An Analysis on Bipartite Matching and Matroids

Miguel Tavares
83528

Ricardo Brancas
83557

1 Theoretical Background

1.1 Bipartite Matching

Consider a bipartite graph $G = (W, J, E)$, where $V = W \cup J$. Let W be called the set of workers, and J be called the set of jobs. A matching between W and J over E consists of a subset of the edges E, such that selecting edge (w, j) means that worker w is assigned to job j, where each worker can have at most one job, and each job can be worked by at most one person.

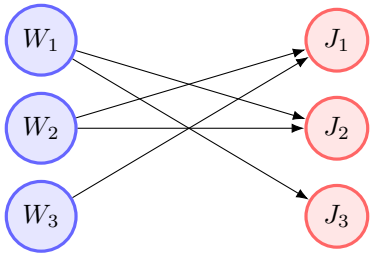

Figure 1: Example of an instance of the bipartite matching problem.

1.2 Maximum Cardinality Bipartite Matching

The Maximum Cardinality Bipartite Matching problem consists on finding the maximum size bipartite matching in a given graph $G = (W, J, E)$. The objective is to find a matching such that the number of assignments is maximal.

1.2.1 Reduction

One way to solve this problem is to reduce it to the classical Max-Flow problem. We introduce two new nodes, s and t; for each node $u \in W$ we add an edge (s, u) and for each node $v \in J$ we add an edge (v, t). The cardinality of the maximum matching is then the value of the max-flow from s to t, and the matching itself consists of the edges used for the flow.

1.2.2 Solution

One possible algorithm to solve the Max-Flow problem is Dinitz' Algorithm which runs in $O(|V|^2 |E|)$ time [5]. In practice, however, when using it to solve bipartite matching problems,

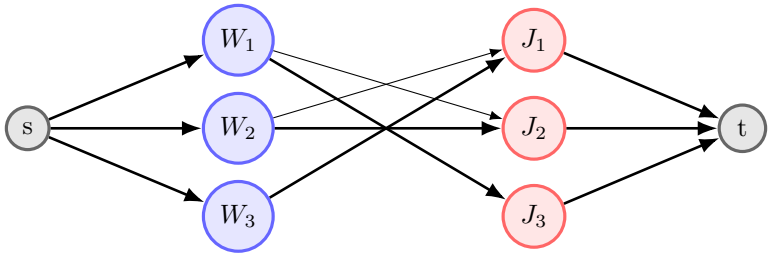

Figure 2: Example of the reduction to Max-Flow, with the optimal solution marked in bold.

the algorithm runs much faster; in fact it can be seen as a simplified form of the Hopcroft–Karp algorithm, which runs in time $O(|E|\sqrt{|V|})$ [5]. The algorithm uses $O(|V| + |E|)$ memory.

Furthermore, for the case of maximum cardinality bipartite matching all weights are unitary and it is unnecessary to explicitly represent the flow.

1.3 Maximum Weight Bipartite Matching

Let us now consider the weighted version of the Maximum Bipartite Matching problem. Consider a graph $G = (W, J, E)$ and a function $r : E \mapsto \mathbb{R}$. As before the problem consists of selecting a subset of E subject to each worker having at most one job, and each job being worked by at most one person. The goal is now to optimise this subset, $X \subset E$, such that $\sum_{x \in X} r(x)$ is maximal.

This problem is the maximisation version of the Assignment Problem[3].

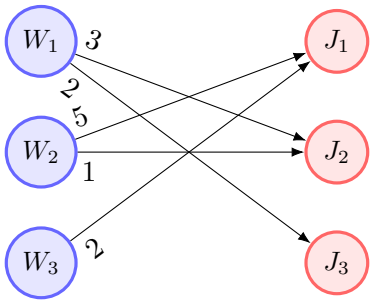

Figure 3: Example of an instance of the weighted bipartite matching problem.

1.3.1 Reduction

In order to use this algorithm we need to convert our problem to a minimisation problem. This can be done by creating a new r function, r' such that:

$$r'(e) = \max_{e' \in E}(r(e')) - r(e), \forall e \in E.$$

Moreover, the input to the problem is classically provided in matrix form: in tables 1 and 2 we can see the original and transformed matrix, respectively, corresponding to the example in figure 3.

