CO759 Term Project Report

Cutting Plane Method for Matching and Its Application in Solving Uncapacitied b-Matching via Data Reduction

Tian QIAO

Department of Applied Mathematics, University of Waterloo

&

Yinuo LIU

Computational Mathematics, University of Waterloo

Department of Combinatorics and Optimization, University of Waterloo,

Waterloo, Ontario, Canada N2L 3G1

Abstract

The most famous algorithm for general matching is general blossom algorithm described by Pullybank in his Phd thesis [1]. However, this algorithm has many difficulties to implement, and it is not a polynomial algorithm in any case. In this report, we will present another algorithm to approach uncapacitied b-matching. This algorithm is strongly polynomial time for some special b vector and sparse graph. The algorithm has three steps: First, the algorithm modifies the original graph G to Gb. Second, it solves optimal 1-perfect matching in Gb and obtain a matching M' in Gb. Finally, it will correspond M' to an uncapacitied b-matching M in G, and M is the optimal uncapaci -tied b-matching in G. The first step and third step are called data reduction; the second step is called perfect matching, which will be solved via cutting plane method [2].

Cutting Plane for Perfect Matching [2]

Consider a Matching instance: Find a minimal perfect matching M in graph G=(V, E).
Let x_e to be the characteristic vector of matching for all edge $e \in E$. We have following linear programming:
(P)

$$\min c_e^T x_e$$
$$x\big(\delta(v)\big) = 1\,, \forall v \in V\ (each\ vertex\ can\ only\ be\ matched\ once)$$
$$x\big(\gamma(S)\big) \leq \frac{|S|-1}{2}, \forall S \subset V, |S| = odd\ (Blossom\ inequality)\ \ (*)$$
$$x_e \geq 0, \forall e \in E$$

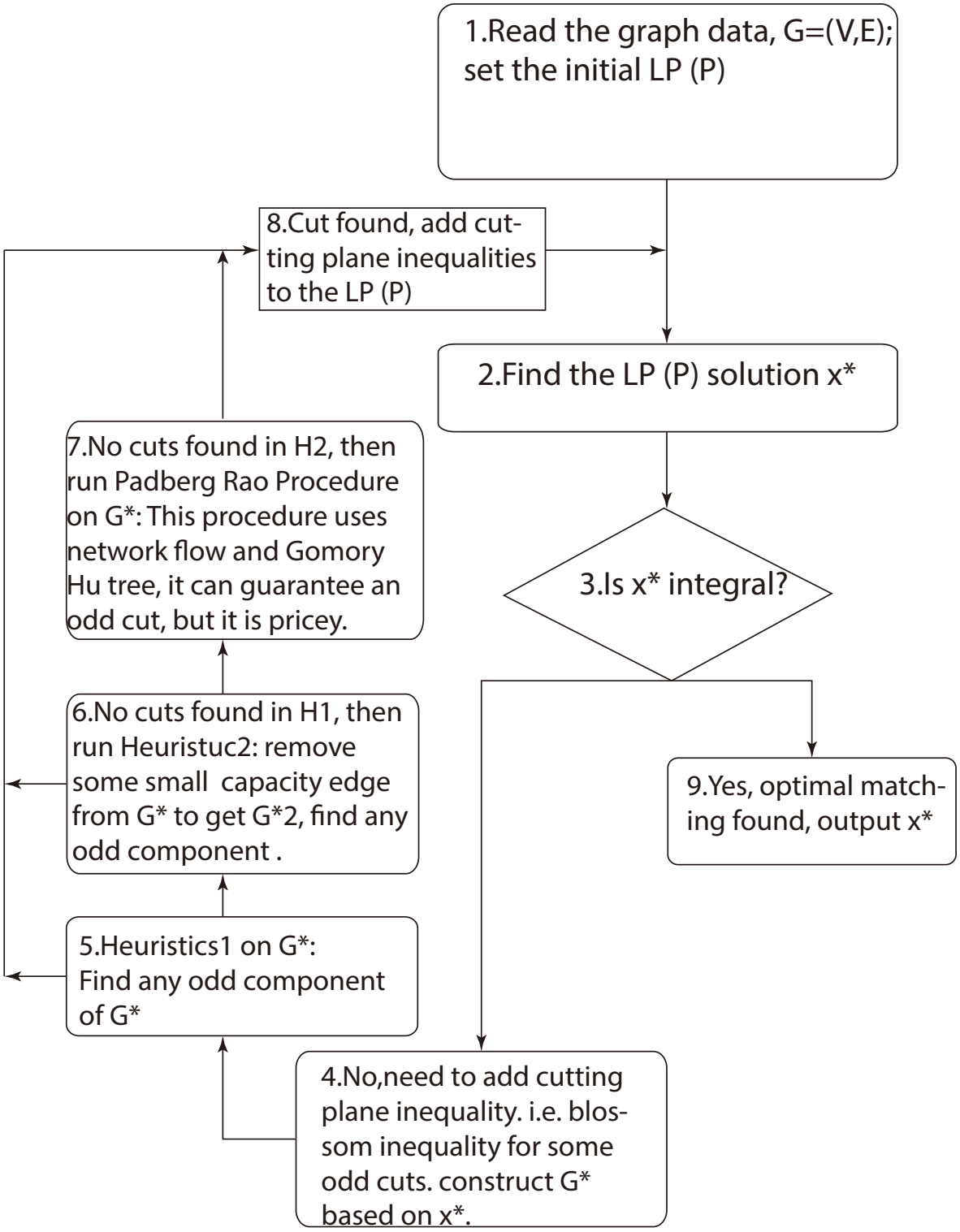
It has proven that the matching polytope is defined by the constraints in (P) [3]. That means each vertex of (P) is a matching. If we run simplex on (P), it will give us a matching which minimizes the constraint. However, this procedure is impossible in practical because blossom inequality $(*)$ is hold for every possible subset S contained in V, which means we have $2^{|V|-1}$many inequalities in $(*)$; it is almost impossible to store an exponential large data or solve such a big linear programming. Thus, we only need to add inequality $(*)$ when it is necessary. This method is called cutting plane method.

