Optimizing Bipartite Matching with Interleaved and Injective Mappings: Implementing and Evaluating the k-Swap Heuristic

Gonzague Yernaux(✉), Manel Barkallah, Mikel Vandeloise, Wim Vanhoof, and Jean-Marie Jacquet

Faculty of Computer Science, University of Namur, Namur, Belgium
{gonzague.yernaux,manel.barkallah,mikel.vandeloise,
wim.vanhoof,jean-marie.jacquet}@unamur.be

Abstract. Bipartite matching problems play a crucial role in various software engineering tools, including resource allocation and task assignment. In this paper, we address a specific variant of the bipartite matching problem, called the Double Assignment Problem (DAP), focusing on the allocation of machines to workers in a production environment. The objective is to maximize the number of worker-machine associations, subject to a second, orthogonal matching problem: associating some worker W to a machine M implies that the (ordered) list of servers employed by M are dedicated to the respective programs used by the worker W. Since DAP is, in general, NP-hard, we introduce a heuristic that quickly approximates candidate results. The heuristic is called k-swap stability and has originally been formalized to tackle a specific DAP instance arising in the niche field of anti-unification. We extend the definition to our more general setting and give promising preliminary results obtained by applying our k-swap implementation on a testbed of examples.

Keywords: Double Assignment Problem · combinatorial optimization · approximation algorithms · escape game puzzles · resource allocation

1 Introduction

In the world of combinatorial optimization, the problem of matching elements from two disjoint sets under certain constraints, known as the Assignment Problem, is fundamental and has widespread applications. This paper tackles a nuanced variant of this bipartite matching problem, in which the purpose is to optimize the allocation of *machines* to *workers* in industrial settings while ensuring compatibility based on machine, worker *types* and *properties*. Note that, while we will use the machine/worker terminology in the paper, the problem can arise in a range of different applications.

K. Daimi and A. Al Sadoon (Eds.): ACR 25, LNNS 1346, pp. 51–62, 2025.
https://doi.org/10.1007/978-3-031-87647-9_5

Let us first give the intuition of our Assignment Problem variant. Consider a bipartite graph where vertices represent machines and workers. The objective is to establish a pairing among machines and workers that maximizes productivity (being defined as the number of worker-machine couples in the pairing), subject to the following critical constraints. First, machines, and workers are categorized into *types*; a machine of some type T can only be assigned a worker of the same type T (machines and workers are each given only one type). This represents the fact that a worker is only trained to manipulate a certain machine requiring certain technological knowledge, see e.g. [2]. Additionally, machines require access to *servers*, and workers utilize specific *computer programs*. Associating a worker W with a machine M implies installing the worker's sequence of programs on the machine's sequence of servers. Each server can only support one program, and vice versa. Formally, this means that the list of programs of W must be mapped onto the list of servers used by M in a way that must ensure *injectivity*, i.e., each server paired with a program cannot be mapped to another program in another worker-machine association.

As an example highlighting the injectivity constraint, consider two machines, M_1 and M_2 of a same type T, with respective lists of servers $[s_1, s_2]$ and $[s_2, s_3]$. Consider two workers W_1 and W_2, also of type T, with respective programs $[p_1, p_2]$ and $[p_3, p_2]$. Then, it is impossible to construct a valid machine-worker pairing of size 2, since it would imply mapping p_2 onto two *different* servers.

In what follows, we will refer to this class of problems as instances of the Double Assignment Problem (DAP), a name chosen to reflect the fact that a second mapping (namely that of servers and programs) is being constructed alongside the main assignment (namely that of machines and workers). Examples of practical use cases in which a DAP instance arises include the following:

- In cloud computing design, one may seek an injective mapping from virtual machines to tasks, while respecting compatibility constraints between hardware configurations (servers) and software dependencies (programs) [1].
- In logistics, assigning delivery methods (servers) to specific routes (programs) sometimes involves respecting an underlying association of specific drivers (workers) to the vehicles (machines) that they are allowed to drive [5].
- In escape games (or rooms), players often have to solve small-scale mathematical puzzles to obtain some clue as to their following task [8]. Interestingly, small DAP instances are, in fact, often used for such purposes.

