# Klein ’67 Cycle Cancelling Algorithm Implementation and Animation

Alex Gyori

December 8th, 2017

CS4447A, Roberto Solis-Oba

## Introduction

In this project, we explore an implementation and animation of the Klein ’67 cycle cancelling algorithm for minimum cost maximum flow. We will also explore the algorithm itself. The algorithm and animation was implemented in Unity 5, a modern graphics engine. During the development of this project, there were three primary goals:

**(i)** Create a learning tool that allows people to quickly and easily learn the Klein ’67 cycle cancelling algorithm.

**(ii)** Visualize the algorithm in such a way that a user can understand the individual steps of the algorithm and understand where the algorithm is inefficient. Hopefully, this can lead to improvements in the algorithm or the creation of a new algorithm entirely.

**(iii)** Give the user enough input choices so that they can be creative and interact with the animation and the algorithm. This goal lead to the creation of a fairly extensive graph creating and editing component of the program.

# The Klein '67 Cycle Cancelling Algorithm

## Description

The Klein cycle cancelling algorithm is a minimum cost maximum flow algorithm, where a given graph must be directed and connected graph with a source and target with each edge containing three values: a flow and a capacity value as usual, and additionally a cost value. This cost value is per unit flow, meaning that if an edge has a cost value of five and there are two units of flow flowing positively through the edge, then the total cost the edge contributes to the graph is ten[1]. Like other algorithms that solve the minimum cost maximum flow problem, the algorithm's first priority is to find a maximum flow, and once that is found the algorithm will begin augmenting the path to find the minimum cost while still satisfying maximum flow. More specifically, the algorithm is represented by the following pseudocode:

**Figure 1: Cycle Cancelling Algorithm Pseudocode** [2]

```
Initialize: Find some max flow x in G and form the residual
graph Gr(x).
while Gr(x) contains a negative-cost cycle
     Find some negative cost cycle W in Gr(x)
     Set $\delta$ = min{$r_{ij}$: $(i, j) \in W$}
     Augment $\delta$ units of flow around W and update Gr(x)
end while
```

Clearly, the steps are not specific in how they are performed. To implement this algorithm at each step, we will require the use of other algorithms. To fully understand the solution to the minimum cost maximum flow problem as presented by the Klein ’67 cycle cancelling algorithm, it is important to understand all of the sub-algorithms used. The first of these algorithms is for the initialize step, where we find a valid maximum flow without any regard for the cost whatsoever. This can be accomplished by the Ford-Fulkerson algorithm for maximum flow (which was studied in class). The Ford-Fulkerson algorithm iteratively adds flow in the form of augmenting paths until there are no augmenting paths left [4]. To do this, it requires the usage of a residual graph [4]. Note that when we create the residual graph, the back edges have a cost value of -cost(edge) [5]. Specifically, the Ford-Fulkerson is as follows:

**Figure 2: Ford-Fulkerson Algorithm Pseudocode** [4]

```
Ford-Fulkerson Algorithm
1) Start with initial flow as 0.
2) While there is an augmenting path from source to sink.
          Add this path-flow to flow.
3) Return flow.
```

This will result in a maximum flow for the given directed graph, but not necessarily the most cost-efficient maximum flow. To implement the main body or while loop of the Klein cycle cancelling algorithm we will need to use the Bellman-Ford algorithm (which was also studied in class). The goal of this is to detect the existence of a negative-*cost* cycle. It is important to understand that these cycles are formed by negative costs introduced by adding back edges when we created the residual graph. So why would we use the Bellman-Ford algorithm to accomplish this? As a refresher, the Bellman-Ford algorithm solves the shortest paths problem but is far less efficient than other algorithms for shortest paths such as Dijkstra's algorithm. Its pseudocode is displayed here:

**Figure 3: Bellman-Ford Algorithm Pseudocode** [6]

```
BellmanFord(G, V, E):
       for v in V:
           v.distance = infinity
           v.predecessor = None
       source.distance = 0
       for i from 1 to |V| - 1:
           for (u, v) in E:
               relax(u, v)
relax(u, v):
       if v.distance > u.distance + weight(u, v):
           v.distance = u.distance + weight(u, v)
           v.predecessor = u
```

However, the reason we want to use the Bellman-Ford algorithm is because it works when a graph has negative edges [7]. When a negative edge is present, each iteration will result in one or more of the values becoming smaller even after |V-1| iterations where we are guaranteed to have every vertex reach every other vertex. Using this knowledge, if we run the Bellman-Ford algorithm with the *sink* as the origin vertex and do a $|V|^{th}$ iteration and a value is decreased then we know there must be a negative cycle in the graph.

