The equivalence between two classic algorithms for the assignment problem

Carlos A. Alfaro [*]   Sergio L. Perez [†]   Carlos E. Valencia [‡]

Marcos C. Vargas [§]

October 9, 2018

Abstract

We give a detailed review of two algorithms that solve the minimization case of the assignment problem. The Bertsekas' auction algorithm and the Goldberg & Kennedy algorithm. We will show that these algorithms are equivalent in the sense that both perform equivalent steps in the same order. We also present experimental results comparing the performance of three algorithms for the assignment problem. They show the auction algorithm performs and scales better in practice than algorithms that are harder to implement but have better theoretical time complexity.

1 Introduction

The assignment problem can be stated as follows. We have a bipartite graph $G = (U \sqcup V, E)$, with $|U| = |V|$, and an *integer weight function* over the edges of G given by $w : E \to \mathbb{Z}$. The objective is to find a perfect matching of G of minimum weight. We call the pair $\{G, w\}$ an *instance of the assignment problem*, or equivalently an *integer weighted bipartite graph*. A perfect matching of minimum weight is called an *optimum matching*. If the graph has no perfect matchings, then the problem is *infeasible*.

The assignment problem is important from a theoretical point of view because it appears as a subproblem of a vast number of combinatorial optimization problems, and its solution allows the development of algorithms to solve other combinatorial optimization problems.

The roots of the assignment problem can be traced back to the 1920's studies on matching problems. And more remarkably by the 1935 marriage theorem of Philip Hall. The assignment problem can be modeled as a linear program,

---

[*]Banco de México, Mexico City, Mexico (carlos.alfaro@banxico.org.mx)

[†]Banco de México, Mexico City, Mexico (sergio.perez@banxico.org.mx)

[‡]Departamento de Matemáticas, CINVESTAV del IPN, Apartado postal 14-740, 07000 Mexico City, Mexico (cvalencia@math.cinvestav.edu.mx)

[§]Banco de México, Mexico City, Mexico (marcos.vargas@banxico.org.mx)

with the property that its associated polyhedron has all the vertices integer valued. Therefore, this problem can be solved using general linear programming techniques. The problem with these techniques is that they do not perform well in practice. This has pushed the development of specialized algorithms that exploit the particular structure of the assignment problem. For instance, Kuhn proposed in [15] the first polynomial time algorithm for solving the assignment problem. Since then, the assignment problem has been deeply studied, see for instance [16, 1, 6, 18, 15, 16, 14]. For a more detailed account of this fascinating topic, we refer the reader to [9]. The *ε-scaling auction algorithm* [5] and the *Goldberg & Kennedy algorithm* [13] also solve the assignment problem and for long time they have been considered as different algorithms, for instance see Section 4.1.3 of [9]. However, in this paper it will be shown that they are equivalent in the sense that one can be transformed into the other by means of memory optimization.

Through the paper, we will use the following notation: $m = |E|$, $n = |U| = |V|$, and $W = \max_{uv \in E} |w(uv)|$. The theoretical time complexity of these two algorithms is not the best currently known. The best current time complexity for the assignment problem is $O(\sqrt{n}\, m \log(nW))$ proposed by Gabow & Tarjan [10], while the time complexity for the ε-scaling auction algorithm is $O(n\, m \log(nW))$, as well as for the Goldberg & Kennedy algorithm. In our experience, the auction algorithm performs a much better in practice than other algorithms with equal or better theoretical time complexity, including the Goldberg & Kennedy algorithm.

The ε-scaling auction algorithm has been modified onto several variants to directly solve related problems such as *shortest path* [4], *min-cost flow* (see [3] p. 17, [5]), *assignment, transportation* [5] and many others. An important advantage of the auction algorithm is that it can be implemented using parallel computation [2]. It is important to remark that the auction algorithm only works with balanced instances, that is, $|U| = |V|$.

The ε-scaling auction algorithm operates like a real auction, where a set of persons U, compete for a set of objects V. In this scenario, to each object is assigned a price which, in certain sense, represents how important is the object for the persons. The optimization process is done in a competitive bidding, where the prices of the objects are properly reduced in order to make the desired object of a person less desirable to the other persons.

On the other side, the Goldberg & Kennedy algorithm is based on network flow techniques. The algorithm transforms the problem into a minimum-cost flow problem, and aims to build an optimum flow on a couple of auxiliary digraphs. Once we get an optimum flow, the induced optimum matching is easily obtained.

2 The ε-scaling Auction algorithm

In the ε-scaling auction algorithm every object v has a price $p(v)$. This set of prices can be seen as a price function over V defined by $p : V \to \mathbb{R}$. For every

edge $uv \in E$ we define the *reduced cost* $w'(uv)$ of job v for the person u as $w(uv) - p(v)$.

The objective in the auction algorithm is to find prices $p(v)$ for the jobs and a perfect matching M such that every person is assigned to the job that has almost the minimum reduced cost that the person can get. This condition is expressed in the following definition.

Definition 1. Given $\epsilon > 0$. A set of prices p and a perfect matching M are said to satisfy the ϵ-*Complementary Slackness condition*, or ϵ-*CS condition* for short, if they satisfy:

$$w'(uv) \leq \min_{z \in N(u)} w'(uz) + \epsilon, \qquad \forall uv \in M.$$

The following theorem shows the reason of why this condition is important.