The cutting planes we want to add is basically blossom inequality $(*)$. All we need is a feasible solution for some initial (P), and find a subset S that violates $(*)$; then we add the blossom inequality for S to (P) and solve the new (P) until we have an integral solution for (P); the integral solution will be the optimal perfect matching. We can start with (P) without $(*)$. The initial LP (P):

$$\min c_e^T x_e$$
$$x\big(\delta(v)\big) = 1\,, \forall v \in V\ (each\ vertex\ can\ only\ be\ matched\ once)$$
$$x_e \geq 0, \forall e \in E$$

The flow chart for the algorithm [2] is described below. Following the chart we will describe in detail about the procedure to find a violated S.

The Cutting-Perfect-Matching(G):



We implement this algorithm in C++, solving linear programming by ILOG Cplex. Since C++ is an object oriented programming language, there will be a lot of class and objects in the code. They will all be explained while going through the process of the flow chart.

Steps 1,2,3,8,9 are typical steps. There are no algorithmic difficulties in these steps. We (almost Yinuo Liu) define a class called ***RelaxedLP*** (it can be found in *RelaxedLP.h* and *RelaxedLP.cc*) to do all of them. This class has several private variables to store the number of nodes and edges of the graph, the edge array for graph, the linear programming objects for the graph, the solution of the stored linear programming and the objective value. The class also has public functions to solve the LP, read problems from txt files to the ***RelaxedLP*** object (Also modify data is in this process) and add constraints to the LP. Basically, it can deal with everything in steps 1,2,3,8,9.

If our solution x* is non-integral, there will be some odd cuts S such that S violate the blossom inequality. Steps 4,5,6,7 are looking for these cuts S.