At this point, our next step in the Klein '67 cycle cancelling algorithm is to find a negative cycle if one has been proven to exist. If a negative cycle is shown to not exist then we terminate the Klein '67 cycle cancelling algorithm. There are a number of to determine what nodes are in the negative cycle, but the most obvious to me is to utilize the predecessor fields of the vertices in G (which store what vertex the new distance value came from and were created during execution of the Bellman-Ford algorithm) and "follow the trail" of predecessors. We take the predecessor of the predecessor of the predecessor etcetera until we have found a repeating vertex (determined by id). In the actual implementation of the animation, this is performed by pushing the vertices onto a stack and when we find a vertex that is already in the stack, we empty the stack from the bottom-up to just before the matching node. This results in the complete list of nodes in the negative cycle (or at least a single negative cycle, since there may be more than negative cycle in the graph).

After we have our list of nodes in the found negative cycle, we need to redirect some of the flow to the more cost-effective path. But how much flow can we augment at a time. This is why we need to find δ**,** the smallest amount we can augment for the found negative-cost cycle. This value is equal to the weight of the smallest residual edge (and can be either a backward or forward edge). If the edge is a back edge we take away δ units of flow from the edge in G and if the edge is a forward edge we add δ units of flow.

Finally, we have reached the end of our while loop and adjusted a negative-cost cycle. Now we need to see if we either terminate or if there are more negative cycles to be adjusted. We retest the condition of our while-loop by running the Bellman-Ford algorithm as described earlier. If there are no more negative-cost cycles found after execution, then the while statement evaluates to false and we terminate the algorithm, resulting in a graph with flow values that are have a minimized net cost and a maximized net flow. Otherwise, we continue the while loop and continuously eliminate negative-cost cycles from the residual graph.

**Analysis of the Algorithm**

Overall, the algorithm has a non-polynomial time complexity of $O(nm^2CU)$ where n is the number of vertices, m is the number of edges, U is the maximum capacity of an edge, and C is the edge's maximum absolute value cost [8]. It can be plainly seen that this is not a very efficient algorithm especially when compared to more recent algorithms that address the same problem. One such algorithm is the minimum mean cycle cancelling algorithm that yields a strongly polynomial solution of $O(m^2n\log(n))$ [9]. Still, Klein '67 cycle cancelling algorithm is still important to learn about since it is one of the most fundamental solutions to the minimum cost maximum flow problem.

This algorithm works because it follows a theorem called the Negative Cycle Optimality Conditions which states that a feasible flow x* is optimal if and only if G(x*) contains no negative-cost directed cycles. The proof hinges on the fact that increasing flow around negative-cost directed cycle results both in a feasible flow and [2]

There are many useful applications for this algorithm, even if it is not the most efficient. One of the major uses of this algorithm is to solve supply and demand problems where there are multiple source nodes that have finite amount of flow to give (supply) and sink nodes that have a flow deficit and require flow to be given (demand). To move flow from a supplier to a demander, the flow must travel over edges with a cost and a capacity. In real life, this problem could be replicated with a shipping company with multiple warehouses of supplies to ship and several people in different locations ordering their supply. To get the supplies to the customers, the supplies would need to be sent down roads which have a traffic capacity as well as a cost (gas, tolls, etc.). These supply-demand scenarios are actually a subset of the problem we looked at where there is a single source and a single sink. To reduce the supply-demand problem into the problem we are familiar with, we can add source and sink node to the directed graph where the source points to the suppliers and the demanders point to the sink [2]. The cost for these edges would be 0 and the capacities would equal the absolute value of the supply/demand.

