TU WIEN Informatics

Bayessche Hyperparameter Optimierung mit ALASPO und Anwendungen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Dave Pfliegler, BSc
Matrikelnummer 11811384

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter
Mitwirkung: Projektass. Dipl.-Ing. Tobias Geibinger, BSc

Wien, 2. Mai 2025

Dave Pfliegler	Thomas Eiter

TU WIEN Informatics

Bayesian Hyperparameter Optimization with ALASPO and Applications

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Dave Pfliegler, BSc
Registration Number 11811384

to the Faculty of Informatics

at the TU Wien

Advisor: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter
Assistance: Projektass. Dipl.-Ing. Tobias Geibinger, BSc

Vienna, May 2, 2025

Dave Pfliegler	Thomas Eiter

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dave Pfliegler, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 2. Mai 2025

Dave Pfliegler

Danksagung

Ich möchte mich bei meinen Betreuern Prof. Thomas Eiter und Dipl.-Ing. Tobias Geibinger für ihre Unterstützung und ihr Feedback während dieser Arbeit sowie für die Möglichkeit, in ihrem Forschungsgebiet zu arbeiten, bedanken.

Weiters möchte ich mich bei meinen Kollegen Bini, Peter und David für ihre wertvollen Beiträge, ihre Unterstützung und ihre Freundschaft nicht nur während der Entstehung dieser Arbeit, sondern während unserer gesamten gemeinsamen Studienzeit bedanken.

Abschließend möchte ich meiner Familie für ihre stetige und überwältigende Unterstützung während meines Studiums danken.

Acknowledgements

I would like to thank my advisors Thomas Eiter and Dipl.-Ing. Tobias Geibinger for their support, feedback and guidance throughout this thesis and for the opportunity to work in their field of research.

I would also like to thank my close colleagues Bini, Peter, and David for their valuable input, support, and friendship, not only during the writing of this thesis, but throughout the entire time we spent studying together.

Finally, I would like to thank my family for their immense support and encouragement throughout my academic pursuits.

Kurzfassung

Das ALASPO Framework wendet Large Neighbourhood Search (LNS) auf Answer Set Programming (ASP) an und hat sich bereits bei der Lösung verschiedener Optimierungsprobleme bewährt. Um das Verhalten der LNS-Schleife zu steuern, bietet ALASPO eine Vielzahl von Konfigurationsoptionen, die in Kombination mit allen möglichen Parametern des zugrundeliegenden ASP-Solvers zu einem riesigen Konfigurationsraum führen.

Um diese Problematik zu adressieren, integrieren wir das Framework für Bayessche Optimierung *SMAC3* in ALASPO, um ein automatisches Tuning von ALASPO Konfigurationen zu ermöglichen. Weiters gruppieren wir die bereitgestellten Instanzen anhand ihrer Eigenschaften, so dass wir mehrere optimale Konfigurationen erhalten, eine für jede Gruppe ähnlicher Instanzen. Das Ergebnis des Tunings kann dann für zukünftige ALASPO Durchläufe mit bisher unbekannten Instanzen verwendet werden, um automatisch eine geeignete Konfiguration basierend auf den Eigenschaften der jeweiligen Instanz auszuwählen und anzuwenden.

Wir evaluieren unsere Methodik anhand von vier Benchmark-Problemen und zeigen, dass unser Tuning bei einigen dieser Probleme zu signifikanten Verbesserungen gegenüber der Standardkonfiguration führen kann, während wir ebenso ihre Grenzen aufzeigen, wenn keine ausreichenden Verbesserungsmöglichkeiten vorhanden sind.

Abstract

The ALASPO framework applies Large Neighbourhood Search (LNS) to Answer Set Programming (ASP), and has proven to be effective in tackling a number of different optimization problems. To control the behaviour of the LNS loop, ALASPO offers many configuration options, that, when combined with all of the possible solving and grounding parameters of the underlying ASP solver, result in a vast configuration space.

To address this challenge, we integrate the SMAC3 framework for Bayesian Optimization into ALASPO to enable the automatic tuning of ALASPO configurations. Additionally, we cluster the provided instances based on their characteristics, allowing us to obtain multiple optimal configurations, one for each cluster of similar instances. After tuning, the result can be used for future ALASPO runs on unseen instances to automatically select and apply an appropriate configuration based on the characteristics of the given instance.

We evaluate our methodology on a set of four benchmark problems and show that tuning can, for some of them, provide significant improvements over using the default configuration, while also demonstrating its limitations whenever there are insufficient opportunities for advancement.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Research Question . . . 2
1.2 Contributions and Findings . . . 2
1.3 Organization . . . 3

2 Background **5**
2.1 Answer Set Programming . . . 5
2.2 Large Neighbourhood Search . . . 9
2.3 Hyperparameter Optimization . . . 14

3 Integrated Automated Tuning in ALASPO **19**
3.1 Overview . . . 19
3.2 Feature Extraction . . . 20
3.3 Clustering . . . 21
3.4 Implementation . . . 21

4 Experiments **29**
4.1 Experimental Setup . . . 29
4.2 Hypotheses and Methodology . . . 30
4.3 Test Laboratory Scheduling Problem (TLSPS) . . . 31
4.4 Valves Location Problem (VLP) . . . 37
4.5 Travelling Salesman Problem (TSP) . . . 42
4.6 Multi-Agent Path Finding (MAPF) . . . 46
4.7 Discussion . . . 52

5 Related Work **55**

6 Conclusion and Future Work **57**

Overview of Generative AI Tools Used **59**

List of Figures **61**

List of Tables **63**

Bibliography **65**

Appendix **73**

CHAPTER 1

Introduction

Answer Set Programming (ASP) [MT98, EIK09, BET11, Lif19, GKKS17] is a declarative paradigm rooted in logic programming, and is, unlike procedural languages, not used to encode a problem solving algorithm, but rather the problem itself. ASP in combination with the Large Neighbourhood Search (LNS) [Sha98, PR10] metaheuristic, which iteratively destroys and repairs parts of a solution to find an overall better result, specifically the Adaptive Large-Neighbourhood ASP Optimiser *ALASPO* [EGH+22], has already proven to be effective in tackling typical optimization problems [EGH+22, EGH+24]. Such optimization problems involve various criteria that contribute to an objective value which reflects the quality of candidate solutions. These problems are then typically optimized by iteratively finding multiple solutions, each better than its predecessor with an objective value closer to the optimum.

ALASPO uses an ASP solver internally to facilitate both the finding of an initial solution as well as the repair step of the LNS. This solver offers an extensive range of solving and grounding parameters that can greatly influence its performance. In addition, ALASPO offers numerous configuration options that control the behaviour of the LNS process, together resulting in a considerable configuration space [EGH+22]. Consequently, any exhaustive exploration of different parameters and combinations thereof, to identify the optimal configuration, is infeasible by hand and calls for using an automated approach, namely Hyperparameter Optimization (HPO) [KJ95, KFS95]. HPO is the process of identifying the best performing parameters for a given function, by evaluating them on trials, thereby reducing the effort of manual experimentation, improving reproducibility, and potentially improving performance through more thorough exploration.

The topic of Hyperparameter Optimization has gained traction primarily due to the rise of machine learning algorithms and their performance being strongly dependent on the correct parameter choices [FH19, YS20]. The basic HPO forms of Grid Search and Random Search [BB12], which evaluate the function with all possible combinations of parameters or a random subset thereof, require an extensive number of trials. In contrast,

Bayesian Optimization [Moc94] takes advantage of a surrogate model and may reduce the number of required function evaluations by balancing both exploration and exploitation using learned knowledge from previous function calls [JSW98, FH19, BCdF10].

Given that Bayesian Optimization improves upon existing black-box HPO approaches, it has been investigated and successfully implemented in various diverse applications: For instance, in tuning machine learning models to predict the choice of work travel mode [AAJ+23], network intrusion detection [MSH+21], or brain tumor classification [AAXKM22]. Furthermore, it has been successfully used in tools like *auto-sklearn* [FKE+15] as well as *Auto-Pytorch* [ZLH21] [LEF+22].

1.1 Research Question

In the light of these successes, a natural issue is to provide HPO in ALASPO and to study whether similar benefits can be achieved for challenging problems of relevance in practice. This has been the motivation for this thesis, which proceeds along the following lines.

We integrate the Bayesian Optimization framework for Hyperparameter Optimization *SMAC3* [LEF+22] into ALASPO, to automatically find optimal configurations, eliminating the need for time-consuming and tedious manual trial runs. Choosing which aspects of an ALASPO configuration should be tuned is done by providing a custom configuration space, specifying the properties and different values that should be considered for them. Furthermore, we implement a clustering of instances based on their characteristics, allowing us to obtain multiple optimal configurations, one for each cluster, that exploit similarities between instances in order to improve overall performance. In addition, we allow the customization of the instance characteristics considered in this clustering, to best represent their features for a specific problem at hand. The result of a finished tuning process can then be used for future ALASPO runs on new instances to automatically select and apply an appropriate configuration based on the characteristics of the instance provided.

We subsequently investigate the effectiveness of our tuning methodology in finding optimal ALASPO selection strategies on a set of four different benchmark problems, using different clustering methods and varying the number of trials. The particular benchmark problems we choose for the evaluation are the Test Laboratory Scheduling Problem (TLSPS) [GMM21, MM18, MM21], the Valves Location Problem (VLP) [GS09, GS10, CGN+11], the Travelling Salesman Problem (TSP) [Men32, Rob49, DFJ54], and finally Multi-Agent Path Finding (MAPF) [SSF+21].

1.2 Contributions and Findings

Summing up, our contributions are briefly as follows:

(i) The design and integration of a customizable Bayesian Hyperparameter Optimization approach into the ALASPO framework using SMAC3, to find best performing configurations.

(ii) A customizable clustering of problem instances based on their characteristics, enabling the tailoring of configurations to homogeneous groups of instances, exploiting their similar optimization behaviour, for an overall heterogeneous set of instances.

(iii) An experimental evaluation of the effectiveness of the newly integrated features on a set of four benchmark problems of high industrial relevance.

(iv) Detailed discussions of the experimental results, including potential factors influencing the observed outcomes.

We show that our hyperparameter tuning is able to provide significant performance improvements for some problems, already with a small number of trials, while also demonstrating its limitations: For the TLSPS, tuning resulted in about 7% better objective scores on average, up to 22% for some instances. The VLP saw more moderate improvements, with an average of 5% better results, while a quarter of the instances still performed worse than the default configuration. Next, the TSP saw significant improvements, with average gains of 28%. Finally, tuning for the MAPF could not provide meaningful improvements over the default configuration, suggesting that the ALASPO baseline configuration is already best suited for this problem.

1.3 Organization

The rest of the thesis is structured as follows: In Chapter 2, we introduce the underlying concepts on which this work builds upon, followed by Chapter 3, where we present the automated tuning we integrate into the ALASPO framework, along with a closer examination of its core features and their implementation. Next, Chapter 4 contains the experimental evaluation of the newly implemented tuning on four different benchmarks, accompanied by a discussion of their results. Finally, in Chapter 5, we briefly present related works that use similar methodologies, and summarize our findings in Chapter 6.

CHAPTER 2

Background

In this chapter, we introduce the underlying concepts this work builds upon, namely, the notion of Answer Set Programming, Adaptive Large Neighbourhood Search, and Hyperparameter Optimization.

2.1 Answer Set Programming

Answer Set Programming (ASP) [MT98, EIK09, BET11, Lif19, GKKS17] is a declarative problem solving paradigm, rooted in logic programming, knowledge representation, and reasoning. Unlike procedural programming languages, ASP does not encode a problem solving algorithm, but rather the problem itself. Given an encoding and an instance of a problem, an ASP solver computes models, so-called *answer sets*, that represent solutions to the particular problem [Lif19, BET11].

2.1.1 Logic Programs

A disjunctive logic program is a finite set of rules following the form:

$$a_1 \vee \cdots \vee a_m \leftarrow b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_n \tag{2.1}$$

where $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$ are *atoms*, referred to as the head and body of the rule, respectively. *Literals* are *atoms* or their negation, as indicated by the keyword **not** preceding $b_{k+1}, \ldots, b_n$, making $not\, b_{k+1}, \ldots, not\, b_n$ *negated literals* and $b_1, \ldots, b_k$ *positive literals* [EIK09].

Atoms are of the form:

$$p(t_1, \ldots, t_n) \tag{2.2}$$

where p is a predicate symbol with arity $n \geq 0$ and $t_1, \ldots, t_n$ are *terms*, which are either constants or variables.

A rule is called (i) *fact* if its body is empty ($n = 0$), (ii) *non-disjunctive* if its head contains only one literal ($m = 1$), (iii) *basic* if there are no negated literals in its body and its head is not empty ($k = n,\ m \geq 1$), (iv) *normal* if it is non-disjunctive and contains no strong negation, and (v) *horn* if it is basic and normal. Additionally, a rule is *ground*, if all of its literals are *ground*, i.e., do not contain any variables.

A rule without a head, such as:

$$\leftarrow a, b \tag{2.3}$$

acts as a *constraint*, disallowing any solution (answer set) containing both a and b. Such constraints are a central and widely used construct in ASP.

Rule Semantics A rule of a logic program can be interpreted as implication: where rule 2.1 is similar to the following implication:

$$(b_1 \wedge \cdots \wedge b_k \wedge not\, b_{k+1} \wedge \cdots \wedge not\, b_n) \implies (a_1 \vee \cdots \vee a_m) \tag{2.4}$$

and can be intuitively read as: *If the body of a rule is true, then the head of the rule must be true as well* [BET11].

An example of a rule containing variables:

$$\text{canFly}(X) \leftarrow \text{bird}(X), not\, \text{penguin}(X) \tag{2.5}$$

can be interpreted as: *If X is a bird and it is not known that X is a penguin, then X can fly.*

Negation The already mentioned keyword **not** denotes the so-called *default negation*, which means that an atom is *not derivable*. In contrast, the so-called *strong negation*, denoted by the usual symbol $\neg$, holds, if the negation of an atom is derivable. Since any program containing *strong negation* can be restated as one without, by introducing a new atom in place of the strongly negated literal, it is not a strictly necessary construct. It is, however, useful in order to express defaults, as in the following example, which models inertia:

$$\text{stationary}(T+1) \leftarrow \text{stationary}(T), not\, \neg\text{stationary}(T+1). \tag{2.6}$$

which can be interpreted as: *If some object is stationary at time T, then it is also stationary at time $T+1$ if we can not derive that the opposite is true (i.e., that it is moving).* It is thus a straightforward way of solving the frame problem [BET11].

2.1.2 Answer Sets

An *answer set* is an *interpretation* M of a *ground* logic program P that is a *minimal model* of the *Gelfond-Lifschitz reduct* P^M of P.

To elaborate: An *interpretation* M is a set of ground literals that is *consistent*, meaning that it does not contain both an atom a and its negation $\neg a$. A logic program is

ground if all of its rules are. An interpretation M is a *model* of program P iff for every rule, whenever the body is *true* ($b_1, \ldots, b_k \in M$ and $b_{k+1}, \ldots, b_n \notin M$), its head is too ($\{a_1, \ldots, a_m\} \cap M \neq \emptyset$). A *model* for P^M is *minimal*, if there exists no subset $N \subset M$ which is also a *model* of P^M. The *Gelfond-Lifschitz reduct* [GL88] P^M of program P and interpretation M is defined as follows:

$$P^M = \Big\{ a_1 \vee \cdots \vee a_m \leftarrow b_1, \ldots, b_k \mid (a_1 \vee \cdots \vee a_m \leftarrow b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_n) \in P, \{b_{k+1}, \ldots, b_n\} \cap M = \emptyset \Big\} \quad (2.7)$$

Which means that P^M is *positive*, i.e., does not contain any default negation.

Example 1 Following example shows how a model could be found for a very simple program.

$$a \leftarrow \quad (2.8)$$
$$b \leftarrow a \quad (2.9)$$
$$c \leftarrow not\, b \quad (2.10)$$

Since the first rule is a fact (i.e., contains no body), `a` *must be contained in the resulting answer set. Given that the body of the second rule is now true,* `b` *must also be part of the model. Now, since* `b` *is derivable,* `not b` *cannot be true, making the body of the third rule false and consequently* `c` *not part of the solution. Thus, the resulting model is* `{a,b}`.

2.1.3 Extensions

There are several extensions to ASP that simplify or increase its modelling capabilities and expressiveness, such as allowing *disjunction* in the head of rules (see 2.1). It affects the notion of answer sets, since the Gelfond-Lifschitz reduct could now contain rules with such disjunctions, potentially resulting in multiple minimal models. Consequently, an answer set is now defined as being one of these minimal models [BET11].

Another often used construct are the so-called *choice rules* of the form:

$$l \diamond \{a_1; \ldots; a_n\} \diamond u \leftarrow b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_n \quad (2.11)$$

which express that if the body is true, any number of atoms of $\{a_1, \ldots, a_n\}$ should be included in the resulting answer set, the number of which is determined by the cardinality constraints l and u. Where the operators $\diamond$ can be any of the comparisons in $\{<, \leq, >, \geq, =, \neq\}$. More commonly, *conditional* choice rules of the following form are used:

$$l \diamond \{a_1 : L_1; \ldots; a_n : L_n\} \diamond u \leftarrow b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_n \quad (2.12)$$

where $L_1, \ldots L_n$ are each a list of multiple literals, acting as conditions possibly restricting the set $\{a_1, \ldots, a_n\}$ of which atoms are to be chosen from [CFG+19, GKKS12, EGH+24].

Additionally, ASP solvers like *clingo* [GKKS17] support *aggregates* such as $\#count$, $\#sum$, $\#min$ and $\#max$ of similar form:

$$\alpha\ \{t_1 : L_1; \ldots; t_n : L_n\} \tag{2.13}$$

where $t_1, \ldots, t_n$ are terms, $L_1, \ldots, L_n$ are each a list of literals and α is one of the aggregate functions. Note that the chosen function will be applied to the *set* of terms that are produced by their corresponding conditions, which means that duplicates will be disregarded [Unib, GKKS12].