In Step 4, we need to construct the so-called LP graph G* based on G and LP solution x*. For an edge ,if $e \in E, x_e \neq 0$, we add e to G* with capacity $c_e = x_e, for\ e \in G^*$. Notices that since $x\big(\delta(v)\big) = 1\ , \forall v \in V$, every vertex is incident to at least one edge with nonzero LP solution . Thus the vertex of G* is same as G=(V, E). We can define G*=(V, E*, c). In our implement, we define a ***graph*** class (it can be found in *graph.h* and *graph.cc*) to construct G*(using function `construct_g_star()`) from the ***RelaxedLP*** class. Moreover, ***graph*** class will be used almost everywhere in the rest step.

In Step 5,which is the Heuristic 1, we first look for any "strong" components of G* that violate the blossom inequality. If there are some odd components in G*, the cuts yield by these components will definitely violate the blossom inequality. This step is $O(V)$.

If no odd components are found in Step 5, we move to another heuristic procedure called Heuristic 2. We first remove all edges in G* with LP solution less than 0.3, this yields a new graph G*2; then we do Step 5 on G*2. This step is $O(V)$.

The Step 7, which is the most cruel and time-consuming process, could guarantee at least one odd cut S that violates the blossom inequality. If no odd component is found in Step 5 and 6, we run Padberg-Rao's Procedure [6] on G*, This procedure output an minimum capacity odd cut S which will violate the blossom inequality. However, the procedure can output more than one odd cut that violates the inequality. In fact, any odd cuts in G* with total capacity less than one will violate the blossom inequality, and this procedure can return odd cuts in a sorted capacity order. We just following the order and output all odd cuts with capacity less than one.

In order to perform Padberg-Rao, we need to construct Gomory-Hu tree of G*(G-H tree). Let PATH(u,v,T) be the path of node u,v in tree T ,MAXFLOW(u,v,G*) be the value of maximum flow between u and v in G*; then GH(G*) is defined as following:

$$V\big(GH(G^*)\big) = V(G), \forall u, v \in V(G^*), if\ uv \in GH(G^*), then\ w_{uv} = MAXFLOW(u, v, G^*);$$
$$if\ uv \notin GH(G^*), then \min\{w_{ij} : ij \in PATH\big(u, v, GH(G^*)\big)\} = MAXFLOW(u, v, G^*);$$
$$let\ mn = \{mn \in GH(G^*) : w_{mn} = min\{w_{ij} : ij \in PATH\big(u, v, GH(G^*)\big)\},$$
$$then\ disconnect\ mn\ in\ GH(G^*) gives\ two\ components\ and\ the\ components\ gives\ a$$
$$minmum\ capacity\ u - v\ cut\ in\ G^*$$

When constructing G-H tree, we need use maximum flow as a subroutine. In our implementation,

we use the Ford-Fulkerson algorithm [5]; we create a class called ***FordFulkerson*** (it can be found in *FordFulkerson.h* and *FordFulkerson.cc*); in its constructor, it takes a ***graph*** object (using the function `FordFulkerson(const Graph&);`), and computes the maximum flow between two nodes (using the public function `max_flow(int, int);`); it can also store the minimum capacity cut when computing the maximum flow (the public function `get_nodeList();`can output the stored corresponding minimum capacity cut).

Once we can compute maximum flow and minimum capacity cuts, we create a ***GomoryHu*** class (it can be found in *GomoryHu.h* and *GomoryHu.cc)* to compute G-H tree. The constructor takes a ***graph*** class (`GomoryHu(const Graph&);`), and construct a ***FordFulkerson*** object based on the input ***graph*** class. The public function `construct_GH_tree()`constructs the tree by using the ***FordFulkerson*** object and a very simple algorithm[4]; it can return a node list with parents information of each node stored in it.

The last class, which performs the actual Padberg-Rao's Procedure, is ***PadbergRao*** class. The constructor `PadbergRao(Graph &)`takes a ***graph*** class, computes ***GomoryHu*** class based on the input and sorts the edges in the output of `construct_GH_tree()`in a increasing order of weights. In the public function `remove_edge()`, it removes the edge in the sorted G-H tree with weight less than one and uses depth first search to see the parity of the component yields by removing the edge; if it is an odd component we add its blossom inequality to our ***RelaxedLP***; otherwise, continue to search for other edges until we go through all edges with weight less than one.