Theorem 2. *Let M^* be a perfect matching of minimum weight and $\epsilon > 0$. If a perfect matching M satisfies the ϵ-CS condition with a set of prices p, then $w(M^*) \leq w(M) \leq w(M^*) + n\epsilon$.*

Proof. Part $w(M^*) \leq w(M)$ follows because M^* is optimum. In the other hand, from the definition of ϵ-CS condition follows that:

$$w(M) = \sum_{uv \in M} w(uv) = \sum_{uv \in M} w(uv) - \sum_{v \in V} p(v) + \sum_{v \in V} p(v) = \sum_{uv \in M} (w(uv) - p(v)) + \sum_{v \in V} p(v)$$

$$\leq \sum_{uv \in M^*} (w(uv) - p(v) + \epsilon) + \sum_{v \in V} p(v) = \sum_{uv \in M^*} w(uv) + \sum_{uv \in M^*} \epsilon = w(M^*) + n\epsilon.$$

$\square$

Since we are dealing with integer-weighted bipartite graphs, the following corollary shows how to obtain an optimum matching via the ϵ-CS condition.

Corollary 3. *If a perfect matching M and a set of prices p satisfy the ϵ-CS condition for $\epsilon < \frac{1}{n}$, then M is of minimum weight.*

Proof. Let $w(M^*)$ be the optimum weight. Theorem 2 implies that $w(M^*) \leq w(M) < w(M^*) + 1$. Since the weights are integral, then $w(M) = w(M^*)$. $\square$

The pseudo-code given in Algorithm 1 shows how to obtain a perfect matching and a set of prices that satisfy the ϵ-CS condition for a given $\epsilon > 0$. If the given instance has perfect matchings, then the procedure always terminates with the correct output as we will see later. It is important to remark that if the instance has no perfect matchings then the procedure will fall into an infinite loop. This algorithm is called the *auction algorithm*.

Algorithm 1: The auction algorithm

```
1       input: an instance {G, w}, ε > 0 and prices p.
2       output: a perfect matching M and prices p satisfying
            the ε−CS condition.
3
4       procedure auction (G, w, ε, p)
5           M = φ
6           while |M| < n do
7               take an unassigned u ∈ U
8               bid(u)
9           end
10          return (M, p)
11      end
12
13      procedure bid(u)
14          Let uv and uz be the edges with the smallest and
            the second smallest reduced costs, respectively.
15          if (v is assigned to some u' ∈ U)
16              remove u'v from M
17          append uv to M
18          γ = w'(uz) − w'(uv)
19          p(v) = p(v) − γ − ε
20      end
```

Note that we allow the procedure to receive initial prices p. Such initial prices have no particular restrictions, they can be any real values. The algorithm will automatically adjust them after each iteration and will end up with the correct output as the following proposition shows, which is proven in [5]. However, as we will see later, the initial prices have a big impact on the running time of the procedure.

Proposition 4. *If the auction procedure, described in Algorithm 1, is applied to a feasible instance of the assignment problem, then the procedure will terminate after a finite number of iterations and will return a matching and a set of prices that satisfy the ε-CS condition, regardless of the initial prices.*

It turns out that if the initial prices p are random and ϵ is very close to 0, then the resulting matching will be optimum or close to optimum, but the running time will be huge, and if ϵ is large then the running time will be small but the matching will be far from being optimum. However, if the initial prices have a structure close to the ϵ-CS condition, then the running time is proven to be $O(nm)$ [5]. The ϵ-scaling auction algorithm exploits this behavior to efficiently find a perfect matching M that satisfies the ϵ-CS condition for our small $\epsilon < \frac{1}{n}$. The pseudo-code is given in Algorithm 2, and makes use of the *auction* procedure. The scaling factor $\alpha > 1$ is a custom parameter and remains constant during the execution.

Algorithm 2: min_epsilon_scaling_auction

4

```
1    input: a weighted bipartite graph {G, w}.
2    output: a minimum weight perfect matching.
3
4    procedure: ε_scaling_auction(G, w)
5        ε = W
6        p(v) = 0,  ∀v ∈ V
7        while ε ≥ 1/n do
8            ε = ε/α
9            (M, p) = auction(G, w, ε, p)
10       end
11       return M
12   end
```

The idea of the ϵ-scaling auction algorithm is to iteratively find a sequence of pairs $\{M, p\}$ that satisfy the ϵ-CS condition, where the sequence of values for ϵ is $\{W/\alpha, W/\alpha^2,$
$W/\alpha^3, \ldots, W/\alpha^k\}$, with $W/\alpha^k < 1/n$. Note that the resulting prices that satisfy the ϵ-CS condition for $\epsilon = W/\alpha^i$ almost satisfy the ϵ-CS condition for the next iteration with $\epsilon = W/\alpha^{(i+1)}$.

As we mentioned before, each call to the auction procedure takes $O(nm)$ time. And the number of scaling phases is $O(\log(nW))$. This gives us a total of $O(nm\log(nW))$ running time for the ϵ-scaling auction algorithm. The following corollary follows directly from Proposition 4 and Corollary 3.