Although the problem thus does arise in a range of real-life situations (the list above being non-exhaustive), it has, to the best of our knowledge, not been studied formally before. In the remainder of the paper, we first introduce DAP in a more formal way to fill this gap, then prove it to be NP-hard. Since it appears from our research that no existing heuristic or algorithm can as-is treat the problem in a fairly efficient or scalable way – due to the subtlety of the injectivity constraints – we subsequently develop a dedicated heuristic and compare its performances with those of generic classes of known algorithms selected for their capability to solve or approximate such search problems. To achieve this, we introduce the notion of k-swap stability, where the parameter k controls

the approximation level, considering as *stable* those solutions that cannot be extended with a new machine-worker couple despite *swapping* (i.e. replacing) of most k previously selected machine-worker couples. The core of our algorithm revolves around iteratively swapping pairs within the matching under construction to incorporate some new machine-worker couple, as such enhancing the so-called *quality* of the matching while upholding adherence to the inherent DAP constraints. To validate our approach, we implement the algorithm in Java, and give quantitative results of its runs.

Throughout the paper, we will use the running example of an escape room puzzle that relies on DAP logic – as briefly discussed above. The puzzle basically boils down to finding pairs of *dominos* decorated by *symbols*, ensuring that all these pairs are compatible with one another. While such examples allow to visualize DAP instances easily, recall that more scalable or real-life DAP occurrences – i.e. those for which we are looking for an efficient approximation – tend to harbour drastically more machines, workers, servers, and programs.

2 Setting the Stage: The Double Assignment Problem (DAP)

We start by defining what assignments are considered *valid* within DAP.

Definition 1. *Let us consider a set of machines $\mathcal{M}$, a set of servers denoted $\mathcal{S}$, and a function $S : \mathcal{M} \mapsto (\mathbb{N} \mapsto \mathcal{S})$, where $\forall m \in \mathcal{M}$ the sequence $S(m)$ is called the* list of servers *of m. To denote such a sequence we will sometimes use the notation $\langle s_1^m, \ldots, s_n^m \rangle$, where $n \in \mathbb{N}$, and to further ease notation, given such a sequence u we will refer to its ith element by u_i. Let us similarly consider a set of workers $\mathcal{W}$ and a set of programs $\mathcal{P}$, such that each worker $w \in \mathcal{W}$ is associated to a sequence referred to as its* list of programs *through a function $P : \mathcal{W} \mapsto (\mathbb{N} \mapsto \mathcal{P})$. Both machines and servers are also given a type, represented by integers and retrievable through a function $t : \mathcal{M} \cup \mathcal{W} \mapsto \mathbb{Z}$.*

A pair of mappings $(\phi : \mathcal{M} \mapsto \mathcal{W}, \psi : \mathcal{S} \mapsto \mathcal{P})$ is said to be valid *if and only if (1) $\forall (m, w) \in \phi : t(m) = t(w)$, i.e. coupled machines and workers must be of the same type; (2) $\forall m_1, m_2 \in \mathcal{M} : m_1 \neq m_2 \Rightarrow \phi(m_1) \neq \phi(m_2)$, i.e. ϕ is injective; (3) $\forall s_1, s_2 \in \mathcal{S} : s_1 \neq s_2 \Rightarrow \psi(s_1) \neq \psi(s_2)$, i.e. ψ is injective; (4) $\forall (m, w) \in \phi : |S(m)| = |P(w)| \wedge \forall i \in 1..|S(m)| : \psi(S(m)_i) = P(w)_i$, i.e. each server of m is associated (through ψ) with exactly one program of w, corresponding to their position in the lists of servers and programs.*

Given two machine-worker couples (m, w) and (m, w'), we will sometimes say that the couples are *compatible* when there exists a mapping ϕ containing both couples and being part of a valid pair of mappings; otherwise the couples are said to be *incompatible*. Abusing terminology, we will also sometimes use the term *(in)compatible* for entire (parts of) mappings instead of individual couples.

