Istvan Gates

Data Structures

Self-Study Project

Template Methods with Graph Traversals

**Introduction**

This topic involves design patterns and in particular, Template Methods. It is often useful for software engineers to follow frameworks that will enable them to produce object-oriented software which is more precise, correct, and reusable. These frameworks can be called design patterns and when followed, can help programmers to write code that follows a series of underlying logic and principles. For this project, I have chosen to solve a problem with the Template Methods design pattern, and paired it with Graph Traversals.

**Data Structure and Design Pattern**

The template method pattern is a behavioral design pattern that defines the program skeleton of an algorithm, and allows a programmer to redefine certain steps of an algorithm. It is important to note that this use of ‘template’ is not related to C++ templates!

It can be useful to follow this design pattern when you have many related algorithms, or if there is one problem that may be solved in many related ways. Such an example is the task of finding the minimum distance to from one vertex to another in a graph structure.

A graph data structure is a pictorial representation of a set of vertices where some pairs of vertices are connected by ‘edges’. An edge is a simply a link which connects two vertices. Two common implementations of graphs are Adjacency Matrices, and Adjacency Lists. In a Matrix a two-dimensional array is used, where the indices will serve as the vertices, and the values will serve as the edge weights. This implementation is more costly in terms of space used by the structure, but can simplify multiple algorithms. An Adjacency List presents the same concept where it is commonly an array of Lists (or dynamic vectors), and each node will have an associated weight and to\_node. Weights may be both positive and negative. A positive weight can be thought of as the ‘cost’ that is associated with taking that path, and a negative weight can be thought of what is gained. So when presented with two options paths to take, one with a negative weight and the other with positive, the negative weight will commonly be prioritized.

Graphs can be used to model many real world scenarios, such as a map or in circuit design. Perhaps the most common problem is how to minimize the cost of travelling from one spot on a map to another. This paper focuses on solving the travelling problem by using an Adjacency Matrix, and pairing it with the Template Methods software design pattern. Two different algorithms are implemented to solve this problem, where the algorithms share a base class and common variables and functions. These are Dijkstra's pathfinding algorithm, and the Floyd-Warshall algorithm to find the shortest path between every pair of vertices in the graph.

In this study, only positive weights are considered in order to match the problem I am solving (when travelling a road there are no ‘positive’ costs). The following weighted, undirected graph is considered for all test cases.

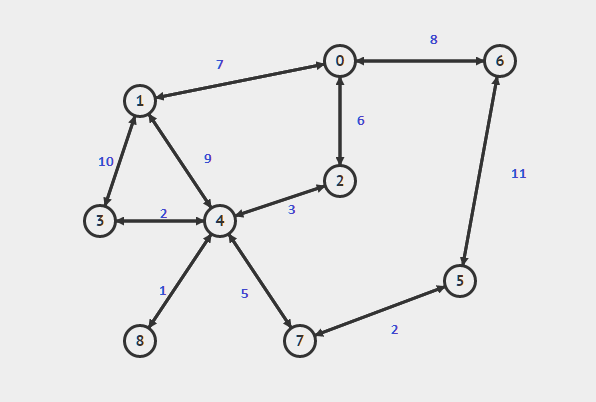


Figure A: The graph used in test cases, and implemented as Adjacency Matrix

Dijkstra's algorithm works by finding the lowest-cost path from a source vertex to all the other vertices. After the algorithm concludes, an array holds all the distances to each vertex.

Floyd-Warshall’s algorithm is a bit more complex, but simple in implementation. This algorithm uses a technique called dynamic programming. Dynamic programming works by breaking down a complex problem into a collection of simpler sub problems, and each subproblem is solved only once. The solution to each of these subproblems is stored for later use. This way when a large problem is encountered, it can be broken down to subproblems (for which the solutions are already known) and the main problem is solved very easily. By using this technique, the Floyd-Warshall algorithm can find the shortest paths between every pair of vertices on the graph. This differs from Dijkstra's algorithm, as Dijkstra's find the shortest path to every vertex from one particular source vertex.

