On Implementing a solution for the Matching Problem over Complete Bipartite Graphs using Johnson-weighted Augmenting Paths

Garrenlus de Souza
Informatics Institute
Federal University Of Rio Grande do Sul - UFRGS
Porto Alegre, Brazil
gsouza@inf.ufrgs.br

I. Introduction

A. Johnson's technique

Johnson's approach on applying a weighting function over the original weights is key to the current implementation. Such approach can be summarized on the following equation:

$$w'_{u,v} = w_{u,v} - (p_v - p_u) \tag{1}$$

This is useful as it allows us to use Dijkstra's algorithm over graphs that originally have negative weighted arcs. Notice how this possibility is conditioned to the way the function p_v is defined as some values could render $w'_{u,v}$ negative, in turn ruling out the application of Dijkstra's approach on finding shortest paths. For starters we could start by defining the initial values for p_v. This value is $-max(w_{u,v})$ for every node in T. This guarantees that the smallest weight $w'_{u,v}$ under Equation 1 is 0 when the algorithm is being executed. Notice that there's no requirement on the minimum and maximum values for the weights, so it's perfectly reasonable that the minimum and maximum values differ in module $(\neg(|min(w_{u,v})| = |max(w_{u,v})|))$. That way we must guarantee that queries for the weight of unmatched arcs don't violate Equation 1. This can be assured by multiplying the original weight for -1 in such cases while returning the original value with no changes otherwise (the demonstration of such statement is left to the reader).

The next step would be to provide a suitable update method that keeps the properties of Johnson's technique on re-weighting. This consists of assigning to p_v the smallest distance between v and any free vertex u in S according to the original weights. This can be achieved by simply adding the re-weighted closest distance (to a free node in S, on the same premises) to the (correct) potential at hand [1].

II. Methodology

A. Environment

For the current study, the applied methodology consisted primarily into the analysis of running time statistics made over a reference implementation written in C++. The operating system is Ubuntu 20.04 LTS, running over a 32 GB RAM, 11th Gen Intel Core i7-1185G7 at 3.00GHz clock-speed system. All the code was compiled using g++ 9.4.0 (built for this very platform) through a CMake enabled build pipeline set to *"Release"* mode. The instances evaluated are available at [2] and the heap data-structure was previously tested and deemed correct in a previous assignment [2]. The timing fixtures are native to C++. All the data structures employed come from the *STL*[3] unless otherwise stated.

B. Algorithm

The essentials of the algorithm can be stated quite simply as seen in Algorithm 1. After initializing the potentials (to be used under Johnson's invariants), and properly defining all nodes as unmatched, the main loop begins. This main loop is kept running as long as there is a free node and consists of three main tasks: 1a) Finding out what is the closest path between a free node in S and a free node in T, 1b) Finding out the minimum distances between the free nodes in S and every vertex v, 2) Updating the current matching accordingly to this newly discovered path and 3) Updating the potentials for every node using the distances found in 1.

The $find\_augmenting\_path$ procedure is key on the complexity analysis as $update\_matching$ and $update\_potentials$ are bound to the longest non-looping path in the graph, which is bound to $O(n)$. The algorithm for $find\_augmenting\_path$ consists in an extended implementation of Dijkstra's algorithm. It extends the canonical form by checking two aspects when expanding a node: 1) The partition it belongs to and 2) its current mate. The actual algorithm can de described as in Algorithm 3.

