 // Initializes the parameters necessary for the algorithm to run  
  
 // dictionary of distances  
 distances = {  
 forward: {},  
 backward: {},  
 }  
  
 // dictionary of distances  
 paths = {  
 forward: {},  
 backward: {},  
 }  
  
 // heap of (distance, node) tuples for extracting next node to expand  
 // inits with nodes to visit  
 fringe = {  
 forward: [ //to visit from source],  
 backward: [ //to visit from destination ],  
 }  
  
 // nodes who have already been seen and investigated  
 visited = {  
 forward: {source: 0},  
 backward: {target: 0},  
 }  
  
 // holds the information the results  
 // and necessary for the algorithm to run  
 finalPath = []  
 directionNum = 'backwards'  
 finalDist = 0  
 }  
   
 bidirectionalDijkstra(stepping == False){  
   
 }  
}

The state of the class is initialized when the constructor is called. All the dictionaries holding the information necessary for the algorithm are parameters of the class. The solving function, is only concerned with the stepping, it will use the class arguments regardless.

## Bidirectional Dijkstra, solving function

As mentioned before, to keep the pseudocode clear the logging logic is omitted. The algorithm, does however allow the user to call the logger at any point during the solving of the algorithm to provide information on the resolution.  
The rendering is also called only once, when the solving is complete. The outside wrapper takes care of further calls to update the view when stepping.

bidirectionalDijkstra(stepping == False):  
  
 // if source and destination match, we already have the shortest path  
 if source == target:  
 return distance and path  
  
 while fringe[forward] and fringe[backward] are not empty:  
  
 // reverses the lookup direction at each iteration  
 lookupDirection = inverse(lookupDirection)  
  
 distance, v = getNextNode(fringe[lookupDirection])  
  
 if distances[lookupDirection] contains v  
  
 // update the distance to v  
 distance[lookupDirection][v] = distance  
  
 // if we have already visited the node bidirectionally, we are done  
 if distances[inverse(lookupDirection)]:  
 if not stepping:  
 renderResults()  
   
 for w in getNeighbours(v):  
 // regardless of the lookup direction, we always want to go from source ---> target,  
 // in a directed graph, target --> source nodes only nodes, are not considered  
 if lookupDirection is forward:  
 minWeight = getWeightOf(v,w)  
 else:  
 minWeight = getWeightOf(w,v)  
   
 vwLength = distances[lookupDirection][v] + minWeight  
  
 // catches the exception caused by potential negative paths  
 // this is visible if minWeight is negative  
 if vwLength < distances[lookupDirection][v]:  
 return -1  
   
 // first or shorter path from source => w or w => target found  
 else self.visited[lookupDirection] not contains w or vwLength < self.visited[lookupDirection][w]:  
 // updates parameters  
 self.visited[lookupDirection][w] = vwLength  
 fringe[lookupDirection].update(vwLength)  
 paths[lookupDirection][w] = paths[lookupDirection][v] + [w]  
 // has w been visited backward and forward?  
 if visited[forward] contains w and visited[backward] contains w:  
 // we have found a path  
 newDistance = visited['forward'][w] + visited['backward'][w]  
 if not finalPath or finalDistance > newDistance:  
 finalDistance = newDistance  
 // adds the path to reach w, to to the path from w to target,  
 // reversing the latter and removing w (otherwise we would have w => w)  
 finalPath = paths[forward][w] + reverse(paths[backward][w])[1:]  
   
 if stepping:  
 updateState()  
 return;  
 // if we are out of the while loop, and all fringe buffers  
 // are empty, and we still haven't returned, this means there is not path source ==> target  
 return:  
 -1, []

Keeping renderer and logger outside the main logic, avoids polluting the algorithm solving logic with unnecessary one used for keeping the state. Moreover, the general logging calls can be moved and replaced at will all over the code where necessary.

Keeping the state of the function, outside the function itself also allows better housekeeping and avoid messing up the logic that solves the graph, with the one necessary to step through the execution or even provide the undo and redo functionality.

Renderer and Logger, are simple function past in parameters inside the class and can be invoked whenever necessary.

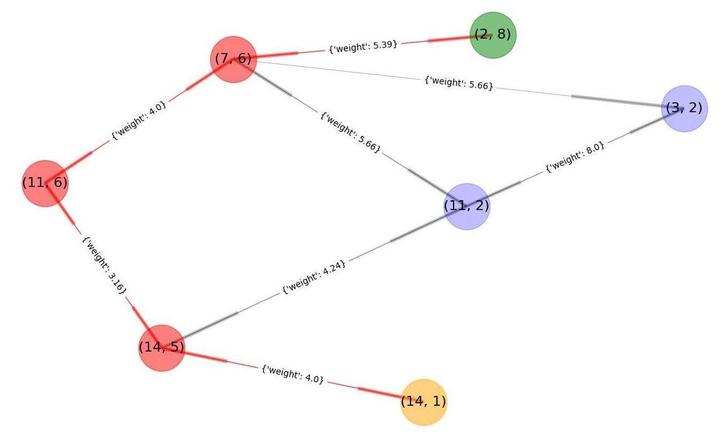
Part3: Program

*As a quick brief note regarding the program.*

*As mentioned earlier, the code responsible for writing the program relies on Python 3.6 and uses a small series of library to achieve the result.*

*This are:*

* *PyQT5 for Python 3: PyQt is one of a few of QT binding libraries for Python. It allows the usage of the QT GUI/XML and SQL framework for C++ for Python. The cross-platform library is comes with different licencing, including the free Open Source version used for this project.*
* *NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. In this project, NetworkX has been used in order hold the parsed data and provide a comfortable interface for plotting and visualizing the graph*
* Matplotlib is a Python 2D plotting library easily interfaceable with NetworkX, its purpose is to provide a visualization for the graphs while being somewhat compatible with PyQt5 Backend.



