Implementing Blossom V Algorithm for Computing Minimum Cost Perfect Matching in a General Graph

GSoC 2019 Proposal
Julia - LightGraphs
Mentors : Mathieu Besançon, Simon Schoelly

Tushar Sinha

April 8, 2019

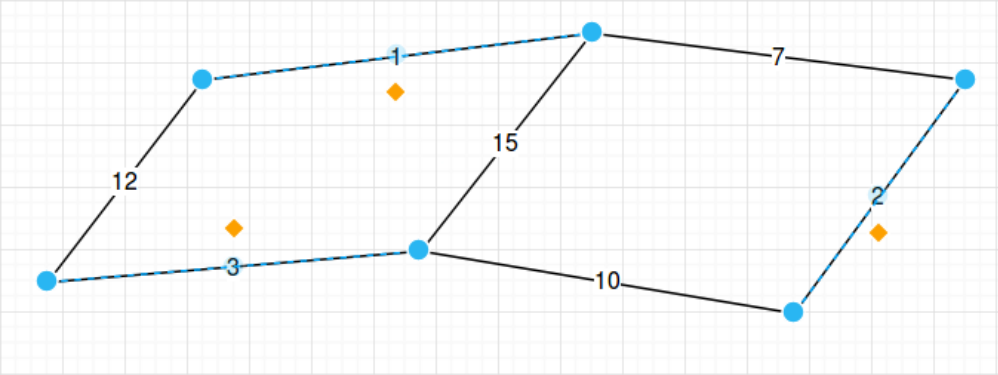
1 INTRODUCTION

1.1 Matching problem in graphs

Let G = (V,E,c) be an undirected weighted graph where V is the set of vertices, E the set of edges and c denotes the cost of edges. A matching M in G is a set of pairwise non-adjacent edges, none of which are loops; that is, no two edges share a common vertex.

A maximum matching M of a graph G is a matching that contains the maximum possible number of edges from the graph. That is, for every matching M′of G, $|M| \geq |M'|$.

A matching is called a perfect matching if all the vertices of the graph are saturated i.e. they are the end points of some edge of the matching M.



Often the graphs are weighted and in that case we are interested in finding minimum/maximum cost matching of a given cardinality in the graph. The most popular amongst these is perfect matching in a graph in which exactly $|V|/2$ number of edges are part of the matching. The cost

of a matching is defined as the sum of weights of edges in matching M. Minimum cost perfect matching in a graph is depicted in above image.

Matching problems have some variations which are as follows:

- Maximum bipartite matching problem.
- Minimum weight perfect bipartite matching problem.
- Maximum matching problem.
- Minimum weight perfect matching.

All the four problems mentioned above are closely related. In fact, solutions of harder problems are based on the ideas used for easier problems. It is easier to solve matching problems in bipartite graphs because we can quickly search for augmenting paths because we explore from each vertex only once [2,3]. The major ideas in solving these problems are:

- Finding augmenting paths to increase the cardinality of the matching.
- Contraction of odd length cycles into single pseudonodes.
- Growing the alternating tree to find an augmenting path, and using dual-primal approach to convert linear programming problem into purely combinatorial algorithm. This idea involves implicit formulation of the dual linear program, maintaining its feasible solution, and increasing the cardinality of the matching with respect to the complementary slackness condition so that the final solution is also feasible to the primal linear program which ensures optimality.

Jack Edmonds discussed the problem in [2] and proposed a polynomial-time algorithm based on ideas described above for solving this problem in [3]. His approach was based on the fundamental fact, proven in [6], that a matching is maximum if and only if there is no alternating path connecting two unmatched vertices. Further researches were focused on improving average running time. Improvements to the Edmond's Blossom I algorithm include:

- Using more efficient data structures to improve the running time.
- Using different ways to perform dual updates.
- Involvement of additional initialization techniques.

Using above ideas, the worst-case complexity of the blossom algorithm has been steadily improving. First, a running time of $O(V^3)$ was achieved which was improved to $O(EVlogE)$, which was further improved to $O(E(Vloglogl og_{max\{m/n,2\}}V + VlogV))$. The current best known result in terms of E and V is $O(V(E + logV))$.

My project is focused on developing an implementation of the fast version of Edmond's Blossom I algorithm, namely Blossom V, for minimum weight perfect matching in general graphs. This algorithm was developed by Kolmogorov [8].

Implementation of Blossom IV and Blossom V don't have such a good worst-case complexity. Instead, they possess very good average running time, which gives them advantage over other algorithms in this field (in fact, Blossom IV has $O(V^3E)$ worst-case complexity observed in [8]). That is why improvements described above usually don't reduce the worst-case running time. This area is an active field of study of the best techniques to use in practice. As a result, implementations of Blossom IV and V have different subversions that use various strategies.