Corollary 5. *If the ϵ-scaling auction procedure is applied to a feasible instance of the assignment problem, then the procedure will terminate after a finite number of steps and will return an optimum matching.*

3   Goldberg & Kennedy algorithm

The Goldberg & Kennedy algorithm [13] applies network flow techniques to the assignment problem, which is a special case the minimum-cost flow problem. Their algorithm is base on the push-relabel technique [11]. The complexity of the resulting algorithm is $O(n\,m\,\log(nW))$ and an overview is presented following.

Given a weighted bipartite graph $\{G = (U \sqcup V, E), w : E \to \mathbb{Z}\}$, it is transformed into an instance of the min-cost flow problem $\{\hat{G} = (U \sqcup V, \hat{E}), w, c, d\}$. First, $\hat{G}$ is considered as a directed graph with the same vertex set $U \sqcup V$ and the arc set $\hat{E}$ equal to the edges of E but oriented from U to V. We denote the arc that goes from u to v by $\overrightarrow{uv}$. The *capacity function* is given by $c(\overrightarrow{uv}) = 1$, $\forall \overrightarrow{uv} \in \hat{E}$. And the *supply function* is given by $d(u) = 1 \; \forall u \in U$, and $d(v) = -1$ $\forall v \in V$.

A *pseudoflow* is a function $f : \hat{E} \to \mathbb{Z}$ such that $f(\overrightarrow{uv}) \leq c(\overrightarrow{uv})$ for every arc $\overrightarrow{uv} \in \hat{E}$. We define the *excess flow* of a vertex $x \in U \sqcup V$ by:

$$e_f(x) = d(x) + \sum_{\overrightarrow{yx} \in \hat{E}} f(\overrightarrow{yx}) - \sum_{\overrightarrow{xy} \in E} f(\overrightarrow{xy}). \tag{1}$$

A node v satisfying $e_f(v) > 0$ is called *active*. A *flow* is a pseudoflow with no active nodes. The weight of a pseudoflow is:

$$w(f) = \sum_{\overrightarrow{uv} \in \hat{E}} w(\overrightarrow{uv}) \cdot f(\overrightarrow{uv}). \tag{2}$$

The objective is to find a flow of minimum weight.

Given a pseudoflow, the *residual capacity* of an arc $\overrightarrow{uv} \in \hat{E}$ is $c_f(\overrightarrow{uv}) = c(\overrightarrow{uv}) - f(\overrightarrow{uv})$. The *residual graph* induced by the flow is $G_f = (U \sqcup V, E_f)$, where the set of *residual arcs* E_f contains the arcs that satisfy $c_f(\overrightarrow{uv}) > 0$ and the reversed arcs $\overrightarrow{vu}$ such that $c_f(\overrightarrow{uv}) = 0$. The weight function w_f defined over the residual arcs is $w_f(\overrightarrow{uv}) = w(\overrightarrow{uv})$ for forward arcs, and $w_f(\overrightarrow{vu}) = -w(\overrightarrow{uv})$ for reversed arcs.

Let $p : U \sqcup V \to \mathbb{R}$ be a function that assigns prices to the vertices. The *reduced cost* of an arc $\overrightarrow{xy} \in E_f$ is given by $w_p(\overrightarrow{xy}) = w_f(\overrightarrow{xy}) + p(x) - p(y)$. The *partial reduced cost* of an arc $\overrightarrow{uv} \in \hat{E}$ is $w'_p(\overrightarrow{uv}) = w(\overrightarrow{uv}) - p(v)$.

Given $\epsilon > 0$, a pseudoflow f is said to be ϵ-*optimal*, with respect to the price function p, if every arc of E_f satisfies the following:

For a reversed arc $\overrightarrow{vu}$:	$w_p(\overrightarrow{vu}) \geq -\epsilon,$	(3a)
For a forward arc $\overrightarrow{uv}$:	$w_p(\overrightarrow{uv}) \geq 0.$	(3b)

Theorem 6. *Let $\epsilon < 1/n$. If f is ϵ-optimal with a price function p, then it is of minimum weight.*

The proof of this theorem is given in [12] and [13]. Note that given a pseudoflow, a matching is induced by the arcs that carry one unit of flow. The full pseudocode of the algorithm is shown in Algorithm 3. The real number $\alpha > 1$ is a custom parameter and remains constant during the execution of the algorithm. During the algorithm, we assume that we construct and keep track of the directed graphs $\hat{G}$ and G_f. To keep track of the matching induced by the resulting flow, we use the variable M, where $M(v) = u$ if and only if $f(\overrightarrow{uv}) = 1$.

Algorithm 3: Goldberg & Kennedy algorithm

```
1     Input: a weighted bipartite graph {G, w}.
2     Output: a matching of minimum weight.
3
4     procedure Goldberg_Kennedy(G, w)
5         ε = W
6         p(v) = 0, ∀v ∈ V
7         while ε ≥ 1/n do
8             ε = ε/α
9             (f, p) = refine(G, w, ε, p)
10        end
11        return M (Matching induced by f)
12    end
13
14    procedure refine(G, w, ε, p)
```

```
15          f(\vec{uv}) = 0 , \ \forall \vec{uv} \in \hat{E}
16          p(u) = -\min_{\vec{uz} \in \hat{E}} \ w'_p(\vec{uz}) , \ \forall u \in U
17          while f is not a flow do
18              take an active u \in U
19              double_push(u)
20          end
21          return (f,p)
22      end
23
24      procedure double_push(u)
25          let \vec{uv} and \vec{uz} be the arcs with the smallest and
26           second smallest partial reduced costs,
            respectively
27          p(u) = -w'_p(\vec{uz})
28          send one unit of flow from u to v
29          if (e_f(v) > 0)
30              send one unit of flow from v to its match M(v)
31          M(v) = u
32          p(v) = p(u) + w(\vec{uv}) - \epsilon
33      end
```

The following theorem, which proof can be found in [13] and [12], states the correctness of the algorithm.

