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

Stavros Kanias
Student at the Department of Electrical and Computer Engineering, University of Patras, Greece
up1066563@upnet.gr

Abstract

Despite their significant solving capabilities ASP solvers haven't yet crossed the line between academic and widespread industrial use. Even in areas where classical AI solutions developed with the ASP framework exceed solutions utilizing procedural algorithms, in terms of raw performance, the later approach is most commonly preferred. The lack of integration between ASP solvers and of standard software development components, such as SQL databases, is commonly claimed as an argument for this tendency. In this paper a Python interface for converting a database to a logic program knowledge base is presented[1]. This structure was utilized to solve the problem of medical appointment rescheduling yielding great results.

1 Introduction

Since its inception in 1972, logic programming has developed to the point where contemporary solvers, such as clingo, are able to solve very complex (mostly hard NP-complete) problems when they are expressed under answer set semantics (Vladimir Lifschitz, 2019). There has been though, a gap between ASP solvers and the industry since modern software applications involve components such as a large-scale database. To be more precise, as of today a solution to create a knowledge base for an ASP solver from a database with an automated process has not been developed. In this paper the form and use of a simple interface between a PostgreSQL database and a clingo logic program is presented. The final software solution greatly simplifies the data flow between the database and the solver, bringing the possibility of using ASP solvers in the backend of industrial scale software applications to the spotlight. The programming language used for its development was Python and the code was based on the Potassco backed CLORM interface module. The problem of appointment scheduling and rescheduling was chosen to showcase the immense power of ASP over classical programming techniques in applications that require processing of large-scale sets of data combinations and optimization. The healthcare framework was also chosen to make the application more practical and also to provide a clear example of the ways society can benefit from the use of an AI approach to (re)scheduling problems. 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 and an AI approach to the problem. In Section 3 the ASP solution is presented. First, the structure of the SQL database is analyzed and then follows the logic program used to solve the problem. The third subsection is dedicated to the form and usage of the Python interface. Furthermore, in Section 4, various statistics describing the performance of the solution for different datasets are listed 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 Problem

Assuming a hospital with thousands of patients and dozens of doctors. Each doctor works for 8 hours providing 8 one-hour appointment timeslots. 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 which blocks 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 developed in order 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 from the queue. In this case, given that the patients will 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

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

can occur through which a much better 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 the following table. The scores are derived from the formula:

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

Timeslot	Patient Queue	Request Score
Monday at 9 am	Nikos (Granted)	90
	Giorgos	87
	Kostas	86
Tuesday at 1 pm	Giorgos (Granted)	72
	Ioannis	60
	Maria	55
Thursday at 11 am	Ioannis (Granted)	45
	Maria	40
Friday at 3 pm	Kostas (Granted)	71
	Despoina	70

Table 1: Initial Schedule

In the above scenario, if we define 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 Giorgos and thus a community benefit of 258 will emerge. In contrast the AI approach mentioned above and described in depth in the following paragraph will result in the schedule seen in Table 2.

Timeslot	Patient Queue	Request Score
Monday at 9 am	Kostas (Granted)	86
Tuesday at 1 pm	Giorgos (Granted)	72
	Ioannis	60
	Maria	55
Thursday at 11 am	Ioannis (Granted)	45
	Maria	40
Friday at 3 pm	Despoina (Granted)	70

Table 2: Schedule after AI rescheduling

The result is a schedule that provides a higher community benefit of 273.

3 The Solution

3.1 Database Design

A necessary step for developing and testing the clingo – database interface is the design and implementation of the database itself. The relational model was chosen for the design to make the structure of the database clearer. 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 below.