The topmost part of Fig. 1 is an instance of a game that we will call DominAP, incarnating a logic puzzle based on small instances of DAP. The upside-down dominos on the top play the role of the *machines*; they feature everyday life

Fig. 1. An illustrative DominAP instance and two valid pairings.

objects, such as an hourglass and a ball, that represent the underlying *servers*. Similarly, the right side up dominos below them are the *workers* and contain animals (for *programs*). Given such an initial configuration, the player of a DominAP instance is asked to form as many pairs of dominos as possible. The *type* of a domino is represented by the number of bullets that are drawn on its light-coloured triangle, and two dominos in a pair must have the same type. Of course, it is also imperative that each pair be composed of an object-domino (machine) and an animal-domino (worker), and that the injectivity constraint is respected, i.e. creating a pair of dominos implies mapping the animals of its worker on the objects of its machine, in the order in which they appear on the surface when placed as in the figure. Obviously, this strictly corresponds to the search for two mappings in a DAP instance – with the subtle variation that the player must try to maximize the number of domino pairs.

Example 1. In the situation depicted in Fig. 1, it is obviously not possible to find a valid double matching such that the mapping ϕ contains 4 pairs of dominos (because of the different types and the injectivity constraints). It is however straightforward to find *two* compatible pairs of dominos, e.g. by mapping the bird on the hourglass. Such a mapping, depicted in the lower left part of Fig. 1,

cannot further be extended due to the injectivity constraint of the underlying mapping ψ between animals and objects. Indeed, the two dominos having three symbols cannot be added as a pair to the matching under construction, since it would imply associating the bird with the ball, when the bird is already paired with the hourglass in the two already chosen pairs. In contrast, the three pairs of dominos on the bottom-right side of the figure are compatible with one another; each animal is mapped on a different object (and vice versa), which means that the only recurring symbols (namely the bird and the ball) are coherently mapped onto one another. This evidently induces a mapping ϕ of maximal size ($|phi| = 3$). The same, as it happens, holds here for ψ ($|\psi| = 5$ being maximal), but this is not necessarily the case in general. Also note that while, in general, more than one domino mapping ϕ of maximal size can exist, the purpose of the DominAP game is to find *one* of such largest mappings.

The example above hints that, while finding valid mappings is a rather straightforward task, some of these mappings might be considered suboptimal. To remain parametric regarding the underlying context of application, we define such a sense of optimality in terms of a *quality function* ω.

Definition 2. *Let ϕ and ψ be a valid pair of mappings as per Definition 1. Then, a* quality function *is any function ω that takes such a pair of mappings as input and outputs a real number, i.e. a function that follows the signature $\omega : (\mathcal{M} \mapsto \mathcal{W}) \times (\mathcal{S} \mapsto \mathcal{P}) \mapsto \mathbb{R}$.*

In practice, one useful incarnation of the quality function ω is simply the function $\hat{\omega}$ counting the number of machines that have found a matching worker, i.e. $\hat{\omega}(\phi, \psi) = |\phi|$. It is this quality function that needs to be maximized in DominAP instances. While particularly straightforward, the function $\hat{\omega}$ reflects the search for a machine-worker assignment allowing for as many machines to be handled at once; this corresponds to a common concern in assignment (and, more globally, optimization) problems. As such, we will from now on facilitate our discussion and consider $\hat{\omega}$ as our working quality function. Recall however that many other incarnations of quality measures exist, an example being a function counting how many *different types* are captured by the mappings.

We can now define DAP and show that, when instantiated on the quality function $\hat{\omega}$, the problem is intrinsically hard.