Theorem 7. *If G is feasible and balanced, then the Goldberg_Kennedy procedure will finish and will return a matching of minimum weight.*

4 Equivalence of the ϵ-scaling auction and the Goldberg & Kennedy algorithms

It is worth mentioning that the pseudocode of the previous algorithms were presented a bit different from the original versions, while maintaining the same flow of operations. The objective in doing so is to make easier, to the reader, to observe the similitude between the algorithms.

The central idea to see the equivalence is to make changes to the G&K algorithm that turn it into the ϵ-scaling auction algorithm. The changes are focused in getting rid of the auxiliary directed graphs used in the G&K algorithm, which turn out to be redundant, inducing more expensive computations and an unnecessary increase of the memory space, as well as hard to follow.

We are going to go through the Goldberg & Kennedy algorithm as described in Algorithm 3. First note that *reduced cost* from the auction algorithm and *partial reduced cost* from Goldberg & Kennedy are the same thing. Given an underlying edge uv, both are defined as $w(uv) - p(v)$.

Let us go through the *refine* procedure line-by-line to show the equivalence with the *auction* procedure described in Algorithm 1. At first, the flow vanishes, which is the same as setting the induced matching empty. The second line only initializes the prices of the set U which, as we will see later, are redundant

and a waste of computation, we can safely remove this initialization. The while condition is the same as testing whether the induced matching has less than n edges. And finally, taking an active vertex is the same as taking a vertex which is not assigned under the induced matching. Therefore, we can rewrite the refine procedure as in Algorithm 4.

Algorithm 4: Modified refine procedure

```
1    procedure refine (G, w, ε, p)
2        M(v) = φ,  ∀v ∈ V;
3        while |M| < n do
4            take an unassigned  u ∈ U;
5            double_push(u);
6        end
7        return (M, p);
8    end
```

Continuing with the *double_push* procedure of Algorithm 3, the first instruction is clearly the same as the first instruction of the *bid* procedure of Algorithm 1. For the second instruction, note that the previous value of the price of u is completely ignored, because it is replaced by the minimum partial reduced cost given by its neighbours. This observation proves that the initialization discussed in the previous paragraph, inside the original refine procedure, is indeed redundant. Back to the *double_push* procedure, note that the only place where the price of u is used is in the last instruction, therefore we can directly replace the computed price in this last instruction and forget about the second instruction. There are no more places in the pseudocode where the prices of U show up, this shows that the prices for the vertices of U are not needed. Once we substitute the computed price of the second line into the last line, the last instruction is equivalent to $p(v) = p(v) - \gamma - \epsilon$, where $\gamma = w'_p(uz) - w'_p(uv)$, as the following chain shows:

$$p(v) = -w'_p(uz) + w(uv) - \epsilon = p(v) - w'_p(uz) + w(uv) - p(v) - \epsilon = p(v) - (w'_p(uz) - w'_p(uv)) - \epsilon$$

The instruction where we send one unit of flow from u to v and the instruction $M(v) = u$ in the double_push procedure are equivalent, but since we want to get rid of the flow, we reject the first one and just conserve the second one. Finally, the condition in the *if* instruction is equivalent to test if the vertex v is already assigned under the induced matching, and sending one unit of flow from v to $M(v)$ is equivalent to remove the edge $M(v)v$ from the matching. Then the double_push procedure can be rewritten in the equivalent form shown in Algorithm 5.

Algorithm 5: Modified double_push procedure

```
1    procedure double_push (u)
2        let uv and uz be the arcs with the smallest and
3            second smallest partial reduced costs,
            respectively;
```

```
4              if ( v  is  assigned  to  some  u' ∈ U )
5                  M(v) = φ ;
6              M(v) = u ;
7              γ = w'_p(uz) − w'_p(uv) ;
8              p(v) = p(v) − γ − ε ;
9          end
```

As we can see, we have removed the need of keeping the matching in two equivalent structures, the flow and the matching. Therefore we can also replace the flow by the matching in the Goldberg_Kennedy procedure. If we compare side by side this equivalent version of the Goldberg & Kennedy algorithm with the ϵ-scaling auction algorithm, we will notice they are the same.

One final thing to do is to make sure that the optimality condition of the Goldberg & Kennedy algorithm is equivalent to the optimality condition of the ϵ-scaling auction algorithm. Since the optimality condition makes use of the prices of the vertices of U, which were removed in the new versions, we will focus on the original pseudocode of the G&K algorithm.