1.2 The LightGraphs Package

LightGraphs.jl [10] is a Julia package that implements several commonly used graph algorithms. LightGraphs.jl has many sister packages one of which is LightGraphsMatching.jl [11] which supports commonly used matching algorithms in graph theory including Blossom V algorithm. This package is dependant on BlossomV.jl [12] package which is a Julia wrapper around Kolmogorov's Blossom V algorithm. The code by Kolmogorov[1] is implemented in C and is called from Julia using the following ccall :

```
ccall((:matching_solve, $_jl_blossom5), Cvoid, (PtrCvoid,), matching.ptr)
```

However, author's code has UCLB Licence ALBL 1.02 which prohibits its commercial use and is licensed for research purposes only. Another problem is that calling this code from Julia often leads the code to segfault. Also, a lot of Julia tools can't be used because BlossomV.jl calls C functions from [1].

I propose to write a native Julia implementation of Blossom V algorithm in a new package named BlossomMatching.jl thus totally removing the dependency on C code. There are only a couple of other implementations available for this algorithm. LightGraphs will become one of the first graph package to have implementation of Blossom V algorithm and it will make the JuliaGraphs ecosystem more feature complete, rich and self-contained with lower external dependencies

2 PROJECT DESCRIPTION

In this section I describe the different logical and structural aspects of the implementation.

2.1 INTERFACE OF THE IMPLEMENTATION

In this sub-section I'm going to describe the interface of the implementation. I plan to provide the following options in the implementation :

- Initialize the solver with a weighted graph and solve problem instance. Obtain the global solution (set of matched edges), check, whether a particular edge is matched, or get the corresponding matched vertex of a given vertex which is also called the mate of the vertex.
- Specify the configuration of the algorithm (the file, to which the solution should be saved, type of initialization, etc).
- Obtain the certificate of optimality (values assigned to the dual variables, whose sum equals to the weight of the matching).
- Get the statistics of the algorithm execution, i.e. number of primal operations (shrink, expand, etc.), time spent to perform them. This is one type of benchmarking.
  Goal: Provide information for further research or for additional improvements.

2.2 DATA STRUCTURES

This algorithm needs additional data structures to optimize time complexity. Some of these like Fibonacci heaps, pairing heaps are not yet implemented in DataStructures.jl and hence I need to implement them from scratch.

2.2.1 Nodes

During the execution of the algorithm we need to maintain alternating trees, which help us to find alternating paths and augment the matching. Nodes of the trees need to store following information:

- whether the node is a vertex or it is a pseudonode.
- whether it is an outer node or is contained in another blossom.
- its label (+, - or ∞, see [8]). Besides that it stores information needed to grow a tree, to perform an augmentation, etc.

One way to store this data is to have global dictionaries for all vertices. In case the vertices are integral and in the range 1:N, then using a vector is also sufficient.
Responsibilities:

- store dual variables, edge pointer and other information needed for the execution of the algorithm.
- encapsulate operations performed on this information in order to speed up primal and dual updates.

2.2.2 Edges

Edges also have information attached to them. For instance, an edge has a slack. Edges are store in priority queues, so they store references to these data structures for quick access.
Responsibilities:

- store edge slacks and other attached information.
- encapsulate operations performed on this information in order to speed up primal and dual updates.

2.2.3 Trees

This data structure can be implemented either explicitly or implicitly. In the first case it will encapsulate all operations needed to grow the tree, shrink, expand nodes, etc. This interface then will be used in the primal updater function. In the second case these operations will be directly implemented in the primal updater function. A set of alternating trees will be, in fact, an array of references to the root nodes. In particular, the latter approach is used in [1].

2.2.4 Priority Queues

This data structure is used to find an edge with a minimum slack or to find the ”-” pseudonode of a tree with the minimum dual variable value in O(1).
Responsibilities:

- store edges.
- speed up dual updates.

There are two possible alternatives as described :

- **Fibonacci Heaps** : This is a well-known data structure. It was used in the first version of Blossom V.
- **Pairing Heaps** : Pairing heaps are a simplified version of Fibonacci heap. They take less memory (only 2 pointers per node) and are more efficient in practice. Fibonacci heaps were replaced by them in the second version of the [1]. According to [1], pairing heaps are marginally faster. They are described in [13].

