An AI Approach to Large-Scale Medical Appointment (Re)Scheduling Using ASP

Stavros Kanias

University of Patras, University Campus Rion, Patra, GR26504, Greece

Abstract

Despite their significant solving capabilities ASP solvers have not yet crossed the line between academic and widespread industrial use. Even in areas where classical AI solutions developed in the ASP framework exceed solvers that utilize procedural algorithms in terms of raw performance, the latter approach is usually preferred. The lack of integration between ASP solvers and standard software development components, such as SQL databases, is commonly cited as a reason for this tendency. To this end, a Python interface for converting an SQL database (where the healthcare system's data reside) to a logic program knowledge base was developed[1]. This interface supports all the basic SQL transactions adding a level of abstraction on top of the Potassco backed CLORM[2] module. Simultaneously, a new approach to medical appointment (re)scheduling is proposed taking into consideration the total community benefit, showcasing the immense potential of ASP in the modern software industry. An application that uses the ASP-SQL interface to solve the medical appointment (re)scheduling problem with this approach is the final product of this research.

Keywords

Answer Set Programming, Appointment Rescheduling, Common Benefit Maximization, Clingo, Python

1. Introduction

Since its inception in 1972 [1], logic programming has developed to the point where contemporary solvers, such as clingo, are able to solve complex, mostly NP-complete [2], optimization problems when they are expressed under answer set semantics. Based on those capabilities and the fact that it can greatly reduce development time due to its declarative nature, ASP has been successfully used in several research areas, including Artificial Intelligence, Bio-informatics , and Database querying while recently entering the software industry as a viable software alternative to procedural algorithms [3]. Specifically, modern healthcare systems pose a plethora of challenges in the road to full digitalization and the improvement of patient experience, one of which is the management of appointments in a way that maximizes the community benefit. This challenge arises from the sheer scale of data that characterizes a healthcare system and the ethical implications that come to the forefront when a patient's health is at stake. In fact, digital systems are essential to overcome this challenge and as the paper proposes, AI and in particular ASP, can provide innovative perspectives and new capabilities to modern digital healthcare systems. Before solving the medical rescheduling problem with AI reasoning a technical challenge must be overcome. The gap between ASP solvers and the software industry, since every modern application consists of many separate components interacting with one another, is a phenomenon commonly attributed to the absence of abstract interfaces between ASP solvers and modern high-level languages. For example, although few attempts to integrate SQL databases with logic programs have been made [4] an automated solution for converting an SQL database into an ASP knowledge base in a high-level programming language environment has yet to become available. Thus, in this paper the structure and use of an interface between a PostgreSQL database and a clingo logic program developed to simplify the data flow between the database and the solver is presented.

The final software brings the possibility of using ASP solvers in the backend of industrial scale software applications to the spotlight. The Python programming language is used for the interface development utilizing the Predicate class of the Potassco backed CLORM module to encode the database records and the Control class to execute the logic program on the derived knowledge base. In Section 2 a more in-depth examination of the medical appointment (re)scheduling problem is presented followed by a simple example comparing a classical to an AI approach for attaining the optimal solution to the problem. Section 3 contains a structural and logical analysis of the ASP rescheduler program. First, the structure of the SQL database is presented. The second subsection is dedicated to the structure and usage of the Python interface. In the third subsection, the code

ENIGMA-23, September 03-04, 2023, Rhodes, Greece

up1066563@upnet.gr

0009-0003-2328-9660

[1] The source code for the application can be found on https://github.com/StavrosKanias/Medical_Appointment_Rescheduling_App

[2] At the time of writing CLORM is an open-source project with its source code residing on https://clorm.readthedocs.io

structure rule content of the logic program used to solve the problem is thoroughly analyzed. Furthermore, in Section 4, various statistics describing the performance of the solution for different datasets are presented and comments are made on the factors that affect the program's execution time. Finally, in Section 5, possible implications of a real-world application based on this solution are presented.

