Fast algorithm for the minimum weight perfect matching problem

Timofey Chudakov

1 Introduction

Matching problem has several variations. Namely, there are maximum bipartite matching problem, minimum weight perfect bipartite matching problem, maximum matching problem, and last but not the least minimum weight perfect matching. 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 [11].

2 Description of the problem in general

All four algorithms mentioned in the previous section are closely related. In fact, solutions of harder problems are based on the ideas used for easier problems. This ideas include 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. The last 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 [9] and proposed a polynomial-time algorithm based on ideas described above for solving this problem in [8]. His approach was based on the fundamental fact, due to Berge [3], 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 data structures to improve the running time.
- Using different ways to perform dual updates.
- Involvement of additional initialization techniques.

2.1 Complexity

At this moment there exists implementation which achieves $O(mn\,log(n))$ running time complexity. 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 $O(n^3m)$ worst-case complexity observed in [11]). 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.

In the next sections different logical and structural aspects of the implementation are discussed.

3 Interface of the implementation

In this section I'm going to describe the interface of the implementation. There will be options to

- 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.
- 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.

4 Data structures

The algorithms needs additional data structures to optimize time complexity. They are described below.

4.1 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 [11]). 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 maps for all vertices. I consider first approach to be more object-oriented, in particular, it adds encapsulation of the operations performed on nodes.

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.

4.2 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.

4.3 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 module. In the second case these operations will be directly implemented in the primal updater module. 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 [5].

4.4 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 $O(1)$. There are two possible alternatives.

Responsibilities: store edges, speed up dual updates.

4.4.1 Fibonacci heaps

This is a well-known data structure. It was used in the first version of Blossom V. Also, it has been already implemented in JGraphT.

4.4.2 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 Blossom V. According to [5], paring heaps are marginally faster. They are described in [10].

5 Modules

Here I'll describe the way I'm going to decouple the implementation into separate modules. Even though, the algorithm can be partitioned into certain parts, the modules responsible for these parts can't be completely independent, because they all work with the same data structures. My task is not to mess up everything into one heap. This decoupling is aimed on making subsequent extensions easier.

5.1 Initialization module

This module will be used to precompute some intermediate matching to improve the running time of the algorithm. Different strategies for this task are discussed in [1].

Responsibilities of the module: find initial matching to start the execution of the algorithm with non-empty set of edges and non-zero dual variables.

5.1.1 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 $(1/2)min\{w_{v,u} : u \in N(v)\}$. Then we traverse the nodes again, greedily increase dual variables and choose matching edges, if it is possible.

5.1.2 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. As shown in [1], it is also much faster comparing to the Blossom algorithm.

5.2 Primal Updater

The module uses 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 of the module: 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 module should also correctly report these situations.

5.3 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 [11].

- 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 [7] and mentioned in [11] 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 use 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 [5].

Responsibilities of the module: manipulate the dual variables assigned to vertices and pseudonodes and increase the dual objective function.

5.4 Main module

Responsibilities of the module: controls the execution of the algorithm, calls the methods of primal and dual modules, handle notifications from these modules, 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.

5.4.1 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 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 $O(1)$.

6 My approach

I think reasonable approach would be to (i) start with creation of the working prototype, (ii) refine the structure of the implementation and ensure modularization in order to avoid rewriting the code more than it is needed, (iii) then begin to improve individual modules and implement advanced techniques to gradually achieve better performance, (iv) and finally prepare the basis for further improvements/changes/variations.

The first stage includes implementing data structures, greedy initialization, all operations of primal updater, single tree approach in dual updater and main module. The next step will be to implement fractional initialization and multiple tree variable δ approach. In my opinion, these parts are very important for the algorithm, because they substantially improve its running time.

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.

I also want to make my implementation more object-oriented and decouple individual modules where it makes sense. This includes making some algorithms or data structure reusable, which will also help in the development of the library in the future.

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 [11] 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.

I'll break the development of the project in following blocks:

1. Creating prototype $\rightarrow$ 2.5 weeks.
2. Ensuring good modularity/writing unit for the prototype $\rightarrow$ 1 week.
3. Adding additional functionality for achieving good running time $\rightarrow$ 2 weeks.
4. Creation of supporting elements, i.e. special a data structure or specific algorithms $\rightarrow$ 2 weeks.
5. Testing overall algorithm, writing benchmarks, and correcting code $\rightarrow$ 1 weeks.
6. Writing comprehensive documentation & adding support for future extensions $\rightarrow$ 1.5 weeks.
7. Unpredicted situations like some component turned out to be harder that it was assumed $\rightarrow$ 2 weeks.

It is worth noting that some tasks can be done in parallel, i.e. developing prototype and writing tests for it.