As shown, this procedure need to compute maximum flow whose complexity is $O(VE^2)$, compute the G-H tree by using a simple but inefficient algorithm which needs $O(V^2)$ times of maximum flow [4], sort edge needs $O(VlogV)$ and depth-first search on G-H tree needs $O(V)$. The total estimated complexity of Step 7 is $O(VE^2)O(V^2) + O(VlogV) + O(V) = O(V^3E^2)$ which is prohibited for large scale problem. Thus, we run Step 5 and 6 before Step 7 to acquire cuts via cheap methods and try not to reach this step.

There is one improvement of Padberg-Rao in G*: We can remove all vertices that incident to an edge has value 1 in LP solution. This will give us a much smaller graph G*'. It is obvious that any minimum odd cut in G*' is same as G*, and vice versa. In our implementation, we use ***Bimap*** class to re-index G*' and correspond cuts in G*' back to G*.

Once we have integral solution x*, it is a matching and we go to Step 9: output x* and corresponding matched edges. We denote this algorithm on G by "cutting-perfect-matching(G)" , this is not an actual function in the code. It is just for convenience in explaining data reduction method.

Data reduction Method for Uncapacitied b-Matching [3]

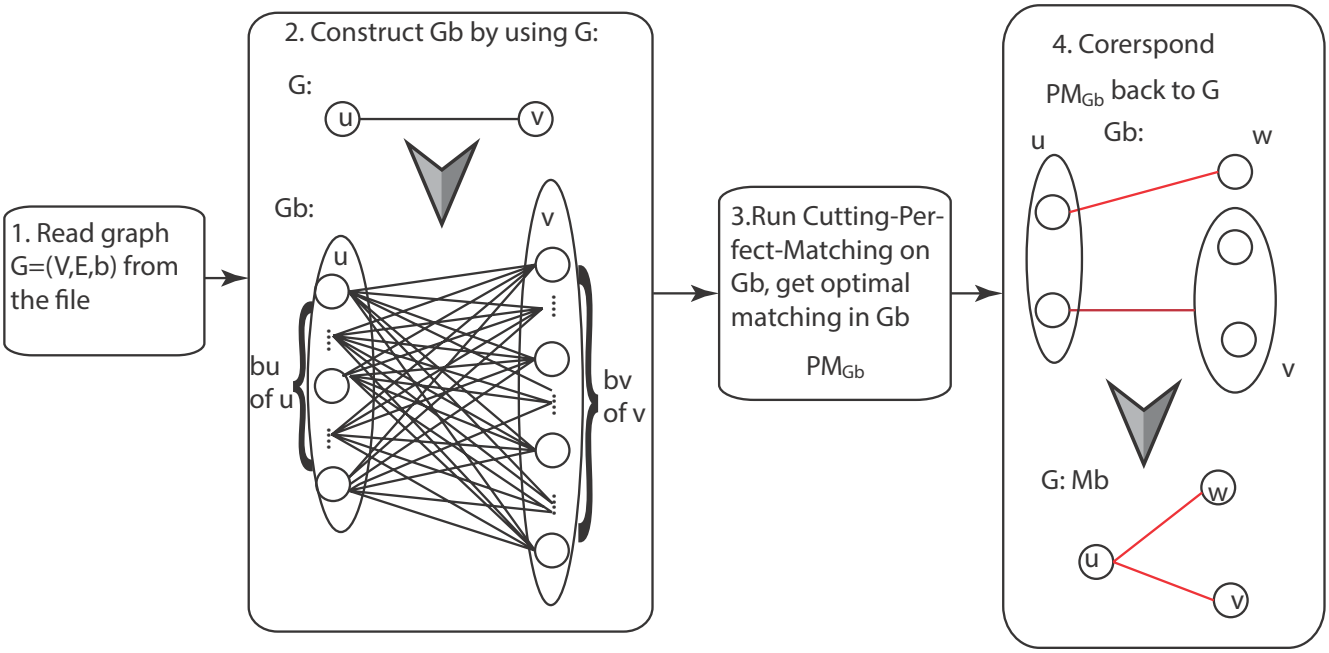
This method is described and proven in [3]. The b-matching is a matching M_b, for vector b with dim(b)=|V|:

$$\forall v \in V, |\mathrm{M_b}(v)| = b_v$$

If there is no restriction on the capacity of edges, the b-matching is called uncapacitied b – matching. In this report, without other specify, b-matching is believed as uncapacitied.