2. The Appointment (Re)Scheduling Problem

Consider a hospital with thousands of patients and dozens of doctors. Each doctor works for 8 hours daily providing 8 one-hour appointment timeslots per day from Monday to Friday. In most contemporary healthcare systems patients can book an appointment only if it is currently available. As soon as a vacant timeslot is requested it is granted by the system to the patient preventing all the other patients from requesting the same timeslot. The problem that arises from this policy is that in the case of a cancellation the timeslot will again become available but will most likely be wasted since no waiting queue was formed for another patient to claim it instantly. Even in the case where a waiting queue exists a first-come-first-served approach will be followed, and the system will grant the request with the highest score for the timeslot in question. In this case, given that the patients would freely choose a better timeslot for themselves if asked, the community optimal assignment may be missed. As described in the following example, using an AI approach to maximization, if the system can choose not to give the canceled appointment to the highest-scoring request of the queue, a chain-reaction of rescheduled appointments can occur through which a much higher community benefit can emerge. A scenario that allows patients to select more than one timeslot with ascending preference order, where 6 patients claim 4 timeslots can be seen in Table 1.

Table 1
Initial Schedule

Timeslot	Patient Queue	Request Score
Mon (9 am)	Nikos (Granted)	90
	George	87
	Kostas	86
Tue (1 pm)	George (Granted)	72
	John	60
	Maria	55
Thu (11 am)	John (Granted)	45
	Maria	40
Fri (3 pm)	Kostas (Granted)	71
	Despoina	70

The score of each request is calculated as the weighted sum of two factors, the request's preference, and the patient's priority. As preference we define the request's temporal order among all the patient's requests. For example, the first request made by the patient to the system will be registered with priority 1, the next with priority 2 etc. The second factor, priority, is considered as a combined metric of the patient's health calculated by the system according to the patient's history. It is used to determine the urgency attributed to a particular request. In the current implementation the scores are derived from the following formula.

$$Score = \frac{0.3}{preference} + 0.7 \cdot priority \quad (1)$$

In the scenario presented in Table 1, after defining the community benefit as the sum of the scores of the granted requests, a community benefit of 278 is offered by the healthcare system to society. In the case where Nikos cancels his appointment, the first timeslot becomes unclaimed. If a first-come-first-served approach is used the timeslot will be granted to George and thus a community benefit of 258 will emerge. In contrast the AI approach previously mentioned and described in depth in the next section will create the schedule seen in Table 2 which yields a higher community benefit of 273.

Table 2
Schedule after AI rescheduling

Timeslot	Patient Queue	Request Score
Mon (9 am)	Kostas (Granted)	86
Tue (1 pm)	George (Granted)	72
		60
		55
Thu (11 am)	John (Granted)	45
	Maria	40
Fri (3 pm)	Despoina (Granted)	70

3. Developing an AI solution using SQL, Python and ASP

3.1. Database design

A thoughtfully designed and implemented database was necessary to develop and test the clingo – database interface. The relational model was chosen for the design to make the structure of the database more obvious. Since the interface is developed with backend applications in mind the PostgreSQL environment was used alongside the Python psycopg2 module. Simulating a typical hospital application, the database was structured as can be seen in Figure 2. This diagram was transformed into the final database schema seen in Figure 3 and translated to Python code as a dictionary of dictionaries.

To be more specific, the database schema in the source code is a dictionary with the keys being the entities and the values being subdictionaries with the

entity's attributes as keys and tuples of two to four elements as their values. Each value follows the following convention, the first element is the type of the attribute, and the second element is a boolean signifying if the attribute is a primary key. The third element (if it exists) signifies the uniqueness of the

```
'REQUEST': {
"ID": ('integer', True)
"PATIENT_ID":
('text', False, 'PATIENT', 'ID'),
"TIMESLOT_ID":
('integer', False, 'TIMESLOT',
'ID'),
"PREFERENCE": ('integer', False),
"SCORE": ('integer', False),
"STATUS": ('integer', False)
}
```

Figure 1: Conversion of the Request entity from the database schema to Python

attribute in a three-element tuple or the foreign entity that this attribute points to if it is a foreign key. Finally, the fourth element appearing only in foreign keys is the foreign attribute to which the attribute refers to. An example of this conversion methos for the request entity can be seen in Figure 1.

Although in a real healthcare system the complexity of such a database would be much higher, only the request, timeslot and doctor entities are needed to feed the clingo rescheduler with data to output a general solution for the healthcare system as can be seen in the following section, making every other information stored in the database irrelevant.