If the implementation of the Fibonacci heaps suits for the algorithm, then it will be used initially. Nevertheless, pairing heaps are claimed to be more lightweight then Fibonacci heaps and will be implemented in the case the latter data structure doesn't supplies the needs of the algorithm.

2.3 CODE STRUCTURE

Here I describe the way I'm going to decouple the implementation into different directories, files and functions. My task is not to mess up everything into one heap. This decoupling is aimed on making implementation process less error prone and also to make subsequent extensions easier.

2.3.1 Initialization Function

This function will be used to precompute some intermediate matching to improve the running time of the algorithm.
Responsibilities of the function: Find initial matching to start the execution of the algorithm with non-empty set of edges and non-zero dual variables.
Two different strategies for initialization are discussed in [8] which are described as follows :

- **Greedy initialization** : This is the simplest type of initialization. For each node v in graph G = (V, E) we set the dual variable y_v to $(1/2)min\{w_{u,v} : u \in N(v)\}$. Then we traverse the nodes again, greedily increase dual variables and choose matching edges, if it is possible.
- **Fractional initialization** : The idea is to solve a linear program for a graph instance, which doesn't include constraints for all sets of vertices of odd cardinality. In fact, this LP formulation is identical to the one used in the Hungarian algorithm. In general graphs this is not sufficient and as a result, primal variables can be equal to 0, 0.5 or 1. The next step is to fix the solution and begin the execution of the main algorithm with obtained matching.
  Advantages : The fractional matching problem is an efficient way to initialize the matching. It is also much faster compared to the Blossom algorithm.

2.3.2 Primal Updater

This function will use information stored in vertices and edges to grow trees, find alternating paths and augment the matching. It performs shrinking, expanding, growing and augmenting operations.
Responsibilities:

- perform primal operations.
- update data stored in the vertices and edges of the trees.

There exist cases, when primal updates can't be performed exactly after dual update. This can happen when there exist no perfect matching in the graph (note: in this case the algorithm doesn't guarantee to choose the minimum weight matching among all existing). This function should also correctly report these situations.

2.3.3 Dual Updater

Algorithm needs dual variable updates in the case when primal updater can't grow trees any further. There are several possible ways to perform dual updates:

- **Single tree approach** : We choose a tree, compute δ for it by means of information stored in priority queues, and update dual variables of nodes or pseudonodes of the chosen tree.
  Advantages: This method is pretty simple and efficient at the beginning of the algorithm.

- **Multiple tree single fixed δ approach** : We consider all constraints imposed by the vertices of all trees. Then we compute fixed δ and update dual variables.
  Disadvantages: This method implies examining all trees. With fixed the dual update is determined as a bottleneck among all trees, and thus can be quite small [8].

- **Multiple tree variable δ approach**: Again we take vertices of all trees in consideration, but allow each tree to have its own δ_{T_i} . There are several complications with this technique, for example, (+, -) edges may create circular dependencies between the δ_{T_i} , so not all δ can be different. This technique also involves using supporting algorithms for the computation of the δ_{T_i} .
  Advantages: This method performs better in some cases comparing to the fixed δ approach. Furthermore, due to its flexibility it can increase dual objective by a much larger amount, which can lead to more primal updates.

Here I need to make one note. It is shown in [9] and mentioned in [8] that final 5% of augmentations can take 50% of time and it often happens. That is why it makes sense to use single tree approach to perform first 95% of augmentations and multiple tree fixed δ or variable δ approach for the remaining 5%. Variable δ variant is used in Blossom V, as it has a couple of advantages over fixed δ approach. As mentioned above, this technique is pretty advanced. It formulates a linear program for δ's of different trees, which turns out to be a dual to the minimum cost maximum flow problem. It solves the dual problem with the successive shortest path algorithm of Ford and Fulkerson [1].
Responsibilities: Manipulate the dual variables assigned to vertices and pseudonodes. Also increase the dual objective function.

2.3.4 Controlling Function

Responsibilities: Controls the execution of the algorithm, calls the functions of primal and dual part, handle notifications from these parts, starts the executions with an appropriate initialization. It also decides, which type of dual updates to use at a particular stage of the execution. There is also one technique worth noting here which is described as follows:

- **Price and repair** : This technique is mainly used to solve large problem instances. The complexity of the algorithm is heavily influenced by the number of edges in the graph. Initially the sparse subset of edges is selected and the problem is solved on this subgraph. Then the slack of every edge is computed (pricing). Finally edges with negative slack are added to the subgraph and the procedure repeats. There are different techniques of choosing a sparse subgraph, for instance, k-nearest neighbor - selecting of k edges with minimum weights incident to a vertex for some k.
  Advantages: This technique optimizes the running time of the algorithm for large problem instances.