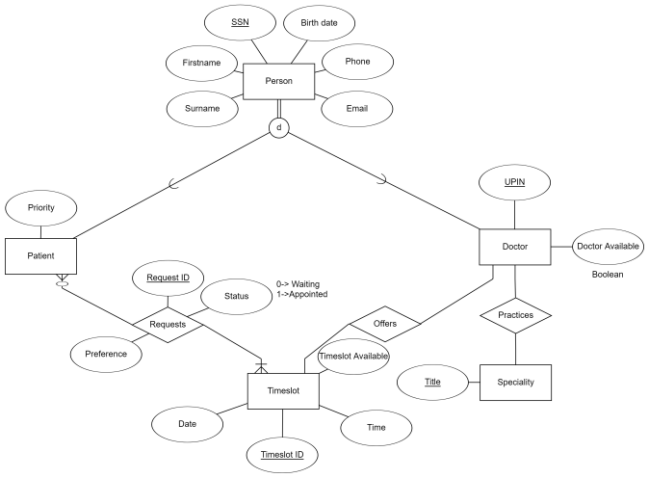

Figure 1: Database ERD diagramm developed with ERD plus.

The above diagram was transformed into the final database schema seen in Figure 2 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,

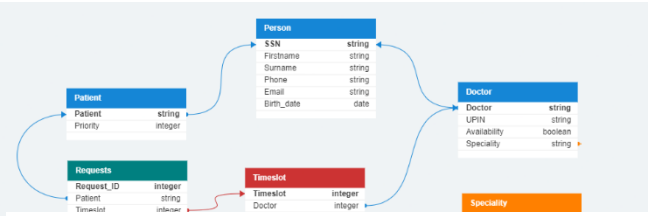

Figure 2: The database schema developed with DB designer

the first element is the type of the attribute, 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 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 particular attribute corresponds to.

Although in a real healthcare system the complexity of such a database would be much higher, only the request entity is needed to feed the clingo rescheduler as can be seen in the following paragraph. Thus, in reality only the entities directly linked to the Request entity are required for the application to be functional.

```
'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 3: Snapshot of the database schema translation in Python for the Request entity

3.2 The Python interface

Although many attempts have been made to solve the scheduling problem with ASP, the effort of fueling clingo with large scale data has been lacking behind. To enable the data flow between a database and the clingo solver a KnowledgeBase Python class was developed. 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 able to be utilized by the solver provided by the Control class of the clingo Python module. A typical use case of the interface can be seen below.

Utilizing the KnowledgeBase class

Input: Database name, schema, credentials, and conditions
Solver parameters: Path to solver, Output predicates, Time and model limits, and Data subset
Output: The optimized answer set and optimization statistics

1: kb = KnowledgeBase (name, schema, credentials, conditions)
2: kb.toFile() (optional)
3: Create the output predicates
4: solution = kb.run(path, outPreds, searchDuration, models, subKB)
5: Process the solution data (optional)
6: kb.update(entity, conditions, values)

Figure 4: A typical use case for the KnowledgeBase class

As for the translation process, two approaches were implemented. Both of them extensively use the Python **dynamic typing** feature for creating the KB predicates based on the database schema. 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. In the **split** approach, the **linguistic** aspect was taken predominantly into consideration. In order for the clingo code to resemble natural language as much as possible each attribute of the database is directly translated to a predicate. This means that for every attribute a new class is created containing in the case of a primary key, only the key's value and for all the other attributes 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 complexity of the code needed to implement the basic database actions (select, insert, update, delete). To be more specific, to implement the aforementioned actions it is necessary to first select all the primary keys that conform to the given conditions and then select all the needed attributes matching the obtained primary key. The **merged** approach was developed to focus specifically on **performance,** intending on minimizing the number of predicates in the knowledge base assuming that this action will lead to smaller execution times. As shown in Section 4 that does not appear to be the case. In this approach all the attributes of a record are merged in one predicate thus leading to an identical size between the knowledge base and the database simultaneously offering an easier way to implement the basic database actions. The great disadvantage of this approach is that it creates logic programs with many anonymous variables making the code less readable and intuitive and also restricts the expression of more complex logical rules.