To facilitate optimization in ASP based on an objective value, so-called *weak constraints* are available in modern solvers. They are of the form:

$$:\sim b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_n.\ [w@l, t] \tag{2.14}$$

where t is a tuple of terms, w is the weight (or cost) that the tuple t contributes to a cost function, and $@l$ is an optional level, defaulting to 0, that can be used to prioritize the optimization of some objectives over others (higher level equals higher priority) when using multiple weak constraints. The ASP solver clingo provides the shorthand:

$$\#minimize\{w_1@l_1, t_1 : L_1, \ldots, w_n@l_n, t_n : L_n\} \tag{2.15}$$

where $L_1, \ldots, L_n$ are each a list of literals. These directives represent n regular weak constraints 2.14 [CFG+19, Unib, EGH+24].

2.1.4 Computing Answer Sets

To compute answer sets for a given problem, ASP solvers will start by first *grounding* the program and subsequently *solving* it.

Grounding The ground program $grnd(P)$ of P is obtained by essentially instantiating each rule with every possible combination of constants found in P. The set of these constants is called the *Herbrand Universe* $HU(P)$. However, following this approach naively can result in unreasonably large groundings, which is why modern ASP solvers implement many optimizations to produce a smaller program without altering its answer sets [BET11].

Solving Next, the ASP solver generates answer sets for the grounded program by exploiting methods borrowed from satisfiability solving [BET11].

Native ASP solvers use a backtracking approach similar to that of SAT solvers, while additionally verifying the *foundedness* of incumbent models. The notion of *foundedness* ensures that every atom can be derived by a rule of the program. Whenever a contradiction occurs or the current model is not *founded*, the search will backtrack to correct earlier decisions which proved to be wrong [BET11].

There exist other ASP solvers that are based on actual reductions to SAT. Consequently, models of the corresponding SAT problem are answer sets of the initial problem [BET11].

2.1.5 Modelling

When modelling a problem in ASP, a useful approach is the so-called *guess-and-check* method: Generate (*guess*) multiple candidate solutions to the problem at hand by using nondeterministic features such as disjunctive rule heads or choice rules 2.12. Next, *check* said candidates by introducing additional rules and constraints that discard illegal solutions [EIK09].

Example 2 Following example, as described by Eiter et al. [EIK09], demonstrates the *guess-and-check* methodology using the 3-colour problem:

$$r(X) \vee g(X) \vee b(X) \leftarrow node(X) \tag{2.16}$$
$$\leftarrow r(X), r(Y), edge(X,Y) \tag{2.17}$$
$$\leftarrow g(X), g(Y), edge(X,Y) \tag{2.18}$$
$$\leftarrow b(X), b(Y), edge(X,Y) \tag{2.19}$$

where instances are expected to have the facts node(n) for $n \in V$ and similarly edge(u, v) for $(u, v) \in E$ for a given graph $G(V, E)$. Rule 2.16 uses disjunction to produce the candidate solutions by colouring each node, while Rules 2.17-2.19 discard candidates containing equally coloured neighbouring nodes [EIK09].

2.2 Large Neighbourhood Search

Large Neighbourhood Search (LNS) [Sha98, PR10] is a local search heuristic often used in optimization problems. It was first described by Shaw [Sha98] as continuous relaxation and re-optimization and proceeds as follows:

First, an initial solution to the current problem must be obtained, for example, by using a construction heuristic that produces a suboptimal or even trivial solution. Next, a subset, called the neighbourhood, of the current incumbent solution is destroyed (or relaxed). The resulting partial solution is then rebuilt in an attempt to find a new feasible solution of better quality. If this new solution is closer to the optimum, it becomes the new incumbent (otherwise it is discarded). Subsequently, the destroy and repair operations are repeated in a loop until some stopping criterion is met or an optimum is found [PR10, Sha98].

Naturally, the effectiveness of the LNS optimization is highly dependent on the neighbourhood selection: If it is too small, the search may get stuck in a local optimum due to its limited exploration of the search space; if it is too large, the heuristic may repeat certain optimizations at each step, resulting in poor efficiency or poor solution quality [PR10].

2.2.1 Adaptive Large Neighbourhood Search

Adaptive Large Neighbourhood Search (ALNS) [RP06] extends regular LNS by allowing for multiple destroy and repair operators in one search. The operators are dynamically

selected throughout the optimization based on their associated weight, which is continuously updated after each iteration based on their performance, e.g., success rate. The degree to which each operator's weight is affected after each iteration can be controlled by a *decay* parameter [PR10].

The basic idea of ALNS is therefore to favour good performing destroy and repair operators with a higher probability of being selected. To counteract a possible bias towards more complex operators due to their higher quality solutions, performance can be normalized by execution time [PR10].

2.2.2 ALASPO: Adaptive Large-Neighbourhood ASP Optimiser

The Adaptive Large-Neighbourhood ASP Optimiser *ALASPO*[1] is a system introduced by Eiter et al. [EGH+22, EGH+24] implementing ALNS for ASP optimization problems. It supports the ASP solver *clingo* and its extensions *clingcon* and *clingo-dl* of the Potsdam Answer Set Solving Collection *Potassco* [Unia].

ALASPO allows for a portfolio of different search and and neighbourhood (relax) operators, implements different operator selection strategies and can be configured in detail through a JSON configuration. Figure 2.1 by Eiter et al. [EGH+24] demonstrates the workflow and architecture of the ALASPO system.

Out-of-the-Box ALASPO can be used without the need for any configuration, by making use of a sensible default strategy and problem independent search and relax operators: By default, the *dynamic* strategy is used, with two neighbourhoods selecting random atoms or random constants with sizes [0.1, 0.2, 0.4, 0.6, 0.8] and [0.1, 0.2, 0.3, 0.5] respectively, where the sizes represent the percentages of total atoms (or constants) to relax, and a search operator with timeouts of [5, 15, 30, 60] seconds. The available strategies, relax and search operators are outlined in more detail in the following paragraphs.

For a particular problem with its ASP encoding written in a file `program.lp`, the following starts ALASPO:

```
> alaspo -i program.lp
```

The command-line interface also includes options to set the time limit, a seed, select the desired solver, and select one of the available neighbourhood types with corresponding relaxation rates, as well as search timeouts. For further customization and control of the LNS loop, a JSON configuration file can be passed to ALASPO [EGH+24].

Selection Strategies ALASPO's selection strategies determine which relax and search operators are chosen from the given portfolio at each iteration of the LNS loop. The following strategies are currently implemented:

[1] `https://gitlab.tuwien.ac.at/kbs/BAI/alaspo/-/commit/9f8d5492f249d7eef2917f2c8d5d583b2d5cb235`, accessed: April 15, 2025.

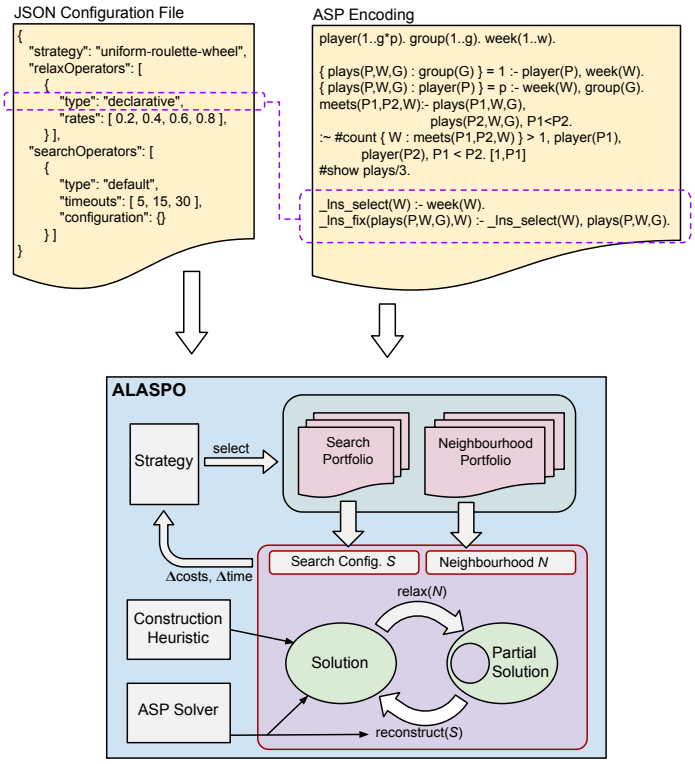


Figure 2.1: An overview of the ALASPO system by Eiter et al. [EGH+24].

(i) The **dynamic** strategy aims to increase relaxation rates and time limits to escape local optima during search. It does so using the *unsatStrikes* parameter, which dictates the number of times the solver must report unsatisfiability in order to switch to the next larger neighbourhood. The *timeoutStrikes* option specifies how often the solver should time out before either increasing the time limit or decreasing the size of the neighbourhood, the choice of which is decided by coin flip. If it is not possible to increase the relaxation rate or time out, i.e. if they are already at their maximum, a new operator is chosen at random.

(ii) The **roulette-wheel** strategy assigns each pair of relax and search operators the same initial weight. At the end of each iteration of the LNS loop, the currently used operators' weight is updated based on the formula $w_{new} = (1-\alpha) \cdot w_{old} + \alpha \cdot r$, where r is the improvement in objective score over the operators' runtime, and α is a parameter controlling the *learning* rate. The weight of an operator pair determines its probability of being selected in future iterations.

(iii) The **uniform roulette-wheel** or **random** strategy selects operator pairs with equal probability without adapting their weights.

(iv) Finally, a novel **power law** strategy samples operators according to a power law

distribution based on their indices: Each i-th operator is assigned a weight $w_i = \frac{1}{i^\alpha}$, which is then normalized to form the probability distribution. The parameter α is set to 1.5 by default, favouring lower indexed operators, i.e., typically smaller neighbourhoods with shorter time limits. Negative α values can be used to invert this trend.

Note that it is easy to implement new custom strategies by extending the abstract class `AbstractStrategy` in Python code [EGH+24].

Neighbourhood Types ALASPO offers two predefined neighbourhood types that are problem independent:

(i) The **random-atoms** neighbourhood chooses a random set of the visible atoms to relax. By default, clingo treats all atoms as visible; however, the *#show* directive can be used to manually specify which atoms should be visible.

(ii) The **random-constants** neighbourhood picks a random sample from all constant symbols present in visible atoms and relaxes every atom containing any of the selected constants.

Random-atoms can be quite effective, but may struggle whenever there are a lot of dependencies between different atoms. The *random-constants* neighbourhood tries to combat this by relaxing groups of related atoms sharing the same constant.

In addition, ALASPO allows user-defined neighbourhoods as part of the problem encoding, so-called **declarative** neighbourhoods, using the predicates `_lns_select/1` and `_lns_fix/2`:

The `_lns_select/1` predicate can be used to specify a set of terms S from which a subset is selected to be relaxed in each iteration of the LNS loop.

The `_lns_fix/2` predicate is used to provide a mapping from the terms of S to atoms that should be fixed under assumptions during solving. The first argument is the atom to be fixed, the second is the corresponding term of S.

For every term $t \in S$ that is not to be relaxed in the current iteration, all atoms of the current incumbent identified by the `_lns_fix/2` predicate that contain any of the terms t are fixed, i.e., provided to the solver and assumed to be *true* (meaning they must be included in the solution), destroying the rest of the incumbent. Consequently, this results in the relaxation of every selected term $s \in S$ and its corresponding atoms (along with all other non-fixed atoms).

Figure 2.1 shows the use of a declarative neighbourhood in the JSON configuration file, and highlights the corresponding `_lns_select/1` and `_lns_fix/2` predicates in the problem encoding: The encoding depicted belongs to the Social Golfer Problem (SGP), where a number of golfers must be grouped into g groups, each containing p players, over

```
{
    "strategy": {
        "name": "dynamic",
        "unsatStrikes": 3,
        "timeoutStrikes": 1
    },
    "relaxOperators": [
        {
            "type": "randomAtoms",
            "sizes": [ 0.1, 0.2, 0.4, 0.6, 0.8 ]
        },
        {
            "type": "randomConstants",
            "sizes": [ 0.1, 0.2, 0.3, 0.5 ]
        }
    ],
    "searchOperators": [
        {
            "name": "default",
            "timeouts": [ 5, 15, 30, 60 ]
        }
    ]
}
```

Figure 2.2: The default portfolio of ALASPO.

a period of w weeks, such that no two players are placed in the same group more than once. The two special predicates at the end of the encoding are then used to define whole weeks as neighbourhoods for ALASPO.

Finally, custom neighbourhoods can also be implemented in Python code by extending the abstract class `AbstractRelaxOperator`, which allows defining specialized and possibly more complex neighbourhoods that would otherwise be difficult or impossible to realize as part of the encoding [EGH+24].

Advanced Configuration A portfolio of different relax and search operators can be specified trough a JSON configuration file: The default configuration is depicted in Figure 2.2. A search and neighbourhood portfolio can each contain multiple different relax and search operators, each with distinct relaxation rates and timeouts.

In addition, search operators can contain a configuration of solver specific options that should be applied to the selected ASP solver during search. Furthermore, a so-called initial operator can be provided as part of the configuration, which can be used to let the ASP solver pre-optimize the problem for a given time before the main LNS loop starts. A solver configuration can be applied to the initial operator as well. Both of these features are demonstrated in Figure 2.3.

For further details on ALASPO, we refer to Eiter et al. [EGH+24, EGH+22].

```
{
    /* ... */
    "searchOperators": [
        {
            "name": "default",
            "timeouts": [ 5, 15, 30, 60 ],
            "configuration": {
                "solver": {
                    "opt_strategy": "bb,lin"
                },
                "configuration": "handy"
            }
        }
    ],
    "initialOperator": {
        "timeout": 60,
        "configuration": {
            "solver": {
                "opt_strategy": "usc,3"
            },
            "configuration": "jumpy"
        }
    }
}
```

Figure 2.3: An excerpt of an ALASPO configuration file containing ASP solver specific options and an initial operator.

2.3 Hyperparameter Optimization

The topic of *Hyperparameter Optimization* (HPO) [KJ95, KFS95] has gained traction primarily due to the rise of machine learning algorithms and their performance being strongly dependent on the correct parameter choices [FH19, YS20]. It is the process of identifying best performing parameters for a given function by evaluating them on trials. Automating this procedure is a natural approach to reduce the effort of manual experimentation, improve reproducibility, and possibly improve the performance of the examined function by exploring the configuration space more thoroughly.

The basic forms of HPO are the well known Grid Search and Random Search [BB12]. The former evaluates the given function with all possible parameter combinations, resulting in a considerable number of required trials. This is especially true for complex configuration spaces, since increasing the dimensionality of the space leads to an exponential growth in the number of function evaluations. The latter, samples the configuration space at random until a given budget is exhausted, and is proven to work better than Grid Search for applications where the performance impact of parameters is very uneven [FH19, BB12].

The field of Algorithm Configuration (AC) – which focuses on finding the correct parameter settings, i.e., configurations, for a given algorithm on a set of different problem instances – has developed methods to effectively solve AC scenarios: Racing algorithms evaluate multiple candidate configurations sequentially, and discard worse performing ones as

soon as they fall too far behind. Stochastic Local Search (SLS) [HS15] algorithms use local search in the configuration space, and lastly, Sequential Model-Based Optimization (SMBO), adopted by SMAC3 2.3.1, uses an internal model to estimate the performance of parameter settings [Hoo12].

2.3.1 Bayesian Optimization

Bayesian Optimization (BO) [Moc94] balances both exploration and exploitation of the search space by using learned knowledge from previous function calls and evaluating the most promising next trial at each iteration, thus possibly reducing the overall number of required function evaluations [FH19, BCdF10].

To do so, it uses a surrogate model in combination with an acquisition function: At each iteration, the model is fitted to the previously made observations. The acquisition function is an easy to optimize estimate of the utility of points that can be sampled, and is used to find the best possible trial to evaluate next [FH19, BCdF10].

Example Scenario Figure 2.4 shows a Bayesian Optimization of a simple one-dimensional example function, with the intention of finding its optimum: The dashed line is the actual objective function, while the solid line is the surrogate function, which represents the current estimate of the objective function with its uncertainty highlighted in blue. The acquisition function is superimposed at the bottom of each plot in green; its maximum determines the next value to be evaluated. The example shows that the acquisition function estimates the utility of values around observations as low, while it is high if it either predicts a high objective (exploitation) or the uncertainty is high (exploration). However, at iteration $t = 4$, the acquisition function chooses a value close to other observations, but it does so as it correctly predicts that this will yield a new maximum [BCdF10, FH19, SSW$^+$16].

The mean and uncertainty estimates are referred to as the *posterior* because they are derived using Bayes' theorem, where the *prior* holds the beliefs or assumptions about the objective function, and the *posterior* represents the updated beliefs after incorporating the observations [BCdF10].

Surrogate Models Bayesian Optimization typically uses Gaussian Processes (GPs) [RW06] as surrogate models. A GP is a probabilistic distribution over functions that allows an acquisition function to rely on its mean and variance predictions. However, standard GPs do not scale well for high-dimensional data or large data sets. Therefore, other machine learning models can be adapted for use in place of GPs, due to their higher flexibility and scalability. One of which are Random Forests (RFs) [HHLB11]; they are fast, can cope with large and complex configuration spaces better than GPs [JAGS17], and scale better [FH19].

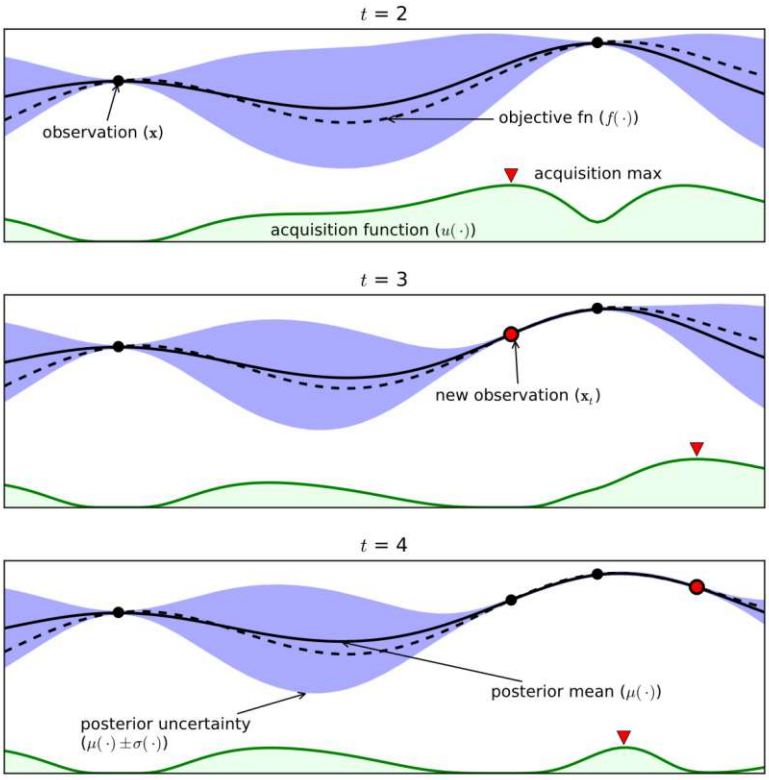


Figure 2.4: An example Bayesian Optimization of a one-dimensional function, by Brochu et al. [BCdF10].