	J_1	J_2	J_3
W_1	0	3	2
W_2	5	1	0
W_3	2	0	0

Table 1: Original matrix, r.

	J_1	J_2	J_3
W_1	5	2	3
W_2	0	4	5
W_3	3	5	5

Table 2: Modified matrix, r'.

1.3.2 Solution

A possible algorithm to solve the Assignment Problem is the Munkres Algorithm (also known as the *Hungarian method* and several other names) [5]. The algorithm starts with an empty matching, then it successively finds "augmenting paths" such that a new matching with greater value is obtained. The algorithm as described by Jin Kue Wong [2] runs in time $O(|V|^3)$. As for memory, the algorithm uses several auxiliary matrices/buffers amounting to $O(|V|^2)$.

1.4 Matroids

A matroid M is a pair $(E, \mathcal{F})$, where E is a a set (called the ground set) and $\mathcal{F}$ is a family of *independent* subsets of E, subject to the following restrictions:

- $\emptyset \in \mathcal{F}$,
- if $X \in \mathcal{F}$ and $Y \subseteq X$, then $Y \in \mathcal{F}$,
- if $X, Y \in \mathcal{F}$ and $|X| < |Y|$, then there is $y \in Y$ such that $X \cup \{y\} \in \mathcal{F}$.

1.4.1 Matroid Intersection

Let $M_1 = (S, \mathcal{F}_1)$ and $M_2 = (S, \mathcal{F}_2)$ be two matroids, on the same ground set S. Then $\mathcal{F}_1 \cap \mathcal{F}_2$ corresponds to the common independent sets of both matroids, ie. their intersection.

The algorithm for computing the intersection of 2 matroids, as defined by Schrijver [5], can be shown to run in $O(n^2 m(n + Q))$ time, where n is the maximum size of a common independent set, m is the size of the ground set, and Q is the running time of the independence oracle [5].

1.4.2 Bipartite Matching as a Matroid Problem

Consider again the problem of bipartite matching in a graph $G = (W, J, E)$, where $V = W \cup J$. Let $\delta : V \mapsto 2^E$ represent the edges incident on a given vertex, $v \in V$. Let also:

- $\mathcal{F}_W = \{X \subseteq E : |X \cap \delta(v)| \leq 1, \forall v \in W\}$
- $\mathcal{F}_J = \{X \subseteq E : |X \cap \delta(v)| \leq 1, \forall v \in J\}$

Then $M_W = (E, \mathcal{F}_W)$ and $M_J = (E, \mathcal{F}_J)$ are matroids [1] (known as partition matroids). Furthermore, the largest cardinality element in the intersection corresponds to a maximum matching in G.

The independence oracle for this matroid can be implemented in $O(1)$ time and $O(|V|)$ space, just by keeping a counter for each vertex representing the number of incident edges.

1.4.3 r-Arborescences as a Matroid Problem

Consider a directed graph $D = (V, E)$ and a special root vertex, $r \in V$. Then an r-arborescence is a directed spanning tree oriented away from r. We assume that graph D has no incoming arcs into vertex r.

Let G be the undirected counterpart of graph D. Note that if graph D already contains edges (u, v) and (v, u) then they will also be duplicated in graph G. Let also:

- $\mathcal{F}_1 = \{X \subseteq E : \text{acyclic}(X)\}$,
- $\mathcal{F}_2 = \{X \subseteq E : |X \cap \delta^-(v)| \leq 1, \forall v \in V \setminus \{r\}\}$,

where $\delta^-(v)$ corresponds to the set of edges incoming to v.

Then $M_1 = (E, \mathcal{F}_1)$ and $M_2 = (E, \mathcal{F}_2)$ are matroids (a graphic matroid and a partition matroid, respectively) [1]. Moreover their intersection is the set of all sets of edges that are acyclic and where each vertex has at most one incoming edge, ie. an arborescence; maximising the cardinality of these sets makes it so that all paths must start in r resulting in an r-arborescence.