If we keep track of the prices of the vertices of U in the original pseudocode, it is not hard to find that at the end of every iteration in the loop, the price of any vertex $u \in U$ is equal to $p(u) = -min_{uz \in \hat{E}} w'_p(uz)$. This is clear in the initialization, but at the beginning of the double_push procedure the price $p(u)$ is modified such that it is assigned the negative second minimum partial reduced cost for u, but at the end of the procedure the price $p(v)$ is also changed in such a way that the second minimum is now the minimum partial reduced cost. Therefore, the original optimality condition (3) is equivalent to the following:

1. For a reversed arc $\overrightarrow{vu}$ (i.e. for every uv in the matching):

$$
\begin{aligned}
w_p(\overrightarrow{vu}) &\geq -\epsilon & \Longleftrightarrow \\
w_f(\overrightarrow{vu}) + p(v) - p(u) &\geq -\epsilon & \Longleftrightarrow \\
-(w(\overrightarrow{uv}) - p(v)) &\geq p(u) - \epsilon & \Longleftrightarrow \\
w(\overrightarrow{uv}) - p(v) &\leq -p(u) + \epsilon & \Longleftrightarrow \\
w'_p(uv) &\leq min_{uz \in \hat{E}} w'_p(uz) + \epsilon,
\end{aligned}
\tag{4}
$$

2. The second condition is redundant as we can see following, proceeding similar to the previous case. For a forward arc $\overrightarrow{uv}$ (i.e. for every other edge uv):

$$
w_p(\overrightarrow{uv}) \geq 0 \qquad \Longleftrightarrow \qquad w'_p(uv) \geq min_{uz \in \hat{E}} w'_p(uz)
$$

Which is the same as the optimality condition of the ϵ-scaling auction algorithm.

5 Performance analysis

This section is dedicated to present experimental results about the performance of three different algorithms: the *ϵ-scaling auction algorithm* with theoretical

time complexity of $O(nm \log(nW))$, the *Hungarian algorithm* [15] with time complexity of $O(mn + n^2 \log n)$ and the *FlowAssign algorithm* [20] with time complexity of $O(m\sqrt{s} \log(sW))$. Where $\{G = (U \sqcup V, E), w\}$ is a weighted bipartite graph not necessarily balanced, with $n = |U| \geq |V| = s$. The objective is to show that a very easy-to-understand and easy-to-implement algorithm like the auction algorithm performs and scales a lot better in practice than algorithms that are harder to implement and, as in the case of the FlowAssign algorithm, have better theoretical time complexity. The reason we do not include the Goldberg & Kennedy algorithm in this analysis is that we have just shown that this algorithm performs redundant computations and memory wastes, therefore it is expected a worst performance respect to the auction algorithm. Furthermore, we have also proven that if we optimize the implementation of the G&K algorithm we will end up implementing the auction algorithm.

To compare the performance of the algorithms we will construct different types of random instances with different structures. This is motivated by the fact that each algorithm may perform better on some graph structures than others and we want to explore how the algorithms behave to different graph configurations.

An instance of the assignment problem has two components, the bipartite graph and the edge weights. We will define constructions for each component separately since we can pick one of each to form an instance. Given a bipartite graph G, we define its *density* as $\rho(G) = \dfrac{|E|}{n \cdot s} \in [0, 1]$. Note that in a complete bipartite graph $K_{n,s}$ we have $\rho(K_{n,s}) = 1$.

5.1   Models for generating random instances

The first model for generating random bipartite graphs is the *Erdös-Renyi* model. Here, every possible edge of the bipartite graph has probability d to stay in the graph, under a Bernoulli distribution. The pseudocode to generate this type of random bipartite graphs is given in the following procedure, which takes as argument: the number of vertices in U, the number of vertices in V and the target density d.

Algorithm 6: Erdös-Renyi model for generating random bipartite graphs

```
1       procedure erdos_renyi(n, s, d)
2           U = {u_1, ..., u_n},  V = {v_1, ..., v_n},  E = φ;
3           for each u ∈ U, v ∈ V:
4               if (getBernoulli(d)==1) add edge uv to E;
5           return G = (U ⊔ V, E);
6       end
```

The function *getBernoulli(d)* returns 1 with probability d and 0 with probability $(1 - d)$. It is not difficult to see that in the resulting graph $\rho(G) \approx d$. Observe that in this model, the degrees of the vertices satisfy that $deg(u) \approx d \cdot s$ for $u \in U$ and $deg(v) \approx d \cdot n$ for $v \in V$. In other words, we get almost null variability in the distribution of the degrees in both sides.

The second model is the *dispersed-degree* model. It is designed to make the distribution of the degrees in the side U more variable while preserving the target density d. Basically, we define the degree of each vertex $u \in U$ as an integral uniform random number in the interval $[d \cdot s - r, d \cdot s + r]$, where r is a custom *dispersion radius*. Note that $r \leq s \cdot \min(d, 1 - d)$, to keep the degrees in the valid range $[0, s]$. Since different densities induce different upper bounds for the dispersion radius, then we define the *normalized dispersion radius* as the quotient of the dispersion radius by the upper bound. Thus the normalized dispersion radius is always in the range $[0, 1]$. Therefore, since the interval is centered at $d \cdot s$, then $\rho(G) \approx d$. Once we have defined the degree $deg(u)$ of a vertex $u \in U$, its neighbors are a random subset of V of size $deg(u)$. The following pseudocode shows how to generate this class of graphs. It takes as input: the number of vertices in U, the number of vertices in V, the desired density d, and the custom radius of dispersion $0 \leq r \leq s \cdot \min(d, 1 - d)$.