Acquisition Functions Acquisition functions determine the points to evaluate during optimization and are thus crucial in determining the ratio of exploitation and exploration and its overall success. Following are some of the most prominent functions:

(i) **Probability of Improvement (PI)** selects trials based on the probability that the objective value will be greater than the current incumbent. It will, however, tend to strongly exploit already good solutions rather than explore uncertain regions [BCdF10, SSW+16].

(ii) **Expected Improvement (EI)** does balance exploitation and exploration by additionally accounting for the potential magnitude of improvement [BCdF10, SSW+16].

(iii) **Upper/Lower Confidence Bounds (UCB/LCB)** are metrics used for maximization and minimization, respectively: $\mathrm{UCB}(x) = \mu(x) + \kappa\sigma(x)$, $\mathrm{LCB}(x) = \mu(x) - \kappa\sigma(x)$, with κ controlling the balance between exploration and exploitation. Given $\kappa = 1$ they could be intuitively understood as the top and bottom edges of the blue shaded uncertainty drawn surrounding the posterior mean in Figure 2.4 [BCdF10, SSW+16].

SMAC3: A Versatile Bayesian Optimization Package for Hyperparameter Optimization SMAC3[2] [LEF+22] is a Bayesian Optimization framework for hyperparameter tuning written in Python. By default, it uses Random Forests as surrogate model for hyperparameter optimization, given the previously mentioned benefits over standard Gaussian Processes. It implements several so-called facades tailored to specific use cases, and allows for custom combinations of different modules. In addition, it is capable of optimizing a function based on a given set of instances, evaluating the performance of a configuration based on trials with multiple instances. To enable the parallelization of trials, SMAC3 can make use of custom DASK [Roc15] clients. DASK is a Python library that can be used to efficiently implement parallel computing; DASK clients can be passed to SMAC3 to parallelize its trials, including across HPC clusters [LEF+22].

To identify best performing parameter configurations, SMAC3 uses an aggressive racing algorithm [BSPV02, HHLBS09] that, if provided with multiple instances, discards less promising candidates as soon as possible by evaluating them on only a few instances [LEF+22].

SMAC3 has been successfully used in tools such as *auto-sklearn* [FKE+15] as well as *Auto-Pytorch* [ZLH21], and was part of a winning solution [ASD+20] for the 2020 NeurIPS Black-Box Optimization Challenge (BBO) [TEM+21] [LEF+22].

[2] `https://github.com/automl/SMAC3`, accessed: April 15, 2025.

CHAPTER 3

Integrated Automated Tuning in ALASPO

In this chapter we will describe the main ideas and building blocks of the tuning approach integrated into the ALASPO system.

3.1 Overview

In addition to the extensive range of solving and grounding parameters of the underlying ASP solver, ALASPO offers numerous configuration options that control the behaviour of its LNS process, both resulting in a considerable configuration space. Consequently, any exhaustive exploration of different parameters and combinations thereof, to identify optimal configurations, is infeasible by hand.

To address this, we integrate the SMAC3 2.3.1 framework for Bayesian Hyperparameter Optimization into ALASPO, to automatically tune ALASPO configurations for a varied set of instances. We we want to, among other things, allow a workflow where we have a small set of diverse instances for a problem, and supply them to the tuning process. ALASPO then clusters these instances based on their characteristics, and produces a best configuration for each. Furthermore, the finished tuning process produces an artifact that can be used for future invocations of ALASPO with new instances to automatically select a fitting configuration.

Figure 3.1 shows the general tuning approach in ALASPO: The tuning configuration file contains general settings such as the total trial budget, the time limit per run, the configuration space for the optimization, as well as the problem encoding and the instances to be considered. Optionally, a custom feature extractor or a custom DASK 2.3.1 configuration can be provided. ALASPO then uses its feature extractor (either the default or a custom one) to obtain a set of features for each instance. This dataset is then used to

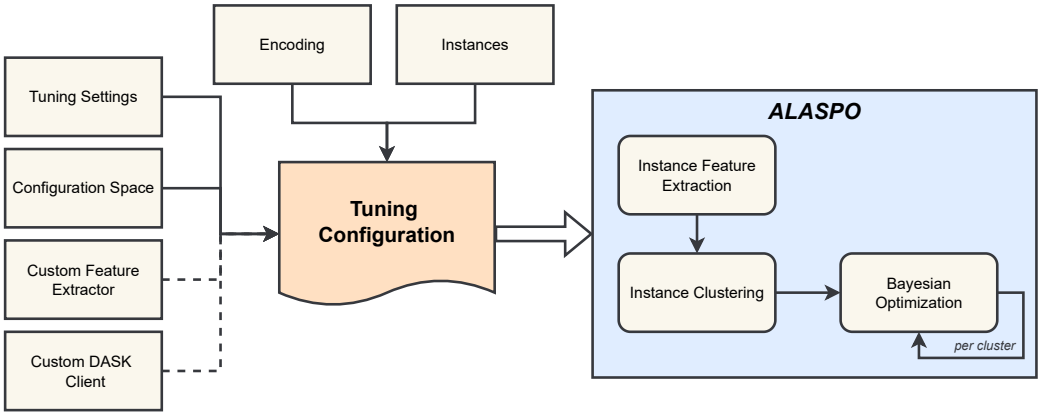


Figure 3.1: Overview of the automated tuning in ALASPO.

cluster the provided instances, so that instances with similar characteristics are grouped together. Finally, Bayesian optimization is performed on each instance cluster, resulting in a best found configuration for each. ALASPO will return a `clustering.dill` file after the tuning is finished, which can then be used for future ALASPO invocations, automatically applying a best found configuration based on the given (unseen) instance.

3.2 Feature Extraction

For the clustering of the provided instances, each instance must be associated with a set of features which are extracted by a *feature extractor*.

3.2.1 Default Feature Extractor

Since the default method needs to be problem agnostic, we opted for counting the number of occurrences of different facts for each instance. However, the success of this approach is very dependent on the particular problem and its encoding, as the number of facts may, in the worst case, be exactly the same across all instances, wheras their characteristics could differ significantly depending on the unaccounted-for terms of the predicates.

3.2.2 Custom Feature Extractor

To enable problem-specific instance feature extraction, we allow a custom feature extractor to be passed to ALASPO. This custom feature extractor is a method that takes the path to an instance file as parameter, and must return a list of numeric values (floats) that best represent the characteristics of the instance. The extracted features will subsequently be used to produce a convincing clustering of the given instances.

To illustrate why this could be beneficial, lets look at the already introduced Social Golfer Problem (SGP) 2.2.1, wherein golfers have to be grouped over w weeks, while

ensuring that no two players are in the same group more than once: If we assume that an ASP encoding does not expect the weeks to be represented by multiple facts of the form `week/1`, but only by a single fact `weeks/1` indicating the total amount, the default approach of counting facts is unable to distinguish instances by the number of weeks. However, it is evident that two similar instances that differ only in their duration vary in difficulty, as it becomes increasingly difficult to avoid similar pairings. In order for the instance clustering to account for this aspect, a custom feature extractor can be provided that returns the actual number of weeks as a feature.

3.3 Clustering

We cluster the instances based on their characteristics and use SMAC3 to find best performing configurations for each cluster separately. We do this in the hope of exploiting similarities between instances to improve the quality of configurations by tailoring them to related groups of instances.

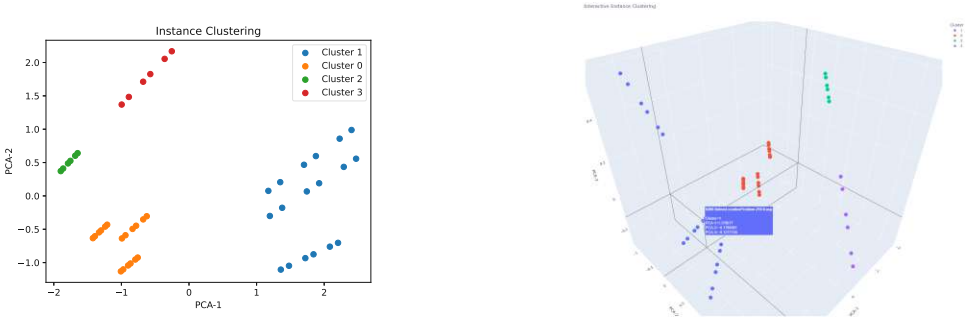
For the clustering, we use the K-means algorithm [HW79] on the dataset of extracted instance features. To determine the best number of clusters, we use the Silhouette Coefficient [Rou87] to evaluate the quality of the clustering. It ranges from -1 to 1, with the upper bound indicating that samples are far from neighbouring clusters, i.e., a good clustering, while values around 0 and below indicate that clusters overlap. We therefore find the number of clusters that maximizes the Silhouette Coefficient of the resulting clustering. Alternatively, the number of clusters can be specified using a corresponding parameter in the tuning configuration.

3.3.1 Visualization

We provide the option to visualize the resulting clustering of instances, which may help in understanding the instance set and its features. To do so, we reduce the dimensionality of the feature dataset using Principal Component Analysis (PCA) [Pea01], which we then use to visualize the clustering two-dimensionally, as well as three-dimensionally with an interactive plot. Figures 3.2a and 3.2b show examples of such visualizations.

3.4 Implementation

We perform the hyperparameter optimization with SMAC3 for each cluster of instances. Specifically, we use its `HyperparameterOptimizationFacade`, which, in turn, uses a Random Forest as the surrogate model. For the initial design – the component that determines which configurations are evaluated before the Bayesian optimization loop – we use the `DefaultInitialDesign` instead of the `SobolInitialDesign` used by the selected facade. This ensures that the actual provided default configuration is evaluated initially, rather than a *random* set sampled from the configuration space.



(a) 2D cluster visualization. (b) 3D cluster visualization.

Figure 3.2: Comparison of 2D and 3D cluster visualizations for the same instance set.

For the serialization of the tuning results, we use the dill [MSS+12] library, as it allows us to serialize complete functions, such as the feature extractor, which is needed for future ALASPO runs on new instances. In addition to the feature extractor, the resulting `clustering.dill` file also contains the feature scaler, a dictionary specifying the best configuration per cluster, and the K-means estimator used to predict the label of future instances.

3.4.1 Tuning Configuration

To start the hyperparameter tuning described above, a tuning configuration has to be passed to ALASPO. This can be done using either a JSON file or directly through Python. For a complete list and description of available parameters as well as examples we refer to the tuning README[1].

JSON Configuration

The JSON configuration file is an extension of a standard ALASPO configuration file. It contains an additional `tuning` node, under which all tuning related parameters are specified. To be valid, it must contain a time or a trial limit, the time limit of each trial, a problem encoding, instances, a configuration space, and a default configuration. Note that the JSON configuration itself acts as the default configuration.

Configuration Space The configuration space is configured as a JSON node where: a *key* represents the JSONPath[2] [GNB24] to a specific property of the default configuration, and the associated *value* is an array (or range) of candidate values to be tested for that property. Figure 3.3 shows an example JSON tuning configuration file. In the configuration space shown, the various available ALASPO strategies are tuned along with their respective settings. However, the parameters of the selection strategies should only

[1] `https://gitlab.tuwien.ac.at/kbs/BAI/alaspo/-/blob/9f8d5492f249d7eef2917f2c8d5d583b2d5cb235/examples/tuning/README.md`, accessed: April 15, 2025.

[2] `https://www.rfc-editor.org/rfc/rfc9535`, accessed: April 15, 2025.

be sampled by SMAC3 when the corresponding strategy is selected. Therefore, when tuning entire strategies (e.g., not just the alpha parameter of the current one) using the `strategy` key, the values are treated specially: The different strategies and their attributes are essentially flattened to regular hyperparameters, while the attributes are resolved as conditional hyperparameters for their corresponding strategies.

To illustrate tuning other parts of the configuration, Figure 3.4 shows a configuration space that tunes only the number of `unsatStrikes` of the dynamic strategy 2.2.2 in Line 2 (instead of tuning complete strategies), as well as the sizes of the first relax operator in Line 3.

To express simple uniform ranges for hyperparameters within the configuration space using a JSON file, we allow using the string prefix `$eval:` to enable the use of Python literal types. This is particularly useful for specifying Python tuples, which are interpreted differently: Lists are interpreted as categorical hyperparameters, while tuples are interpreted as either uniform float or uniform integer hyperparameters. An example can be seen in Lines 39 and 43 of Figure 3.3, which specify the range for the alpha attributes of their associated selection strategies.

DASK client Using a custom DASK client is useful for parallelizing trials and can be used to run the workers on a HPC cluster. To use this feature, it is necessary to specify the actual class of the desired cluster with the `type` key, as well as the number of `daskWorkers`. Each additional key-value pair is passed to the constructor of the specified cluster type. The available cluster types and their parameters can be found in the documentation for the DASK jobqueue package[3]. Figure 3.5 shows an example for a SLURM cluster, where 8 workers (Line 3) will be spawned as jobs, each processing one job at a time (Line 6), with Line 9 activating the correct conda environment for the worker processes.

Feature Extractor To use a custom feature extractor, the path to a Python file has to be provided. This file must contain a function named `extractFeatures` that takes the path to an instance as parameter and returns a list of numeric values (float or integer) that best characterize it. Note that import statements required by the provided function must be written inside the function itself for it to work after deserialization.

Python

Using Python directly can overcome some limitations imposed by the JSON standard. It allows passing a custom feature extractor function as well as a custom DASK client directly, and can be used to specify more complex types of hyperparameters. Examples of such advanced hyperparameters, and how to define them, can be found in the ConfigSpace

[3] `https://jobqueue.dask.org/en/latest/clusters-api.html`, accessed: April 15, 2025.

documentation[4]. A simple example of how to configure and start the tuning process directly in Python is shown in Figure 3.6.

3.4.2 Setup

To use the tuning integrated into ALASPO, there exist the optional dependencies `tuning` and `tuning-viz` which can be installed using `pip install alaspo[tuning]` and `pip install alaspo[tuning-viz]`, respectively. The latter contains additional dependencies needed for visualizing the clustering via the `visualizeClustering` option.

3.4.3 Usage

Given a tuning configuration has the name `tuningConfig.json`, the following can be used to start the tuning:

```
> alaspo --tuning-config tuningConfig.json
```

The resulting `clustering.dill` file can then be used as input for an (unseen) instance `newInstance.lp` as follows:

```
> alaspo --use-clustering clustering.dill -i encoding.lp
--instance newInstance.lp
```

The instance file must be specified explicitly using the `-instance` option, as this file will determine which configuration will be automatically applied.

[4] `https://automl.github.io/ConfigSpace/latest/reference/hyperparameters/`, accessed: April 15, 2025.

```
{
    "strategy": {
        "name": "dynamic",
        "unsatStrikes": 3,
        "timeoutStrikes": 1
    },
    "relaxOperators": [
        {
            "type": "randomAtoms",
            "sizes": [ 0.1, 0.2, 0.4, 0.6, 0.8 ]
        },
        {
            "type": "randomConstants",
            "sizes": [ 0.1, 0.2, 0.3, 0.5 ]
        }
    ],
    "searchOperators": [
        {
            "name": "default",
            "timeouts": [ 5, 15, 30, 60 ]
        }
    ],
    "tuning": {
        "trials": 1000,
        "trialTimeLimit": 600,
        "encoding": [
            "./encoding.lp"
        ],
        "instances": "./instances.txt",
        "configurationSpace": {
            "strategy": [
                {
                    "name": "dynamic",
                    "unsatStrikes": [ 1, 2, 3, 4 ],
                    "timeoutStrikes": [ 1, 2, 3, 4 ]
                },
                {
                    "name": "powerlaw",
                    "alpha": "$eval:(-5.0, 5.0)"
                },
                {
                    "name": "roulette",
                    "alpha": "$eval:(0.0, 1.0)"
                },
                {
                    "name": "random"
                }
            ]
        }
    }
}
```

Figure 3.3: An example JSON tuning configuration for ALASPO, with a configuration space that tunes different ALASPO strategies with their appropriate settings.

```
"configurationSpace": {
    "strategy.unsatStrikes": [ 1, 2, 3, 4 ],
    "relaxOperators[0].sizes": [ [ 0.1, 0.2, 0.3 ], [ 0.5, 0.6, 0.7, 0.8 ] ]
}
```

Figure 3.4: An example configuration space tuning the `unsatStrikes` parameter of the dynamic ALASPO strategy 2.2.2, and the sizes of the first relax operator.

```
"dask": {
    "type": "SLURMCluster",
    "workers": 8,
    "walltime": "-1",
    "cores": 2,
    "processes": 1,
    "memory": "25600MB",
    "job_script_prologue": [
        "source .../miniconda3/bin/activate alaspo"
    ],
    "job_extra_directives": [
        "--cpu-freq=2400000-2400000:performance",
        "--partition=sunnycove",
        "--output=slurm-%A_%a_stdout.log",
        "--error=slurm-%A_%a_stderr.log"
    ]
}
```

Figure 3.5: An example DASK configuration for parallelizing the tuning on a SLURM cluster.

```python
import json

import alaspo.json_config
from alaspo.tuning import TuningConfig, TunerFactory

def extractFeatures(instance: str): # dummy custom feature extractor
    from alaspo.tuning_util import parseInstance
    return [float(x) for x in parseInstance(instance).values()]

alaspoConfig: dict = json.loads(alaspo.json_config.DEFAULT_CONFIG)

configurationSpace = {
    "strategy": [
        {"name": "roulette", "alpha": (0.2, 0.8)},
        {"name": "dynamic", "unsatStrikes": (1, 4), "timeoutStrikes": [1, 2, 5, 6]}
    ],
}

tuningConfig = TuningConfig(
    timeLimit=None,
    trials=15,
    trialTimeLimit=5,
    encoding="./encoding.lp",
    instances="./instances.txt",
    configurationSpace=configurationSpace,
    defaultConfig=alaspoConfig,
    clusterInstances=True,
    customFeatureExtractor=extractFeatures,
)

TunerFactory.get(tuningConfig).tune()
```

Figure 3.6: An example of configuring the tuning process directly in Python.