The maximization variant of the problem differs by the stopping condition taking into consideration the net value that would result from updating the current matching given the last augmenting path found (Algorithm 2, step 3). This version of the algorithm introduces *net_score*, a procedure that calculates

Algorithm 1 Johnson's based Minimum Perfect Matching

```
Require: $S, T$ bipartite undirected graph $G = (V, A, W)$ such that $w_a$ denotes the weight of arc $a \in A$. Every vertex must be accessible from the other partition and $|S| = |T|$.
Ensure: The matching $m$ found is the minimum perfect matching possible
        - $m_v$, the mate of a node $v$
        - $d_v$, the min distance from a free node in $S$ to $v$
        - $l$, the shortest path between free nodes in $S$ and $T$
1: $p_u \leftarrow -max(w_a)$; $p_v \leftarrow 0$; $\forall u \in S$ and $\forall v \in T$
2: while $\neg matching\_is\_perfect(m)$ do
3:     $l, d, e \leftarrow find\_augmenting\_path(m)$
4:     $m = update\_matching(l, e, m)$
5:     $p = update\_potentials(d)$
6: end while
7: return f
```

the net score. This must be done taking into account what arcs are currently matched and the invariant at Equation 1.

Algorithm 2 Johnson's based Maximum Matching

```
Require: $S, T$ bipartite undirected graph $G = (V, A, W)$ such that $w_a$ denotes the weight of arc $a \in A$. Every vertex must be accessible from the other partition and $|S| = |T|$.
Ensure: The matching $m$ found is the maximum matching possible
        - $m_v$, the mate of a node $v$
        - $d_v$, the min distance from a free node in $S$ to $v$
        - $l$, the shortest path between free nodes in $S$ and $T$
1: $p_u \leftarrow -max(w_a)$; $p_v \leftarrow 0$; $\forall u \in S$ and $\forall v \in T$
2: $l, d, e \leftarrow find\_augmenting\_path(m)$
3: while $\neg(e = None)$ and $net\_score(e, l, m) > 0$ do
4:     $m = update\_matching(l, e, m)$
5:     $p = update\_potentials(d)$
6:     $l, d, e \leftarrow find\_augmenting\_path(m)$
7: end while
8: return f
```

Notice that the node expansion (steps 8-23 of Algorithm 2) proceeds differently depending on the partition of the node and its corresponding matching state at the moment. When the node u under expansion is in S we make sure to use a non-matched arc to get to the other partition (T in this case) so the path l is sure to preserve the alternating pattern (*free-matched-free*, *S-T-S*). Something analogous can be seen if $u \in T$.

When expanding a node in T we can maintain the alternating property of an augmenting path by adding only its mate to the exploration frontier. If there is no match we just update the minimization accordingly, doing nothing to the priority queue. At the end we have all that is needed in Algorithm 1, a predecessor function (that contains the alternating path of interest), the distance function and the last node of the alternating path (if there is one). The $decrease\_key$ procedure (Algorithm 3) is quite similar to Dijkstra's canonical form, added by the tracking of the predecessor and the use of Johnson's weighting function (Algorithm 4, step 5 and Equation 1).

Algorithm 3 Find Augmenting Path

```
Require: A valid matching $m$, a potential function $p$, the neighbourhood function $N$.
Ensure: $d_v$ is the minimal distance from the free nodes in $S$ to any node $v$ and $l$ is an augmenting path.
1: $d_u \leftarrow \infty \ \forall u \in V : u \in S$ or $m_u = None$ else $d_u \leftarrow 0$
2: Insert all the free nodes in S into a priority queue $pq$
3: $e_v \leftarrow false$
4: $closest\_free\_node = None$
5: while $\neg pq.empty()$ do
6:     $u \leftarrow pq.pop\_min()$
7:     $e_u \leftarrow true$
8:     $label \leftarrow get\_label(u)$
9:     if $label = S$ then
10:        for $v \in N(u)$ do
11:            if $e_v = false$ and $\neg(v = m_u)$ then
12:                $decrease\_key(u, v, d, l, p, m)$
13:            end if
14:        end for
15:    else
16:        if $m_u = None$ then
17:            if $\neg(closest\_free\_node = None)$ or $d_u < d_{closest\_free\_node}$ then
18:                $closest\_free\_node = u$
19:            end if
20:        else
21:            $decrease\_key(u, m_u, d, l, p, m)$
22:        end if
23:    end if
24: end while
25: return $l, d, e$
```

C. Analysis & Implementation

The implementation leveraged some inherent facts of the augmenting path-based approach and the fact that the graph is bipartite and complete. A linear buffer was the structure of choice when it came to storing the edge weights. While the graph is being read from standard input (as required for the assignment), the values were added to a linear structure, resembling some sort of deconstructed (or abstractly deconstructed) 2-dimensional container with just enough dynamically allocated space to hold all the necessary values. This was done on the impossibility of allocating a matrix on the fly under C++ (using STL), which in turn demanded that the authors themselves endured the task of guaranteeing a safe access to the arc weights.