A simple way to check independence of a given set is to just check if the corresponding graph is acyclic, using for example a DFS, resulting in $O(|V| + |E|)$.

2 Implementation Details

2.1 Dinitz' Algorithm

As mentioned in section 1.2.2 when solving maximum cardinality bipartite matching the flow on a given edge is always either one or zero, and as such it is possible to avoid representing the flow and the residual capacity separably, as one can be derived from the other. Moreover, if we modify the original graph to represent the residual capacity, then the algorithm needs no extra memory.

2.2 Munkres Algorithm

It is important to point out that we assumed our input graphs to be dense, which is why we chose the Munkres algorithm to solve this problem. If, however, the graphs were mostly sparse there would be two possible solutions:

- Modify the algorithm to use a sparse matrix. This however comes with a lot of problems as we need to ensure that the cells needed at each time step are *active* in the matrix.
- Reduce the problem to min-cost max-flow, and use a different algorithm altogether.

The version of the Munkres algorithm we implemented is the one described by R. A. Pilgrim [4], although we optimised a few cycles, particularly in step 4.

2.3 Matroid Intersection Algorithm

We created a generic `Matroid` class which includes the matroid intersection algorithm. Creating new kinds of matroids consists of creating subclasses and implementing the oracles.

It is also worth to mention that our Matroid interface requires two functions for the oracle:

- `is_independent_with(e)`: $E \mapsto$ `bool`
  This function checks if adding element `e` to the current independent set maintains independence or not;
- `is_independent_except_with(e1, e2)`: $E \times E \mapsto$ `bool`
  This function checks if by removing element `e1` and adding `e2`, the current set is still independent.

This allows for much faster oracle calls for some matroids (like the partition matroid).

3 Experimental Analysis

In this section each data point corresponds to the average of 11 executions of different problems generated with the same parameters. We also present in red the regression of the execution time and memory usage plots with respect to the theoretical complexities (note: to simplify the computation of the regressions we assumed the worst case of $|E| = |V|^2$).

Considering we used Python to implement our algorithms memory usage data might not be relevant, as it is a garbage collected language, nevertheless we present the results.

3.1 Maximum Cardinality Bipartite Matching

Explanation of the parameters:

- p: probability of a given edge (w, j) existing;
- size: number of elements on each partition.

From the analysis of figure 4 we can see that the running time of the algorithm seems to fit within the theoretical bounds (as stated in section 1.2.2). Moreover the space complexity also seems to fit the excepted behaviour.

Complementary, by analysing figure 5 we can see that both time and spatial complexities are linear in the number of edges (when fixing the number of vertices), as would be expected.

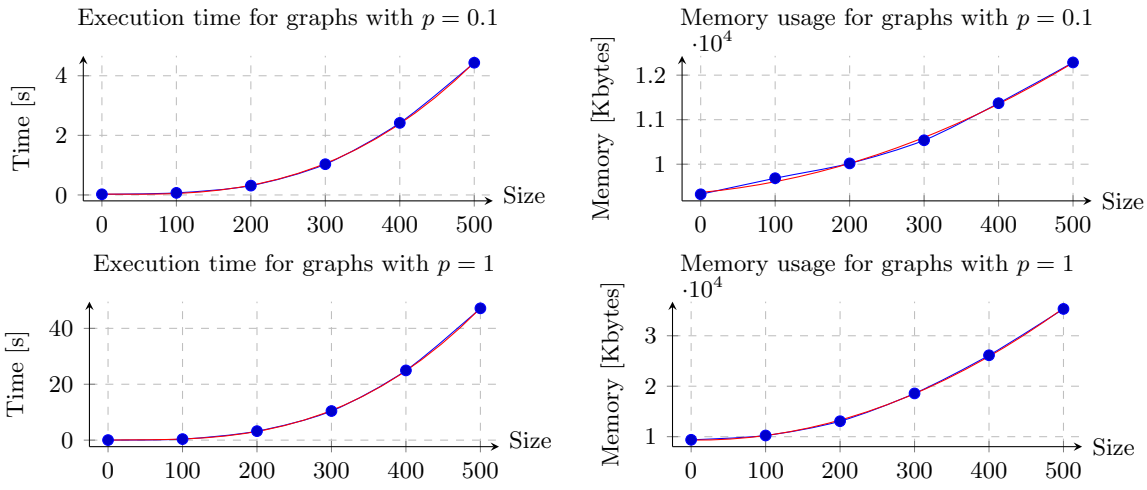

Figure 4: Execution results of the maximum cardinality bipartite matching problem.