Notice that if $b = \mathbb{1}^{|V|}$, then the b-matching is the perfect matching.
The implement of this method can be found in *data_reduction.h* and *data_reduction.cc.* It takes a graph (not the ***graph*** class) G=(V, E, b) (the number of nodes and edges, the edge list and the b vector), and transforms it into a new graph $G_b = (V_b, E_b)$. It splits node v into b_v copies of v for all v in V and adds these copies of nodes to V_b, and then if uv is an edge in E, we add edges of all copies of u and v in E_b with same weight as uv. There is no doubt that this modification will increase the graph instance largely.

After we have G_b, we just need to run algorithm cutting-perfect-matching(G_b); the output will gives us a optimal perfect matching in G_b denoted PM_{G_b}. Then we corresponds PM_{G_b} back to M_b in G by finding the source of each node in M, i.e. if uv is in PM_{G_b}, then find the vertex u' in G such that u is one of the copies of u', and do the same thing for v to find v' in G; finally add edge u'v' in M_b. M_b is the optimal b-matching in G. The flow chart of the method is described below:



In *data_reduction.h* and *data_reduction.cpp,* we have a function `construct_b_graph(int , int , int* ,int*,int* , int &,int &)` will construct Gb and write it into a file called "b-graph.edg" ,and function `correspond_map(int, int*)` will return a map that maps the node index from b-graph to the original graph.

Usage, Test and Complexity

In our implement, we compile cutting plane for perfect matching (the compiled file is *perfect_matching*) and data reduction method (the compiled file is *data_reduction*) independently (use same makefile) because we want to test perfect matching individually on some large and dense graph (e.g. K_{1000}). For example, we want to compute b-matching of graph edge file *g.edg*, and b vector file *b*. Then we do the following:

1. Construct Gb: in command line, type "*./data_reduction g.edg b*", this will call *data_reduction* and create a new file called "*b-graph.edg*". This is the b-graph Gb.
2. Run cutting-perfect-matching on Gb: in command line, type "*./perfect_matching*

b-graph.edg". It will find perfect matching on "*b-graph.edg*",and output a file called "*matching.out*" which is the perfect matching in Gb ("*b-graph.edg*").

3. Corrsponding matching to b-matching: in command line, type "*./data_reduction g.edg b matching.out* ". It will correspond "*matching.out*" in "*b-graph.edg*" to a file "*b-matching.out*" which is the b-matching in *g.edg*

Notice that if *data_reduction* takes two arguments (order is: graph file, b vector), it will construct b-graph; if it takes three arguments (order is: graph file, b vector, matching file; do not put b-graph as the first argument), it will correspond the1-matching in Gb to b-matching in G.

In this case, we test perfect matching and data reduction individually.

Tests for perfect matching

The key part of the project is *perfect_matching*. Thus we compile it separately in order to test on adequate samples. We test the code on *g10000.18780.edg* from course website, random geometric K_{100}, random geometric K_{500} and 1000 of random geometric K_{1000}. Here are some results:

The running time for *g10000.18780.edg:*

```
v1020-wn-89-164:co7591 VeniceX$ ./perfect_matching g10000.18780.edg
the length of optimal matching is 315429
number of total iterations is 222
the running time is31.8643
```

Not too bad for a graph with 10000 nodes, but the graph is sparse.