```
% INTEGRITY CONSTRAINTS
%% A timeslot can be appointed only to one patient
:- grant(R1), timeslot_id(R1, T), grant(R2), timeslot_id(R2, T), R1 != R2.
%% Only one timeslot can be appointed to a patient
:- grant(R1), patient_id(R1, P), grant(R2), patient_id(R2, P), R1 != R2.
% %% If a request is claimed all the other requests for the corresponding timeslot can't be granted
:- grant(R), timeslot_id(R,T), claimed(X), timeslot_id(X,T), R != X.
```

(a)

```
% INTEGRITY CONSTRAINTS
%% A timeslot can be appointed only to one patient
:- request(R,_,T,_,_), grant(R,_), request(X,_,T,_,_), grant(X,_), R != X.
%% Only one timeslot can be appointed to a patient
:- request(R,P,_,_,_), grant(R,_), request(X,P,_,_,_), grant(X,_), R != X.
% %% If a request is claimed all the other requests for the corresponding timeslot can't be granted
:- request(R,_,T,_,_), grant(R,_), request(X,_,T,_,_), claimed(X), R != X.
```

(b)

Figure 5: The first three integrity constraints expressed in clingo for the split approach (a) and the merged approach (b)

3.3 Optimal (Re)Scheduling with ASP

Typically, every ASP program starts with the creation of its knowledge base which represents its input. 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 the 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 express all the possible states that can occur in the problem's state space. In order to come to a solution that satisfies the real problem integrity

constraints must be applied, that is to say, the logical constraints that the real world imposes upon the set of possible solutions have to be taken into consideration. As these sets often describe a possible set of actions the set generation rules are often referred to as effect axioms. Rules that specify immutable subsets of the answer sets (frame axiom) can also appear in this section. Finally, an aggregate rule can be used for optimization. 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 answer set that satisfies the problem created by the previous commands. If the

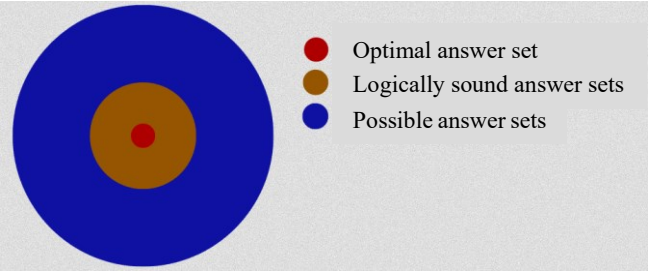

Figure 6: Venn diagramm describing the ASP development process.

user wants to print a predicate that appears on an answer set, a show command can also be used.

The clingo rescheduler was developed according to the ASP model described above. Independently of the approach used to codify the knowledge base (split or merged), six auxiliary predicates were used:

1. The **granted** predicate that is satisfied by all requests that are granted (having a state value equal to one) with the current schedule.
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 particular timeslot.
6. The **bestSingleRequest** predicate that is satisfied by the request that has the greatest score among all the requests of a patient that satisfy the singleRequest property. It is generated by arguing that there is no other single request belonging to this particular patient with a higher score.

The generated answer sets cover the following three possible types of requests that can be granted:

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 community 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 community 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 can't 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.

The danger that appears when someone approaches this problem with the maximization of the community benefit in mind is that many times the system might behave 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 community benefit. If we don't apply this constraint a patient with only one request will most likely never receive an appointment.

The only acceptable case in which the individual gain will be sacrificed in order for a greater community benefit to be achieved is when a better option becomes available for an already appointed patient but if granted will have a negative impact on the community benefit. As can be seen in the example given at Section 2 Giorgos' best request was

not granted to him as he already has an appointment and this action led to a greater community benefit.

Finally, to reduce the execution time of the clingo rescheduler four assumptions were provided reducing the number of requests to be examined for granting in order for the optimal answer set to be found.

1. All requests that claim an already claimed timeslot won't be taken into consideration.
2. All the single requests that are not the patient's best single request won't be taken into consideration.
3. If a request is the best only option 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 request is the best only option of an unappointed patient, it will automatically be assigned to the patient blocking 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 also offers the highest community benefit through the grant action.

4 Evaluation

To assess whether the solution examined in this paper is a viable option for a realistic 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 but also 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 ecosystem 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 particular case, a timespan of two weeks (10 working days) was assumed for all the datasets. Finally, the option of creating the schedule described in Section 2 intended for validity check purposes is also provided.

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