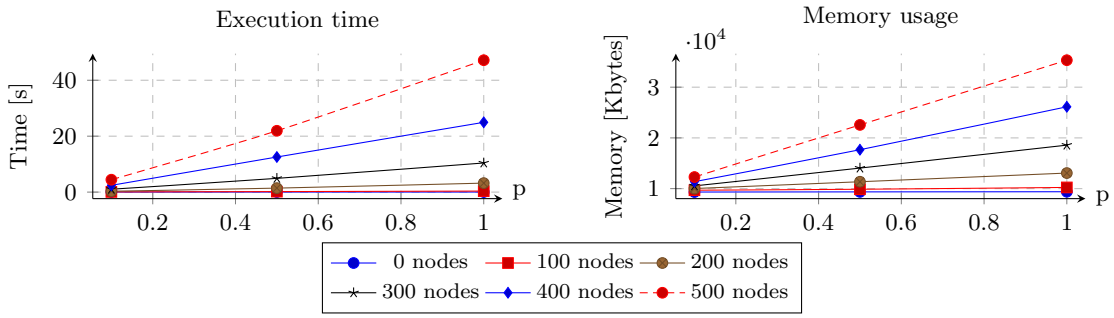

Figure 5: Evolution for different values of p in the maximum cardinality bipartite matching problem.

3.2 Maximum Weight Bipartite Matching

Explanation of the parameters:

- p: probability of a given edge (w, j) existing;
- w: maximum weight possible for any edge (weights follow a discrete uniform distribution $]0, w]$);
- size: number of elements on each partition.

Looking at figures 6 and 7 we conclude that the running time of this algorithm is also within the theoretical bounds (as stated in section 1.3.2). Furthermore the space complexity also corresponds to what was expected.

Figure 8 seems at first to be nonsensical, as the algorithm runs faster for problems with more edges. This is however because, as previously referred, the Hungarian method works best when graphs are dense.

As for the effect of the weights' magnitude, even though just by looking at figure 9 one might think that they do have an impact, this is actually because increasing the number of possible weight values, also increases the number of possible optimal solution values. If we were to change the magnitude of the weights uniformly (ie. multiplying all of them by some large number) we would see that it doesn't actually impact the performance of the algorithm, as would be expected. The differences in memory usage are because of the way Python handles small integer numbers.

3.3 Matroid based Maximum Cardinality Bipartite Matching

Explanation of the parameters:

- p: probability of a given edge (w, j) existing;
- size: number of elements on each partition.

The first thing to notice when looking at figure 10 is that this algorithm is much slower than the max-flow reduction in section 3.1, however (assuming we already have a matroid intersection algorithm) there is almost nothing to implement: just the oracles.

As before, analysing figures 10 and 11 leads us to believe that our algorithm fits the expected time and space requirements; in particular figure 11 indicates that the algorithm in linear in the number of edges, as was indicated by the theoretical bounds.