In the real world it is important to classify the problem that is being solved and apply the appropriate algorithm. By using template methods, the skeleton for solving the problem can be defined, but the ‘nitty gritty’ for each solution can be easily implemented to suit the solution requirements. For example, when given a weighted graph, it is common to find the most efficient way to travel from one spot to another. If a user was using a real time system, such as Google Maps, then Dijkstra's algorithm would be more effective because given a ‘source’ vertex (the user’s original location) and the ending requirement, the optimal path can be found quickly. The resulting cost to the destination can be returned to the user in this case, where a quick calculation is needed. But for a city planner who wishes to minimize the overall cost of traversing a city and does not worry about how long the matrix takes to compute, the Floyd-Warshall algorithm proves to be more useful. By finding the shortest path for every pair of vertices on the map, the city planner has a ‘big picture’ overview of the city and can quickly make changes and immediately see how those changes would affect all other paths, determining if the change had a net positive or net negative effect on travel times.

But more importantly if a new problem came to rise and neither of the given algorithms would be effective then, because of the software design patterns that were followed, a new algorithm can be quickly defined and integrated with the existing system. Following the template methods design pattern allows for an easily scaleable codebase, and helps in keeping the entire system very organized.

This paper solves the common problem of finding the minimum cost to a certain location on a map, supplies different algorithms that provide a solution, and analyzes them while following the Template Methods design pattern.

**Analysis**

Before solving a problem using graphs, it is important to distinguish between Adjacency Lists and Matrices. Each structure has its own space and time costs, and depending on the constraints of the system it will be clear which structure should be prioritized. In this case, the problem to solve is a city planner (or a road navigation system), and so the Adjacency Matrix is used. This is because the city graph is static, and the extra space required by the Adj. Matrix does not pose a problem. Compare this to a navigation system which is computed on a microprocessor. Here, the space saved by an Adj. List could prove to be beneficial for the system as a whole.

The space complexity is O(|V|) x O(|V|) = O(|V|^2). This is because each row in the matrix has a complexity of O(|V|), where V is how many vertices are in the graph. Thus the complexity of the matrix will be the area of the matrix.

It is important to note that the constant factor here is very small, as each entry in the matrix is just an int (16 bits). Because of this, the Matrix is very useful the more dense a graph gets, and there is no extra cost when deciding between using an undirected or a directed graph. As such, the total space used by this adjacency matrix is 16\*V^2 = 16\*(81) = 1296 bits.

Each algorithm will have a different worst case time complexity when computing least-cost paths. For Dijkstra's, the proof is as follows:

Each path that was explored to a particular vertex will be marked and not be traversed again. It will also depend on how many edges exist on the graph as a scalar multiple. Thus, the worst case complexity is

O(E \* log(|V|))

For Floyd-Warshall’s algorithm, the worst case time complexity is easily computed. Because there are three nested for loops which iterate over every vertex, the complexity is derived as follows:

O(V) \* O(V) \* O(V + 1) = O(V^3)

The third loop has an if statement, which will always perform in constant time.

Because this implementation of Floyd-Warshall’s algorithm uses the original matrix as the place to store its solution, the space complexity will be the same as the matrix: O(|V|^2).

Figure 0 (at the end of the paper) provides the outputs, as well as the timings for each algorithm’s runtime. As expected, both ran in lightning-quick times due to having only nine vertices to compute. In more resource-constrained systems, the timings would differ and would likely be more useful. It is also important to note that a direct comparison of each algorithm does not make sense, because both algorithms are designed to solve two different problems.

**Implementation**

This paper will discuss the implementation in a top-down approach. First the main.cpp file will be discussed, then each algorithm will be described in detail (with notes on how the program follows the software design pattern).

The driver of the entire program is the main.cpp file. This file will decide which

algorithms are called, and can specify any details before calling an algorithm (such as number of vertices that exist in the graph, or the source node).