Disadvantages: This technique is advanced and requires additional data structures and supporting algorithms for computing LCA in O(1).

Price and repair technique improves running time mainly on the large problem instances. Multiple tree fixed δ approach is an alternative for multiple tree variable δ approach. It is mentioned in [8] that latter has its advantages over the other one. That is why I consider the implementation of these techniques as not the task of prime importance. They can be added after the structure of the code is refined appropriately.

2.4 BENCHMARKING

Main goal of benchmarking is to measure average running time, compare the performance with different heap implementations namely Fibonacci heaps and pairing heaps and compare the algorithm with other implementations.

Benchmarking can be performed in a similar to [8,9] way. Namely, large graphs can be generated using existing libraries or as a set of points on a plane with weights as the Euclidean distance between them. This strategy will also be used to test the performance of different implementation options.

In my opinion, the best way to check correctness of this implementation on large instances is to use [1].

3 TIMELINE

Pre-GSoC Period (April 10-May 5)

I have to submit my B Tech. thesis around 15th April and then my end semester exams follow from 20th-29th April. So I won't be able to work during this time. I can restart the work from 30th April. I plan to submit more PRs before 6th May to get myself more familiar with LightGraphsFlows.jl & LightGraphs.jl . The next algorithms which I plan to add are:

- Push Relabel algorithm for computing the maximum flow of flow network in $O(EV + V^2E^{0.5})$ time . Current implementation in LightGraphsFlows.jl has $O(V^3)$ time complexity.
- Finding Lowest Common Ancestor (LCA) of 2 nodes in a tree by reducing it to a Range Minimum Query (RMQ) problem. Each query will require $O(logN)$ time with one time preprocessing in $O(N)$ time.

Also, from whatever time I can manage during April 10-April 29, I plan to go through the codebase of LightGraphs.jl to learn more about the coding style followed and to know more about intricacies of Julia.

Community Bonding Period (May 6 - May 26)

I plan to go through the research papers mentioned in the References section of this proposal again in this time. A lot of implementation details like which data structures to use and what additional information to store with nodes, edges are already covered in the Blossom V paper but due to complexity of the project, a lot of concrete implementation details need to be decided beforehand. I plan to take all the design decisions in this period with the help of my mentors.

First and Second week (May 27 - June 9)

I plan to start by working on the representations of nodes, edges and tress and implementing data structures which store additional information required with them.

Third and Fourth week (June 10 - June 23)

I plan to work on initialization function by implementing greedy initialization. Also, implement all operations of primal updater in primal updater function and implement single tree approach in dual updater and control functions.

Fifth week (June 24 - June 30)

I plan to use this week as a buffer for catching up on any previous work left.

Sixth, Seventh and Eighth week (July 1 - July 21)

I plan to implement fractional initialization and multiple tree variable δ approach in initialization and dual updater function respectively. These parts are very important for the algorithm because they substantially improve its running time.

Implementing Fibonacci and pairing heaps for using them as priority queues and experimenting with them for speed will also be done in these 3 weeks.

Ninth week (July 22 - July 28)

I plan to use this week as a buffer for catching up on any previous work left.

Tenth week (July 29 - August 4)

I plan to spend this week in connecting all the work done as of now and get the algorithm working correctly by the end of 10th week. Also, I plan to connect BlossomMatching.jl with LightGraphsMatching.jl removing its dependancy on BlossomV.jl .

Eleventh and Twelfth week (August 5 - August 18)

I plan to spend this period in testing, benchmarking and writing documentation. Also, I would write tests for the algorithm covering different type of graphs as described in [8].

Thirteenth week (August 19 - August 25)

Wrapping up all the work. Making sure that I've reached all project goals and writing a comprehensive blog showing the benchmarks and comparison with other existing implementations of Blossom V algorithm.

Post GSoC period

Once the Blossom V algorithm is up and running, it can be extended to solve the Maximum Weight Perfect Matching problem and Maximum Weight Matching problem. Galis's paper[5] and Guido Schäfer's Master's thesis as given on [1] describe the ways to extend Blossom V implementation to solve above problems.