# Implementation and Animation

The implementation and animation is a medium-large sized program (in the context of a single person and time constraints) written in C# and running the Unity 5 video game engine.  The program has defined 18 classes and sits at just over 1000 lines of code.

## Usage and Work Flow

To open the program, extract the contents of agyori-cs4445-project.zip into an empty folder.  There will be two different executable files.  One is called agyori-cs4445-project-x86.exe and is compiled for 32-bit Windows operating systems while the other is called agyori-cs4445-project-x86_64.exe and is compiled for 64-bit Windows operating systems.  Ensure that their accompanying Data folder and the executable is at the same level and run the executable according to your system type.

A window should appear asking what resolution you would like to run the program at.  This program runs best at 1024x768 in Windowed mode.  After you have selected your resolution, click the Play button.

After the Unity logo splash screen, the program will appear and you will see the cycle cancelling algorithm's pseudocode in the top right corner.  In the top left corner is a help button that displays the controls (also listed in the following section).

The program has two main states: the design state and the animation state.  The design state allows the user to create a graph quickly, easily, and dynamically. Capacity/cost values are editable, and nodes/edges can be added or deleted.  By pressing the Enter key the program checks if the user inputted graph is valid.  If it is, the program changes to the animation state. In the animation state, the user may use the left and right arrow keys to move forward and backwards through the algorithm's animation steps. As the user moves through the steps, the pseudocode in the top right updates its colours to reflect what step of the algorithm the user is seeing. Pressing Enter again at any time in the animation state will move the user back into the design state. This clears all the flows (but keeps the capacities and costs) of the graph and unlocks it so the user can make modifications to the graph and run the algorithm again. This encourages creativity and exploration of the algorithm, allowing the user to probe for weaknesses in the algorithm. As well, this gives the user the ability to see the cause and effect of adding different paths or modifying cost/capacity values.

## Controls

- spacebar: create node
- left shift + left click + drag: create edge
- right click: delete node or edge
- click on capacities and costs to edit
- press S while hovering over a node to assign it as the source node
- press T while hovering over a node to assign it as the target node
- enter: toggle design/animation state
- left/right arrows: previous/next animation

## About Unity

With the aim of understanding the data structures used in this program, a brief explanation of basic Unity principles is in order. Unity, being a video game engine, allows for programmatic translation, rotation, scaling, field updates, and other actions to happen frame-by-frame.  One of the most basic classes in Unity is called GameObject and can represent anything held in the scene.  A GameObject could be empty, where it only stores a transform (position, rotation, and scale values), but more usefully it can be assigned many Components.  A Component is an abstract class that can be thought of as a complex field for a GameObject.  For example, you could represent a text box with a GameObject containing a Text Component.  The Text component then has many different primitive fields that can be added such as text and RGB values.  GameObjects can also hold other GameObjects to form a GameObject hierarchy.

The Script Component is a key part of Unity, since we can attach a script to a GameObject so that it is controlled by the Script.  These scripts may be written in C# or JavaScript and must inherit Unity's MonoBehaviour class.  Objects that inherit MonoBehaviour have access to frame-by-frame functions such as Start (which is called on the first frame of the objects instantiation) and Update (which is called every frame). Furthermore, MonoBehaviours cannot be instantiated with the "new" keyword, and instead must be instantiated with the Instantiate keyword and the result is a new GameObject in the scene.

GameObjects can be made in the Unity editor and be unique to the scene or can be saved off into "prefabs".  In this program, all repeating GameObjects such as Nodes, Edges, and EdgeTags are prefabs, while global singleton objects like Graph and GameController are not prefabs.  GameObjects can have references to prefabs allowing them to create other objects (GameController holds all the prefabs).

The key ideas the reader should understand is that GameObjects are rendered in the scene and can form a GameObject hierarchy, and that references to prefabs can be stored in a script by setting the field to public and then going into the Unity Editor and linking the field to the content.