Definition 3. *Let* DAP *denote the following problem: for given sets $\mathcal{M}, \mathcal{W}, \mathcal{S}$ and $\mathcal{P}$, find a pair of valid mappings ϕ and ψ such that $\nexists$ another valid pair of mappings ϕ' and ψ' (with $(\phi, \psi) \neq (\phi', \psi')$) verifying $\omega(\phi', \psi') > \omega(\phi, \psi)$.*

Proposition 1. *Let DEC-DAP refer to the decision-problem "Given a DAP instance, does it admit a valid pair of mappings (ϕ, ψ) such that $\hat{\omega}(\phi, \psi) = \min(|\mathcal{M}|, \mathcal{W})$?". DEC-DAP is NP-complete.*

Proof. Proving the belonging of DEC-DAP to NP is immediate, since the verification of an adequate solution can be achieved simply by computing $\hat{\omega}$ values as well as the input sets' cardinalities, which is done in linear time.

We will now perform a reduction from the Induced Subgraph Isomorphism Problem (ISIP) [11], generalizing the proof given in [14]. ISIP can be formulated as follows. Given two non-oriented and unweighted graphs, (V_1, E_1) and (V_2, E_2), with $|V_1| \leq |V_2|$ and where for each graph (V_i, E_i), V_i denotes the set of vertices and E_i the set of edges between vertices from V_i, then ISIP is the problem of deciding whether (V_1, E_1) is isomorphic to an induced subgraph of (V_2, E_2). For such an isomorphism to be found, there needs to exist a (total) injective function $f : V_1 \mapsto V_2$ such that $\forall x, y \in V_1$, there is an edge $(x, y) \in E_1$ if and only if there is an edge $(f(x), f(y)) \in E_2$. The problem is NP-complete [11].

Let us transform an arbitrary instance of ISIP into an instance of DEC-DAP as follows. Given the graphs (V_1, E_1) and (V_2, E_2) (with $|V_1| \leq |V_2|$), we define a set of machines $\mathcal{M}_1 = \{M_i | i \in V_1\}$ that are all compatible with the corresponding set of workers $\mathcal{W}_1 = \{W_i | i \in V_2\}$, such that $\forall i \in V_1 : S(M_i) = \langle i \rangle$ and $\forall i \in V_2 : P(M_i) = \langle i \rangle$. We then define a second set of machines $\mathcal{M}_2 = \{M'_{i,j} | (i, j) \in E_1\}$ and a second set of workers $\mathcal{W}_2 = \{W'_{i,j} | (i, j)\ in E_2\}$. Again, these machines and workers are supposed to be of the same type. This time, the servers, and programs are such that $\forall (i, j) \in E_1 : S(M'_{i,j}) = \langle i, j \rangle$ and $\forall (i, j) \in E_2 : P(W'_{i,j}) = \langle i, j \rangle$. We then have our complete sets of machines $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2$ and $\mathcal{W} = \mathcal{W}_1 \cup \mathcal{W}_2$.

The machines are thus composed of the nodes (having one server) and the edges (having two) of the first graph, while the workers represent those of the second graph, where nodes are similarly encoded as workers having one program, and edges as workers having two. Now if we were able to decide DEC-DAP, we would be capable of knowing whether there exists a total function mapping the vertex identifiers of V_1 onto those of V_2, while ensuring that all edges from E_1 are found – after applying ψ on their constitutive vertices – in E_2, since the injectivity constraint in the mapping ψ would need to be observed. DEC-DAP's answer would thus be "yes" if and only if all vertices and edges of V_1 have an isomorphic counterpart in V_2, i.e. if the answer to ISIP is also "yes". □

In various domains such as cloud computing, telecommunications, healthcare, supply chain management, manufacturing and energy management, the ability to quickly and accurately assign resources while respecting compatibility constraints directly impacts operational efficiency and service quality. Consequently, developing techniques to approximate DAP solutions swiftly while maintaining a high level of quality (ω-wise) is critical. The following section introduces such a heuristic, initially applied in the context of so-called *anti-unification* of logic program artefacts [14], which in fact incarnates a specific case of DAP.