Another important note is that preparations play significant role in this project. So, I'm going to familiarize me with with the topic further and work through details of actual implementation before the beginning of coding period.

7 Personal background

Last summer I took online course Stanford CS 261: Optimization and Algorithmic Paradigms[16]. My theoretical background about combinatorial optimization and linear programming duality is based on it as well as on my university operation research course. Practical knowledge are based on CLRS and on implementation of the LPSolver myself [12]. After LPSolver my next step was to implement algorithms like unweighted Hungarian algorithm and push-relabel in order to compare their performance with the performance of the LPSolver in practice [6].

Also, I took other online courses like MIT 6.006: Introduction to Algorithms [13] and MIT 6.046: Design and Analyses of Algorithms [14] for general background in algorithms. This autumn I've completed the second part of the Java Programming course in Univer Infopulse (certificate upon request) and I'm now completing the third part.

I consider my participation as a lecturer in the Kiev Summer Math School 2017 as a relevant working experience. One of the main topics of my lectures was graph theory and the usage of the probabilistic analyses in it.

8 Benefits to the community

Minimum weight perfect matching is still an area of research. Implementations of the algorithm my project is focused on aren't very widespread. There exists only a couple of implementations of Blossom IV, Blossom V, or other fast variation of the initial Edmond's Blossom I, which are mainly written in C or C++. This open-source implementation, which will be written in Java, will be one of the first in the area. It will enable many programmers to use the fast algorithm for min. weight perfect matching in Java. I believe it will have its impact on the further development of this area.

In addition to its theoretical meaning, this project is also important for practical applications. In [15] the problem of drawing graphs on a mechanical plotter is discussed. The weights of the edges of a graph are Euclidean distances between the vertices. The goal is to minimize wasted plotter-pen movement. The problem is solved by finding perfect matching of minimum weight and computing Eulerian cycle after adding edges of the matching to the graph. The algorithm is also used to solve routing problems. The goal is to produce a schedule with minimum cost. Many variations of this problem are NP-hard and matching based heuristic is used to approximate the solution. This topic is described in [2].

As a result, mplementation of the fast algorithm for solving minimum weight perfect matching will bring opportunity to optimize real world processes.

9 Related works

- Blossom V is implemented in C++, the code is licensed for research purposes [5].
- Blossom IV is implemented in C [4].
- LEDA library has own implementation for minimum weight prefect matching in C++, it are available for college/university departments for research purposes or in terms of commercial use. It also has Java interface for use using this implementation.

10 Benchmarking

Benchmarking can be performed in a similar to [7, 11] 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 [5].

11 Miscellaneous

My University instruction ends on July 8, 2018. After that I have to pass the finals till July 27, 2018. I have one individual German lesson per week, but it will last also till mid-July. Besides that I don't have any other activities till the end of GSoC 2018.

References

[1] David Applegate and William J. Cook. "Solving Large-Scale Matching Problems". In: *Network Flows And Matching.* 1991.

[2] Michael Ball, Lawrence Bodin, and Robert B. Dial. "A Matching Based Heuristic for Scheduling Mass Transit Crews and Vehicles". In: *Transportation Science* (1983).

[3] Claude Berge. *Two theorems in graph theory.* 1957.

[4] *Blossom IV implementation.* URL: `https://www.math.uwaterloo.ca/~bico/blossom4/`.

[5] *Blossom V implementation.* URL: `http://pub.ist.ac.at/~vnk/software.html`.

[6] *Combinatorial Optimization.* URL: `https://goo.gl/UEZBqS`.

[7] William Cook and Andre Rohe. "Computing Minimum-Weight Perfect Matchings". In: *INFORMS Journal on Computing* (1999).

[8] Jack Edmonds. "Maximum mathing and a polyhedron with 0,1 - vertices". In: *Journal of Research of the National Bureau of Standards* (1965).

[9] Jack Edmonds. "Paths, trees, and flowers". In: *Classic papers in combinatorics* (1965).

[10] M.L. Fredman, R. Sedgewick, and D.D. Sleator. "The Pairing Heap: A New Form of Self-Adjusting Heap". In: *Algorithmica* (1986).

[11] Vladimir Kolmogorov. *Blossom V: A new implementation of a minimum cost perfect matching algorithm.* 2009.

[12] *LPSover.* URL: `https://goo.gl/g1e5DF`.

[13] *MIT 6.006.* URL: `https://goo.gl/79qG59`.

[14] *MIT 6.046.* URL: `https://goo.gl/6Kko5v`.

[15] Edward M. Reingold and Robert E. Tarjan. "On a Greedy Heuristic for Complete Matching". In: *SIAM Journal on Computing* (1978).

[16] *Stanford CS 261*. URL: `https://goo.gl/gcvy6H`.