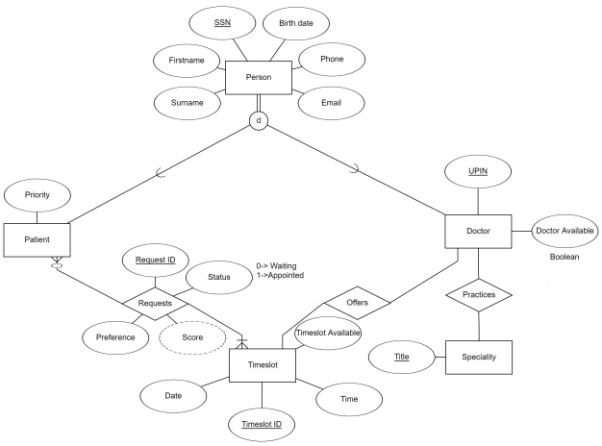

Figure 2: Database Entity Relationship Diagram [9]

3.2. Python interface

To fuel clingo with large scale data efficiently an interface was developed in the Python programming language. The functionalities required to enable the data flow between a database and the clingo solver were implemented as a KnowledgeBase Python class. This class instantiates a knowledge base structure that extends the FactBase class from the Potassco backed CLORM framework. The goal of the interface is to create a one-to-one copy of the database in a format intended to function as input for the solver provided by the Control class of the clingo Python module. A typical use case of the interface can be seen in Figure

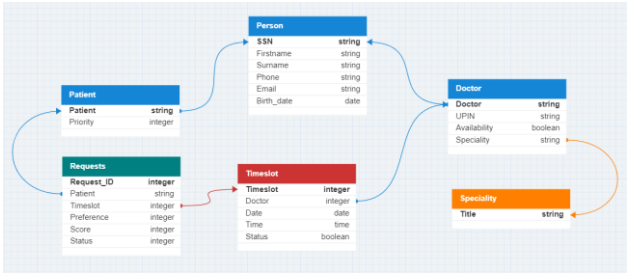

Figure 3: Database relational schema [10]

5.

For the conversion of the database data to clingo predicates, two approaches were taken into consideration, each with its own strengths and weaknesses. Both extensively use Python's dynamic typing capabilities for creating the KB predicates based on the database schema of Figure 3. An example of this record to predicate translation for both encodings can be found in Figure 4.

These predicates extend the Predicate class found in the CLORM framework specifying each time the name and type of each database attribute. At this point it is useful to mention that clingo only supports integers as numerical values and thus the fabricated data used to test the application had to be adjusted to this constraint. Another challenge that arose while attempting to preserve the database properties in the knowledge base was the absence of the concept of foreign keys which was manually implemented.

To this end the merged approach was implemented intending on keeping the information of each database record unified in one predicate and thus enabling the use of joins and other SQL capabilities based on matching foreign keys. Simultaneously it achieves minimization of the number of predicates in the knowledge leading to faster query execution times. Although the ability to solve the problem utilizing the merged encoding is given to the user (the ASP program must conform with the knowledge base encoding) the execution time for the optimization process is lacking behind compared to the second encoding. Another problem with this approach in the context of ASP code development is the creation of logic programs with many anonymous variables making them less readable and intuitive and simultaneously complicating the expression of more advanced logical rules.

	id [PK] integer	patient_id text	timeslot_id integer	preference integer	score integer	status integer
1	2	26057784758	1251	1	37	0

(a)

```
request(2,
26057784758,
1251,
1,
37,
0)
```

(b)

```
request(2)
patient_id(2,26057784758)
timeslot_id(2,1251)
preference(2,1)
score(2,37)
status(2,0)
```

(c)

Figure 4: Database record (a), conversion in merged encoding (b), conversion in split encoding(c)

To resolve these issues different forms of encoding were considered for the optimization process. In the end, using a split approach in the process of translating the database records to knowledge base predicates two significant advantages emerged. The first pertains to the linguistic aspect of ASP as it results in the declarative rules of the ASP program to resemble natural language sentences. This is achieved by splitting each table's attribute and directly translating it to a predicate. This means that for every attribute in the database a new class is created containing in the case of a primary key, only the key's value and for all the other attributes of a table the record's primary key and the attribute's value. Although this approach greatly simplifies the process of developing the clingo code and increases the code's readability it also increases the internal complexity of the knowledge base slowing down the basic database transactions (select, insert, update, delete) thus making it less suitable for the data management inside the knowledge base. For example, in order to resolve a select query to a split encoded knowledge base it is necessary to first select all the primary keys that conform to the given conditions and then collect all the needed attributes matching the obtained primary keys. It also produces a knowledge base with more predicates leading to a larger data structure for the solver to handle.