Some of the vertex related information was aggregated into what became the *vertex struct*. It is interesting to note that the distance function was leveraged directly from the *heap* implementation [4] so no additional container was necessary. Another interesting detail is that we don't actually need to test if all the nodes are matched at every iteration of Algorithm 1.

Algorithm 4 Decrease Key

```
Require: A minimum based priority-queue $pq$, functions $l_v$ (predecessor function), $d_v$ (distance function), $p_v$ (potential function) and a matching $m_v$.
Ensure: $l_v$ and $d_v$ are updated appropriately, as well as the priority queue
 1: $w \leftarrow get\_weight\_johnson(u, v, m, p)$
 2: if $d_v = \infty$ then
 3:     $l_v \leftarrow u$
 4:     $d_v \leftarrow d_u + w$
 5:     $pq.insert(v)$
 6: else if $d_u + w < d_v$ then
 7:     $l_v \leftarrow u$
 8:     $d_v \leftarrow d_u + w$
 9:     $pq.update(v)$
10: end if
11: return f
```

Algorithm 5 Get Johnson weighted arc value

```
Require: Two nodes $u$ and $v$ ($u, v \in V$), a matching $m$, the original weight function $w$ and a potential $p$
1: $new\_weight \leftarrow w_{u,v}$
2: if $\neg(m_u = v)$ then
3:     $new\_weight = -new\_weight$
4: end if
5: $new\_weight = new\_weight - (p_v - p_u)$
6: return $new\_weight$
```

We can leverage the value returned by *find_augmenting_path* and check if e is valid, if not, then we are done as the latter did not go far enough as to iterate even once (no free nodes in S). This works thanks to the properties of the graph G added by the fact that every new augmenting path matches at least two more nodes, reducing the number of free nodes at both partitions evenly until there is none.

This fact sheds some light on the complexity of such algorithm as the main loop can be executed at most $n/2$ times for $n = |V|$, so it is bounded by $O(n)$. When looking at *find_augmenting_path*, we know it is basically a modified version of Dijkstra's algorithm. We also know that such modified version never extrapolates the number of times a node is explored and that it never walks over an arc *(u, v)* twice. This is enough to derive an upper bound as $O((m+n)log(n))$ for its execution, leaving us with an overall $O(n((m+n)log(n)))$ theoretical limit. We know that $m = (n/2)^2$ and for n as some value bounded upwards by 10^c, we can get to something like $O((n/2)^3)$, a function that for practical purposes ($n >= 2$) never overestimates the theoretical worst-case complexity.

The space complexity is bounded by the data-structure that stores the graph itself, in this case a linearized matrix ($n/2 \times n/2$). It holds exatcly $(n/2)^2$ edges, so $O(n^2)$. Of course there are other data-structures needed, such as a container for the matching itself, the distances, the augmenting path, and the potential function (implemented with $O(1)$ for all access operations), though all of them are linearly dependant on $|V| = n$, therefore $O(n)$, which is smaller in order when put next to the graph storage itself.

The same argument holds for the maximum matching with the addendum that it may actually stop way before n iterations over the main loop. This because it is totally reasonable that the maximum matching achievable is not perfect, which means that it does not include all the nodes. This concludes the theoretical discussion over the worst-case time-complexity bounds for the presented algorithm.