CHAPTER 4

Experiments

In this chapter, we present the benchmark problems we use to evaluate the hyperparameter tuning of ALASPO, and discuss the results of the experiments run.

4.1 Experimental Setup

All experiments were run on a SLURM [JW23] high performance computing (HPC) cluster of 10 nodes, each equipped with two Intel Xeon Silver 4314 CPUs and 512GB of RAM, running Ubuntu 22.04 (Kernel 5.15.0-131-generic).

Features such as hyperthreading, Address Space Layout Randomization (ASLR), and Non-Uniform Memory Access (NUMA), as well as unnecessary background services, have been disabled in favour of reproducibility. Several additional measures have been taken to ensure stable, parallel, and repeatable experiments by: (i) taking the memory architecture of the hardware used into account by performing cache partitioning, (ii) ensuring low measuring overhead by using the Linux performance events subsystem (perf), and (iii) adopting a strict SLURM setup to isolate individual job executions through cgroups restrictions, all of which are discussed in detail by Fichte et al. [FGHS24]. In practice, running and scheduling a multitude of experiments for specific solvers, configurations, and problem instances that conform to the described measures was automated using the copperbench[1] tool [FGHS24].

Experiments were run at a fixed CPU frequency of 2.4GHz and a 25600MB memory limit, using Python v3.9.19, clingo v5.6.2, clingcon v5.2.0, smac v2.3.0, and ALASPO[2].

Used encodings, instances, instance generators and experiment logs are available at[3].

[1] https://github.com/tlyphed/copperbench, accessed: April 15, 2025.

[2] https://gitlab.tuwien.ac.at/kbs/BAI/alaspo/-/tree/9f8d5492f249d7eef2917f2c8d5d583b2d5cb235, accessed: April 15, 2025.

[3] https://github.com/DaveWasTakn/DIPLOMA-THESIS, accessed: April 18, 2025.

4.2 Hypotheses and Methodology

The methodology for evaluating the effectiveness of the newly integrated tuning and our hypotheses about its behaviour are the same for each of the four benchmark problems:

Hypotheses With our experimental evaluations, we want to investigate the following hypotheses about the tuning of ALASPO configurations:

(H1) The resulting configuration of a tuning run should perform better than the default ALASPO configuration.

(H2) Clustering problem instances based on their characteristics should result in multiple best configurations, one for each group of similar instances, thus achieving better objective values for the validation instance set than would a single configuration.

(H3) Using a custom feature extractor to produce a more representative feature dataset of the problem instances should lead to a better clustering and, in turn, better configurations.

(H4) Tuning approaches without clustering may perform better if the instances are quite similar in their optimization behaviour.

(H5) Increasing the number of trials run by the tuning should yield configurations of better quality, and consequently improve overall performance.

Methodology For the experimental evaluations, we tested our approach with the same configuration space contained in Figure 3.3. We thus tried to find the optimal ALASPO selection strategy along with its best performing parameters for each problem.

In addition to the fact that the selected benchmark problems are all highly relevant to industry, we chose the *Test Laboratory Scheduling Problem* [GMM21, MM18, MM21] as it has already been examined in previous work [EGH+24, GMM21] and has proven to be a computationally expensive scheduling task. The *Valves Location Problem* [GS09, GS10, CGN+11] is a typical combinatorial network design problem and was included as benchmark in the Fifth Answer Set Programming Competition[4]. The *Travelling Salesman Problem* [Men32, Rob49, DFJ54] is one of the most prominent optimization problems, with a very simple instance format (for the encoding we choose), containing only the layout of the graph, making the extraction of features from instances for their clustering of particular interest. Finally, *Multi-Agent Path Finding* [SSF+21] is a contemporary planning and coordination problem that has recently received increased attention due to its range of diverse applications; it is a highly conflict-intensive optimization task involving many inter-agent dependencies.

[4] `https://www.mat.unical.it/aspcomp2014`, accessed: April 15, 2025.

As we want to enable the overall workflow described in Section 3.1, we use separate sets of instances for the tuning and its validation.

For each problem, we present the results in a figure showing a series of box plots of the experiments using the described encoding, instances, and configuration space: Both the ALASPO baseline and the tuning validations are run 5 times each and averaged. The results are plotted against the ALASPO default baseline, with the objective values represented as relative difference. The box plots are grouped by the number of trials carried out by the tuning, where each group consists of four runs:

(i) **Default Clustering** uses the default feature extractor 3.2.1 for the instance clustering,

(ii) **Custom Clustering** uses a custom feature extractor that has been tailored to best capture the instance characteristics of the specific problem,

(iii) **No Clustering** does not cluster the instances by their features before running the Bayesian Optimization for each, but rather just supplies the features to SMAC3 and optimizes over all instances,

(iv) Whereas **No Clustering No Features** runs the Bayesian Optimization once, but without passing any instance features to SMAC3.

4.3 Test Laboratory Scheduling Problem (TLSPS)

The Test Laboratory Scheduling Problem (TLSPS) [GMM21, MM18, MM21] is an extension of the well known *Resource-Constrained Project Scheduling Problem* (RCPSP). It involves multiple projects, each with a number of tasks, that have to be completed. These task are grouped into jobs, that have specific resource requirements, different modes they can be completed in, and a time slot, in which the job must be started and completed. The constrained resources that need to be shared among all jobs are workbenches, equipment and employees.

A solution to this problem is a schedule, with each job being assigned a mode, the start and end time slots, a workbench, equipment, and employees. The quality of a solution is determined by the sum of multiple soft constraint violations, such as due date infractions or the time taken to complete each project.

4.3.1 Problem Definition and Encoding

Based on evaluations of ALASPO on the TLSPS in previous work [EGH+24, GMM21], we choose a clingcon based encoding that exploits clingcon specific features like domain constraints, that performed best among other encoding variants.

```
1  &dom{R..D} = start(J) :- job(J), release(J, R), deadline(J, D).
2  &dom{R..D} = end(J) :- job(J), release(J, R), deadline(J, D).
3
4  1 {modeAssign(J, M) : modeAvailable(J, M)} 1 :- job(J).
5  duration(J, T) :- job(J), modeAssign(J, M), durationInMode(J, M, T).
6  &sum{end(J); -start(J)} = T :- job(J), duration(J, T).
7  &sum{start(J)} >= end(K) :- job(J), job(K), precedence(J, K).
8  &sum{start(J)} = 0 :- job(J), started(J).
9
10 1 {workbenchAssign(J, W) : workbenchAvailable(J, W)} 1 :- job(J), workbenchRequired(J).
11 R {empAssign(J, E) : employeeAvailable(J, E)} R :- job(J), modeAssign(J, M),
   ↪  requiredEmployees(M, R).
12 R {equipAssign(J, E) : equipmentAvailable(J, E), group(E, G)} R :- job(J), group(_, G),
   ↪  requiredEquipment(J, G, R).
13
14 :- job(J), job(K), linked(J, K), empAssign(J, E), not empAssign(K, E).
```

Figure 4.1: The hard constraints for the TLSPS encoding.

Hard Constraints

Figure 4.1 shows the hard constraints of the encoding, which need to be satisfied by any solution:

Lines 1 and 2 restrict the integer values of `start(J)` and `end(J)` to be within the domains `&dom{R..D}`, thus ensuring that the start and end times of each job `J` are within its release and deadline window. Line 4 assigns one of the available modes to a job, while Lines 5 and 6 introduce a job's `duration` based on its selected mode and associate it with the difference between its start and end time. The next rule in Line 7 deals with precedence between jobs; it ensures that if job `K` is predecessor of job `J`, then `J` can only start after `K` has finished. Line 8 simply states that any job that is specified as started – problem instances can include a set of jobs that are already running – has its start time set to 0.

Next, Rules 10–12 deal with resource allocation. They assign each job an available workbench, employees, and equipment based on its requirements. Finally, the constraint in Line 14 ensures that if two jobs are linked, they must be processed by the same employees.

In addition, Figure 4.2 contains the unary resource constraints. They enforce that no resource is used by more than one job at any given time, using the precedence relation discussed earlier: If two jobs are assigned the same resource, then the disjunction in the rule heads implies that one of the jobs must precede the other.

Soft Constraints

The soft constraints of the TLSPS encoding are depicted in Figure 4.3:

The first two lines contain optimization directives that minimize the number of not preferred employees assigned to jobs, as well as the total number of employees assigned

```
16 precedence(J, K), precedence(K, J) :- job(J), job(K), workbenchAssign(J, W),
   ↪ workbenchAssign(K, W), J < K.
17 precedence(J, K), precedence(K, J) :- job(J), job(K), empAssign(J, E),
   ↪ empAssign(K, E), J < K.
18 precedence(J, K), precedence(K, J) :- job(J), job(K), equipAssign(J, E),
   ↪ equipAssign(K, E), J < K.
```

Figure 4.2: The unary resource constraints for the TLSPS encoding.

```
20 #minimize{ 1, E, J, s2 : job(J), empAssign(J, E), not employeePreferred(J, E) }.
21 #minimize{ 1, E, P, s3 : project(P), empAssign(J, E), projectAssignment(J, P) }.
22
23 &sum{delay(J); T} = end(J) :- job(J), due(J, T), &sum{end(J); -T} > 0.
24 &sum{delay(J)}=0 :- job(J), due(J, T), &sum{end(J); -T} <= 0.
25 delay(J, T) :- job(J), &sum{delay(J)} = T, T = 0..M, M = #max{D : deadline(J, D)}.
26 #minimize{ T, J, s4 : delay(J, T), job(J)}.
27
28 &dom{0..H} = projectStart(P) :- project(P), horizon(H).
29 &dom{0..H} = projectEnd(P) :- project(P), horizon(H).
30 1 {firstJob(J) : job(J), projectAssignment(J, P)} 1 :- project(P).
31 &sum{projectStart(P)} = start(J) :- firstJob(J), projectAssignment(J, P).
32 &sum{projectStart(P)} <= start(J) :- job(J), projectAssignment(J, P).
33 1 {lastJob(J) : job(J), projectAssignment(J, P)} 1 :- project(P).
34 &sum{projectEnd(P)} = end(J) :- lastJob(J), projectAssignment(J, P).
35 &sum{projectEnd(P)} >= end(J) :- job(J), projectAssignment(J, P).
36 &sum{projectEnd(P)-projectStart(P): project(P)} = projectDelay.
37 projectDelay(D) :- &sum{projectDelay} = D, D = 0..H*C, horizon(H),
   ↪ C = #count{ P : project(P) }.
38 #minimize{D, s5 : projectDelay(D)}.
```

Figure 4.3: The soft constraints for the TLSPS encoding.

to projects.

Each job has a release, the earliest possible start time, a due date, the time by which it should be completed, and a strict deadline. Lines 23–26 are used to minimize the delay of jobs, i.e. the time that a job exceeds its due date by. First, the delay of a job is defined as the difference between its end time and its due time in case it is completed late. Next, Line 24 ensures that the delay is set to 0 if a job finishes within the desired time window. Finally, Lines 25 and 26 introduce a standard predicate `delay/2`, whose second term captures the actual value of the integer variable (theory atom) `delay/1`, which is then minimized for all jobs in Line 26.

The last part of the encoding, Lines 28 through 38, introduces the necessary constructs to ultimately minimize the project delay. In Lines 28 and 29, both the start and end of a project are defined as integer variables with their values restricted to be within the scheduling horizon. Rules 30–32 determine the first job of each project to then derive the actual value for the `projectStart` variable. The next three lines analogously set the value for projectEnd. In Line 36, the `projectDelay` variable is defined as the sum of the differences between project end and start times over all projects. Subsequently,

this variable is used to derive a standard predicate `projectDelay/1`, which is finally minimized in the last line of the encoding.

The constants $s2$ to $s5$ within the minimization statements are used to make the derived tuples unique across the directives.

Show Statements

Since ALASPO selects a subset of the *visible* atoms, we use the `#show` directives contained in Figure 4.4 to limit the selection pool to only those atoms that describe all relevant parts of a solution to the TLSPS.

```
40 #show modeAssign/2.
41 #show workbenchAssign/2.
42 #show empAssign/2.
43 #show equipAssign/2.
44 #show start/2.
```

Figure 4.4: The `#show` directives used for the TLSPS encoding.

4.3.2 Instances

For the evaluation we use the set of randomly generated instances[5] created using the instance generator introduced by Mischek et al. [MM18], one half of which are labelled *labStructure*, and the other *general*. The former being modelled as close to *real-world* test laboratories as possible, while the latter are constructed with more varied features. Additionally, the set contains 3 *real-world* instances of an industry partner that were anonymized.

For the tuning we select a set of 31 instances, including the same generated instances used by Geibinger [Gei20], and one of the *real-world* instances. We subsequently validate the tuning results on a different set of 74 instances, containing 36 *labStructure* and 36 *general*, as well as the remaining 2 *real-world* instances.

The characteristics of the tuning instances, along with all three *real-world* instances, can be seen in Table 4.1, created by Geibinger [Gei20].

4.3.3 Experimental Evaluation

For the experimental evaluation we use the methodology as described in Section 4.2, with a time limit of 600 seconds for the tuning trials and 1800 seconds for the validation runs.

For the **Custom Clustering**, we implement a custom feature extractor that extracts the number of projects and jobs, how many jobs are linked and how often a precedence relation is expressed, the number of available modes, as well as the size of the scheduling horizon. In addition, we calculate the following two metrics:

[5] `https://www.dbai.tuwien.ac.at/staff/fmischek/TLSP`, accessed: April 15, 2025.

#	Data Set	ID	$\|P\|$	$\|J\|$	h	$\|E\|$	$\|B\|$	$\|G^*\|$	$\overline{\|E_j\|}$	$\overline{\|B_j\|}$	$\overline{\|G_{gj}\|}$
1	General	000	5	7	88	7	7	3	2.08	3.57	1.5
2	General	001	5	8	88	7	7	3	4.88	3.63	15.67
3	LabStructure	000	5	24	88	7	7	3	1.84	3.38	11.67
4	LabStructure	001	5	14	88	7	7	3	4.36	3.5	0.36
5	General	005	10	29	88	13	13	4	4.04	3.48	5.76
6	General	006	10	18	88	13	13	6	5.56	4.22	13.28
7	LabStructure	005	10	37	88	13	13	3	6.16	4.03	0.65
8	LabStructure	006	10	29	88	13	13	3	6.21	3.76	21.01
9	General	010	20	60	174	16	16	5	7.42	4.42	11.36
10	General	011	20	84	174	16	16	4	7.31	4.3	3.7
11	LabStructure	010	20	65	174	16	16	3	6.28	4.43	26.26
12	LabStructure	011	20	62	174	16	16	3	7.27	4.24	1.21
13	General	020	15	29	174	12	12	5	5.76	3.97	1.12
14	LabStructure	020	15	53	174	12	12	3	6.28	4.47	20.63
15	General	025	30	113	174	23	23	3	8.26	4.41	5.71
16	LabStructure	025	30	105	174	23	23	3	7.52	4.25	39.63
17	General	015	40	126	174	31	31	3	9.26	4.48	29.53
18	LabStructure	015	40	138	174	31	31	3	7.36	3.57	41.93
19	General	030	60	208	174	46	46	6	9.85	4.11	31.45
20	LabStructure	030	60	212	174	46	46	3	9.28	4.17	78.16
21	General	035	20	76	520	6	6	5	4.24	3.62	8.08
22	LabStructure	035	20	71	520	6	6	3	4.3	3.42	11.70
23	General	040	40	196	520	12	12	4	6.95	4.47	4.24
24	LabStructure	040	40	187	520	12	12	3	6.55	4.51	1.38
25	General	045	60	260	520	18	18	6	7.65	4.52	23.95
26	LabStructure	045	60	239	520	18	18	3	7.44	4.42	33.65
27	General	050	60	270	782	13	13	4	6.89	4.39	3.89
28	LabStructure	050	60	247	782	13	13	3	6.97	4.21	23.42
29	General	055	90	384	782	19	19	5	7.27	4.29	26.89
30	LabStructure	055	90	401	782	19	19	3	7.34	4.53	36.76
31	2019-04	-	74	297	606	22	17	1	5.49	3.06	1*
32	2019-10	-	59	223	572	19	17	1	5.70	3.48	1*
33	2019-07	-	59	251	700	24	22	1	5.33	3.17	1*

Table 4.1: Characteristics of the TLSPS instances used. Shown are the type of the instance (which data set it was taken from) and their ID. The following columns list the number of projects, jobs and the length of the scheduling period, followed by the number of employees, workbenches, and equipment groups. The last columns contain the mean of qualified employees and available workbenches per job, as well as the mean of available devices per job and equipment group (only over jobs that require at least one device of the group, about 10% of all jobs).
*The discrepancy compared to the other instances is caused by the fact that some equipment groups were not yet considered at the time this instance was created. [Gei20]

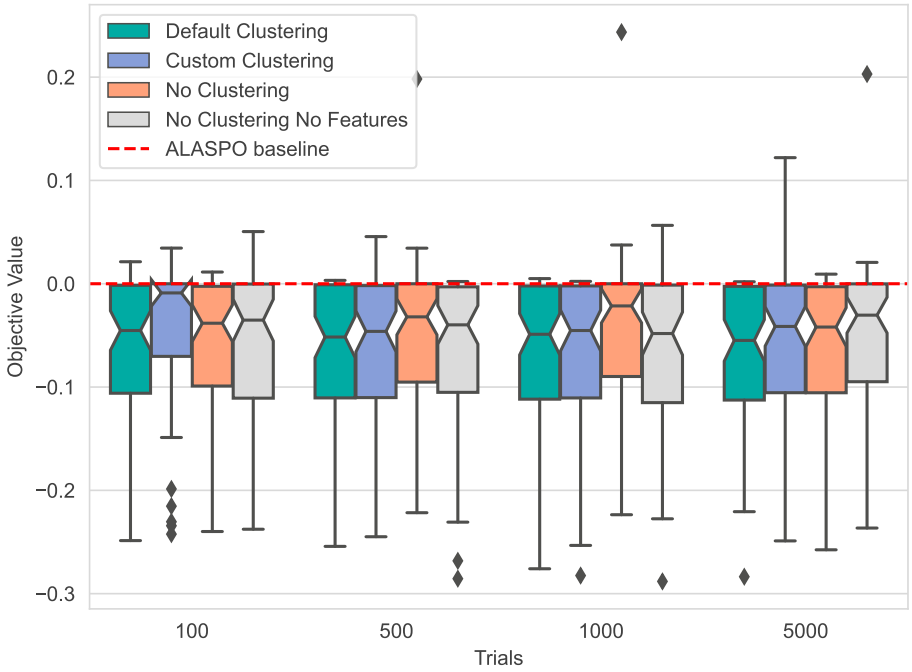


Figure 4.5: Tuning results for the TLSPS. Objective values are relative differences to the ALASPO baseline. Detailed results are available in Table A.1 in the Appendix.