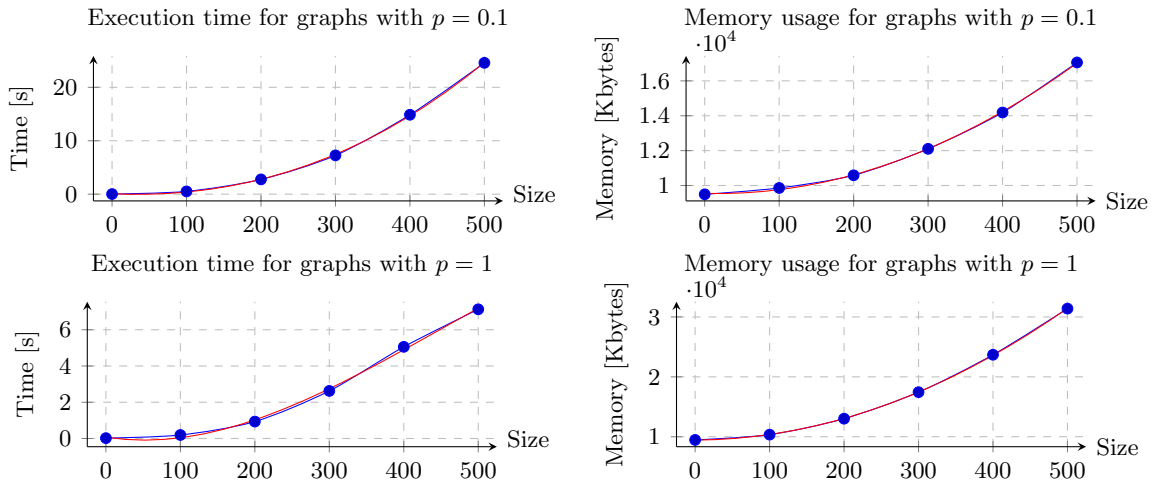

Figure 6: Execution results of the maximum weight bipartite matching problem, with $w = 100$.

3.4 Matroid r-Arborescences

Explanation of the parameters:

- p: probability of a given edge (u, v) existing;
- size: number of elements on each partition.

The first thing to notice when observing figure 12 is that the number of instances has been cut down from the previous problem, in particular when the graph is dense we actually had to stop the computation because it was taking too long. We see that for $p = 1$ the interpolation of our data points is above the obtained regression but theorise that it is because the data has too little resolution compared to the rate of growth of the running time.

In figure 13 we were expecting to see a non-linearity because of the running time of the oracle being $O(|V| + |E|)$, which would result in the time being quadratic with respect to $|E|$. However from the little data points obtained the algorithm appears to still be linear on the number of edges; this might be because although a DFS depends linearly on both the number of vertices and edges, particular applications might be more sensitive to one or the other.

References

[1] Michel Goemans. *Handout 14: Lecture notes on matroid intersection*. Mar. 2011. URL: `http://math.mit.edu/~goemans/18433S11/matroid-intersect-notes.pdf`.

[2] Jin Kue Wong. "A new implementation of an algorithm for the optimal assignment problem: An improved version of Munkres' algorithm". In: *BIT Numerical Mathematics* 19.3 (Sept. 1979), pp. 418–424.

[3] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Algorithms and Combinatorics 21. Springer-Verlag Berlin Heidelberg, 2012.

[4] R. A. Pilgrim. *Munkres' Assignment Algorithm*. URL: `http://csclab.murraystate.edu/~bob.pilgrim/445/munkres.html`.

[5] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Algorithms and Combinatorics 24. Springer-Verlag Berlin Heidelberg, 2013.