Interface structure

Input: Database name, schema, credentials, and conditions
Solver parameters: Path to solver, Output predicates, Time and model constraints, Input data subset and the type of encoding as a boolean
Output: The optimized answer set and optimization statistics

A typical use case

1: kb = KnowledgeBase (name, schema, credentials, conditions) # Create the knowledge base
2: kb.toFile() (optional) # Export the knowledge base as a text file for a better data overview
3: Create output auxiliary predicates to express the solution
4: solution = kb.run (path, outPreds, searchDuration, models, subKB, merged) # Run an ASP program on the KB data
5: Process the solution data (optional)
6: kb.update(entity, conditions, values) # Update KB and DB with the results of the optimization process

Figure 5: Basic documentation for the Knowledge Base class

3.3. Optimal rescheduling with ASP

Typically, every ASP program starts with the creation of its knowledge base which functions as its input [5]. Using the knowledge base, auxiliary predicates required to express facts and relations necessary for the problem's description are created. At this point the core rules for the generation of all possible answer sets are stated. These answer sets, in their current form, are not always either correct (able to satisfy the real problem) or optimal. They just describe a potential solution of the problem's state space. To reach a solution that satisfies the real problem, integrity constraints are applied, that is to say, each potential solution is tested according to the logical constraints imposed by the real world, becoming an actual solution if it complies.

As each answer set describes a possible set of actions the set generation rules are often referred to as effect axioms. On the contrary, rules that specify immutable subsets of the answer sets are labeled as frame axioms. Finally, an aggregate rule is often used to specify the parameters and the type of the optimization process. In our case a maximization command was given taking into consideration the score of all the granted requests. What happens at a technical level is that the solver sums the parameter specified for every actual solution. If the user wants to print one or more predicates from an answer set, a show command can also be used.

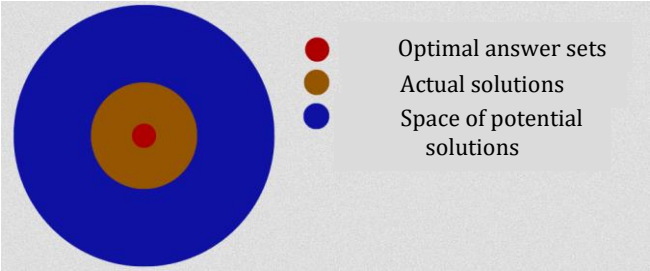

Figure 6: Venn diagram describing the ASP development process.

The clingo rescheduler was developed according to the ASP model described above. Independently of the approach used to encode the knowledge base (split or merged), six auxiliary predicates are used to fully describe the initial schedule.

1. The **granted** predicate that is satisfied by all requests that are granted thus having a state value equal to one.
2. The **appointed** predicate that is satisfied by all the patients having a granted request.
3. The **best** predicate that is satisfied by the single highest scoring request among all the patient's requests. It is generated by arguing that there is no other request belonging to the specific patient that has a greater score.
4. The **onlyOption** predicate that is satisfied by the request belonging to a patient that has made one and only request. It is generated by arguing that there is no other request belonging to the specific patient in general.
5. The **singleRequest** predicate that is satisfied a request that is the first and only request in a timeslot's queue. It is generated by stating that there is no other request targeting this timeslot.

6. The **bestSingleRequest** predicate that is satisfied by the patient's best (highest scoring) request from those requests satisfying the singleRequest property. It is generated by arguing that there is no other single request belonging to this patient with a higher score.

The generated answer sets cover the following three possible types of requests eligible for the system to grant:

1. The **granted** but **not best** requests implying that a patient can retain their currently owned appointment if it appears in the answer set that maximizes the common benefit.

2. The **not granted** requests of a patient already having a **granted** request, that present a **higher score** than the one **currently appointed** to the patient. This enables a patient to receive a request with higher priority if it helps to maximize the common benefit.

3. All the requests that belong to a patient who has not been **appointed** a timeslot if this leads to the optimal answer set.

As for the frame axiom, it generates the **claimed** predicate which is satisfied for every request that is already **granted** and is also the **best** request of the corresponding patient. This practically means that all the already claimed requests cannot be granted to another patient.

At this point, eight **integrity constraints** were applied to the answer sets targeting three specific areas, **logic**, **justice,** and **optimization**. For the logic part two constraints were applied.