(i) The average ratio of the mean duration of a job, based on the different available modes in which it can be processed, to its maximum makespan, i.e., the available time from release to deadline. A high ratio would indicate that jobs are time-critical with little room for adjustment, while the opposite would indicate more flexibility in scheduling jobs, which may affect the optimization behavior of the problem instance.

(ii) The average ratio of the penalty window, i.e., the time between the due date and the deadline, to the maximum makespan of jobs. This metric reflects whether an instance allows more opportunities for optimization when its penalty window is large, or whether it is more dependent on strict deadlines.

Results and Observations

Figure 4.5 shows the results of the TLSPS tuning. They demonstrate that the different clustering approaches lead to relatively similar improvements compared to the baseline. However, the default clustering is the most consistent over the increasing number of trials, and is almost strictly better than default ALASPO at 5000 trials, with only two instances performing negligibly worse than the baseline. It also performs marginally better than the other variants in all four scenarios, with a median improvement of about 5% and average

improvement of 7% over the baseline. Both of these observations generally support our previously formulated hypotheses **H1** and **H5**, wherein tuning should result in better performance over the ALASPO baseline, and increasing the number of trials yields better results.

At 100 trials, the custom feature extraction leads to the least improvements, while at 500 and 1000 trials it performs roughly the same as the default clustering approach. Notably, it performs worse on 5000 trials, with objective scores worse than the default ALASPO configuration, at a maximum of about 12%. This directly contradicts hypothesis **H3**, suggesting that the features used for the custom clustering did not adequately capture the optimization behaviour of different instances.

The tuning runs without clustering, both with and without instance features, contain quite high outliers that perform about 20% worse than the baseline for 500 and 1000 and 5000 trials, respectively. However, at only 100 trials, **No Clustering** performs on par with the default clustered approach, with the same finding for **No Clustering No Features** at 500 trials.

Overall, the experiments show that already a small number of trials for the hyperparameter tuning can improve the performance of ALASPO on the TLSPS, with an average improvement of almost 6% for 100 trials with the default method. Moreover, the **Default Clustering** generally performs best in finding the right ALASPO configuration for solving the TLSPS instances, with a quarter of the objective values being about $11\% - 22\%$ better than the baseline. The fact that a clustered approach performs better than finding a single best configuration for all instances supports our hypothesis **H2**. However, as mentioned above, the results also indicate that **Custom Clustering**, using the different metrics of our custom feature extractor, does not lead to better performance than the default count-based clustering, which contradicts **H3**.

4.4 Valves Location Problem (VLP)

Next, we consider the problem of isolation valve placement in water distribution networks [GS09, GS10, CGN+11], which we will refer to as the Valves Location Problem (VLP). It is the practical problem of placing a number of valves in a water distribution network such that when certain pipes need to be isolated from the rest of the system by turning off a subset of valves, for maintenance or unplanned interruptions, the disruption to the network is minimal. Since the number of valves is limited, the setting does not allow the trivial case of simply placing two valves on each individual pipe. Therefore, in most cases isolating solely the desired pipe is not possible; the affected isolated pipes are each associated with a water demand, the sum of which must be minimized [GS10, CGN+11].

4.4.1 Problem Definition and Encoding

The ASP encoding for this problem is introduced by Gavanelli et al. [GNP13] and is available from the Fifth Answer Set Programming Competition (ASPCOMP 2014)[6].

The basic objective of the encoding is to place valves at positions, such that every pipe can be isolated, and the maximum undelivered demand (UD) is minimized [GNP13].

Preprocessing

Figure 4.6 contains preprocessing steps, needed for later rules: They introduce the `swap/2` and subsequently the `sym_pipe/2` predicate to represent the undirected network. Additionally, the last two rules establish an ordering of pipes that will be used in later stages to break ties in water delivery scenarios.

```
1 swap(pipe(A,B),pipe(A,B)) :- pipe(A,B).
2 swap(pipe(A,B),pipe(B,A)) :- pipe(A,B).
3 symm_pipe(A,B) :- swap(P,pipe(A,B)).
4
5 less_ico(pipe(A,B),pipe(C,D)) :- pipe(A,B), pipe(C,D), A < C.
6 less_ico(pipe(A,B),pipe(A,D)) :- pipe(A,B), pipe(A,D), B < D.
```

Figure 4.6: The necessary preprocessing steps of the VLP encoding.

Valve Placements

In Figure 4.7, possible valve locations within the network are determined. Lines 8 and 9 are used to eliminate potentially bad valve locations. Such undesirable locations are represented by atoms of the predicate `drop/2`, either at the far end of a pipe directly connected to a water tank (if the instance only allows one valve per pipe), or at simple junctions of pipes of size 2.

Next, the choice rule in Line 11 selects a subset of desirable locations for every available valve. In addition, Line 12 now enforces that a valve is placed *directly* next to each tank, which is why valves at the far end of pipes connected to water tanks were previously disallowed if `valves_per_pipe(1)`. The last line then globally ensures that no pipe contains more than one valve, if specified by the instance.

Water Flow

The part of the encoding shown in Figure 4.8 is responsible for modelling how water propagates through the network, and which parts of it become isolated (broken).

Lines 15 through 17 introduce the `broken/2` predicate, which extends from a broken pipe to all connected pipes as long as there is no valve (or tank) to stop it, expressed by the `extend/2` relation.

[6] `https://www.mat.unical.it/aspcomp2014`, accessed: April 15, 2025.

```
8  drop(B,A) :- symm_pipe(A,B), tank(A), valves_per_pipe(1).
9  drop(A,C) :- symm_pipe(A,B), symm_pipe(A,C), B < C, not tank(A),
   ↪  #count{ D : symm_pipe(A,D) } < 3.
10
11 N <= { valve(A,B) : symm_pipe(A,B), not drop(A,B) } <= N :- valves_number(N).
12 :- symm_pipe(A,B), tank(A), not valve(A,B).
13 :- valves_per_pipe(1), valve(A,B), valve(B,A).
```

Figure 4.7: The part of the VLP encoding responsible for valve placement.

Similarly, Lines 19–21 model the water flow, starting from a water tank, following non-broken pipes, using the `reached/2` and `deliver/2` predicates.

```
15 broken(P,P) :- swap(P,P).
16 broken(P,Q) :- extend(P,A), swap(Q,pipe(A,B)), not valve(A,B).
17 extend(P,A) :- broken(P,Q), swap(Q,pipe(A,B)), not valve(A,B), not tank(A).
18
19 reached(P,A) :- swap(P,P), tank(A).
20 reached(P,A) :- deliver(P,Q), swap(Q,pipe(A,B)), not extend(P,A).
21 deliver(P,Q) :- reached(P,A), swap(Q,pipe(A,B)), not broken(P,Q).
```

Figure 4.8: The rules that propagate the effects of a pipe failure based on water flow for the VLP encoding.

Minimizing Undelivered Demand

Finally, Figure 4.9 contains the rules for finding the worst case scenario and minimizing the maximum expected undelivered water demand.

The first two rules examine different isolation scenarios pairwise and introduce the `compare/4` predicate, containing the water demand that the scenarios would contribute to the difference in delivered demand. For each pipe `pipe(A,B)`, if it is delivered in scenario `P` but not in scenario `Q`, it contributes its demand to the comparison as a negative difference, or, if the opposite is the case, as a positive difference. For instance, if we have atom `compare(P, Q, pipe(3, 4), -6)`, then scenario `P` is better, indicating that `P` delivers 6 more units of water than `Q` for this pipe.

Next, in Lines 26 and 27, all the different isolation scenarios are ordered based on their relative differences in delivered demands, by summing up the previously generated `compare/4` atoms (i.e., their last term), to overall establish an ordering of scenarios expressed by the `lower/1` relation.

Note that all four rules make use of the ordering `less_ico/1` introduced in the first section of the encoding 4.6, in order to compare any two scenarios only once, as well as to break ties between equally good isolation scenarios.

Finally, Line 29 introduces atoms of the form `worst_deliv_dem/3`, for every pipe that *does* deliver water in scenario `P`, where `P` is the actual worst case scenario (there is no

other scenario that delivers a lower amount of the demanded water). Now the directive in Line 30 can minimize the demands of all of the pipes that are *not delivered* with water in the worst case. To reiterate, the encoding minimizes the sum of the total undelivered water demands in the worst isolation scenario, where `not worst_deliv_dem(A,B,N)` filters out all pipes `pipe(A,B)` that *are* delivered with water, leaving only the isolated pipes, whose demands `N` are then minimized.

```
23 compare(P,Q,pipe(A,B),-N) :- less_ico(P,Q), dem(A,B,N), deliver(P,pipe(A,B)), not
   ↪  deliver(Q,pipe(A,B)).
24 compare(P,Q,pipe(A,B), N) :- less_ico(P,Q), dem(A,B,N), deliver(Q,pipe(A,B)), not
   ↪  deliver(P,pipe(A,B)).
25
26 lower(P) :- less_ico(P,Q), #sum{ N,R : compare(P,Q,R,N) } < X, X=0.
27 lower(Q) :- less_ico(P,Q), not lower(P).
28
29 worst_deliv_dem(A,B,N) :- deliver(P,pipe(A,B)), dem(A,B,N), not lower(P).
30 :~ dem(A,B,N), not worst_deliv_dem(A,B,N). [N,A,B]
31
32 #show valve/2.
```

Figure 4.9: The final part of the VLP encoding, minimizing the maximum undelivered water demand.

4.4.2 Instances

We use the instances provided alongside the encoding for the VLP, made available by the ASPCOMP 2014[6]. The total instance set contains 318 instances, the upper half of which we select 112 instances for the validation, and 48 for the tuning.

The instances contain the predicates `valves_number/1` and `valves_per_pipe/1`, which specify the available number of valves and whether a pipe has one or two valves, respectively. Furthermore, they contain the water tanks `tank/1`, and a network of pipes `pipe/2` with their water demands `dem/3`.

4.4.3 Experimental Evaluation

For the experimental evaluation, we use the same general methodology as described in Section 4.2.

The experiments were run with a time limit of 600 seconds, for both the tuning trials and the validation runs.

For the **Custom Clustering**, we implement a feature extractor that characterizes instances by: (i) The ratio of available valves to the number of pipes in the network; a high value indicates more flexibility in valve placement, while a low ratio may indicate a more challenging instance, where valve locations have a greater impact on solution quality. (iii) The valve budget relative to the sum of demands and (iii) the average water demand per pipe.

However, though the visualization of the custom clustering (depicted in Figure 3.2) did not seem inadequate at first, we will show below that the characteristics we chose for the custom feature extractor are worse at capturing the difficulty and optimization behaviour of the problem instances than the default implementation.

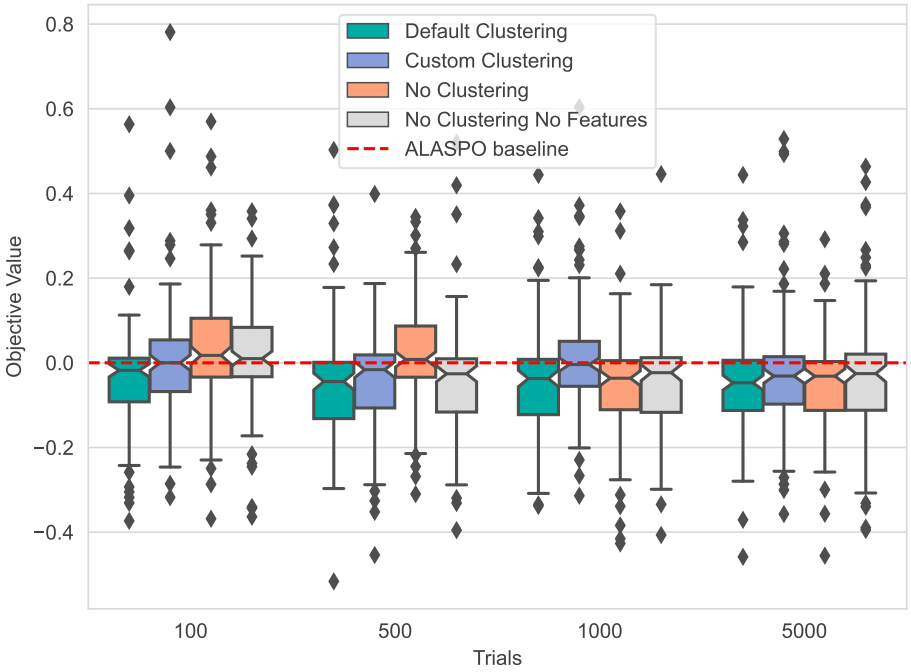


Figure 4.10: Tuning results for the Valves Location Problem (VLP). Objective values are relative differences to the ALASPO baseline. Detailed results are available in Table A.2 in the Appendix.

Results and Observations

The results of the tuning for the VLP are shown in Figure 4.10. They illustrate that the tuning of the ALASPO selection strategies did not work particularly well for this problem, as the results are very volatile across all approaches and trial budgets.

The default clustering produces the best median improvements and is the most consistent as the number of trials increases. It consistently outperforms the other approaches, thus partially supporting our hypothesis **H2**. The custom clustering, however, performs worse than the default clustering at any number of trials, with an outlier at 100 trials that is 78% worse than the ALASPO baseline. The results indicate that the chosen features do not represent the optimization behaviour of the instances well; it is worse than simply counting the occurrences of facts, which directly contradicts our assumption formulated in **H3**.

Interestingly, **No Clustering** produces worse objective values than both **Default Clustering** and **No Clustering No Features** at 100 and 500 trials. That is, providing SMAC3 with the instance feature dataset used for the default clustering, results in worse performance than without any features, even though the clustered approach based on these features performs best.

Also, **No Clustering No Features** is marginally worse at 5000 trials than at both 1000 and 500, with respect to the number of instances worse than the baseline, while this is not the case for any other approach. Although this observation challenges the hypothesis **H5**, the fact that the same hypothesis – namely, that increasing the number of trials improves overall performance – holds for most other clustering approaches and trial counts suggests that this is an anomaly rather than a refutation of our hypothesis.

Overall, the four variants provide similar objective values when run with a trial limit of 5000 and 1000 (excluding **Custom Clustering**). However, although most combinations achieve a small improvement for the majority of instances, with maximum reductions in objective values of about $30\% - 40\%$, all approaches perform worse than ALASPO with its default configuration for at least a quarter of the instances, including significant outliers. Our hypothesis **H1**, which states that configurations resulting from a tuning should perform better than the default ALASPO configuration, is challenged by the overall observations. However, depending on how *better* is interpreted for a particular use case, **H1** could still theoretically hold. For example, at 5000 trials, the average relative improvements over the baseline across the different clustering strategies are in the range of 2.7% to 5.5%, with about 73 instances performing better and only about 36 worse than the baseline.

4.5 Travelling Salesman Problem (TSP)

The Travelling Salesman Problem (TSP) [Men32, Rob49, DFJ54] is a well known problem with a wide range of practical applications. It involves finding the shortest path between a set of nodes, visiting each exactly once, and finishing at the starting point. In other words, it tries to find a Hamiltonian cycle with minimal costs.

4.5.1 Problem Definition and Encoding

The encoding for the TSP is shown in Figure 4.11; it originates from the asparagus platform[7] and was investigated previously by Eiter et al. [EGHR+22].

A solution to a given instance is represented by the `cycle/2` predicates of the visited edges.

In the first two lines of the encoding, choice rules are used to express the fact that every vertex `vtx/1` must be included in the resulting cycle by having exactly one edge entering and one edge leaving it (i.e., it enforces $indeg(v) = outdeg(v) = 1$ for every vertex v).

[7] `https://asparagus.cs.uni-potsdam.de`, found to be inaccessible as of April 15, 2025.

Next, Lines 4–6 introduce the notion of vertices being *reached*, starting with the first vertex and propagating through vertices connected by the `cycle/2` predicate, to then ensure that every vertex can be reached within the cycle in Line 6.

Finally, Line 8 contains a minimization directive, minimizing the sum of the edge weights for all edges included in the cycle.

The last line of the encoding is used to limit the visible atoms to only include `cycle/2`.

```
1  { cycle(X,Y) : edge(X,Y); cycle(X,Y) : edge(Y,X) } = 1 :- vtx(X).
2  { cycle(X,Y) : edge(X,Y); cycle(X,Y) : edge(Y,X) } = 1 :- vtx(Y).
3
4  reached(1).
5  reached(Y) :- reached(X), cycle(X,Y).
6  :- vtx(X), not reached(X).
7
8  :~ cycle(X,Y), edgewt(X,Y,C). [C,X,Y]
9
10 #show cycle/2.
```

Figure 4.11: The encoding for the Travelling Salesman Problem.