3 The *k*-Swap Stability Abstraction

The k-swap stability abstraction is a heuristic particularly adapted to problems involving assignments under constraints. It is based on the following key notion.

Definition 4. *Let k be a natural, and ϕ and ϕ' be two injective mappings of elements from disjoint sets, such that $|\phi|$ ¿ $|\phi'|$. We say that ϕ' is a* k-swap *of ϕ if and only if $|\phi| - |\phi \cap \phi'| \leq k$.*

In other words, a mapping is a k-swap of another mapping if it can be obtained by "swapping" (i.e. removing or replacing) at most k pairs in it. The stability property can then be defined as follows.

Definition 5. *Let us consider an instance of DAP, as well as two mappings* (ϕ, ψ) *that form a valid pair of mappings for it. The pair* (ϕ, ψ) *is said to be* k-swap-stable *if and only if there does not exist* (ϕ', ψ') *and* $(\hat{\phi}, \hat{\psi})$*, two valid pairs of mappings w.r.t. the DAP instance, such that* ϕ' *is a* k*-swap of* ϕ *and* $\phi' \subseteq \hat{\phi}$ *and* $|\hat{\phi}| > |\phi|$*.*

Intuitively, a pair of mappings that is not k-swap-stable is thus a pair (ϕ, ψ) where ϕ admits a k-swap that can readily be extended (without breaking the injectivity rules) into some larger mapping ($\hat{\phi}$ in the definition). The k-swap-stability criterion can thus be understood as an indication of the fitness of an assignment under construction. If a pair of mappings is not k-swap-stable (where $k \in \mathbb{N}$ is a parameter determined beforehand), it means that a "better" assignment can be found (at least w.r.t. $\hat{\omega}$) at the cost of swapping up to k pairs in the machines-workers mapping ϕ. On the other hand, if the mapping is stable already, then it is considered a "good enough" approximation of an optimal (double) assignment.

Example 2. Let us reconsider the DominAP instance from Fig. 1. The mapping in the lower left part is 1-swap-stable: one could not remove a pair of dominos and replace it by another pair that could lead to admitting a third pair in the mapping. However, the solution is not 2-swap-stable, since the replacement of both dominos can lead to the mapping of size 3 depicted in the lower right part.

Note that the choice of a judicious value for k is crucial, since it will dictate the level of backtracking allowed in the search process. A value of zero means that no backtracking should be performed at all, meaning that the mapping ϕ is built by simply collecting machine-worker pairs that are all compatible with each other, in a somewhat greedy manner. When k is set to a value at least equal to that of all machines (or all workers), the backtracking is exhaustive, since in that case, all the couples forming a mapping under construction can be completely swapped away and replaced in the search process.

While the k-swap-stable notion allows to elegantly *characterize* interesting solutions to DAP instances, it does not describe how such solutions should be *computed* in practical situations. Indeed, computing *all* the possible k-swaps of a mapping ϕ under construction can quickly become intense in itself. To tackle this, we will develop hereunder an algorithm that *approximates* k-swap-stable solutions, by incorporating one greedy choice when having to chose which couple could be added to ϕ next.

The resulting simple algorithm can be formulated as follows. We start by initializing the mappings $(\phi, \psi) = ([], [])$. We will then iteratively try and add the *most promising* couple (m, w) to ϕ. Such a most promising couple is defined as the couple introducing as few injectivity conflicts as possible w.r.t. all the other possible couples (that are not yet in ϕ). If (m, w) can readily be added to

ϕ without conflict, then we update $\phi = \phi \cup \{(m, w)\}$ and continue the algorithm. If (m, w) introduces conflicts in ϕ, we remove the conflicted couples S from ϕ; if $|S| \leq k$ then we examine if there exists a set of available couples C that can replace S in ϕ, that is $|C| = S \wedge \phi' = \phi \setminus S \cup C$ is a k-swap of $\phi \wedge \phi' \cup \{(m, w)\}$ is a valid mapping. The search is performed using a queue to allow backtracking on potential ex aequo candidates (both for the choice of (m, w) and for the set C to be swapped with S). If at one point no such k-swap can be found, then the current search branch is pruned. (A formalization of the algorithm depicted in the lines above can be found in [14].)