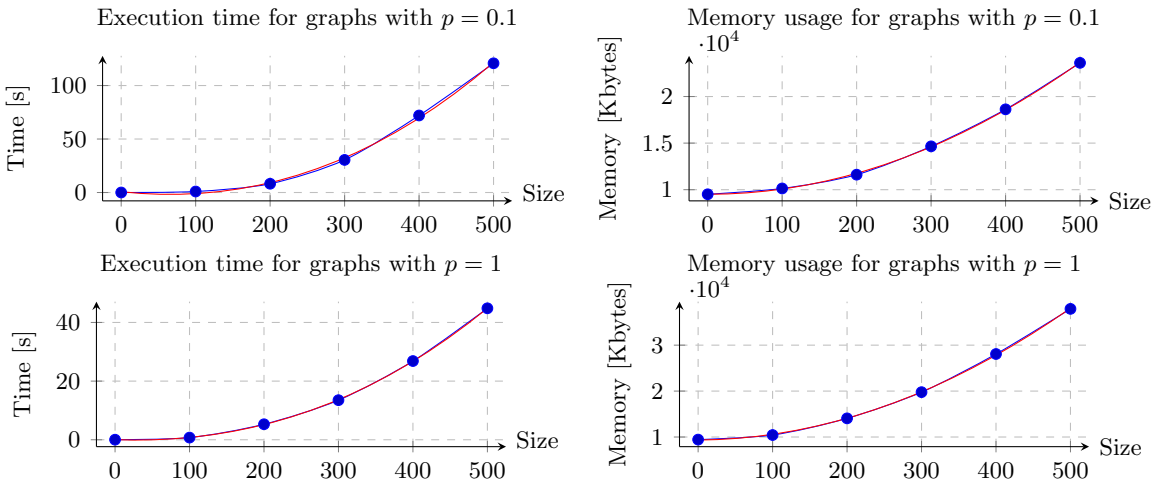

Figure 7: Execution results of the maximum weight bipartite matching problem, with $w = 5000$.

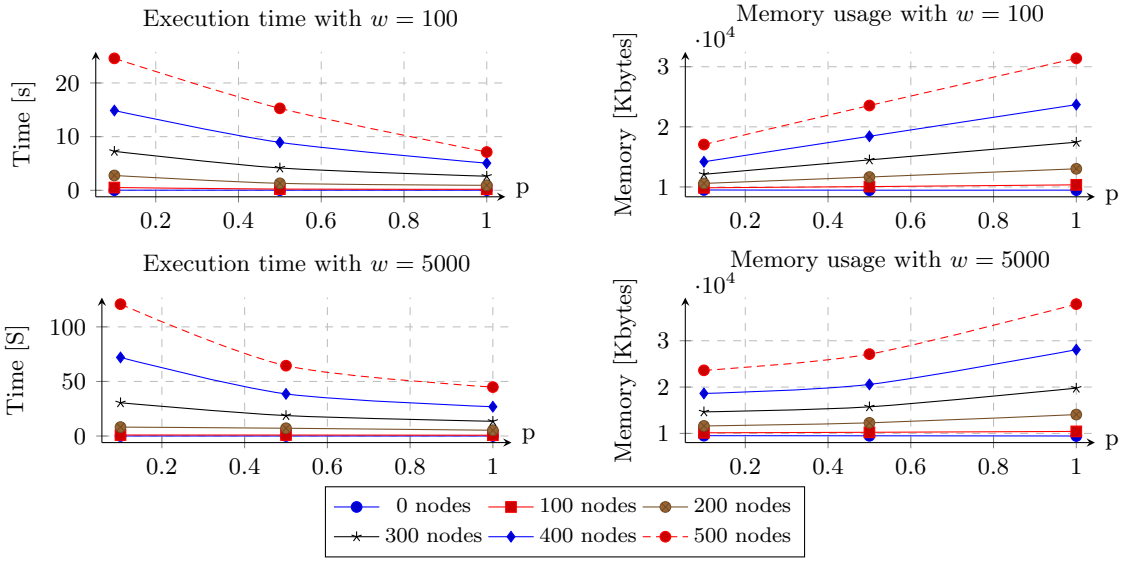

Figure 8: Evolution for different values of p in the maximum weight bipartite matching problem.