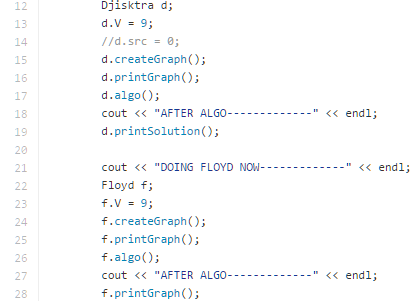


Figure B: Abstracted view on how to implement and execute algorithms

First, the derived class is initialized (lines 12, 13, 22, 23). Then the graph is created. The test case graph is defined in Figure A. The algorithm is called, and the solution is printed. In this abstracted view, only four steps are required to execute any algorithm (with one optional step to print solutions, and to print the original graph).

Because this process is made to be universal, even after implementing a new algorithm the workflow will not change. This allows for more advanced systems to integrate this workflow in more advanced operations (such as automatically iterating over each base class and performing the common set of steps to execute an algorithm).

The logic of each algorithm is contained in algorithms.hpp. First, the base struct is created.

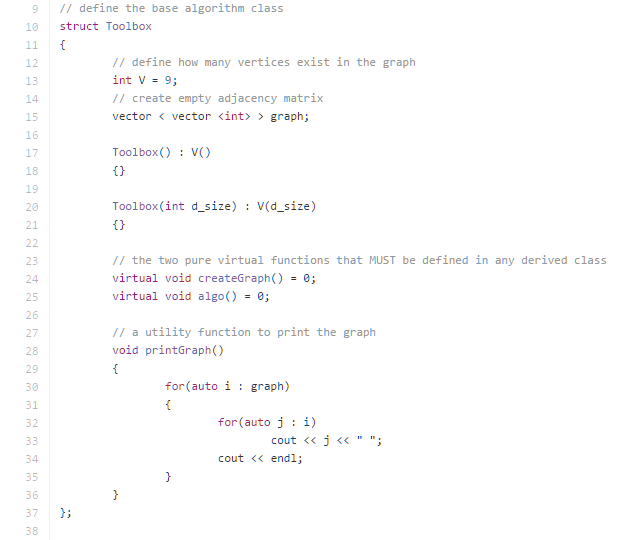


Figure C: Defining the base class, from which all other algorithms are derived from

The base class is fairly simple. It serves as the framework from which all other algorithms are based off of. Toolbox contains information about how many vertices exist, a few constructors, a utility function to print out the graph, and most importantly it holds two pure virtual functions. By having these core functions as pure virtual, it forces the derived classes to provide definitions.

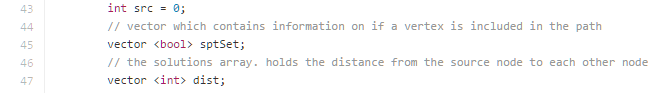
The first derived class that was made is the Dijkstra struct. Not surprisingly, this class contains all information about Dijkstra's pathfinding algorithm. 

Figure D: Member variables for Dijkstra

Figure D shows the member variables used in Dijkstra's algorithm. The src variable holds information about which vertex is considered the ‘source’, sptSet is a vector which serves to mark nodes on if the vertex is included in the current path, and the dist vector is holds all final paths.

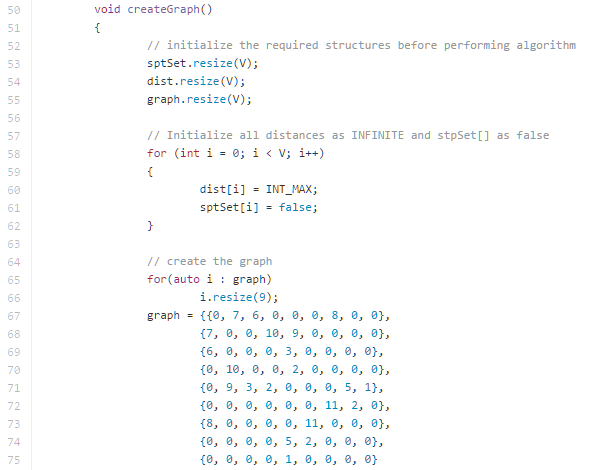
The graph is first initialized in the createGraph function (figure E).

Figure E: Initialization of a graph

It simply resizes each vector to be of length V, and initializes the test case graph. Further logic can be provided to allow a user to create the graph, but the researcher felt that operation was out of scope of this case study. The graph is held in a two-dimensional int vector, and represents the Adjacency Matrix.