## Data Structures

### Part 1 – Global Objects

**Camera**: An orthographic camera is set up in the main scene and points along the Z-axis, giving the application a 2D feel (while in reality, all the objects are held in 3D space).

**GameController**: This object holds the GameController script (a MonoBehaviour) which is in charge of managing all states of the game, including both design and animation states. It is the most macroscopic object in the program and holds the Graph, AlgorithmController, and Pseudocode.

**Graph**: The graph holds a collection of Node and Edge objects which are further described in the following section. It is the parent GameObject for all created Nodes and Edges (i.e. top of the hierarchy).

**Pseudocode**: A simple GameObject container for some text fields that represent a different step of the algorithm. Uses codes from the GraphState to highlight the correct step.

### Part 2 – The Graph Editor

**Node**: This prefab/MonoBehaviour represents a node in the graph. It has an Id and a colour value that can be manipulated for animation purposes. This class also handles MouseOver interactions with the GameObject such as clicking and dragging to move the node around. As well, nodes contain a Predecessor object that is utilized during algorithm execution to store data such as where the node was visited from and from what direction. This type is explained further in the following section.

**Edge**: A prefab/MonoBehaviour that represents an edge in the graph. The edge contains flow, capacities, and costs and can be locked/unlocked for editing. Edges contain a single EdgeTag.

**EdgeTag**: An EdgeTag is a prefab/MonoBehaviour that is assigned to an Edge and holds the different InputFields for an Edge such as flow, capacity, and cost. An event listener in this object updates the actual Edge values whenever the InputField in the EdgeTag is updated.

## Part 3 – The Algorithm Execution

**Predecessor**: This object is used by the CycleCancellingAlgorithm (and in theory other Algorithms too if more were to be added later). It holds information pertaining to things like who last accessed this node using what edge and whether it was a forward or back edge. This field is important for generating the maximum flow with the Ford-Fulkerson algorithm and finding negative-cost cycles with the Bellman-Ford algorithm.

**GraphValidator**: This object sole responsibility is to take a graph as input in the constructor and when the validate function is called it returns either true or false. A graph is valid if all the capacities and costs have been initialized, there is an assigned source and target node, and the source node can reach the target node.

**AlgorithmController**: The role of this object is to control the playback of an algorithm. Taking an algorithm in the constructor, the controller has three main functions utilized by the GameController: Play, Previous, and Next. When AlgorithmController.Play is called, the Algorithm is executed *to completion*. During the execution of the algorithm, the algorithm generates a list of GraphStates ordered in chronological order. The controller accesses these GraphStates by using an internal step counter that keeps track of what "slide" of the animation the user is on. These GraphState "slides" can be navigated by using the Previous and Next methods.

**Algorithm**: An abstract class for an algorithm. All algorithms must contain a Graph, a list of GraphStates (used for animation purposes) and provide an Execute method to kick off the algorithm.

**CycleCancellingAlgorithm**: This object implements the Algorithm class. There are three major components held together by a body method. One is the FordFulkerson algorithm which generates a maximum flow on the graph, another is BellmanFord to find a negative-cost cycle, and finally is the AdjustCycle to augment the cycle. During execution, Nodes and Edges are painted and the Flow values changed. These changes get saved into GraphStates and stored in a list sorted in chronological order. Every time we take a snapshot, we reset the colours so we do not interfere with the next "slide" of the animation.

**GraphState**: A GraphState is an object that holds a snapshot in time for a Graph. A graph can package itself up into a GraphState which can then be stored in a collection of GraphStates by an Algorithm. A GraphState holds a list of NodeStates and a list of EdgeStates that were packaged by the Graph when creating the GraphState. The reason we do not want to store actual Graph, Node, and Edges themselves is because they would need to Instantiated as GameObjects, and thus rendered in the scene which is far less efficient than creating regular objects.  Graph states also may have a code that can be utilized by the Pseudocode to tell it what step of the algorithm the user is looking at.

**NodeState**:  Similar to the GraphState and EdgeState, this is an object that stores the state of a node in a snapshot in time.  It holds an Id and a Colour value.