Algorithm 7: Dispersed-degree model for random bipartite graphs
---

```
1      procedure dispersed_degree(n, s, d, r)
2          U = {u_1, ..., u_n},  V = {v_1, ..., v_n},  E = φ;
3          for each u ∈ U define deg(u) := rand(d · s − r, d · s + r);
4          for each u ∈ U:
5              take deg(u) random elements of V as neighbors
           of u;
6          return G = (U ⊔ V, E);
7      end
```

For the structure of the random weights, we assume that the weights will be in the integer range $\{1, \ldots, 100000\}$. We have three models for assigning random weights to the edges. The first model is the *uniform-weights model* that assigns to every edge an uniform random weight in the range $\{1, \ldots, 100000\}$. The second model is the *uniform-low-high-weights model*. This model randomly partitions the set of edges in two parts, the low-weights part and the high-weights part, according to a parameter $p \in [0, 1]$. Every edge has probability p of being in the low-weights part, under a Bernoulli distribution, whose size is $\approx p \cdot |E|$ and the weights of these edges are chosen random uniform in the range $\{1, \ldots, 1000\}$. The size of the high-weights part is $\approx (1 - p) \cdot |E|$ and the weights of these edges are chosen random uniform in the range $\{1001, \ldots, 100000\}$. The third model is the *low-or-high-weights model* and is similar to the uniform-low-high-weights model, the difference is that the weights of the low-weights part are fixed at 1 and the weights of the high-weights part are fixed at 100000.

From the time complexities of the algorithms we can see that if $|V|$ is asymptotically smaller than $|U|$, then the FlowAssign algorithm is expected to be on advantage. Furthermore, the random graphs that we will generate can be unbalanced, in this case the objective is to find a minimum-weight matching that covers all the vertices in the smaller side V. The Hungarian and the FlowAssign algorithms are designed to directly solve the assignment problem on unbalanced graphs, while the auction algorithm is not. The auction algorithm must address that problem by working on an induced balanced graph that has the double

of vertices and edges, therefore the first two algorithms are expected to be on advantage on unbalanced instances.

5.2 The experimental results

To generate the random instances to be solved by the algorithms, we defined a list of values for each parameter and solved instances for all the possibilities of parameter configurations. The list of values are:

1. Edges distribution = {erdos-renyi, dispersed-degree},
2. Costs distribution = {uniform, uniform-low-high, low-or-high},
3. $n = \{1000, 2000, 4000, 8000\}$,
4. $s = \{\log n, \sqrt{n}, n\}$,
5. $Density = \{0.1, 0.5, 1.0\}$,
6. Normalized dispersion radius = $\{0.1, 0.5, 1.0\}$ (only for dispersed degree),
7. Low costs portion = $\{0.1, 0.5, 0.9\}$ (only for uniform-low-high and low-or-high).

Note that we include families of asymptotically smaller values of s respect to n to explore the advantage of the FlowAssign algorithm's time complexity. Each instance is defined by picking one of each parameter. Since the instances are generated randomly and the computer can have different workload when solving an instance, we decided to take the solving time of a fixed set of parameters as the average solving time of solving ten different random instances generated under the same parameters. Also, in the following experimental results, given a subset of parameters at fixed values, its solving time is the average time of all the instances that share the same values in such parameters. The machine used to solve the instances is Windows 8.1 (64-bit), Intel i5-4670 at 3.4 GHz (4 CPUs) and 16 GB RAM.

In Figure 1, we explore the sensitivity of the algorithms respect to the normalized dispersion radius parameter for the dispersed-degree random graph model. We can observe that the auction and FlowAssign algorithms perform worst at high normalized dispersion radius, while the Hungarian algorithm seems to benefit a little at high values of the parameter.

In Figure 2, we explore the sensitivity of the low-costs edges portion for the low-or-high-weights costs model. Its interesting that this parameter seems completely irrelevant for the Hungarian algorithm, while the FlowAssign algorithm seems to perform significantly better at high values of the parameter. For the auction algorithm, the best value seems to lie somewhere in the middle. This behavior may be influenced from the fact that the probability that the low-cost edges contain a (one-side) perfect matching increasing quickly with the number of edges, due to the way the edges are randomly generated.

In Figure 3, we explore the sensitivity of the low-costs edges portion for the uniform-low-high-weights costs model. This time the Hungarian algorithm is sensitive to the parameter and performs worst at high values of the parameter,

as well as the auction algorithm. The FlowAssign algorithm behaves similar than in the low-or-high-weights costs model.

In Figure 4, we explore the sensitivity of the algorithms to the random graphs models. In this case the auction algorithm performs better in the dispersed-degree random graphs model, while the FlowAssign and Hungarian algorithm performs better in the Erdös-Renyi model. This behavior can be related to the normalized dispersion radius.

In Figure 5, we explore the sensitivity of the algorithms to the random costs models. For this parameter the behavior is interesting. For the auction algorithm, we can observe a significantly worst performance in the low-or-high costs model but only for very unbalanced graphs, while for balanced graphs the algorithm performs its worst with the uniform-low-high costs model. Interestingly, the FlowAssign algorithm benefits a lot from the low-or-high costs model, and performs equal for the other two models. And the Hungarian algorithm performs a little better in the uniform costs model than in the other models. Remember that the auction algorithm has to solve the problem in an auxiliary graph with double number of vertices and edges when the graph is unbalanced.