The running time for random K_{100}:

```
v1020-wn-89-164:co7591 VeniceX$ ./perfect_matching -k100
Random 100 point set, gridsize = 100
Complete graph: 100 nodes, 4950 edges
the length of optimal matching is 349
number of total iterations is 11
the running time is0.011942
```

Too fast

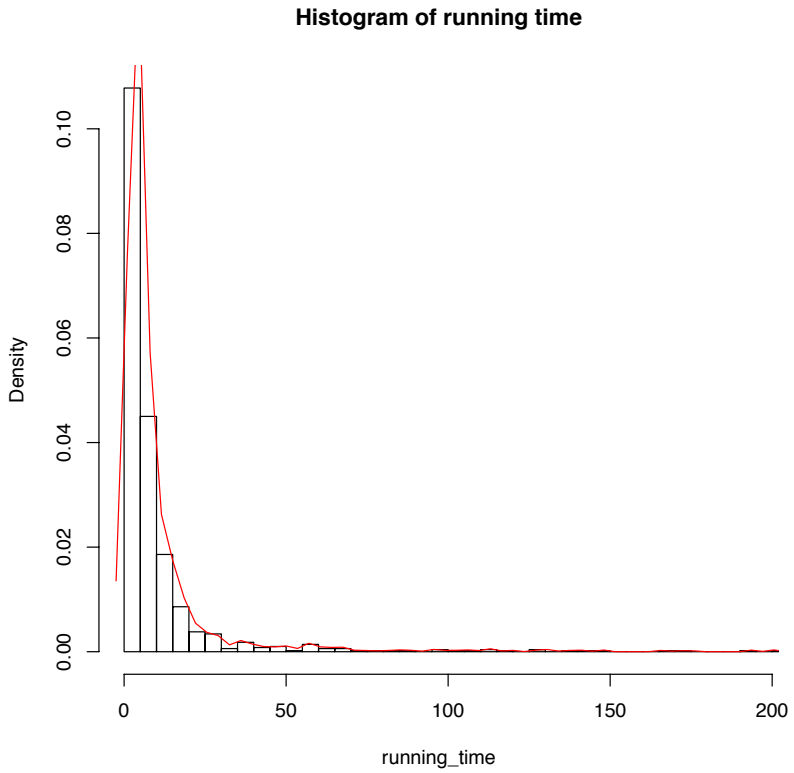
The running time for random K_{500}:

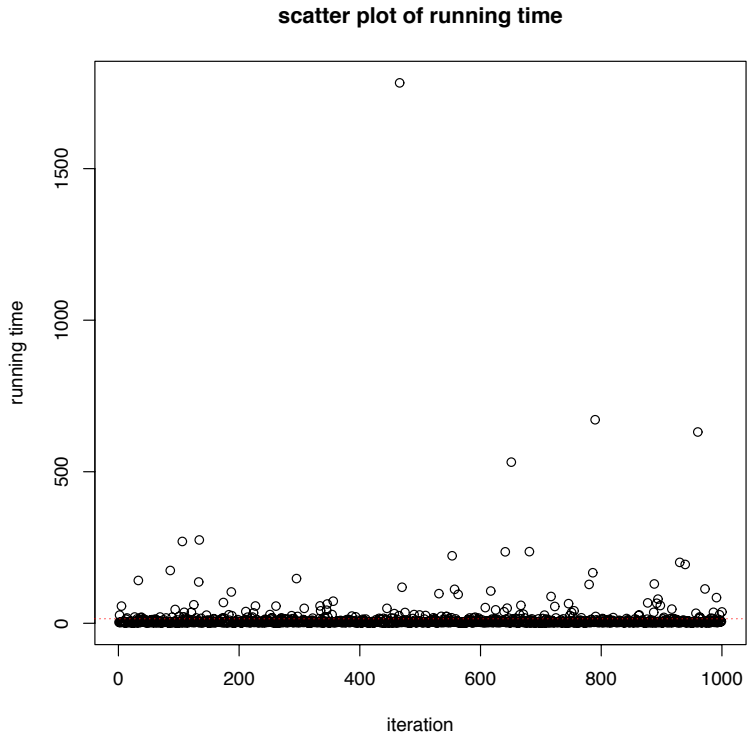
```
v1020-wn-89-164:co7591 VeniceX$ ./perfect_matching -k500
Random 500 point set, gridsize = 100
Complete graph: 500 nodes, 124750 edges
the length of optimal matching is 711
number of total iterations is 32
the running time is0.49118
```

Still fast.