Even if the k-swap mechanism does not *guarantee* convergence to a global optimum, it represents a promising heuristic for tempering the inherent computational demands of DAP while allowing to navigate in and out of local optima (depending on the maximal size of swaps k). In the following section, we develop an implementation of the algorithm sketched above, and we give some preliminary results relative to its performance.

4 Experimental Results

The fact that our k-swap approximation performs close to a polynomial value (regarding the input sizes of $\mathcal{M}$ and $\mathcal{W}$) has been demonstrated before [15], but empirical results were still needed to validate the use of k-swap strategies for general instances such as those considered in (large instances of) DominAP. To that aim, we now evaluate our k-swap heuristic on concrete DAP instances. To benchmark it in a relatable manner, we have implemented a brute force algorithm (which generates all possible mappings and selects the best pair), a greedy algorithm (being the algorithm that systematically selects *a* couple readily compatible with the mapping ϕ under construction that shows as few conflicts as possible with the remaining potential couples) as well as a well-known swarm-based approximation algorithm technique [6], where the size of the swarm of particles depends on the number of input dominos. For each approach, we conducted 44760 automatically generated tests, using each time up to 250 randomly generated *items*, an item being either a machine, a worker, a server, or a program, with the servers and programs being limited to 8 different values each. Recall that, while these might represent small real-life instances of DAP, the number of possible pairs of mappings follows a combinatorial growth that can heavily vary (even for two instances of similar size) due to the disparity of types and server-program associations. Each of our test classes is therefore characterized by a range of such possible matching combinations. As for the value of the parameter k, we have considered the candidate values $\{1, 2, 4, 8\}$, following the approach of the k-swap experimentation described in [14]; these values correspond to a small ($k = 1$) to considerable ($k = 8$) level of backtracking in regard to the number of items appearing in the test cases; indeed, using a certain value of k allows to swap up to k couples *at each iteration*, which can quickly lead to an important amount of computational work, so that higher values of k than those selected above can quickly render the algorithm as little efficient as the

brute force approach. This, of course, depends on the structural properties of the inputs used in the algorithm, and the value of k should systematically be carefully chosen after observing the particulars of the application in which it is used. However, determining *the* best value for k statically is obviously a hard task. Therefore, our implementation dynamically updates the value of k when such a change allows finding a better solution promptly.

We executed the battery of tests on an Apple M2 Max chip with 32 GB of RAM and running on macOS Sonoma 14.6. The raw results are shown in Fig. 2, with the left part depicting mean execution times and the right part plotting the size of the outputted mappings. The values on the right of each graph are the number of servers/programs allocated to each machine/worker present in the instance; these values can be understood as the different test classes mentioned above. For more detailed results and explanations, we refer to the Java implementation available online [13].

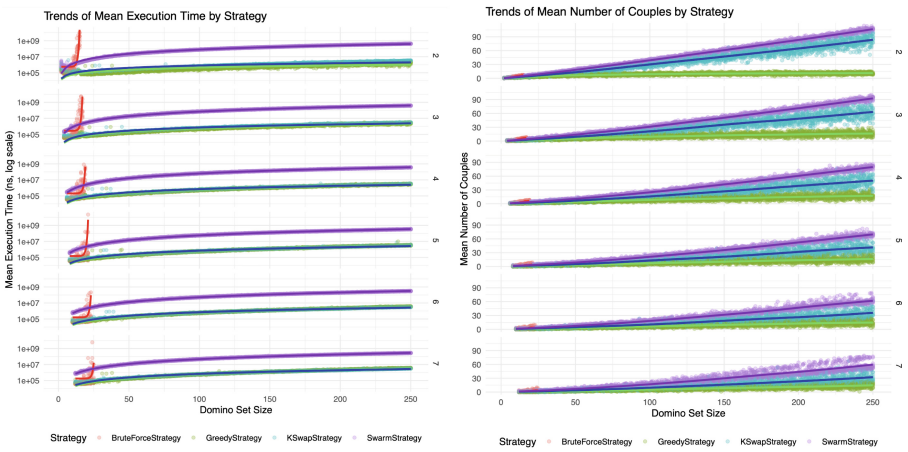

Fig. 2. A comparison of techniques used to solve or approximate the solution of DAP instances: execution time (left) and candidate solution (ϕ) size (right)