1. Each timeslot can be appointed to only one patient.
2. Each patient can only receive one timeslot (from a specific specialty if the general scope is used).

```
:- request(R,_,T,_,_), grant(R,_), request(X,_,T,_,_), grant(X,_), R != X.
```

(a)

```
:- grant(R1), timeslot_id(R1, T), grant(R2), timeslot_id(R2, T), R1 != R2.
```

(b)

Figure 7: The first integrity constraint (only one granted request per timeslot) expressed in clingo for the merged approach (a) and the split approach (b)

The danger that appears when someone approaches this problem with the maximization of the common benefit in mind is that many times the system behaves unfairly towards the individual. To prevent this possibility and thus create a just system the following two constraints were applied.

1. If a patient had an appointment in the previous schedule a timeslot must also be granted to that patient after the rescheduling.
2. If a request is a patient's only option, it cannot be dismissed for the sake of a lower scoring request even if it leads to a chain reaction that maximizes common benefit. If we don't apply this constraint a patient with only one request will most probably never receive an appointment.

The only acceptable case in which the individual gain will be sacrificed is when a better option becomes available for an already appointed patient and if granted will lead to a sub optimal common benefit. As can be seen in the example given in Section 2 George's best request was not granted to him because he already had an appointment, resulting in the maximum common benefit without negatively impacting him.

As for the optimization component, in order to reduce the execution time of the clingo rescheduler four presuppositions were introduced to the system reducing the number of requests to be examined for granting in the pursuit of the optimal answer set.

1. All requests that claim an already claimed timeslot will not be taken into consideration.
2. All the single requests that are not the patient's best single request will not be taken into consideration.
3. If a request is a patient's best single request and it has a better score than the one already granted to the patient it will automatically be assigned to the patient blocking all the other requests made by the patient.
4. If a single request is attributed to an unappointed patient, the system should grant it automatically and block all the other requests made by the patient. Both rules 3 and 4 intend to maximize the number of patients served by the system.

Finally, a maximization rule is given to the solver leading to the selection of the answer set that satisfies all the constraints and simultaneously offers the highest common benefit through the grant action.

4. Evaluation

To assess whether the solution presented in this paper is a viable option for a real healthcare system a data fabricator class was developed in Python to produce pseudo-realistic data according to the database design described in subsection 3.1. The assumption made to produce the timeslot matches most conventional healthcare systems by assigning eight one-hour appointments to each doctor and provides the patients with the freedom to choose as many timeslots as they please with an average of two requests per patient. Also, to be more modular all the parameters needed to create the healthcare system such as the number of patients, the number of doctors, the time period to be simulated (in days), the timeslot availability and the

average demand the healthcare system has to face can be tuned by the user. In this case, a timespan of two weeks (10 working days) was assumed for all the datasets. The tests were first run using a version of the rescheduler which assumes that a patient can request a timeslot from any medical department and is able to only receive one at the time of the rescheduler's execution. This assumption was made to test the rescheduler in a context of a general scope where the AI has knowledge of the whole healthcare system. This does not necessarily have to be the case as described in subsection 3.3 where a solution which also takes specialties into consideration is presented. The per-specialty approach was used to test the performance capabilities of the rescheduler in a very high demand scenario where there is a multitude of requests targeting every timeslot offered by a medical department. In reality, the requests would be more evenly distributed among the medical departments. Finally, the option of creating the schedule described in Section 2 intended for validity check purposes is also provided.

Table 3
Execution time for three different datasets using split encoding

Timeslots	Patients	Requests	Execution Time (s)	
			Best	Worst
400	500	1000	0.06	2.79
960	1200	2400	0.18	152.07
2000	2500	5000	421.4	>1800