Then we test on random K_{1000} graph; the running time differs a lot. We make a histogram and a scatter plot for the running time of 1000 instances of random K_{1000} graph.

The unit for running_time is seconds. The plot and summary is done by R.





Here is the summary of the plot from R:

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	SD
0.8579	2.7610	4.6790	14.9200	9.3440	1783.0000	69.92137

From the summary of the plot of running time of random K_{1000} graph, we see that the average running time is 14.92s; that means the algorithm works fine in K_{1000} in general. However, the peak running time is 1783s, it is infeasible in practical. Also the standard derivation is 67.92,

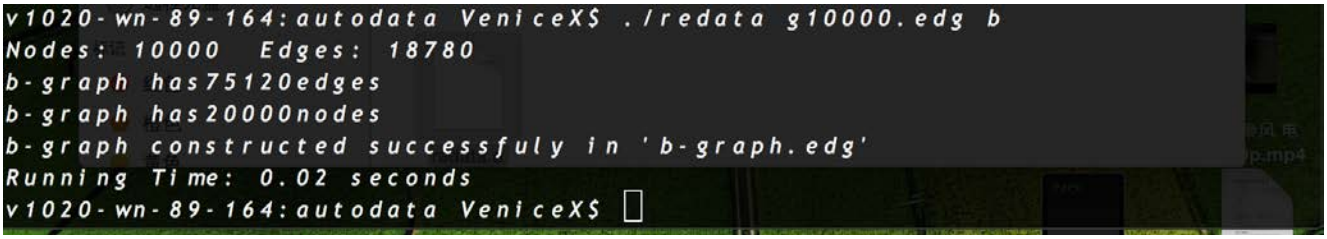
thus the algorithm is not stable, the running time depends on the graph and we do not know the dependence between them.

The death of heuristic

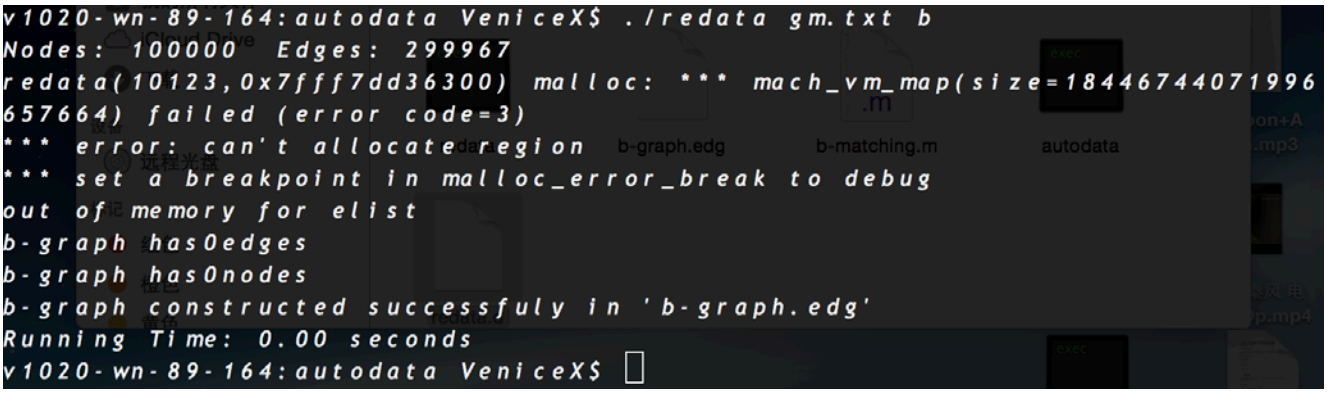
In our first version of perfect matching, we have no restriction on Heuristics (both 1 and 2). As there are components in G*, we will add them as cutting planes to our LP. However, when we run the program on some large problem i.e. *g10000.18780.edg* from course website. The program is keeping finding components in G* and edges in G* is always increasing; it seems to run heuristics forever and never go to Padberg-Rao, so it runs into a dead loop. We cannot find any coding or algorithm reasons for this phenomenon. A conjecture is that the graph has a lot of Combs. The program will keep searching components among these combs and adding useless cutting planes. In order to avoid this situation, the best way is to add comb inequalities as cutting planes; however, comb inequality is beyond our scope of this project; we just use a simple strategy: restriction on the iterations of heuristics; if heuristics cannot change LP object value within 10 iteration, we simply skip the heuristics and perform Padberg-Rao only.