Copies were avoided when possible as to minimize memory overhead issues. As a result, arrays used to implement functions such as matching, potential and distance are passed around by reference or through the use of pointers (something that is leveraged by the min-heap implementation employed in the presented work). Another interesting fact is that the current implementation is non-destructive, which means that queries for maximum or minimum perfect matchings do not tamper on the arc weights, preserving the graph for future use whichever it may be (including new queries as well).

D. Input

For the empirical evaluation of the presented algorithm, some graph instances were generated. Each instance consists in a complete, bipartite graph with non-negative randomly selected weights as instructed by the professor. The values presented in Table I as the partition size would be exactly equal to $n/2$. This information is useful on understanding the difference that exists between the theoretical worst-case complexity and the numbers found in this analysis. The weights are selected uniformly from the range $[0, (n/2)^2)$. For validation purposes, the graph instances provided by the course instructor [5] were also evaluated, its runtime numbers are depicted in Figs 3, 4.

TABLE I
TEST INSTANCES

Instance	n	|S|	weight [0, X)
I1	20	10	100
I2	32	16	256
I3	40	20	400
I4	48	24	576
I5	64	32	1024
I6	96	48	2304
I7	100	50	2500
I8	128	64	4096
I9	192	96	9216
I10	200	100	10000
I11	256	128	16384
I12	384	192	36864
I13	400	200	40000
I14	512	256	65536
I15	1000	500	250000

III. Results

A. Runtime

In order to provide a better visualization, the instances that make out a power series (over r=2) are shown in Fig. 1. In order to integrate the visualizations, the red dot in Fig. 2 corresponds to the biggest value in Fig. 1. In order to more closely capture the growth pattern in terms of runtime, a fitting curve of third degree was introduced. The R^2 value for such curves is shown in both Fig. 1 and Fig. 2.

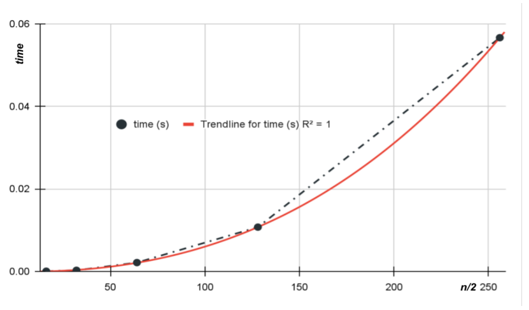

Fig. 1. Runtime as a function of partition size for the instances I2, I5, I8, I11, and I14.

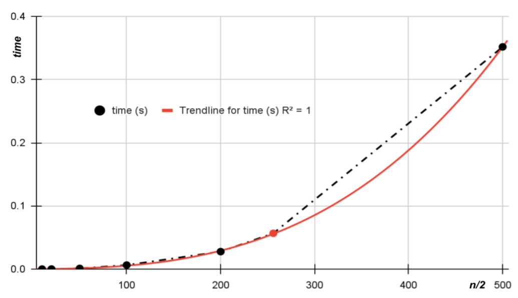

Fig. 2. Runtime as a function of partition size for the instances I1, I3, I7, I10, I13, I14, and I15.

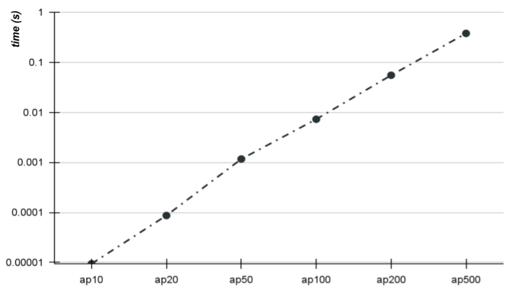

Fig. 3. Median runtime (grouped by n) for the graph instances provided by the course instructor.

The runtime for the reference instances was found to follow an identical growth patterns when compared to the randomly generated instances. The entire test generation source code is available at [2], a private Github repository. This includes random number generator seeds and setup scripts as well as the build configuration files. The statistics are available at [5].