Table 2: Execution time in seconds for three different datasets using split mode.

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 2 and 3 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 receive 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 increases the complexity in need of being addressed by the rescheduler.
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, an unexpected conclusion derived from the testing process is that the number of predicates in a knowledge base does not affect the execution time of the rescheduler to a noticeable degree. In contrast, having predicates with many attributes and thus being forced to use many unnamed (or unused) variables in the rescheduler code drastically increases the execution time especially when the data haven't been scheduled properly prior to rescheduling.

Timeslots	Patients	Requests	Execution Time (s)	
			Best	Worst
400	500	1000	0.09	4.19
960	1200	2400	0.21	448.9
2000	2500	5000	516.9	> 1800

Table 3: Execution time in seconds for three different datasets using merged mode.

At this point, it can also be useful to grasp the scale of the test data and the complexity that the rescheduler has to face. 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, a realistic number for outpatient clinics in Greece.

5 Practical implications

To implement the aforementioned rescheduler in the real world one must consider how the most basic assumption previously made will be realized. It is unorthodox to bluntly assume that given a better option at any point in time a patient will always choose to change 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 in order 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 for it to be realized, thus having the ability to reason on the process of reaching the optimized solution. 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 in order for the patients to accept or discard their new option. Once a patient with a prior place in the chain accepts the better scoring request, the next patient will be notified unlocking the next patient's better request in the case of a positive answer. Thus, each chain produced by clingo will only be realized up to the point where the rescheduling of an unconfirmed appointment fails, resulting in the breaking of the chain. Also, in the case of a second cancelation the notification process, if ongoing will be automatically stopped. Only after a patient has confirmed the appointment change will the database be updated. This means that whenever the second cancelation takes place, if the database has been altered the unconfirmed part of the previous chain must be discarded since it does not align from the current state of the database used to output the next action chain.

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 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 answer sets reaching 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. A more realistic approach is to set a time limit for the optimization process, interrupting it and retrieving the best available model when it expires. After that the system can compare the current community benefit to the newly calculated by the solver and choose the best 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 or each particular doctor. This approach in reality is an attempt to scale down the problem to a department or an individual level, making the timespan and the demand manageable by the rescheduler much wider. The optimality tradeoff increases with each specialization step, and it lies to the system's designer to find the sweet spot 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 such as the doctor's time in this case, a surgical room in ORS (Ivan Porro, Giuseppe Galat`a and Muhammad Kamran Khan, 2021) or a meeting room for a business application.

Conclusion

The ability of ASP solvers to optimize problems with an otherwise unmanageably large search space has yet to be utilized by the industry on a large scale. 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 breaching the gap between the database technologies and the clingo scheduler was presented. Simultaneously this Python solution was used for the development of a fully functional application to manage large scale medical appointment scheduling and rescheduling with attention given to the rescheduling component where the capabilities of ASP solvers can clearly be displayed. A theoretical analysis of the database design, interface structure and clingo code was made in order for the reader to understand 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 more fair and optimized. To the assessment part, the application was thoroughly tested with the results being very promising assuming a 'best request granted' scheduling preceding the application's execution. Even without the scheduling stage the solver can also handle a reasonable amount of data. 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 to enable it to handle real-life scenarios faced by large organizations such as hospitals meeting focused companies. 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

Vladimir Lifschitz, 2019, *Department of Computer Science University of Texas at Austin,, USA. Published by Springer Nature Switzerland AG 2019.*

Ivan Porro, Giuseppe Galat`a and Muhammad Kamran Khan, 2021. *Operating Room (Re)Scheduling with Bed Management via ASP*. *Published online by Cambridge University Press.*