Note the huge solving time difference between the algorithms in all the graphs presented so far. The performance of the auction algorithm is outstanding in every scenario, the performance of the FlowAssign algorithm is acceptable and the performance of the Hungarian algorithm is extremely poor.

In Figure 6, we explore the sensitivity of the algorithms to the unbalanceness gap, that is, to the asymptotic of $s = |V|$ respect to $n = |U|$. In this case we can observe that for $s = \log n$, the Hungarian algorithm performs better in average, followed by the Auction algorithm and then the Flow Assign algorithm, all of them with a significant difference. For $s = \sqrt{n}$, the FlowAssign and Hungarian algorithm perform similar between them and a lot better than the Auction algorithm. Remember that the auction algorithm is in big disadvantage in the unbalanced case in more than one way. For the balanced case $s = n$, the Auction algorithm is in first place, followed by FlowAssign by a significant factor and then Hungarian by a huge factor.

6   Conclusions

We have shown that two apparently different algorithms are rather the same under a few optimizations in allocation space. In other words, if we change a little bit the G&K algorithm in order to get off of redundant objects, we end up exactly with the auction algorithm. Therefore any heuristic developed for one of the algorithms can be implemented in the other without difficulty. Also the auction algorithm automatically induces a parallel implementation [2, 19] on the G&K algorithm.

From the performance analysis, we conclude that the Auction algorithm performs impressively better against other algorithms, even under conditions that put it in disadvantage. The only cases where it performed worst than its competitors is where the time required to solve the instances is very low. But

in general the auction algorithm outperformed its competitors by a big factor, even to the FlowAssign algorithm which has much better time complexity and is designed to exploit the unbalanceness gap of the bipartite graphs.

Acknowledgments

References

References

[1] D. P. Bertsekas, *The auction algorithm: A distributed relaxation method for the assignment problem*, Annals of Operations Research, 14 (1988) 105–123.

[2] D. P. Bertsekas, D. A. Castañon, *Parallel synchronous and asynchronous implementations of the auction algorithm*, Parallel Computing, 17 (1991) 707–732.

[3] D. P. Bertsekas, *Linear network optimization: Algorithms and codes*, MIT Press, Cambridge, MA, 1991.

[4] D. P. Bertsekas, *The auction algorithm for shortest paths*, SIAM J. on Optimization, Vol. 1, 1991, pp. 425–477.

[5] D. P. Bertsekas, *Auction algorithms for network flow problems: A tutorial introduction*, Computational optimization and applications, Vol. 1, pp. 7–66, 1992.

[6] D. P. Bertsekas, D. A. Castañon, H. Tsaknakis, *Reverse auction and the solution of inequality constrained assignment problems*, SIAM J. on Optimization, Vol. 3, 1993, pp. 268–299.

[7] D. P. Bertsekas, J. Eckstein, *Dual coordinate step methods for linear network flow problems*, Math. Progr., Series B, Vol. 42, 1988, pp. 203–243.

[8] D. Bertsimas, J. N. Tsitsiklis, *Introduction to linear optimization*, Athena Scientific, Belmont, MA, 1997.

[9] R. Burkard, M Dell'Amico, S. Martello, *Assignment Problems*, Revised reprint. SIAM, Philadelphia, PA, 2011.

[10] H. N. Gabow, R. E. Tarjan, *Faster scaling algorithms for network problems*, SIAM J. Computation, Vol. 18 No. 5, pp. 1013–1036, October 1989.

[11] A. V. Goldberg, R. E. Tarjan, *A new approach to the maximum flow problem*, Journal of the Association for Computing Machinery 35 (1988) 921–940.

[12] A. V. Goldberg, R. E. Tarjan, *Finding minimum-cost circulations by successive approximation*, Mathematics of operations research 15 (1990) 430–466.

[13] A. V. Goldberg, R. Kennedy, *An efficient cost scaling algorithm for the assignment problem*, Mathematical programming 71 (1995), 153–177.

[14] A. V. Goldberg, R. Kennedy, *Global price updates help*, SIAM J. Discrete Math., Vol. 10, No. 4, pp. 551–572, November 1997.

[15] H. W. Kuhn, *The Hungarian method for the assignment problem*, Naval Res. Logist. Quart., 2 (1955), pp. 83–97.

[16] H. W. Kuhn, *Variants of the Hungarian method for the assignment problem*, Naval Res. Logist. Quart., 2 (1956), pp. 253–258.

[17] E. L. Lawler, *Combinatorial optimization: Networks and matroids*, Holt, Rinehart & Winston, New York, 1976.

[18] J. B. Orlin, R. K. Ahuja, *New scaling algorithms for the assignment ad minimum mean cycle problems*, Mathematical programming, 54 (1992) 41–56.

[19] H. Zaki, *A comparison of two algorithms for the assignment problem*, Computational Optimization and Applications, 4 (1995) 23–45.