**EdgeState**:  Similar to the GraphState and NodeState, this is an object that stores the state of an edge in a snapshot in time.  It holds Id, Colour, Flow.

## Conclusions

Overall, this projected mostly set out what it tried to accomplish.  Let's revisit the three goals we set out to achieve with this project:

**(i)** Create a learning tool that allows people to quickly and easily learn the Klein '67 cycle cancelling algorithm.

I believe this was achieved.  A criticism could be made that I should have made more animation slides for each individual algorithm (F-F, B-F), but one has to argue whether the purpose of this animation is to teach the user about the Klein '67 cycle cancelling algorithm or it's sub-components.  Obviously, at least the priority should be set on the cycle cancelling algorithm.

**(ii)** Visualize the algorithm in such a way that a user can understand the individual steps of the algorithm and understand where the algorithm is inefficient.

Out of all the goals, I think this one was the least achieved.  Because the program at it's current state does not have the capability to display residual edges and their costs, it can be difficult to probe into the nature of the negative-cost cycles as opposed to just acknowledging that they exist.  This is the biggest drawback of my program in my opinion. Preferably there would be a toggle to show/hide the residual edges.

**(iii)** Give the user enough input choices so that they can be creative and interact with the animation and the algorithm.

In contrast to the slightly underwhelming achievement in the second goal, I believe that I have gone beyond in achieving this third goal.  My program has the ability to create any type of directed graph (even with negative costs on initialization), assign a source/sink, customize edges, validate the graph, and reliably have the algorithm run and update the graph.  Furthermore, the animation can be played backwards and forwards and the pseudocode in the top right of the screen gets updated.  What's more is that the animation can be exited at any time and modifications to the graph can be made and put through the animations once more.  This allows the user to get a hands-on feel for how the algorithm operates, both quickly and easily.

Overall, this project has resulted in a solid tool for introducing people to the concepts of cycle cancelling and the problem of minimum cost maximum flow.  With some minor improvements / general refactoring, it could be even more effective.

## References

[1] Zealint. 2016. *Minimum Cost Flow: Part One – Key Concepts.* Accessed November 26, 2017. https://www.topcoder.com/community/data-science/data-science-tutorials/minimum-cost-flow-part-one-key-concepts/.

[2] The University of North Carolina. 2016. *Lecture 12: Basic Algorithms for Min Cost Flow.* Accessed November 28, 2017. http://www.unc.edu/depts/stat-or/courses/provan/STOR724_web/lect12_mcbasic.pdf.

[3] Technische Universität München. 2016. *Cycle Cancelling Algorithm.* Accessed Novemeber 28, 2017. https://www-m9.ma.tum.de/graph-algorithms/flow-cycle-cancelling/index_en.html.

[4] GeeksforGeeks. 2017. *Ford-Fulkerson Algorithm for Maximum Flow Problem.* Accessed November 26, 2017. http://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/.

[5] Skoltech, Karger. 2013. *Lecture 11, 10/07: Min-cost Flows.* October 10. Accessed November 27, 2017. https://www.youtube.com/watch?v=UdtwpgjfR3g&feature=youtu.be&t=31m5s.

[6] Chumbley, Alex, Karleigh Moore, Eli Ross, and Jhimin Khim. 2017. *Bellman-Ford Algorithm.* Accessed November 27, 2017. https://brilliant.org/wiki/bellman-ford-algorithm/#algorithm-psuedo-code.

[7] GeeksforGeeks. 2017. *Dynamic Programming | Set 23 (Bellman–Ford Algorithm).* Accessed November 26, 2017. http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/.

[8] Zealint. 2016. *Minimum Cost Flow: Part Two - Algorithms.* Accessed November 28, 2017. https://www.topcoder.com/community/data-science/data-science-tutorials/minimum-cost-flow-part-two-algorithms/.

[9] Jean Bertrand Gauthier, Jacques Desrosiers, Marco Lübbecke. 2014. *About the minimum mean cycle-canceling algorithm.* August 7. Accessed November 27, 2017. http://www.sciencedirect.com/science/article/pii/S0166218X14003060.