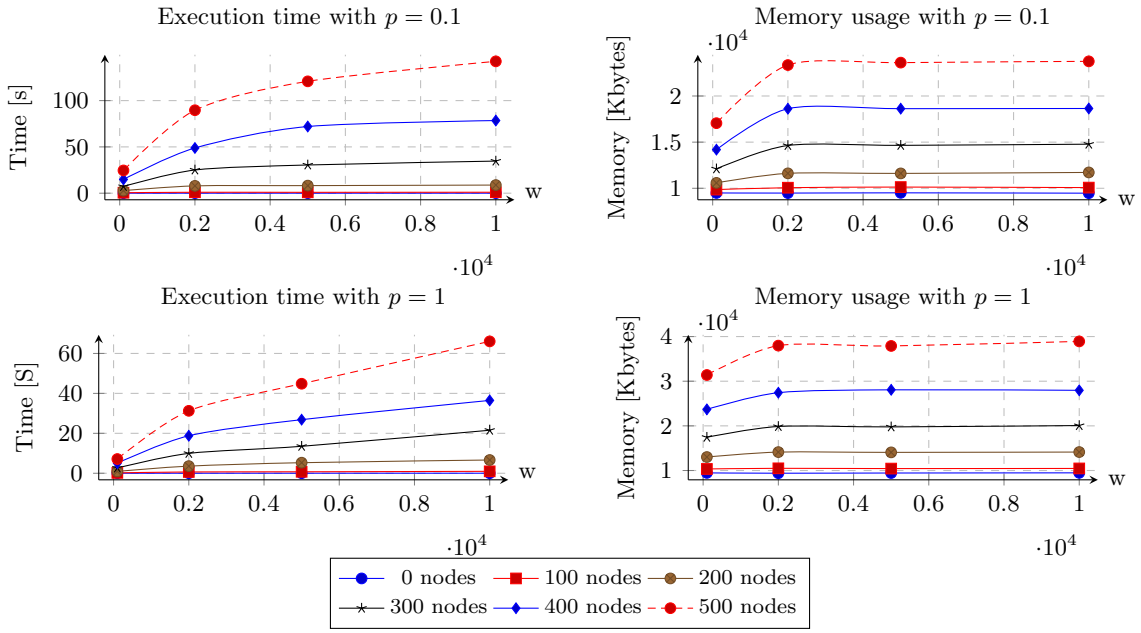

Figure 9: Evolution for different values of w in the maximum weight bipartite matching problem.

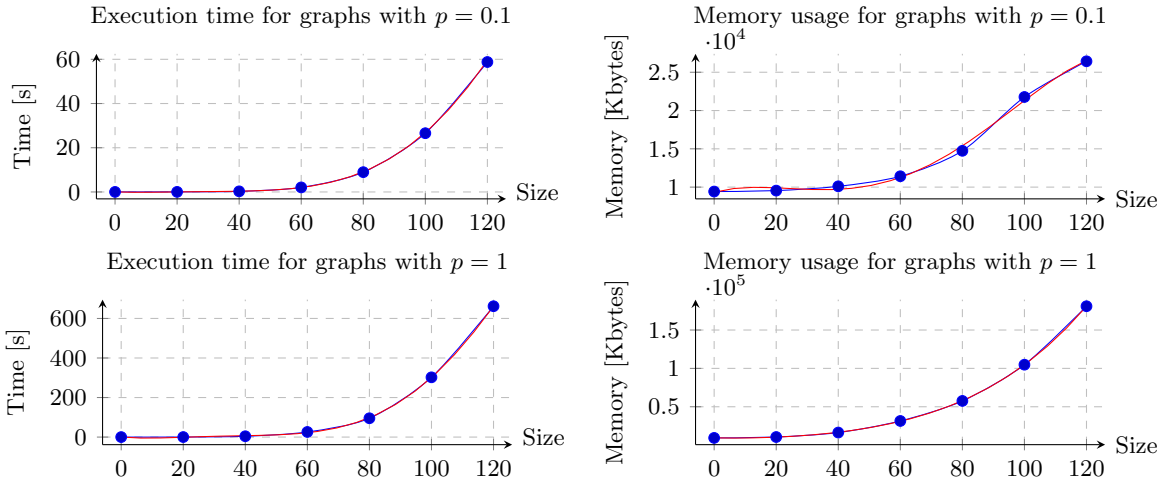

Figure 10: Execution results of the matroid maximum cardinality bipartite matching problem.