4 PREVIOUS CONTRIBUTIONS

- #1173 : Added strongly_connected_components_kosaraju(G) function (with tests) for finding strongly connected components in a directed graph.
- #1182 : Made changes to strongly_connected_components(G) to make it 4-5 times faster. Also after changes, the algorithm required nearly 500 times lesser memory.
- #1187 : Modified floyd_warshall_shortest_paths(G) to indicate negative weight cycles on the path between 2 vertices by outputting dists[i,j] = - Inf . Also made changes to indicate a negative weight cycle in the graph.
- #1189 : Added spfa_shortest_paths() function (with tests) for calculating single source shortest path in a graph with negative weights. Also added has_negative_cycle_spfa(G) function to indicate negative weight cycle in the graph. Both functions are faster than their counterparts viz. bellman_ford_shortest_paths(G) and has_negative_cycle(G).
- #119x (Work in progress) : Added lca(i,j) function (with tests) to calculate lowest common ancestor of 2 nodes i and j in a tree G using binary lifting algorithm.

5 ABOUT ME

I am a 4th-year undergraduate student pursuing my Bachelors of Technology in the Department of Electrical Engineering at Indian Institute of Technology [IIT], Kharagpur, India. I work on Ubuntu 16.04 LTS. I use Juno+Atom as my primary text editor for working in Julia whereas I use Sublime Text for writing codes in C++ and Python. I love both Atom and Sublime Text for their power and flexibility. I'm proficient in C++ and Python. I prefer to use C++ for solving algorithmic coding problems whereas I prefer Python for all other purposes because it lets me easily convert my ideas into code. My favourite programming language is C++ because of its STL library which helps me immensely in solving coding problems.

I am programming in Julia for the past 3 months and now I have learnt it to a fair extent and can program comfortably in it. From my previous programming experience I know that it takes some time and experience in understanding the intricacies of a programming language and hence I have planned to go through the codebase of LightGraphs.jl in pre-GSoC period as I feel that reading efficient code is the 2nd best way to learn the good programming practices of a language. Best way is to do a project using the language.

5.1 Why me?

Implementing Blossom V project will require a strong background in Algorithms, Data Structures, Graph Theory and Linear Programming and I'm proficient in all of these. My knowledge about

these topics is attributed to the following courses that I enrolled and cleared in my college : Algorithms 1, Algorithms 2, Advanced Graph Theory.

Apart from this I have been solving algorithmic coding problems on various competitive programming websites like Codeforces, Codechef, SPOJ, Hackerrank etc. for last 3 years. My programming skills have improved a lot due to this and now I strive to write time and memory efficient code. I'm aware about a lot of algorithms and can think about solutions to complex problems as well. This project is complicated in every sense due to great deal of programming complexity and heavy use of advanced data structures. I think that my knowledge about various algorithms and my ability to think of complex problems and converting them into code will immensely help me in implementing this project.

Also, I have been involved in a lot of projects during 4 years of my college. The projects whose code could be made public have been put under this repository on my github account.

Link to my CV.

5.2 Contact Information

Github: sinhatushar
Email: bikztushar@gmail.com, bikztushar@iitkgp.ac.in
Skype: Reach me by searching for my email-id viz. bikztushar@gmail.com on skype.
Slack Id: sinhatushar
Contact number: (+91) 9932560655
Time Zone: UTC+5:30 (India)

6 SUMMER LOGISTICS

6.1 Working hours

I am hoping to dedicate around 40-45 hours of work per week throughout the summer (6th May onwards) on this project. A reduction in working hours in a week because of unavoidable conditions will be surely compensated later in the following weeks. Also, I will make sure to communicate the same to my project mentors **Simon Schoelly** and **Mathieu Besançon** well in advance. After my graduation in April, I will be joining **JP Morgan Chase & Co.** for a **software developer** role. The joining date is 12th August and I think I will have wrapped up my project by then. So overall I plan to spend around 550-600 hours of work into this project.

7 REFERENCES

[1] Blossom V author's implementation
[2] J. Edmonds. Path, trees, and flowers. Canadian J. Math., 17:449–467, 1965.
[3] J. Edmonds. Maximum matching and a polyhedron with 0-1 vertices. J. of Research at the National Bureau of Standards, 69B:125–130, 1965
[4] LEDA library C++
[5] Zvi Galil, Efficient algorithms for finding maximum matching in graphs, ACM Computing Surveys, 1986.
[6] Claude Berge. Two theorems in graph theory. 1957
[7] Blossom IV implementation.
[8] Vladimir Kolmogorov. Blossom V: A new implementation of a minimum cost perfect

matching algorithm. 2009
[9] W. Cook and A. Rohe. Computing minimum-weight perfect matchings.INFORMS Journal on Computing, 11(2):138–148,February 1999.
[10] LightGraphs.jl
[11] LightGraphsMatching.jl
[12] BlossomV.jl
[13] M.L. Fredman, R. Sedgewick, and D.D. Sleator. "The Pairing Heap: A New Form of Self-Adjusting Heap". In: Algorithmica (1986).