Interestingly, as the figure shows, the k-swap routine operates with execution times comparable to the naive greedy approach, while producing solution mappings ϕ with sizes that are intermediate between those obtained by the swarm approach (the closest to the optimal size) and the greedy approach (often sub-optimal by design). Note that for large instances, the brute force method is infeasible due to combinatorial explosion – hence its absence in some plots. As for the swarm strategy, albeit being the slowest (except brute force) in execution time, it proves to be effective in terms of solution size – paving the way for searching a combination of its features and that of k-swap-based approaches.

5 Discussion and Future Work

In this paper, we have defined and formalized the Double Assignment Problem (DAP), a variant of the classical assignment problem that, to the best of our knowledge, had not been formally studied before. Then, we introduced an approximation scheme called k-swap-stability and have demonstrated its applicability in approximating DAP instances. To verify the relevance of this approach, we developed a k-swap-based routine. Our experimental results, based on a synthetic benchmark of randomly generated instances, have shown that integrating k-swap logic led to significant improvements in efficiency and scalability, particularly in the handling of complex scenarios (i.e. where ϕ and ψ admit many potential combinations). We suspect that the strategy can therefore help in several areas involving (implicit) DAP instances such as supply chain handling, puzzle solving and resource management, to name a few. A thorough study of the occurrences of DAP instances in concrete situations is left for future work.

An interesting alternative formulation of DAP has, in fact, emerged in the context of syntactical anti-unification within (Inductive) Logic Programming [14], where the purpose is to compute, given two sets of Prolog-like atoms, their *most specific generalization*. It turns out that such generalizations involve an underlying injective mapping from the variables appearing in one atom to the variables of the other, parallel to another mapping that needs to be found among the atoms themselves. Apart from this exact correspondence to our DAP formulation, existing approaches in the field of optimization techniques in bipartite graphs did not yet consider instances of what we called the DAP problem. However, some bodies of research did tackle similar problems, in the sense that these also involve finding a matching in the presence of a few additional constraints.

Let us first get back to the Initial Assignment Problem (IAP). IAP involves finding a *maximum weight matching* in a bipartite graph containing tasks and agents, where the sum of the selected edges' weights (being integral numbers) is to be maximized. This problem has been classically resolved in $\mathcal{O}(n^3)$ using the Hungarian method, with n representing the number of vertices in the larger of the two input sets [7]. The Fractional Assignment Problem extends IAP to allow fractional task assignments across multiple agents. It is also solvable by known polynomial-time routines [9]. In contrast, the Generalized Assignment Problem, where each agent has specific capacities and associated costs or benefits, is generally NP-hard [3]. Another intractable version, the Quadratic Assignment Problem, allows for quadratic weights instead of integers only; it is typically approximated by heuristics or evolutionary algorithms [10], much like the *Fuzzy* [4] or *Multi-objective* [12] variants, to name only two of many IAP tweaks.

In future work, we aim to build on the promising results of our implementation by examining its scalability. This will involve incorporating confidence intervals and statistical significance tests in our performance comparisons, and expanding our benchmarks with real-world datasets from domains such as logistics, cloud computing, and network allocation. We also plan to explore advanced optimization strategies, such as hybridizing the k-swap heuristic with swarm optimization or integrating machine learning tools into the implementation.

Extending our approach to new scenarios and investigating variations in the injectivity constraints as they arise in real-world applications will also be key areas of future research. This should help clarify how subtle variations may impact both the problem's complexity and the quality of its solutions.