4.5.2 Instances

For the TSP, we create our own set of random instances using a generator written in R. We make use of the tspgen[8] library developed by Bossek et al. [BKN+19], which implements a set of creative mutation operators aimed at generating diverse instances.

We generate a set of 96 instances in total, with sizes chosen from $40, 60, 80$, and 100 vertices. For each size, we start with a random graph, and use one of four distinct mutations: (i) the `AxisProjectionMutation` of tspgen, (ii) the `GridMutation` of tspgen, (iii) a single cluster mutation, that clusters all points into a single group, and (iv) a triple cluster mutation, which splits the points into three separate clusters. Finally, the resulting instance is perturbed multiple times to obtain different instances with similar global characteristics. Examples of the different types of instances are shown in Figure 4.12.

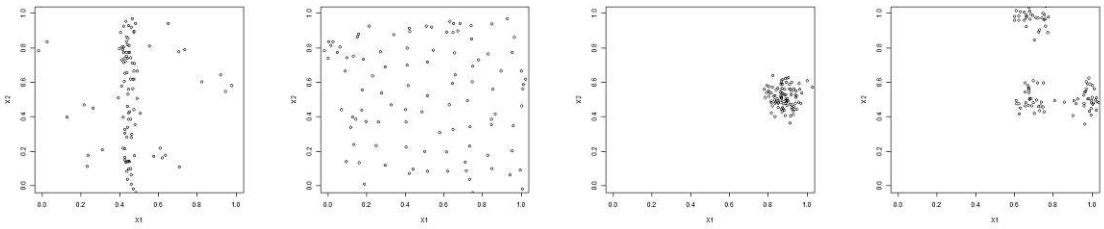
For the experiments, we then split the set, selecting the first instance of each size and type for tuning, while using the rest to validate the tuning results. This results in 16 instances for the tuning and 80 for its validation.

4.5.3 Experimental Evaluation

We employ the general methodology for the experimental evaluation of the TSP as outlined in Section 4.2.

The experiments were run with a time limit of 300 seconds, for both the tuning trials and the validation runs.

[8] `https://github.com/jakobbossek/tspgen/`, accessed: April 15, 2025.



(a) Axis Projection (b) Grid Mutation (c) Single Cluster (d) Triple Cluster

Figure 4.12: Example TSP instances using different mutations.

For the `Custom Clustering`, we implement a feature extractor which, for every instance, reports back the total weight of the Minimum Spanning Tree (MST), the area of its convex hull, as well as the mean and standard deviation of all pairwise point distances. We choose the MST, as it is an easy to compute lower bound for the Hamiltonian cycle, indicating the density of the instance and thus correlating with the length of the TSP tours. The area of the convex hull represents the spread of the different nodes, with a larger area typically indicating a more dispersed layout. Finally, the metrics about the length of edges should provide additional information about the general distribution of nodes. Note that we do not include the number of vertices in the feature set, as we want the clustering to be based on the geographical structure of the instances.

Results and Observations

Figure 4.13 presents the results of the tuning for the TSP. For 100 trials, the **No Clustering No Features** approach clearly outperforms the other three variants, with an average relative improvement of about 18% over the ALASPO baseline and only 3 out of 80 instances performing worse. Interestingly, **No Clustering** – where SMAC3 is also run on the entire instance set, but provided with the default instance feature set – results in significantly worse objective values with the same number of trials. However, for the other runs with 500, 1000, and 5000 trials, the two perform almost identically, consistently better than the default ALASPO configuration with average improvements of about 28%. This observation strongly supports our hypothesis **H5**, since increasing the number of trials clearly leads to better configurations being found.

The clustered approaches are both equally unstable at 100 trials, with 36 of 80 instances above the baseline, suggesting that hypothesis **H2** does not hold when paired with a small trial budget. Their respective clustering is visualized in Figure 4.14, which clearly demonstrates the pitfall of the default feature extraction: Two TSP instances with the same number of vertices and edges can differ significantly in difficulty due to their spatial distributions. However, the generic method of counting facts cannot discriminate any features other than the number of vertices (and edges). Therefore, the **Default Clustering** produces four clusters, one for each instance size, whereas the **Custom Clustering** takes the actual spatial features of the instances into account.

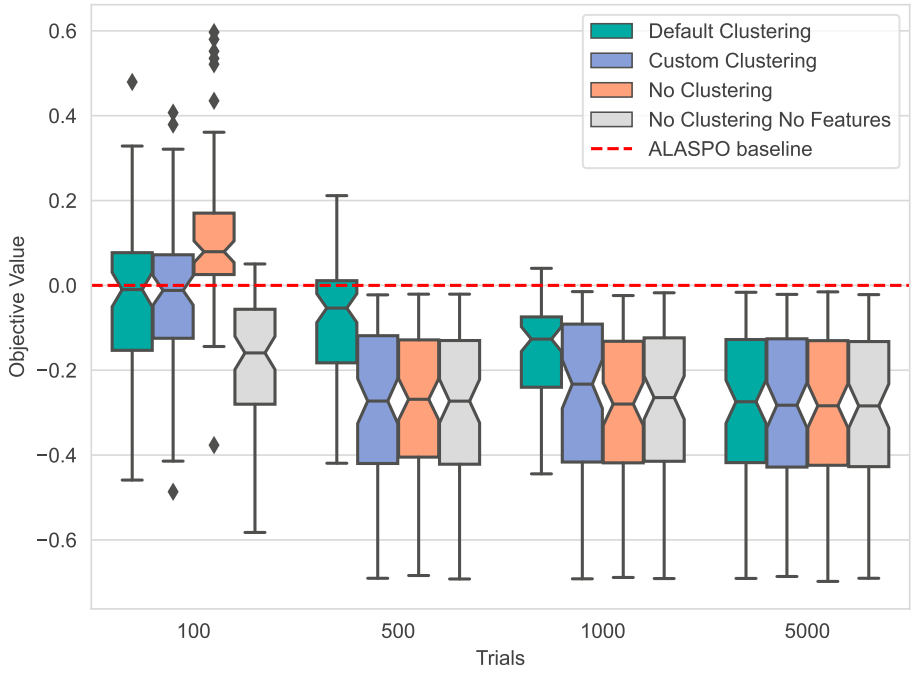
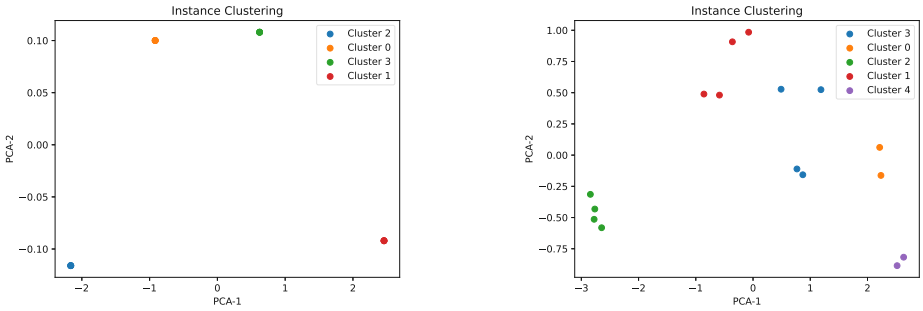


Figure 4.13: Tuning results for the TSP. Objective values are relative differences to the ALASPO baseline. Detailed results are available in Table A.3 in the Appendix.

While the default clustering gradually improves with an increasing number of trials, peaking at practically indistinguishable results with 5000 trials, the custom clustering matches the performance of the other best approaches from 500 trials on. This supports our hypothesis **H3** wherein custom feature extraction creates a more appropriate clustering of instances, thus improving overall performance, albeit only over the default clustering.

Overall, the custom clustering and both unclustered approaches behave almost identically for more than 500 trials (with the custom clustering slightly worse at 1000). They lead to considerable improvements in objective values over the ALASPO baseline, with average improvements of about 28% and maximum gains of just under 70%, strongly supporting hypothesis **H1**.

That the clustered variants, especially the default one, perform worse than **No Clustering No Features** could be explained by the fact that for the TSP, the power law selection strategy 2.2.2 with a large alpha value stands out as the best for all instances regardless of the instance type, which makes clustering of little use. This directly validates our hypothesis **H4**, which states that clustering can be a hindrance in finding good configurations when the optimization behaviour of all instances is quite similar, while also providing another possible explanation as to why **H2** does not hold.



(a) **Default Clustering** (b) **Custom Clustering**

Figure 4.14: Visualization of the instance clustering for the TSP, with the default feature extraction (left) and the implemented custom extractor (right).

4.6 Multi-Agent Path Finding (MAPF)

Multi-Agent Path Finding (MAPF) involves finding conflict-free paths for multiple agents sharing the same environment, each having a start position and a goal position they must reach. Naturally, this is a highly relevant problem for industry, with applications in automated warehouse routing [LTK+21], robotics [CLH+22], and autonomous vehicles [LHL+23] [SSF+21].

Erdem et al. [EKOS13] introduce a formal framework for applying ASP to pathfinding problems with multiple agents, and present encodings that address different path constraints. We will consider the sum-of-cost optimization encoding introduced by Gomez et al. [GHB20].

4.6.1 Problem Definition and Encoding

For our evaluation we use the encoding introduced by Gomez et al. [GHB20]. Specifically, we use the version of encoding with quadratic conflict resolution, rather than their proposed linear encoding, as, although the linear version produces a smaller and faster grounding, we found it to take significantly more time in finding solutions for our instances.

Actions

Figure 4.15 contains rules that both define possible actions, and guess these actions executed by agents.

Lines 1–5 and 7–11 first introduce the possible actions as atoms, i.e., all directions an agent can move to (including the option to wait), and then define the positional delta of an agent after executing one of them. For example, `delta(right, X, Y, X+1, Y)` encodes the fact that an agent starting at position `(X, Y)` and moving to the right will end up at node `(X+1, Y)`.

Next, the choice rule in Line 13 guesses one possible action for each agent at every time step, represented by the predicate `exec/3`. While the last line introduces atoms of the form `at/4`, which specify the location of an agent at any given time.

```
1  action(right).
2  action(left).
3  action(up).
4  action(down).
5  action(wait).
6
7  delta(right, X, Y, X+1, Y  ) :- rangeX(X), rangeY(Y).
8  delta(left,  X, Y, X-1, Y  ) :- rangeX(X), rangeY(Y).
9  delta(up,    X, Y, X  , Y+1) :- rangeX(X), rangeY(Y).
10 delta(down,  X, Y, X  , Y-1) :- rangeX(X), rangeY(Y).
11 delta(wait,  X, Y, X  , Y  ) :- rangeX(X), rangeY(Y).
12
13 1 {exec(A, M, T-1) : action(M)} 1 :- time(T), agent(A).
14 at(A, X, Y, T) :- exec(A, M, T-1), at(A, X', Y', T-1), delta(M, X', Y', X, Y).
```

Figure 4.15: The different possible actions and their effects for the MAPF encoding.

Integrity Constraints and Conflicts

The encoding shown in Figure 4.16 imposes constraints on the possible locations and actions of agents.

Lines 16 through 18 restrict the positions of agents to be within the boundaries of the available grid, whose dimensions are provided with facts `rangeX/1` and `rangeY/1`, and disallow agents at locations that are marked as obstacles.

Next, Lines 20 and 22–23 encode vertex and swap conflicts, respectively. The former simply disallows two different agents to be in the same place at the same time, while the latter ensures that no two agents traverse the same edge in opposite directions at the same time instance, for both the horizontal and vertical case.

```
16 :- at(A, X, Y, T), not rangeX(X).
17 :- at(A, X, Y, T), not rangeY(Y).
18 :- at(A, X, Y, T), obstacle(X, Y).
19
20 :- at(A, X, Y, T), at(A', X, Y, T), A != A'.
21
22 :- at(A, X+1, Y  , T-1), at(A', X, Y, T-1), at(A, X, Y, T), at(A', X+1, Y  , T).
23 :- at(A, X  , Y+1, T-1), at(A', X, Y, T-1), at(A, X, Y, T), at(A', X  , Y+1, T).
```

Figure 4.16: The integrity constraints for the MAPF encoding.

Sum-of-Costs Minimization

Lastly, Figure 4.17 introduces the necessary constructs to then optimize the sum-of-costs, i.e., the number of actions executed by agents before arriving at their goals.

Rules 25 and 26 define the predicate `at_goal/2` representing the time an agent reaches its goal, and enforce that every agent is at their goal at the last time step.

Next, in Lines 28–29 the `at_goal_back/2` predicate is introduced, which is derived for an agent `A` at time `T` if it is at its goal, and from time `T` onwards only waits. It essentially represents if an agent is at its final position without needing to move in the future. The actual predicate is incrementally derived in reverse, starting from the end of the horizon, until an agent performs an action other than `wait`.

Finally, Rules 31–33 introduce the cost of performing penalized actions: For (i) every time instant an agent is not at its goal, (ii) an agent moving away from its goal, and (iii) every time an agent waits at its goal, but later moves away from it (i.e., `not at_goal_back/2`), a cost of 1 is derived for the given agent and time step. The sum of these costs is ultimately minimized in Line 35.

Again, Line 37 limits visible atoms to only those that are relevant.

```
25 at_goal(A, T) :- at(A, X, Y, T), goal(A, X, Y).
26 :- agent(A), not at_goal(A,T), time(T), not time(T+1).
27
28 at_goal_back(A, T  ) :- agent(A), time(T), not time(T+1).
29 at_goal_back(A, T-1) :- at_goal_back(A, T), exec(A, wait, T-1).
30
31 cost(A, T, 1) :- at(A, X, Y, T), not goal(A, X, Y).
32 cost(A, T, 1) :- at(A, X, Y, T), goal(A, X, Y), exec(A, M, T), M != wait.
33 cost(A, T, 1) :- at(A, X, Y, T), goal(A, X, Y), exec(A, wait, T),
   ↪  not at_goal_back(A, T).
34
35 #minimize{C, T, A : cost(A, T, C)}.
36
37 #show exec/3.
```

Figure 4.17: The sum-of-costs optimization for the MAPF encoding.

4.6.2 Instances

For the instances, we implement an instance generator which creates random instances with characteristics close to those used by Gomez et al. [GHB20]: We create *random* and *warehouse* instances, with varying sizes, percentages of obstacles, and number of agents.

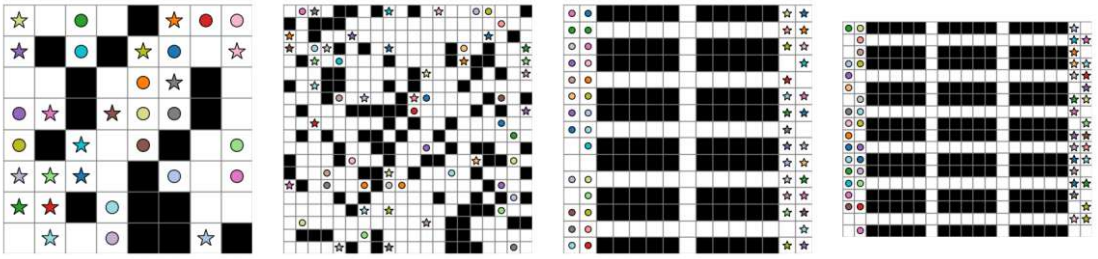
Random instances are created of sizes 8x8 and 20x20, with the number of agents between 10–21 and 16–29 in increments of two, respectively, with $[0\%, 10\%, 25\%]$ of obstacles.

For the *warehouse* instances, we generate 15x15 and 21x18 grids with the number of agents between 16–29 in increments of two.

For each combination of parameters, we generate three instances randomly.

Examples of the different types of generated instances are shown in Figure 4.18.

For our experimental evaluation, we select every third instance for tuning and the rest for validation. Without the unsatisfiable instances (due to the generation process or insufficient time limit), we are left with 39 instances for tuning and 81 for validation.



(a) Random 8x8 (b) Random 20x20 (c) Warehouse 15x15 (d) Warehouse 21x18

Figure 4.18: Example MAPF instances using our instance generator. With agent start positions shown as dots and targets shown as stars.

4.6.3 Experimental Evaluation

In addition to the general methodology 4.2, we will consider an alternative configuration space in combination with a declarative neighbourhood.

The alternative configuration space is shown in Figure 4.19; it contains the dynamic and powerlaw selection strategies with their respective parameters, the `bb,lin` and the `usc,3` optimization strategy as well as different predefined configurations of the clingo solver, that will be applied during search, and finally two sets of search timeouts. Additionally, it contains two types of relaxation rates, the first using absolute values corresponding to the number of agents 4.20, while the second selects them proportionally.

```
"configurationSpace": {
    "strategy": [
        {"name": "dynamic", "unsatStrikes": [2, 3, 4], "timeoutStrikes": [1, 3]},
        {"name": "powerlaw", "alpha": "$eval:(-5.0, 5.0)"}
    ],
    "relaxOperators[0].sizes" : [
        [ 2, 3, 4, 5, 8, 10 ], [ 0.1, 0.2, 0.3, 0.4, 0.5, 0.8 ]
    ],
    "searchOperators[*].configuration.solver.optStrategy" : [ "bb,lin", "usc,3" ],
    "searchOperators[*].configuration.configuration" : [
        "auto", "tweety", "handy", "crafty", "trendy"
    ],
    "searchOperators[*].timeouts": [ [ 5, 15, 30, 60 ], [ 10, 30, 60, 120 ] ]
}
```

Figure 4.19: The alternative configuration space for the declarative MAPF evaluations.

The declarative neighbourhood can be seen in Figure 4.20. It selects a subset of agents along with all of the derived atoms corresponding to their paths for relaxation.

```
39 _lns_select(A) :- agent(A).
40
41 _lns_fix(exec(A, M, T), A) :- _lns_select(A), exec(A, M, T).
42 _lns_fix(at(A, X, Y, T), A) :- _lns_select(A), at(A, X, Y, T).
43 _lns_fix(at_goal(A, T), A) :- _lns_select(A), at_goal(A, T).
44 _lns_fix(at_goal_back(A, T), A) :- _lns_select(A), at_goal_back(A, T).
45 _lns_fix(cost(A, T, 1), A) :- _lns_select(A), cost(A, T, 1).
```

Figure 4.20: The declarative neighbourhood for the MAPF encoding selecting entire agents and their paths for relaxation.