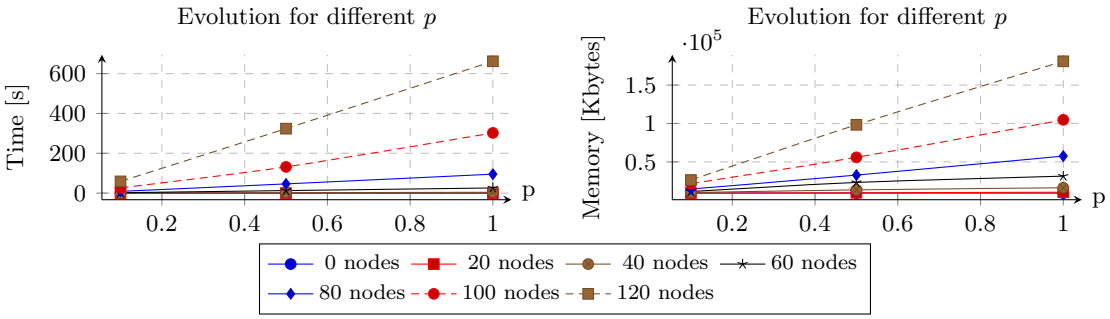

Figure 11: Evolution for different values of p in the matroid maximum cardinality bipartite matching problem.

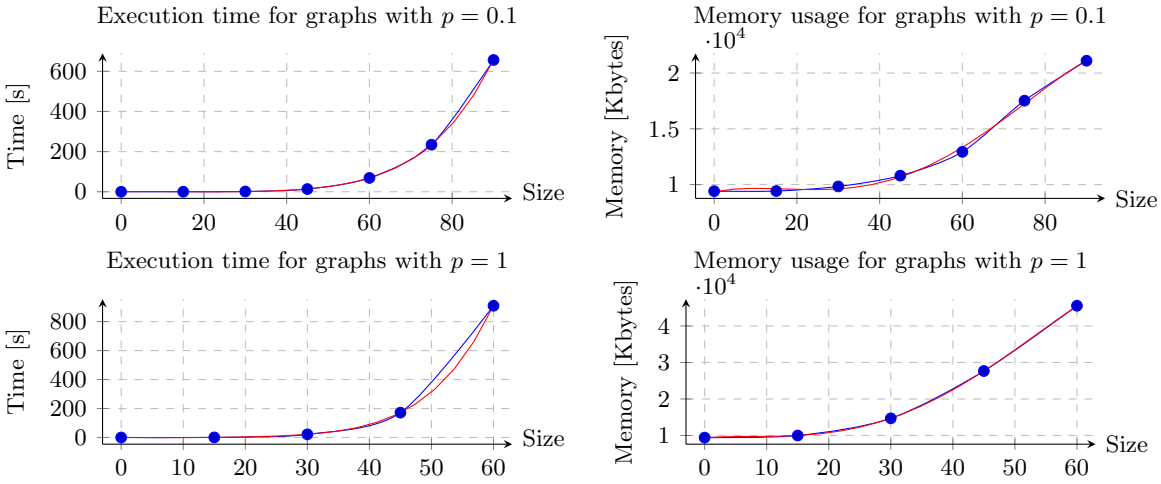

Figure 12: Execution results of the matroid r-arborescence problem.

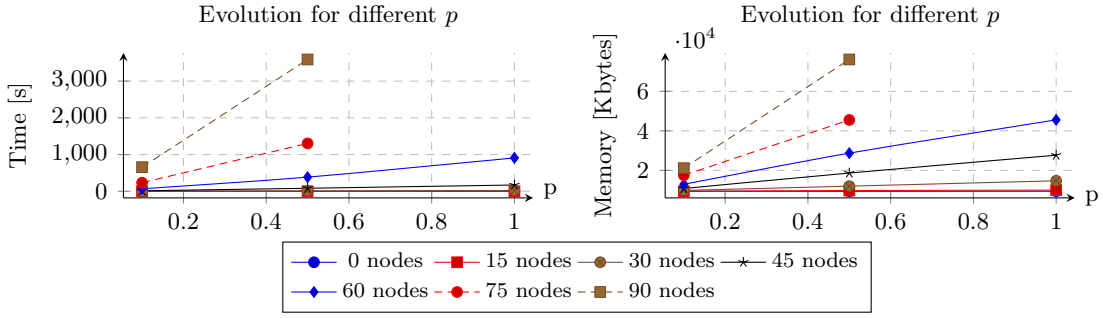

Figure 13: Evolution for different values of p in the matroid r-arborescence problem.