References

1. Alam, M., Mahak, Haidri, R.A., Yadav, D.K.: Efficient task scheduling on virtual machine in cloud computing environment. Int. J. Perv. Comput. Commun. **17**(3), 271–287 (2021). https://doi.org/10.1108/IJPCC-04-2020-0029
2. Casola, V., De Benedictis, A., Rak, M., Villano, U.: Security-by-design in multi-cloud applications: an optimization approach. Inf. Sci. **454**, 344–362 (2018). https://doi.org/10.1016/j.ins.2018.04.081
3. Desaulniers, G., Desrosiers, J., Solomon, M.M.: A branch-and-price algorithm for the generalized assignment problem. Oper. Res. **53**(3), 416–425 (2005). https://doi.org/10.1287/opre.1040.0179
4. Gurukumaresan, D., Duraisamy, C., Srinivasan, R., Vijayan, V.: Optimal solution of fuzzy assignment problem with centroid methods. Mater. Today Proc. **37**, 553–555 (2021). https://doi.org/10.1016/j.matpr.2020.05.582. International Conference on Newer Trends and Innovation in Mechanical Engineering
5. Ha, N.T., Akbari, M., Au, B.: Last mile delivery in logistics and supply chain management: a bibliometric analysis and future directions. Benchmarking Int. J. **30**(4), 1137–1170 (2023). https://doi.org/10.1108/BIJ-07-2021-0409
6. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings of ICNN 1995 - International Conference on Neural Networks, vol. 4, pp. 1942–1948 (1995). https://doi.org/10.1109/ICNN.1995.488968
7. Kuhn, H.W.: The Hungarian method for the assignment problem. Naval Res. Logist. Q. **2**(1–2), 83–97 (1955). https://doi.org/10.1002/nav.3800020109
8. Pais, S., Sousa, A.E.: Using an escape room activity to enhance the motivation of undergraduate life science students in mathematics classes - a case study. In: Proceedings of the 17th European Conference on Games Based Learning, vol. 17, no. 1 (2023). https://doi.org/10.34190/ecgbl.17.1.1431
9. Shigeno, M., Saruwatari, Y., Matsui, T.: An algorithm for fractional assignment problems. Discret. Appl. Math. **56**(2), 333–343 (1995). https://doi.org/10.1016/0166-218X(93)00094-G. Fifth Franco-Japanese Days
10. Silva, A., Coelho, L.C., Darvish, M.: Quadratic assignment problem variants: a survey and an effective parallel memetic iterated tabu search. Eur. J. Oper. Res. **292**(3), 1066–1084 (2021). https://doi.org/10.1016/j.ejor.2020.11.035
11. Syslo, M.M.: The subgraph isomorphism problem for outerplanar graphs. Theoret. Comput. Sci. **17**(1), 91–97 (1982). https://doi.org/10.1016/0304-3975(82)90133-5
12. Tailor, A.R., Dhodiya, J.M.: Multi-objective Assignment Problems and Their Solutions by Genetic Algorithm, pp. 409–428. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72929-5_19
13. Vandeloise, M., Yernaux, G., Barkallah, M., Vanhoof, W., Jacquet, J.M.: Implementation of the k-swap heuristic in java: code and documentation (2024). https://github.com/Vdloisem/DAPDominoGame
14. Yernaux, G., Vanhoof, W.: Anti-unification in constraint logic programming. Theory Pract. Logic Program. **19**(5–6), 773–789 (2019). https://doi.org/10.1017/S1471068419000188

15. Yernaux, G., Vanhoof, W.: Anti-Unification of Unordered Goals. In: Manea, F., Simpson, A. (eds.) 30th EACSL Conference on Computer Science Logic. Leibniz International Proceedings in Informatics, vol. 216, pp. 37:1–37:17. Schloss Dagstuhl (2022). https://doi.org/10.4230/LIPIcs.CSL.2022.37