For the **Custom Clustering**, we implement a custom feature extractor which reports back the following features for every instance: (i) the ratio of agents to the grid area, with a higher value generally indicating a more difficult instance due to space restrictions and the higher chance for conflicts, (ii) the size of the grid, (iii) the average number of line intersections per agent, where we count how often straight start-goal line segments intersect pairwise. This should be a simplified heuristic of the number of potential conflicts between agents, since if more paths intersect, agents are more likely to interfere with each other. Additionally, we compute the Manhattan distances between the agents' starting positions and their destinations, and report (iv) the sum and (v) the average.

Results and Observations

Figure 4.21a shows the tuning results with the configuration space optimizing the different ALASPO selection strategies and their parameters. It is clear that the tuning results for MAPF are suboptimal. Almost every approach has an average objective score that is higher than the ALASPO baseline. The only exception to this rule is the custom clustering run on 1000 trials with a negligible average relative improvement of 0.3%. These results contradict one of our core hypotheses, **H1**, which assumes that tuning should produce better performing configurations than the default.

Despite the unfavourable results, we can make some observations about the relative performance between the different tuning variants as well as the trial budget.

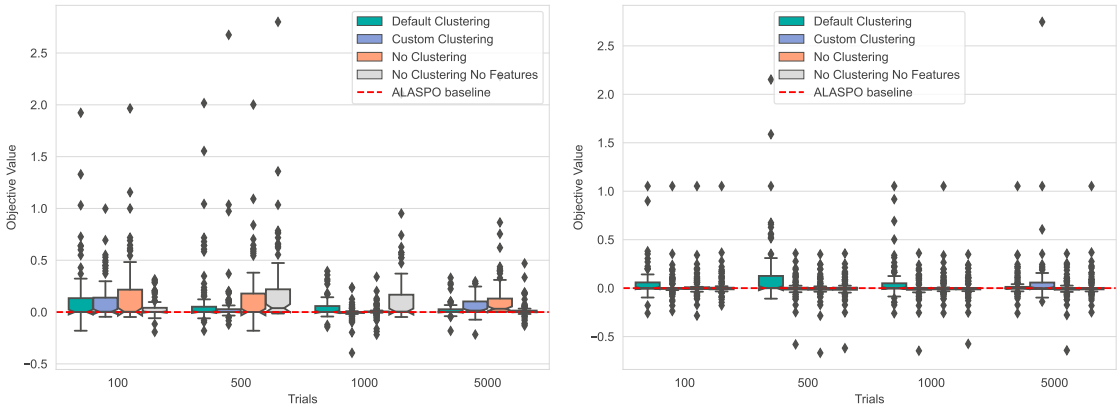
For 100 trials, **No Clustering No Features** produces better objective values than any other method, with moderate outliers in comparison. For the next larger trial size, however, it performs significantly worse than before, and contains the worst outlier overall. At 5000 trials, it again performs comparable to the results at only 100. This inconsistent behaviour contradicts **H5**, which suggests that increasing the number of trials yields better configurations.

In contrast, the default clustering shows a somewhat linear improvement in overall quality with an increasing number of trials, which does, in turn, directly support **H5**. Furthermore, it can be argued that the same hypothesis holds for the other two approaches, **Custom Clustering** and **No Clustering**, particularly with regard to the generated outliers.

Since the results are very volatile in general, it is worth looking at which ALASPO strategies are considered best by the tuning: At 5000 trials, 9 out of 10 clusters choose the dynamic strategy with different parameters for **Default Clustering**, with **No Clustering No Features** also selecting the dynamic strategy. The **Custom Clustering** selects the dynamic strategy for two of its three clusters, with the powerlaw strategy and a high alpha chosen for the third. Finally, **No Clustering** determines that the best strategy is the powerlaw strategy with a high alpha value.

Looking at the results, it is evident that the default ALASPO selection strategy – the dynamic strategy – performs best for the MAPF, leaving virtually no room for improvement for the tuning, making hypothesis **H1** unviable for this problem. This is further highlighted by the fact that **No Clustering**, which uses only the powerlaw strategy, performs strictly equal or worse than the baseline at 5000 trials, with the best objective values only equal to the baseline for 15 out of 81 instances.

Furthermore, it seems that the default values for the parameters **unsatStrikes** and **timeoutStrikes** of the dynamic strategy are also adequate for the tested MAPF instances: **No Clustering No Features** only changes `timeoutStrikes` – the number of times the solver must timeout during the search to either increase the time limit or decrease the relaxation rate – from 1 to 4, resulting in 19 instances with improved results over the baseline. However, the average of these 19 instances is a relative improvement of only 3.7%; while 28 instances perform worse than before, with 8% higher scores on average.



(a) Tuning results for the MAPF with the standard configuration space.

(b) Tuning results for the MAPF with the alternative configuration space 4.19 and declarative neighbourhood 4.20.

Figure 4.21: Comparison of tuning results for MAPF using the standard (left) and alternative (right) configuration spaces. Objective values are relative differences to the ALASPO baseline. Detailed results are available in Tables A.4 and A.5 in the Appendix.

Figure 4.21b shows the tuning results with the alternative configuration space 4.19 and using the declarative neighbourhood 4.20. Unfortunately, the addition of the declarative neighbourhood and the ability to change solver configurations does not result in any

convincing improvements over the baseline, again disproving hypothesis **H1**. In fact, for any number of trials, all approaches result in an average relative difference that is slightly worse than the default ALASPO configuration. Moreover, with the exception of the default clustering at $[100, 500, 1000]$ trials and the custom clustering at 5000, all approaches perform remarkably similarly to each other, contradicting both hypotheses **H2** and **H5**.

4.7 Discussion

Overall, we found that tuning the available ALASPO strategies and their corresponding parameters significantly improves the performance of ALASPO for the TLSPS and the TSP, while producing moderate results for the VLP and virtually unusable results for the MAPF. These findings are generally in support of our original hypothesis **H1**, particularly for the TLSPS and the TSP, while highlighting that it is not applicable to the MAPF, where the default configuration already performs best.

For the TLSPS, tuning with only 100 trials resulted in average improvements of about 5% and lower objective scores in the range of $11\% - 22\%$ for a quarter of the instances, with the default clustering performing best.

The default clustering also performed best for the VLP, with average improvements between $3\% - 5\%$ and little improvement gained by increasing the number of trials after 500.

For the TSP, the approach without clustering and no instance features did best, resulting in lower objective values by 18% on average at 100 trials, and 28% from 500 trials and up.

Finally, tuning for the MAPF did not find any configurations that consistently performs better than the default. Using a declarative neighbourhood that selects a subset of agents along with their paths for relaxation, and allowing solver specific configurations in an alternative configuration space, did not change this unfavourable result.

A key observation is the fact that with a small number of trials, the clustered approaches may perform worse than the unclustered methods whenever there is a single best configuration for all instances. This can be seen at 100 trials for both the TSP and the MAPF, where the powerlaw and dynamic strategies, respectively, perform best for all types of instances. This is due to the fact that the clustered variants have to share the trial budget for each cluster, whereas without clusters SMAC3 has more trials available to find the overall best configuration. In these cases, hypothesis **H2** is directly contradicted, as the implicit assumption of heterogeneous instances does not hold, which in turn supports **H4**.

Furthermore, it is evident that the performance of the custom clustering is highly dependent on the implementation of the custom feature extractor, whereas the default clustering is highly dependent on the structure of the instances. The former is illustrated

by the fact that the **Custom Clustering** performs worse than the default clustering for the VLP. While the TSP instances prove to be the worst case for the latter, providing virtually no information about their characteristics other than the number of vertices. These results show that hypothesis **H3** can only hold if the custom features extracted from instances are successful in capturing distinct optimization behaviours.

We also observe that tuning with 5000 trials almost always gives the best objective values. However, **Custom Clustering** for the TLSPS and **No Clustering No Features** for the VLP performed worse than with fewer trials. A possible reason for this, and by extension why hypothesis **H5** does not hold universally, could be overfitting on the tuning data set.

Tuning the selection strategies for the MAPF also highlights the obvious fact that when the default strategy performs best, tuning is of little use. In cases like these, hypothesis **H1** becomes trivially inapplicable.

Another reason for the suboptimal tuning performance of the MAPF may be that the instances are not diverse enough in their optimization behaviour to justify the use of different selection strategies. Although we generate 4 *visually* distinct types of instances, the random placement of start and goal positions has a huge impact on defining the difficulty and characteristics of an instance. Even a small change in these positions can have a large impact; nuances that our clustering was not able to account for.

CHAPTER 5

Related Work

Hutter et al. [HHS07, HHLBS09] present the ParamILS framework for automated Algorithm Configuration (AC). It iteratively uses local search in the configuration space to find local optima and perturbation phases to escape from these optima, corresponding to a traditional exploitation and exploration step, respectively. Each perturbation phase is followed by a local search phase, after which the framework decides whether the new local optimum is also a global optimum, and if not, reverts to the previous incumbent. The authors report consistent and substantial performance improvements using ParamILS on a variety of different algorithms [Hoo12].

Xia et al. [XS24] explore the problem of SAT based tree decomposition with the TW-SLIM (treewidth SAT-based local improvement method) framework, which, instead of solving a large translation of the problem into SAT, uses multiple SAT calls to find decompositions for trees of smaller width. To improve the performance of TW-SLIM, the authors then investigate how to best choose and apply configurations for the framework. They propose an iterative cascading policy that dynamically applies the best configurations found offline, based on the current updated instance features at each round. To find these best configurations, the authors cluster instances based on their features, and subsequently use the SMAC3 framework to find the best configuration for each cluster separately. The authors report that the proposed policy is effective in improving the performance of the TW-SLIM framework.

Similarly, Song et al. [SLC+23] propose a new instance specific Algorithm Configuration (AC) method for Mixed-Integer Programming (MIP) solvers. The authors first introduce an automated feature extraction for MIP instances, where each instance is represented as a bipartite graph of variables and constraints. Then, random walks extract multiple subgraphs that contain both numerical information as well as structural characteristics. These complex feature sets are then compressed by an auto-encoder and used to cluster instances, resulting in multiple homogeneous groups of instances. Finally, the SMAC3 framework is used to find the best performing configuration for solving each cluster

of instances separately. The authors report that the proposed approach is able to outperform other AC methods on a variety of benchmarks where the set of instances is highly heterogeneous.

Bayesian Optimization (BO) in general has been successfully implemented in several diverse applications such as the following:

Aghaabbasi et al. [AAJ+23] discuss the use of BO for HPO of various machine learning models in predicting the choice of work travel mode. They report that BO is most effective in increasing classification accuracy when tuning k-nearest neighbor models.

In addition, Bayesian HPO has been applied to network intrusion detection based on deep neural networks by Masum et al. [MSH+21]. Their work compares a Deep Neural Network optimized with Random Search HPO to one optimized using BO, and reports that the latter performs significantly better.

Bayesian Hyperparameter Optimization has also been successfully used by Ait Amou et al. [AAXKM22] to automatically tune Convolutional Neural Networks for brain tumor classification, resulting in an overall increase in model performance.

CHAPTER 6

Conclusion and Future Work

We have integrated the SMAC3 framework for Bayesian Hyperparameter Optimization into ALASPO to allow the tuning of ALASPO configurations. In addition, we have implemented a customizable clustering of instances based on their characteristics, allowing us to obtain multiple optimal configurations – one for each cluster. This clustering approach exploits similarities within smaller homogeneous groups for an otherwise heterogeneous set of instances, aiming to improve overall performance. The resulting artifact of a successful tuning run can then be passed to future invocations of ALASPO to automatically select and apply a fitting configuration based on the given problem instance. Subsequently, we have investigated the effectiveness of our tuning approach in finding optimal ALASPO selection strategies on a set of four benchmark problems, using different clustering methods and varying the number of trials.

We show that tuning can provide significant performance improvements for some problems, while also demonstrating its limitations. For the TLSPS, tuning the ALASPO strategies resulted in about 7% better objective scores on average, up to 22% for some instances, with virtually strictly equal or better results than the baseline. The VLP saw more moderate improvements, with an average of 5% better results, while a quarter of the instances still performed worse than the default configuration. Next, the TSP saw significant performance improvements, with average gains of 28% as well as strictly better results than the ALASPO baseline for all instances. Finally, for the MAPF, tuning could not provide meaningful improvements over the default configuration, suggesting that the ALASPO baseline configuration is best suited for this problem.

Overall, the success of the tuning is very dependent on the particular problem at hand. We find it to work best whenever there are sufficient opportunities for improvement over the default configuration, with clustering providing its greatest value over a diverse set of instances.

For future work, it would be interesting to further investigate the performance of the clustered approach on benchmarks with more diverse sets of instances. In addition, since the custom clustering is highly dependent on the features selected by a custom feature extractor, testing and comparing the performance of several such implementations could prove interesting in exploring which instance characteristics are most responsible for particular optimization behaviours. Finally, it may be worthwhile to explore more extensively whether tuning different parts of the ALASPO configuration, besides the selection strategies, can further increase the performance gains of the tuning.

Overview of Generative AI Tools Used

DeepL Write[1] was used throughout this thesis to help find synonyms, correct punctuation and reduce word repetition.

[1] https://www.deepl.com/en/write, accessed: April 15, 2025.

List of Figures

2.1 An overview of the ALASPO system by Eiter et al. [EGH+24]. 11
2.2 The default portfolio of ALASPO. 13
2.3 An excerpt of an ALASPO configuration file containing ASP solver specific options and an initial operator. 14
2.4 An example Bayesian Optimization of a one-dimensional function, by Brochu et al. [BCdF10]. 16

3.1 Overview of the automated tuning in ALASPO. 20
3.2 Comparison of 2D and 3D cluster visualizations for the same instance set. 22
3.3 An example JSON tuning configuration for ALASPO, with a configuration space that tunes different ALASPO strategies with their appropriate settings. 25
3.4 An example configuration space tuning the `unsatStrikes` parameter of the dynamic ALASPO strategy 2.2.2, and the sizes of the first relax operator. 26
3.5 An example DASK configuration for parallelizing the tuning on a SLURM cluster. 26
3.6 An example of configuring the tuning process directly in Python. 27

4.1 The hard constraints for the TLSPS encoding. 32
4.2 The unary resource constraints for the TLSPS encoding. 33
4.3 The soft constraints for the TLSPS encoding. 33
4.4 The `#show` directives used for the TLSPS encoding. 34
4.5 Tuning results for the TLSPS. 36
4.6 The necessary preprocessing steps of the VLP encoding. 38
4.7 The part of the VLP encoding responsible for valve placement. 39
4.8 The rules that propagate the effects of a pipe failure based on water flow for the VLP encoding. 39
4.9 The final part of the VLP encoding, minimizing the maximum undelivered water demand. 40
4.10 Tuning results for the Valves Location Problem (VLP). 41
4.11 The encoding for the Travelling Salesman Problem. 43
4.12 Example TSP instances using different mutations. 44
4.13 Tuning results for the TSP. 45
4.14 Visualization of the instance clustering for the TSP. 46
4.15 The different possible actions and their effects for the MAPF encoding. . 47

4.16 The integrity constraints for the MAPF encoding. 47
4.17 The sum-of-costs optimization for the MAPF encoding. 48
4.18 Example MAPF instances using our instance generator. With agent start positions shown as dots and targets shown as stars. 49
4.19 The alternative configuration space for the declarative MAPF evaluations. 49
4.20 The declarative neighbourhood for the MAPF encoding. 50
4.21 Comparison of tuning results for MAPF using the standard (left) and alternative (right) configuration spaces. Objective values are relative differences to the ALASPO baseline. Detailed results are available in Tables A.4 and A.5 in the Appendix. 51

List of Tables

4.1 Characteristics of the TLSPS instances used. 35

A.1 Detailed experimental results for the TLSPS 4.5. 73
A.2 Detailed experimental results for the VLP 4.10. 74
A.3 Detailed experimental results for the TSP 4.13. 75
A.4 Detailed experimental results for the MAPF 4.21a. 76
A.5 Detailed experimental results for the MAPF 4.21b using the alternative configuration space and declarative neighbourhood. 77

Bibliography

[AAJ+23] Mahdi Aghaabbasi, Mujahid Ali, Michał Jasiński, Zbigniew Leonowicz, and Tomáš Novák. On hyperparameter optimization of machine learning methods using a bayesian optimization algorithm to predict work travel mode choice. *IEEE Access*, 11:19762–19774, 2023.

[AAXKM22] Mohamed Ait Amou, Kewen Xia, Souha Kamhi, and Mohamed Mouhafid. A novel mri diagnosis method for brain tumor classification based on cnn and bayesian optimization. *Healthcare*, 10(3):494, March 2022.

[ASD+20] Noor Awad, Gresa Shala, Difan Deng, Neeratyoy Mallik, Matthias Feurer, Katharina Eggensperger, Andre' Biedenkapp, Diederick Vermetten, Hao Wang, Carola Doerr, Marius Lindauer, and Frank Hutter. Squirrel: A switching hyperparameter optimizer. December 2020. `https://arxiv.org/abs/2012.08180`.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(1):281–305, feb 2012.

[BCdF10] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599, 2010.

[BET11] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, December 2011.

[BKN+19] Jakob Bossek, Pascal Kerschke, Aneta Neumann, Markus Wagner, Frank Neumann, and Heike Trautmann. Evolving diverse tsp instances by means of novel and creative mutation operators. In *Proceedings of the 15th ACM/SIGEVO Conference on Foundations of Genetic Algorithms*, FOGA '19, pages 58–71. ACM, August 2019.

[BSPV02] Mauro Birattari, Thomas Stützle, Luís Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In William B. Langdon,

Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant G. Honavar, Günter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund K. Burke, and Natasa Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, 9-13 July 2002*, pages 11–18. Morgan Kaufmann, 2002.

[CFG+19] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format. *Theory and Practice of Logic Programming*, 20(2):294–309, December 2019.

[CGN+11] Massimiliano Cattafi, Marco Gavanelli, Maddalena Nonato, Stefano Alvisi, and Marco Franchini. Optimal placement of valves in a water distribution network with clp(fd). *Theory and Practice of Logic Programming*, 11(4–5):731–747, July 2011.