An important utility function is shown in Figure F. This function returns the index of the vertex that has the minimum distance from the source. The function is used to find which vertex will be explored next, and is used in comparisons throughout the algorithm.

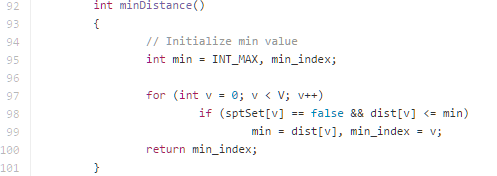


Figure F: The minDistance function, returning the index of the vertex which has the minimum distance

The algorithm itself is in the algo function (figure G). This function is the defined version of the pure virtual function from the Toolbox struct. The algorithm initializes the distance to the src to be zero, and then begins the algorithm. It works by finding the vertex which has the next minimum distance (line 112), and immediately marks the vertex as ‘True’. This marking makes sure that the algorithm does not explore paths that have already been traversed.

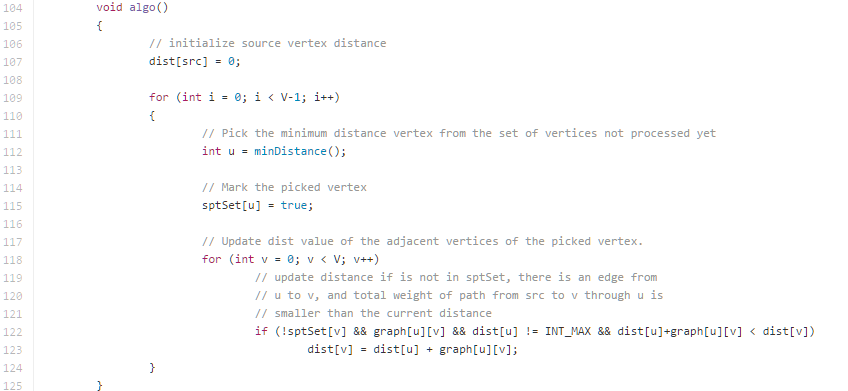


Figure G: Djisktra’s algorithm implementation

On line 118, the distance values will be worked out and updated in order to discover which vertex should be explored next. The distance is only updated if the edge has not been previously explored, the edge exists, and if the distance to that vertex is actually less than the currently recorded distance. It is important to note that the distance array is initialized to have every distance be ‘infinity’ (in reality, the INT\_MAX constant is used). This way, when first encountering a new edge, the computed cost will always be lower than the currently recorded cost.

After the new distances are computed for that vertex, the loop repeats itself and the new minimum distance is computed, and the algorithm repeats itself. In this way, Dijkstra's is called a ‘greedy’ algorithm. It will always explore the edges which have the lowest cost first (compare this to a Depth First Search algorithm used on a weighted graph).

Once the algorithm is computed, a utility function printSolution can print the solution in a formatted and organized way. It is important to note that the information about the path taken is not recorded. To record this information, one would create a new member variable(a vector), and simultaneously update this vector with path information as the shortest paths were computed. Depending on the problem to be solved, having a member variable which holds the path information could be useful.

The second algorithm implemented is Floyd-Warshall’s algorithm. This algorithm differs from Dijkstra's with respect to the final solution; while Dijkstra's is a greedy algorithm, only computing the distance from a source vertex to each vertex, Floyd-Warshall’s algorithm computes every shortest-path cost for every single vertex pair. The implementation itself is much simpler, although the theory behind it might seem more complex. The same createGraph function is implemented as it was in the Dijkstra struct, but no other member variables are needed for this algorithm. As the algorithm computes lowest costs, the original graph is modified. By modifying the original graph, less space is used by the program. This difference from Dijkstra's in space used demonstrates how each algorithm implemented can differ in what member variables and functions they use, but that they must implement the pure virtual functions.