B. Complexity

In order to objectively evaluate if the presented implementation actually was abiding to the theoretical worst-case time

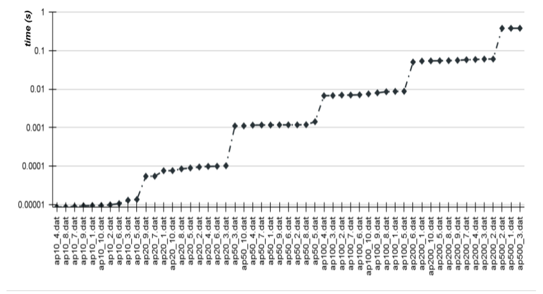

Fig. 4. Individual runtime for the reference graph instances.

complexity, the reference basic operation of choice was the *interchange*, performed by the employed heap implementation (to switch nodes during insertion, deletion or update as to re-establish the heap invariant). Therefore, in each execution the number of operations was accumulated from every call to *find_augmenting_path*. This numbers are depicted in Fig. 5. The proposed ceiling function is $(n/2)^3$, an expressive underestimation of the upper-bound proposed previously.

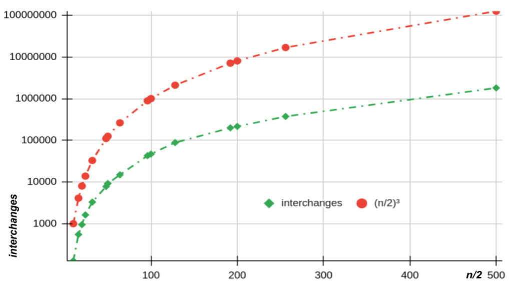

Fig. 5. The total number of interchanges performed by the heap throughout execution vs $(n/2)^3$.

IV. Conclusion

In summary, the runtime grew by an order that closely resembles $(n/2)^3$, though scaled down by a factor that ranged between $7.75\times$ and $70\times$ as can be seen in the last column of Table II (observed/expected interchanges).

TABLE II
RUNTIME & INTERCHANGES

Instance	|S|	time (s)	interchanges	%
I1	10	0.000015867	129	12.90%
I2	16	0.000055416	551	13.45%
I3	20	0.000105495	950	11.88%
I4	24	0.000149433	1621	11.73%
I5	32	0.000278943	3298	10.06%
I6	48	0.00100037	7794	7.05%
I7	50	0.001074388	9203	7.36%
I8	64	0.002145611	14873	5.67%
I9	96	0.004796853	42770	4.83%
I10	100	0.006360558	46883	4.69%
I11	128	0.010743722	88007	4.20%
I12	192	0.025755463	198993	2.81%
I13	200	0.027664877	215046	2.69%
I14	256	0.056680485	372902	2.22%
I15	500	0.351804242	1803092	1.44%

The time taken to compute the maximum matching grew monotonically as the graph instances (Fig. 1,2, 3 & 4) increased in size and was shown to oblige to the theoretical upper limit (time complexity).

Although the minimum perfect matching variant of the problem was not investigated empirically, the fact that the no instance used less than $n/2$ iterations should be enough to convince the reader about what could be expected to be observed had the authors taken the experiment to such extent. The entire source code is available at [2], including the test scripts and build configuration files.

REFERENCES

[1] "How to keep a potential using johnson's weighting," https://www.inf.ufrgs.br/~mrpritt/aa/Hungarian.pdf, Accessed: 2023-3-26.

[2] "Matching through augmenting paths and johnson's weighting," https://github.com/GarrenSouza/inf05016-matching, Accessed: 2023-3-26.

[3] "Containers," https://cplusplus.com/reference/stl/, Accessed: 2023-3-26.

[4] "Implementing dijkstra's algorithm," https://github.com/GarrenSouza/inf05016-dijkstra, Accessed: 2023-3-26.

[5] "Analysis data," https://docs.google.com/spreadsheets/d/1tmOfa0OYLyIeANx1VVZoEaeYFdfQt4vWP-PxLwre0Sg/edit?usp=sharing, Accessed: 2023-3-26.