[20] L. Ramshaw and R. E. Tarjan, *A Weight-Scaling Algorithm for Min-Cost Imperfect Matchings in Bipartite Graphs*, 2012 IEEE 53rd Annual Symposium on Foundations of Computer Science, New Brunswick, NJ, 2012, pp. 581-590.
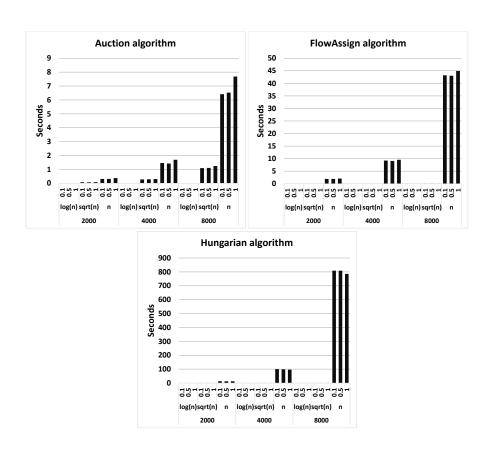
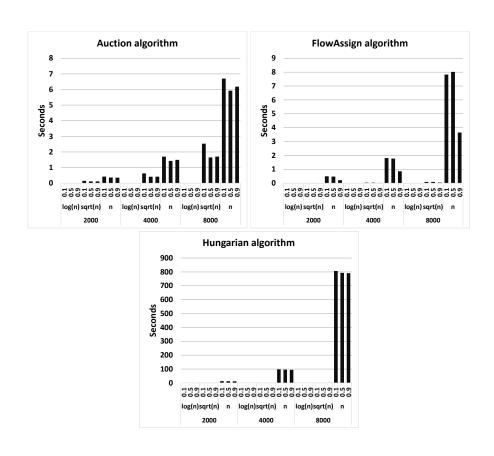Figure 1: Sensitivity analysis of the normalized dispersion radius parameter.

Figure 2: Sensitivity analysis of the portion of low cost edges for the low-or-high-weights costs model.
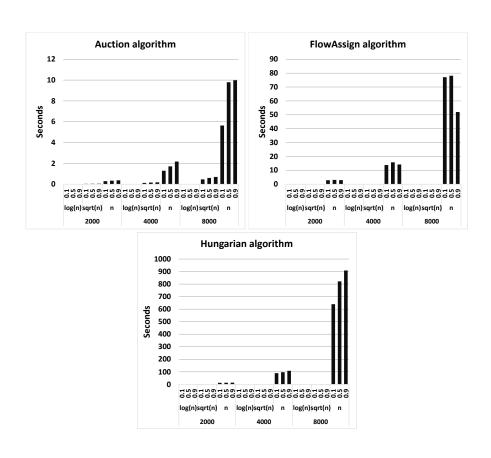
Figure 3: Sensitivity analysis of the portion of low cost edges for the uniform-low-high-weights costs model.
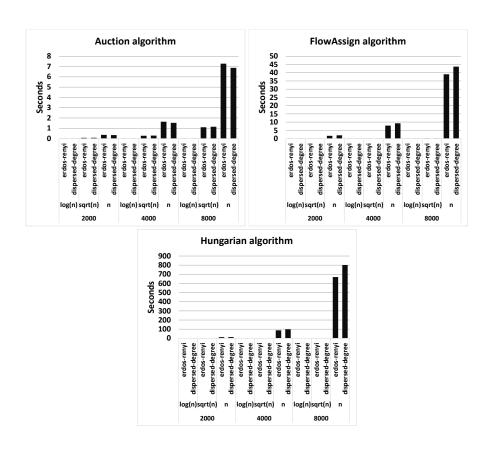
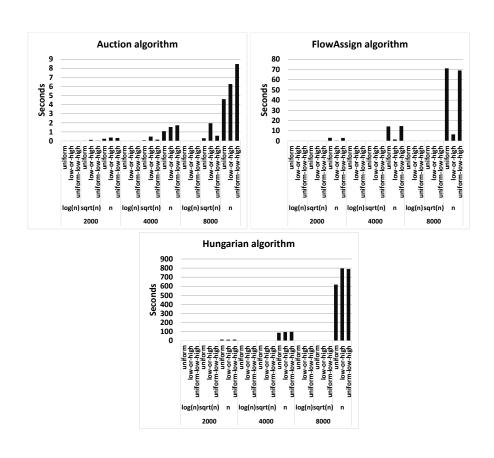Figure 4: Sensitivity analysis of the random graphs models.
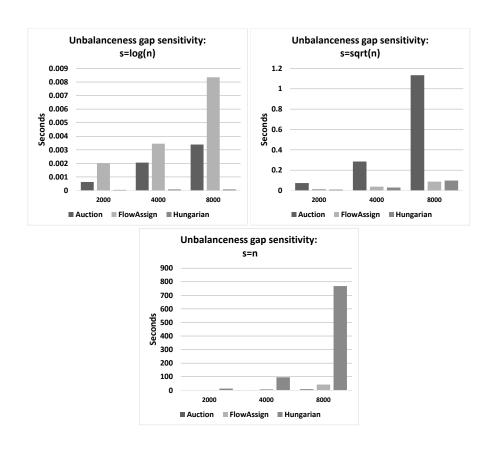
Figure 5: Sensitivity analysis of the random costs models.

Figure 6: Sensitivity analysis of the asimptoticity of $s = |V|$ respect to $n = |U|$.