After extensive testing with various datasets, it seems that the only intrinsic variable meaningfully affecting the execution time of the rescheduler is the number of timeslots offered by the system. An outside factor that also seems to greatly affect the execution time is the prior state of the schedule. If the previous scheduling granted for example the lowest scoring request for every timeslot the rescheduling time appeared to be multiple times higher than in the case of a more logical scheduling that had previously granted the highest scoring requests for every timeslot. For that reason, both scenarios have been measured and assuming a reasonable demand of 2.5 (the requests being 2.5 times the number of timeslots) the best and worst execution times shown in Tables 3 and 4 were obtained. Two other factors observed to affect the execution time to a smaller degree are the patients to requests ratio and the requests to timeslots ratio (demand). In the first case an increase in the ratio leads to a greater execution time since the effect of the integrity constraint that allows each patient to only one appointment decreases. In the second case an increase in the ratio also leads to a greater execution time since the effect of the integrity constraint that allows each timeslot to be granted to only one request decreases. In every case the introduction of variation in the model makes the optimization process more computationally demanding. Moreover, the distribution of requests over the timeslots and patients over requests can affect the execution time. Namely, a more even distribution will lead to more effective combinations increasing the number of possible answer sets to be examined at the optimization stage. Finally, the type of encoding does not affect the result of the optimization. This was expected due to the identical logic statements used to express the problem. Unlike other attempts on ASP scheduling [3] where the difference in execution time between different encodings can be attributed to the difference in the rule content of the ASP program, differences in the execution time for this implementation based on the encoding of the database for two logically identical reschedulers can be caused by the difference in the optimization path followed by the solver. This can manifest in the case where multiple answer sets provide the same maximum community benefit. To be more precise, according to the syntax of the rules and the expression of the predicates clingo might choose a different path to the optimal answer set or reach a different optimal answer set altogether. In terms of performance the two encodings perform similarly for a small number of models. When the number of models becomes higher (above 200) the cost of choosing a longer path to the optimal answer set in terms of execution time becomes more noticeable. The execution times for three different datasets are displayed in tables 3 and 4 for the merged and split encoding respectively. From the above, we can also conclude that the number of predicates in a knowledge base does not affect the execution time of the rescheduler to a noticeable degree.

Table 4
Execution time for three different datasets using merged encoding

Timeslots	Patients	Requests	Execution Time (s)	
			Best	Worst
400	500	1000	0.09	4.19
960	1200	2400	0.15	448.9
2000	2500	5000	516.9	> 1800

At this point, it can also be useful to grasp the scale of the test data and the complexity faced by the rescheduler. Given a scenario of T timeslots, R requests and d demand with a relatively even distribution the possible combinations are 2^R or $2^{d \cdot T}$ since every request can have only two states, granted, or not granted. This leads, even for a relatively small number of timeslots, to an extremely large number of possible combinations. Although this number is greatly reduced by the application of integrity constraints the fact remains that the complexity needed to be handled by the rescheduler is significant to say the least. As for a real-life application, in order to offer 2000 timeslots in two weeks with the assumptions previously mentioned, a hospital has to employ 25 doctors in a certain specialty, a realistic number for outpatient clinics in Greece.

5. Challenges of real-world implementation

To implement the rescheduler in the real world one must consider how the most basic assumption previously made will be realized. It is unreasonable to bluntly assume that given a better option at any point in time a patient will always choose to reschedule their appointment. Thus, for every request that was not granted and now satisfies the grant predicate the system will have to ask the patient for permission to change the appointment. One of the points where ASP solvers shine is that they not only give the optimized solution but are also able to describe the steps needed to reach it, thus having the ability to reason about the optimization process. Using this feature, for every time a granted request is canceled, the rescheduler will be executed, producing as its output an action chain that can lead to the schedule providing the higher community benefit. Following this chain, the system can notify the patients with an automated message giving the opportunity to claim their better option. At this point a time limit must be set for the patients to accept or reject their new option. Once a patient with a prior place in the chain accepts a request of higher preference than the one currently owned, the next patient will be notified continuing the confirmation process of the action chain in the case of a positive answer. Thus, each chain produced by clingo will only be realized up to the request prior to the one where the inability to re-schedule the appointment due to a lack of confirmation arises for the first time, resulting in the breaking of the chain. Only after a patient has confirmed the appointment change will the database be updated. This means that if a confirmation process is ongoing and a new cancelation takes place, the following options appear. The first option is to wait for the ongoing process of validating the previous rescheduling action chain to finish and then run the rescheduler with the new state of the database as its input. Another and probably most realistic approach is to break the confirmation process of the previous action chain and run the rescheduler after updating the state of the database using only its currently confirmed part. In no case can the execution of the rescheduler precede or coincide with the confirmation process as this can lead to fueling the rescheduler with an obsolete input that corresponds to a state of the database where rescheduling actions caused by the last cancelation have not been taken into consideration.