[CLH+22] Jingkai Chen, Jiaoyang Li, Yijiang Huang, Caelan Garrett, Dawei Sun, Chuchu Fan, Andreas Hofmann, Caitlin Mueller, Sven Koenig, and Brian C. Williams. Cooperative task and motion planning for multi-arm assembly systems, 2022. `https://arxiv.org/abs/2203.02475`.

[DFJ54] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, November 1954.

[EGH+22] Thomas Eiter, Tobias Geibinger, Nelson Higuera, Nysret Musliu, Johannes Oetsch, and Daria Stepanova. ALASPO: An adaptive large-neighbourhood ASP optimiser. In *Proceedings of the Nineteenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2022)*. International Joint Conferences on Artificial Intelligence Organization, July 2022.

[EGH+24] Thomas Eiter, Tobias Geibinger, Nelson Higuera Ruiz, Nysret Musliu, Johannes Oetsch, Dave Pfliegler, and Daria Stepanova. Adaptive large-neighbourhood search for optimisation in answer-set programming. *Artificial Intelligence*, 337:104230, 2024.

[EGHR+22] Thomas Eiter, Tobias Geibinger, Nelson Higuera Ruiz, Nysret Musliu, Johannes Oetsch, and Daria Stepanova. Large-neighbourhood search for optimisation in answer-set solving. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(5):5616–5625, June 2022.

[EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems*, pages 40–110. Springer Berlin Heidelberg, 2009.

[EKOS13] Esra Erdem, Doga Kisa, Umut Oztok, and Peter Schüller. A general formal framework for pathfinding problems with multiple agents. *Proceedings of the AAAI Conference on Artificial Intelligence*, 27(1):290–296, June 2013.

[FGHS24] Johannes K. Fichte, Tobias Geibinger, Markus Hecher, and Matthias Schlögel. Parallel empirical evaluations: Resilience despite concurrency. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(8):8004–8012, March 2024.

[FH19] Matthias Feurer and Frank Hutter. Hyperparameter optimization. In *The Springer Series on Challenges in Machine Learning*, pages 3–33. Springer International Publishing, 2019.

[FKE+15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[Gei20] Tobias Geibinger. Investigating constraint programming and hybrid answer-set solving for industrial test laboratory scheduling. Master's thesis, Institute of Logic and Computation, TU Wien, Austria, November 2020.

[GHB20] Rodrigo N. Gómez, Carlos Hernández, and Jorge A. Baier. Solving sum-of-costs multi-agent pathfinding with answer-set programming. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(06):9867–9874, April 2020.

[GKKS12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Answer Set Solving in Practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6:1–238, 12 2012.

[GKKS17] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. May 2017. `https://arxiv.org/abs/1705.09811`.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings Fifth Int'l Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[GMM21] Tobias Geibinger, Florian Mischek, and Nysret Musliu. Constraint Logic Programming for Real-World Test Laboratory Scheduling. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(7):6358–6366, May 2021.

[GNB24] Stefan Gössner, Glyn Normington, and Carsten Bormann. JSONPath: Query Expressions for JSON. RFC 9535, February 2024.

[GNP13] M. Gavanelli, M. Nonato, and A. Peano. An asp approach for the valves positioning optimization in a water distribution system. *Journal of Logic and Computation*, 25(6):1351–1369, December 2013.

[GS09] O. Giustolisi and D. Savic. Optimal design of isolation valve system for water distribution networks. In *Water Distribution Systems Analysis 2008*, pages 1–13. American Society of Civil Engineers, April 2009.

[GS10] Orazio Giustolisi and Dragan Savic. Identification of segments and optimal isolation valve system design in water distribution networks. *Urban Water Journal*, 7(1):1–15, February 2010.

[HHLB11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, pages 507–523. Springer Berlin Heidelberg, 2011.

[HHLBS09] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stuetzle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, October 2009.

[HHS07] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, page 1152–1157. AAAI Press, 2007.

[Hoo12] Holger H. Hoos. *Automated Algorithm Configuration and Parameter Tuning*, pages 37–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[HS15] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search Algorithms: An Overview*, pages 1085–1105. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[HW79] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[JAGS17] Rodolphe Jenatton, Cedric Archambeau, Javier González, and Matthias Seeger. Bayesian optimization with tree-structured dependencies. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1655–1664. PMLR, 06–11 Aug 2017.

[JSW98] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492, 1998.

[JW23] Morris A. Jette and Tim Wickberg. Architecture of the slurm workload manager. In *Job Scheduling Strategies for Parallel Processing*, pages 3–23. Springer Nature Switzerland, 2023.

[KFS95] R. D. King, C. Feng, and A. Sutherland. Statlog: Comparison of classification algorithms on large real-world problems. *Applied Artificial Intelligence*, 9(3):289–333, May 1995.

[KJ95] Ron Kohavi and George H. John. Automatic parameter selection by minimizing estimated error. In *Machine Learning Proceedings 1995*, pages 304–312. Elsevier, 1995.

[LEF+22] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022.

[LHL+23] Jiaoyang Li, The Anh Hoang, Eugene Lin, Hai L. Vu, and Sven Koenig. Intersection coordination with priority-based search for autonomous vehicles. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(10):11578–11585, June 2023.

[Lif19] Vladimir Lifschitz. *Answer Set Programming*. Springer International Publishing, 2019.

[LTK+21] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W. Durham, T. K. Satish Kumar, and Sven Koenig. Lifelong multi-agent path finding in large-scale warehouses, 2021. `https://arxiv.org/abs/2005.07371`.

[Men32] Karl Menger. Das botenproblem. In *Ergebnisse eines Mathematischen Kolloquiums*, pages 11–12. Teubner, Leipzig, Germany, 1932.

[MM18] Florian Mischek and Nysret Musliu. The Test Laboratory Scheduling Problem. Technical Report CD-TR 2018/1, November 2018.

[MM21] Florian Mischek and Nysret Musliu. A local search framework for industrial test laboratory scheduling. *Annals of Operations Research*, 302(2):533–562, 2021.

[Moc94] Jonas Mockus. Application of bayesian approach to numerical methods of global and stochastic optimization. *Journal of Global Optimization*, 4(4):347–365, June 1994.

[MSH+21] Mohammad Masum, Hossain Shahriar, Hisham Haddad, Md Jobair Hossain Faruk, Maria Valero, Md Abdullah Khan, Mohammad A. Rahman,

Muhaiminul I. Adnan, Alfredo Cuzzocrea, and Fan Wu. Bayesian hyperparameter optimization for deep neural network-based network intrusion detection. In *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, December 2021.

[MSS+12] Michael M. McKerns, Leif Strand, Tim Sullivan, Alta Fang, and Michael A. G. Aivazis. Building a framework for predictive science. February 2012. `https://arxiv.org/abs/1202.1056`.

[MT98] Victor W. Marek and Miroslaw Truszczynski. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm, K.R. Apt, V.W. Marek, M. Truszczynski, D.S. Warren (eds.), pp. 375-398. Springer-Verlag, 1999*, September 1998.

[Pea01] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, November 1901.

[PR10] David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of Metaheuristics*, pages 399–419. Springer US, 2010.

[Rob49] J. B. Robinson. *On the Hamiltonian game (a traveling-salesman problem).* RAND Corporation, Santa Monica, CA, 1949.

[Roc15] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 126 – 132, 2015.

[Rou87] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, November 1987.

[RP06] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, November 2006.

[RW06] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, 2006.

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin Heidelberg, 1998.

[SLC+23] Wen Song, Yi Liu, Zhiguang Cao, Yaoxin Wu, and Qiqiang Li. Instance-specific algorithm configuration via unsupervised deep graph clustering. *Engineering Applications of Artificial Intelligence*, 125:106740, October 2023.

[SSF+21] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Kumar, Roman Barták, and Eli Boyarski. Multi-agent pathfinding: Definitions, variants, and benchmarks. *Proceedings of the International Symposium on Combinatorial Search*, 10(1):151–158, September 2021.

[SSW+16] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, January 2016.

[TEM+21] Ryan Turner, David Eriksson, Michael McCourt, Juha Kiili, Eero Laaksonen, Zhen Xu, and Isabelle Guyon. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. April 2021. `https://arxiv.org/abs/2104.10201`.

[Unia] University of Potsdam. Potassco, the Potsdam Answer Set Solving Collection. `https://potassco.org/`. Accessed: 2025-03-21.

[Unib] University of Potsdam. Potassco User Guide. `https://github.com/potassco/guide/releases/tag/v2.2.0`. Accessed: 2025-03-21.

[XS24] Hai Xia and Stefan Szeider. Sat-based tree decomposition with iterative cascading policy selection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(8):8191–8199, March 2024.

[YS20] Li Yang and Abdallah Shami. On hyperparameter optimization of machine learning algorithms: Theory and practice. *Neurocomputing*, 415:295–316, November 2020.

[ZLH21] Lucas Zimmer, Marius Lindauer, and Frank Hutter. Auto-pytorch: Multi-fidelity metalearning for efficient and robust autodl. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(9):3079–3090, September 2021.

Appendix

In this section we present more detailed results for the different benchmark problems described in Chapter 4.

Trials	Clustering	Avg	Min	Q1	Median	Q3	Max	Better	Equal	Worse
100	Default	-0.058	-0.249	-0.106	-0.045	-0.002	0.021	57	11	6
100	Custom	-0.041	-0.242	-0.070	-0.009	0.000	0.035	52	13	9
100	NoC	-0.057	-0.240	-0.099	-0.038	-0.002	0.011	60	11	3
100	NoC NoF	-0.055	-0.238	-0.111	-0.035	-0.000	0.050	56	11	7
500	Default	-0.067	-0.254	-0.111	-0.052	-0.001	0.003	56	14	4
500	Custom	-0.064	-0.245	-0.110	-0.046	-0.002	0.046	60	12	2
500	NoC	-0.050	-0.222	-0.095	-0.032	0.000	0.198	51	14	9
500	NoC NoF	-0.065	-0.285	-0.105	-0.040	-0.003	0.002	61	12	1
1000	Default	-0.067	-0.276	-0.112	-0.049	-0.002	0.005	58	13	3
1000	Custom	-0.064	-0.282	-0.111	-0.045	-0.002	0.002	61	10	3
1000	NoC	-0.045	-0.224	-0.090	-0.021	0.000	0.244	50	12	12
1000	NoC NoF	-0.065	-0.288	-0.115	-0.048	-0.001	0.057	58	11	5
5000	Default	-0.069	-0.284	-0.113	-0.055	-0.003	0.002	62	10	2
5000	Custom	-0.059	-0.249	-0.105	-0.041	-0.001	0.122	59	11	4
5000	NoC	-0.065	-0.258	-0.106	-0.042	-0.003	0.009	59	11	4
5000	NoC NoF	-0.053	-0.236	-0.095	-0.030	-0.000	0.203	55	11	8

Table A.1: Detailed experimental results for the TLSPS 4.5. The summary statistics shown are computed from values representing relative differences compared to the ALASPO baseline and are rounded to three decimal places.

Trials	Clustering	Avg	Min	Q1	Median	Q3	Max	Better	Equal	Worse
100	Default	-0.031	-0.373	-0.092	-0.018	0.011	0.563	75	2	35
100	Custom	0.008	-0.318	-0.068	0.000	0.054	0.781	54	2	56
100	NoC	0.042	-0.368	-0.034	0.018	0.105	0.570	40	2	70
100	NoC NoF	0.023	-0.364	-0.033	0.010	0.084	0.358	42	5	65
500	Default	-0.053	-0.516	-0.132	-0.044	0.001	0.503	80	3	29
500	Custom	-0.042	-0.454	-0.107	-0.016	0.019	0.399	69	4	39
500	NoC	0.021	-0.310	-0.034	0.008	0.087	0.345	48	2	62
500	NoC NoF	-0.037	-0.395	-0.116	-0.026	0.010	0.520	70	6	36
1000	Default	-0.042	-0.338	-0.122	-0.037	0.008	0.444	74	4	34
1000	Custom	0.008	-0.314	-0.055	-0.004	0.051	0.604	58	4	50
1000	NoC	-0.052	-0.427	-0.111	-0.036	0.005	0.358	80	3	29
1000	NoC NoF	-0.042	-0.407	-0.117	-0.023	0.012	0.446	72	3	37
5000	Default	-0.053	-0.458	-0.113	-0.047	0.006	0.444	75	4	33
5000	Custom	-0.027	-0.357	-0.098	-0.031	0.015	0.529	73	4	35
5000	NoC	-0.055	-0.456	-0.112	-0.032	0.003	0.292	78	2	32
5000	NoC NoF	-0.033	-0.395	-0.112	-0.025	0.021	0.463	67	4	41

Table A.2: Detailed experimental results for the VLP 4.10. The summary statistics shown are computed from values representing relative differences compared to the ALASPO baseline and are rounded to three decimal places.

Trials	Clustering	Avg	Min	Q1	Median	Q3	Max	Better	Equal	Worse
100	Default	-0.031	-0.459	-0.153	-0.010	0.077	0.479	43	1	36
100	Custom	-0.025	-0.486	-0.125	-0.012	0.072	0.407	44	0	36
100	NoC	0.106	-0.377	0.026	0.079	0.170	0.597	16	0	64
100	NoC NoF	-0.180	-0.582	-0.280	-0.159	-0.056	0.051	77	0	3
500	Default	-0.080	-0.419	-0.183	-0.054	0.011	0.212	56	0	24
500	Custom	-0.282	-0.690	-0.420	-0.273	-0.119	-0.022	80	0	0
500	NoC	-0.283	-0.684	-0.405	-0.269	-0.128	-0.021	80	0	0
500	NoC NoF	-0.290	-0.692	-0.422	-0.273	-0.130	-0.021	80	0	0
1000	Default	-0.159	-0.444	-0.240	-0.127	-0.074	0.040	78	0	2
1000	Custom	-0.261	-0.692	-0.416	-0.233	-0.091	-0.015	80	0	0
1000	NoC	-0.287	-0.689	-0.418	-0.280	-0.132	-0.024	80	0	0
1000	NoC NoF	-0.287	-0.691	-0.415	-0.265	-0.124	-0.017	80	0	0
5000	Default	-0.289	-0.691	-0.418	-0.274	-0.128	-0.016	80	0	0
5000	Custom	-0.290	-0.686	-0.428	-0.282	-0.126	-0.021	80	0	0
5000	NoC	-0.291	-0.698	-0.424	-0.284	-0.130	-0.015	80	0	0
5000	NoC NoF	-0.291	-0.690	-0.427	-0.284	-0.132	-0.022	80	0	0

Table A.3: Detailed experimental results for the TSP 4.13. The summary statistics shown are computed from values representing relative differences compared to the ALASPO baseline and are rounded to three decimal places.

Trials	Clustering	Avg	Min	Q1	Median	Q3	Max	Better	Equal	Worse
100	Default	0.143	-0.180	0.000	0.000	0.134	1.923	6	36	39
100	Custom	0.110	-0.046	0.000	0.004	0.139	0.998	5	34	42
100	NoC	0.169	-0.048	0.000	0.003	0.216	1.965	6	34	41
100	NoC NoF	0.032	-0.191	0.000	0.000	0.041	0.316	17	35	29
500	Default	0.130	-0.180	0.000	0.000	0.052	2.016	16	32	33
500	Custom	0.080	-0.121	0.000	0.000	0.028	2.675	14	35	32
500	NoC	0.151	-0.180	0.000	0.000	0.179	2.002	7	35	39
500	NoC NoF	0.209	-0.014	0.000	0.038	0.220	2.800	2	31	48
1000	Default	0.040	-0.143	0.000	0.000	0.058	0.395	12	35	34
1000	Custom	-0.003	-0.394	-0.008	0.000	0.000	0.239	27	36	18
1000	NoC	0.006	-0.218	0.000	0.000	0.006	0.340	20	32	29
1000	NoC NoF	0.142	-0.048	0.000	0.004	0.168	2.108	5	32	44
5000	Default	0.021	-0.182	0.000	0.000	0.027	0.331	15	35	31
5000	Custom	0.052	-0.216	0.000	0.014	0.103	0.297	9	25	47
5000	NoC	0.140	0.000	0.004	0.032	0.130	2.277	0	15	66
5000	NoC NoF	0.019	-0.131	0.000	0.000	0.014	0.470	19	34	28

Table A.4: Detailed experimental results for the MAPF 4.21a. The summary statistics shown are computed from values representing relative differences compared to the ALASPO baseline and are rounded to three decimal places.

Trials	Clustering	Avg	Min	Q1	Median	Q3	Max	Better	Equal	Worse
100	Default	0.051	-0.259	-0.006	0.000	0.059	1.053	23	31	27
100	Custom	0.026	-0.236	-0.007	0.000	0.002	1.053	26	34	21
100	NoC	0.025	-0.284	-0.009	0.000	0.009	1.053	24	34	23
100	NoC NoF	0.030	-0.179	-0.012	0.000	0.005	1.053	25	34	22
500	Default	0.142	-0.109	0.000	0.000	0.126	2.153	15	32	34
500	Custom	0.005	-0.580	-0.014	0.000	0.005	0.359	25	33	23
500	NoC	0.003	-0.668	-0.016	0.000	0.000	0.349	27	34	20
500	NoC NoF	0.005	-0.619	-0.017	0.000	0.005	0.358	26	32	23
1000	Default	0.061	-0.260	-0.005	0.000	0.050	1.053	21	33	27
1000	Custom	0.005	-0.647	-0.017	0.000	0.001	0.360	26	34	21
1000	NoC	0.023	-0.257	-0.012	0.000	0.002	1.053	26	34	21
1000	NoC NoF	0.006	-0.575	-0.013	0.000	0.000	0.349	27	35	19
5000	Default	0.033	-0.259	-0.006	0.000	0.013	1.053	24	33	24
5000	Custom	0.078	-0.139	-0.009	0.000	0.058	2.747	23	32	26
5000	NoC	0.002	-0.642	-0.016	0.000	0.001	0.356	27	32	22
5000	NoC NoF	0.028	-0.249	-0.013	0.000	0.002	1.053	26	34	21

Table A.5: Detailed experimental results for the MAPF 4.21b using the alternative configuration space and declarative neighbourhood. The summary statistics shown are computed from values representing relative differences compared to the ALASPO baseline and are rounded to three decimal places.