Tests for data reduction

The data reduction is actually quite efficient. We test *data_reduction* on *g10000.18780.edg*, and it takes only 0.2 second to construct b-graph for b is the 2-vector (vector has value 2 in each coordinate). However, the construction process costs a lot of memory. When we test on *g100000.299967.edg,* it runs out of memory.(see pictures below)



2-graph of graph with 10000 vertices and 18780 edges only takes 0.02s to construct



Construction 2-graph for graph with 100000 vertices fails because of out of memory

Some Paradox

In our project, we encountered some paradox. They seem to be ridiculous at first time, but when we think deeper, they become more interesting.

1. Inefficiency of heuristic: For some problems, adding Heuristic 1 and 2 would not benefit the running time of perfect matching. For the problem instance *g10000.18780.edg,* if we restrict the iterations of Heuristic 1 and 2 to be all zeros i.e. no heuristic methods but only Padberg

-Rao to find cutting planes. Since Padberg-Rao is not very efficient i.e.$O(V^3E^2)$, theoretically this should make program runs slowly. However, the method can find a matching in 20 second. In comparison, if we allow heuristic methods, the running time would be around 60s~70s that means heuristic methods do not contribute to find cutting planes. This result is not strange since the problem instance *g10000.18780.edg* has lots of combs, which will make heuristic methods useless.

2. 2-matching is easier than 1-matching. Still for *g10000.18780.edg,* we compute the 2-graph for the problem; it has 20000 nodes and 75120 edges. If we try to run our perfect matching algorithm on this 2-graph, it should runs slowly because Padberg-Rao is slow for such large problem. However, the program can get optimal matching by using heuristic only, and the running time is only around 3 seconds. The cutting plane method is faster on a 2-graph, which is significantly larger. In fact, this is true in general case. Since the LP relaxation of 2-matching will allow x={0,0.5,1}, the blossom inequality is no longer a constraint. Therefore it is much easier to solve 2-matching which seems to be harder than 1-matching.

Complexity

It has proven that data reduction method for b-matching is strong polynomial time if vector b is bounded by some constant.[3] However, this method increases the size of problem instance which will cause storage problem.

Conclusion and Future

In conclusion, although this method for b-matching is polynomial time, the degree of polynomial is big and thus it is unsolvable for large problems. Data reduction increases the problem size for perfect matching and it requires lots of space. The cutting plane method for perfect matching is still not robust for large random problem i.e. we have no control on the running time for a certain size of problem. The algorithm is good for small, sparse graph and strictly bounded b-vector. It is infeasible for large, dense and arbitrary b-vector.

In future, we can use blossom algorithm for perfect matching, which is more stable than cutting plane method. We can also modify input graph further to add capacity u to each edge; this is called (b,u)-capacitated matching.[3]

Reference

[1]. W.R. Pulleyblank, Faces of Matching Polyhedra, PhD Thesis, University of Waterloo (1973).

[2]. Groetschel and Holland, Solving matching problems with linear programming, Mathematical Programming 33 (1985).

[3]. W.J. Cook, W.H. Cunningham, W.R. Pulleyblank and A. Schrijver, Combinatorial Optimization, Wiley-Interscience (1998).

[4]. Dan Gusfield. 1990. Very simple methods for all pairs network flow analysis. *SIAM J. Comput.* 19, 1 (February 1990), 143-155. DOI=10.1137/0219009

[5]. L.R. Ford AND D. R. Fulkerson, Flows in Networks, Princeton University Press, Princeton, NJ, 1962.

[6]. M. W. Padberg and M. R. Rao. Odd minimum cut-sets and b-matchings. Math. Oper. Res., 7(1):67–80, 1982.