The second aspect of this solution that needs to be addressed is the practical limitations of its execution time. It was observed that even in the best-case scenario after assuming a logical scheduling has preceded the rescheduling, if the number of timeslots exceeds 3000 or the distribution of data is extremely unfavorable, the time needed for the solver to output the optimal answer set renders it almost unusable in real circumstances. To overcome this challenge three actions can be taken. The first is to simply keep optimizing the code by adding rules that reduce the number of actual solutions taken into consideration during the optimization stage. Although the most theoretically correct approach, even if it was possible to forever continue optimizing, eventually each new optimization will have diminishing returns. To overcome this limitation, as is the case with most serial programs, a restructuring of the rescheduler's code to run in parallel [6] will most probably cause a significant decrease in execution time. Another more practical approach is to set a time limit for the optimization process, interrupting it and retrieving the best available model when it expires. Following that, the system can compare the current community benefit to the newly acquired by the solver and choose the highest scoring one between them. This feature has been implemented using an interrupt in the on_model function used by the solver.

The third available solution, which does sacrifice the general optimality of the output but at the same time grants great flexibility to the application, is the use of a batching approach. Choosing to run the rescheduler on a subset of the data is probably the most useful method to reduce execution time for every kind of rescheduler even more so in a scenario like the medical appointment rescheduling where clear divisions can be made to separate the data needed for optimization according to each medical specialty (as was used in Section 4 for testing) or each doctor. This approach is really an attempt to scale down the problem to a department or an individual level, making the timespan and the demand able to be handled by the rescheduler much wider. The optimality tradeoff increases with each specialization step, and it lies with the system's designer to find the right balance between performance and general maximization of the community benefit. It is worthwhile to mention that the last layer of specialization, providing the most localized maximization of community benefit, will always lie to the resource in demand in this case the doctor's time, a surgical room in ORS [7], a nurse's shift in nurse rescheduling [8] or a meeting room for a business focused application.

6. Conclusion

The capabilities of ASP solvers to optimize problems with an otherwise unmanageably large search space has yet to be utilized widely by the software industry. One of the obstacles has always been the lack of interfaces between the fundamental software components and the solvers. In this paper a software solution to bridge the gap between the database technologies and the clingo scheduler on a high-level programming language was presented. This solution developed in Python was used for the development of a fully functional application to manage large scale medical appointment scheduling and rescheduling. A theoretical analysis of the database design, interface structure and clingo code was made to aid the reader in understanding the logic used to solve the problem with a new perspective, the community benefit maximization approach, thus showing how the new software capabilities provided by integrating AI to modern software projects can revolutionize the service sector making it [9]fair and optimized. During the evaluation process, the application was thoroughly tested with the results being very promising especially

when a ‘best request granted’ scheduling precedes the application’s execution. Even without the scheduling stage the solver proved capable of handling a significant number of patients. It is certainly the case that further optimization and improvements can be made both in the clingo and in the Python fronts to make the application more user-friendly, increase performance and provide new features enabling the rescheduler to handle real-life scenarios faced by large organizations such as hospitals and companies with large scale meeting scheduling requirements. As with many other ASP programs, this application is very adaptable to every sector in need of scheduling and rescheduling solutions, thus making the study of this paper beneficial for both understanding the ASP development process and for providing a base for future applications wanting to solve similar problems.

References

[1] V. Lifschitz, Answer Set Programming, Texas: Springer, 2019.

[2] O. E. Khatib, "Job shop Scheduling under Answer Set Programming Environment," *International Journal of Engineering and Innovative Technology (IJEIT),* vol. 5, no. 5, 2015.

[3] C. Dodaro and M. Maratea, "Nurse Scheduling via Answer Set Programming," 2017.

[4] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri and F. Scarcello, "The DLV System for Knowledge Representation," 2003.

[5] G. Brewka, T. Eiter and M. Truszczyński, "Answer Set Programming at a Glance," 2011.

[6] H. Hoos, R. Kaminski, M. Lindauer and T. Schaub, "aspeed: Solver Scheduling via Answer Set," 2013.

[7] C. Dorado, G. Galatà, M. Kamran, M. Maratea and I. Porro, "Operating Room (Re)Scheduling with Bed," 2021.

[8] M. Alviano, C. Dodaro and M. Maratea, "Nurse (Re)scheduling Via Answer Set Programming," 2014.

[9] N. Jukić, S. Vrbsky, S. Nestorov and A. Sharma, "ERD Plus," 2015. [Online]. Available: https://erdplus.com/. [Accessed 2023].

[10] J. Perez, "DB Designer," 2006. [Online]. Available: https://erd.dbdesigner.net/. [Accessed 2023].