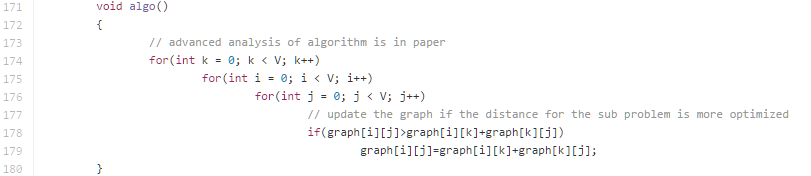


Figure H: Floyd-Warshall’s algorithm implementation.

The only difference is in the implementation of the pure virtual function algo. Figure H shows the algorithm’s implementation, which is only five lines long. The way Floyd-Warshall’s works is by picking two vertices, using the matrix to find the cost, and then attempting to put other vertices ‘between’ the two chosen ones in order to lower the overall cost. The first rule is that the order of putting vertices between must be sequential. For example, the algorithm first attempts to put vertex 1 in between, then 2, then 3, etc. If the computed distance is lower after the vertex is added, then this distance is saved. This is how the algorithm breaks an overall problem (find shortest path between all vertices) into many sub problems (pick two vertices, attempt to put other vertices in the current path. Does this new path have a smaller cost than the original? ). Each sub problem is tackled sequentially, and each successful solution is recorded to the original graph.

Once the algorithm concludes, the output is an Adjacency Matrix which holds the minimum-cost between every vertex. Once again the path information is not saved, and the information may be useful depending on the problem to be solved.

**Conclusion/Summary**

This researcher has successfully implemented two well known algorithms while following a design pattern in order to solve a problem, and analysis has been given about each algorithm.

Just by looking at the complexities, it seems that choosing Dijkstra's algorithm would be the more prudent choice. But it will always depend on the problem to be solved. If considering only the paths taken from a particular source vertex, Dijkstra's would be more efficient. But if all possible path costs must be computed, Floyd-Warshall algorithm will be more useful as it uses the dynamic programming technique to efficiently solve the problem.

Possible areas of improvement is a random graph-generation function which can be used in order to demonstrate the differences in time complexities of both algorithms and could aid in different fields where the graphs used may not be static. For example, a city map where the edge costs change according to traffic patterns, or applying graph theory to biological networks where new vertices (cells) grow and die dynamically. Even in these cases it is important to follow software design patterns, and the template methods may be especially useful as different algorithms may be implemented and prototyped fairly quickly. By having this pattern in place, new developers introduced to the system can be quickly trained in how the entire system works which is very valuable for large companies (i.e Google or Microsoft). Having a developer be able to learn a codebase quickly is one step closer to a more efficient software engineer.

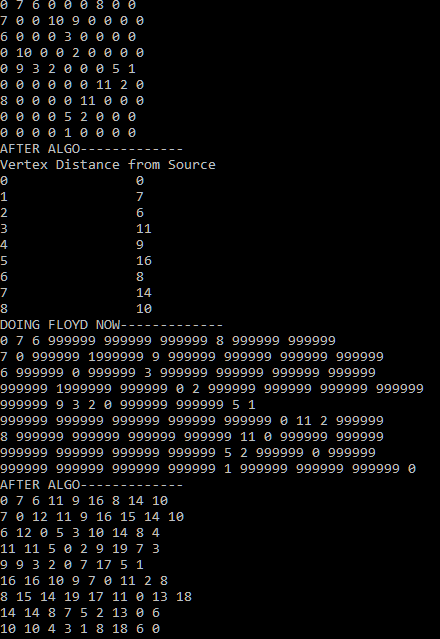


Figure 0: (above) Example output Dijkstra (top) and Floyd-Warshall (bottom). In both cases the graph used is shown as well.

(below) Timings output for a graph with 9 vertices. Left- Dijkstra, right - Floyd-Warshall.



**Full Code**

github.com/gatesyp/Capstone-DS

**References**

<http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm>

https://blog.cedric.ws/c-floyd-warshall-shortest-distance-explanation-code

<https://sourcemaking.com/design_patterns/template_method/cpp/1>

<https://sourcemaking.com/design_patterns/template_method/cpp/2>

https://en.wikipedia.org/wiki/Template\_method\_pattern