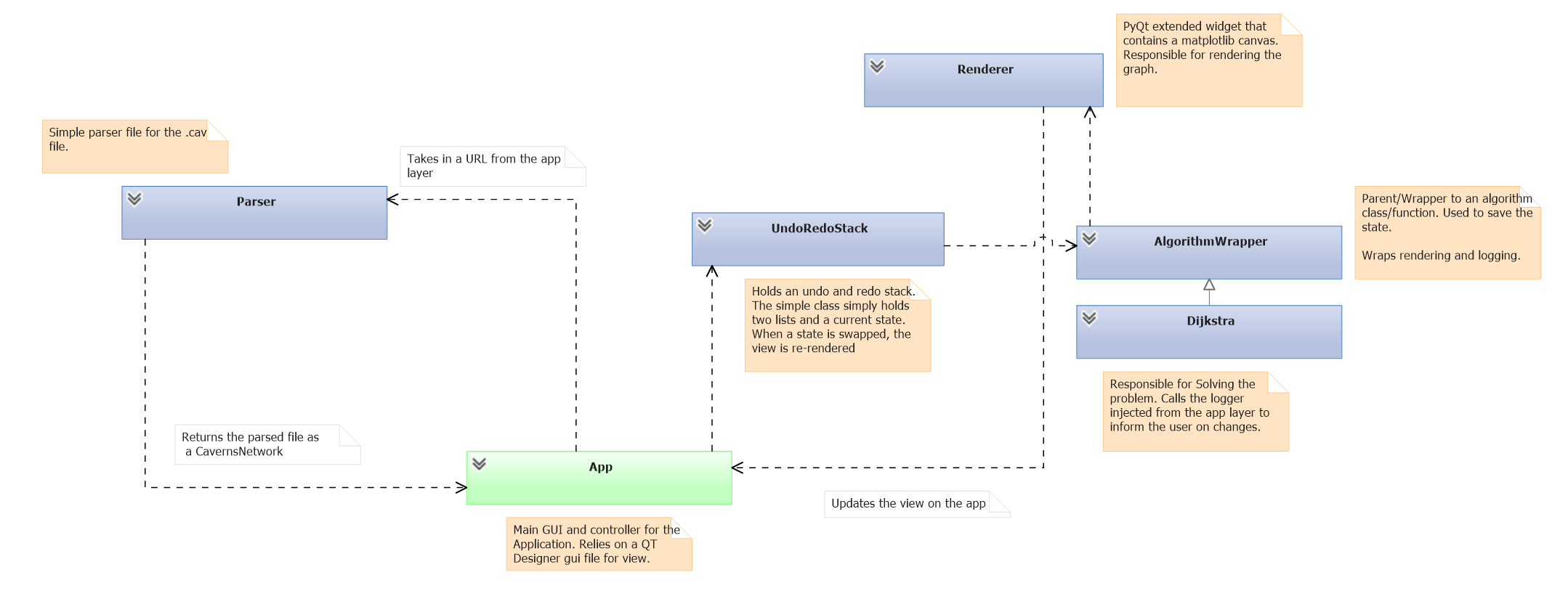
*Matplotlib Graph:  
Rendered from NX*

Part4: implementation

*With the goal of implementing a shortest path algorithm, capable of providing visualization along with a stepping functionality, a few guidelines in the development process felt necessary and have been adopted in trying to achieve a clean expandable result:*

* *Avoid mixing logic: the function which purpose is to find the shortest path, should to exactly that and not much more*
* *Rendering at each step is necessary*
* *Logging information on what is going on is just as important as showing a picture of the state*
* *User should be capable of moving inside the execution*
* *Speed is as important as clean readable logic*
* *The application should allow a backend platform that can allow more algorithm in the future*

*With this in mind the final result, for this MVP of the project is fairly simple and its main classes can be seen in this UML-like diagram (a full size copy can be found in the appendix):*

**

*Along with a quick overview of the project, I will also include more detailed information on the measures undertaken to make sure that the solving function was fast and easily maintainable.*

# Overview and general architecture

## Opening and parsing

The whole program springs from the app class. Simply enough, the app class is a scrip that starts a QT application and window. The GUI, is simple XML generated through the QT designer app, and is not much different than a Java Swing or C# view.

The app layer, which is the front-end of the whole problem is responsible for binding together the GUI loaded from the designer file and from providing the actual bindings between the parsers, undo-redo stack and the actual logic responsible for solving the algorithm.

The user can simply open a file through a classical open file dialog\*, once the URL to the file is solved, this is send to the Parser which will take care of parsing the given file and storing the various details including nodes, edges and their weights into a NetworkX graph.

The choice for using an NetworkX data structure to handle the graph, might seem like an odd decision. It does however make more sense, when after some research is noticeable that NetworkX also offers integration with many different visualization layers, some web based, and some even capable of handling the export of to graph analysis tools. For this reason, instead of re-inventing the wheel, working proper tools seemed a much smarter idea.

When the user selects an algorithm (in this MVP only bidirectional Dijkstra is available), an Algorithm class containing the solving logic is instantiated and its state saved.

## Solving

If the user decides to solve the problem directly, the Undo/Redo stack is disabled and the functional responsible for the Dijkstra implementation is called without breaks. The function solves the algorithm calling the wrapped logger when necessary, and providing a clean and fast solution to the algorithm.

Allowing the solving function to still log, is necessarily a pretty costly tradeoff in terms of performance.  
As you can well understand, each logging call does actually update a GUI component, and scrolls its view to the bottom. Naturally, this slows down the overall efficiency of the function, however, it is worth in the sense that it does provide useful information to the user.

In a different situation, where the overall number of nodes is much larger, or in a real-life situation, it could be worth to:

* Give the users the possibility to choose if they want the information, and otherwise inject a passing lambda instead of a logging function
* Batch the GUI updates by saving the logging to a function, and render it only once at the end of the process.

## Keeping a “fast” state

The solving function, for which the pseudocode can be found in part two, is pretty much exactly the same in its python implementation. The measures taken to make sure that the function ran smoothly and fast are Dictionaries. Dictionaries in Python are fast. The whole state of Python program actually runs in a dictionary that runs in the heap. Moreover, lookups in dicts are incredibly optimized, and without getting into language specific details, it’s worth mentioning that they are amortized to constant complexity running on O(1).  
For this reason, all the state of the program runs in dictionary, this not only makes the program very efficient for most lookup operations, but also allows great readability and maintainability:  
it’s much easier to understand distances[‘forward’] than to store said elements in a set or list!

## Stepping

No unnecessary booking keeping when bookkeeping is unnecessary: The sentence that might sound stupid, simply sums up the idea of separating the logic and how the function handles quickly solving a graph or stepping through one.

The whole process, within the function, does rely on a simple past in flag. When the flag is set to false, the algorithm, as seen before, will simply avoid updating the state conclude its run normally.

On the contrary if stepping is applied, at each iteration the algorithm will:

* Update the state of the wrapping class, providing the node currently being visited and its neighbors
* Update the state of the wrapping regularly, but also keep an eye on whether the computation has finished
* Break after each iteration over the fringe dictionaries

This methodology, will ensure that at the end of each iteration, the state of the wrapping algorithm class will be updated with a working set of information including which node have been visited last, which neighbors were linked to that node and whether the algorithm has already found a result, is still looking or has failed.  
This approach, is consequently used on an more other level. As at each iteration, the overall state of the component is in fact taken and send to the renderer component. The component, which is a fairly dumb view-oriented component, is simply responsible for looking at the graph, and given a series of state-flag choose how to display the graph and its nodes.

Potentially a similar effect could have been achieved by running the solving function asynchronously or in a separate thread, and allow it to wait for user input in order to step to the next section of the algorithm. The approach however, while slightly more elegant wouldn’t have allowed directly the use of something like undo and redo functionality.

## A further level, undo and redo

This last level in the layer of the application, is a simple wrapper around the algorithm class.  
The component, a simple undo and redo dual stack, is simply called whenever a user chooses to step through the solution of an algorithm. At each iteration of the solving function, the undo and redo saves the state of the class. Said state is pushed into an “past” or “redo” stack, which will hold all the previous interaction up to the source or to a decided length.  
Since sadly QT does not offer 2-way binding like modern web framework (React or Angular) or similar GUI frameworks (WPF MVVM), at each iteration the undo/redo stack also manually re-renders the graph with the newly updated information.

When a users chooses to step backward, simply the double stack is updated:

* The current present state is pushed into the redo stack
* The redo stack top tile, is popped into the current state
* A re-render is triggered

The approach offers consequently the possibility to the user, to navigate back and forth (similar functionality, just inverting the stacks) through the solving states of the graph.

Arguably, saving the log state and binding it to the undo-redo stack could also be an option. In this version however, the state of the logger is tied to the application level, allowing the user to red all the operations that have happen to a given point even when stepping backward to check how the graph looked like a few visited nodes ago.

References

* *Wikipedia: Various including Dijkstra, A\*, Bellam-Ford and bidirectional algorithms*
* *Artificial Intelligence: A\* Slides*
* *Python Core: General documentation*
* *PyQt and Qt: General documentation*
* *Algoritmi & Strutture Dati 2: Universita’ Del Piemonte Orientale, Algorithms and Data Structure Slides*
* *Stackoverflow: Various threads to link a dynamic updating matplotlib figure canvas to a PyQt5 back-end capable of working NetworkX render calls.*