Scheduled to Satisfaction: An Exploration of Personal Scheduling with CSPs

Evan Von Oehsen

April 29th, 202

1 Introduction

Given the importance, complexity, and limited amount of the time we have, it only makes sense that many of us look to digital tools in managing our schedules, daily tasks, reminders, and other time-oriented responsibilities. A given digital tool will often encompass one or two of these examples, each with its own specialized eatures and interface. As a result, an average user might use multiple time management tools in conjunction with one-another to meet their overall needs. This leads to a degree of manual work by the user to piece together how they're going to spend their time and stay in sync with their various responsibilities.

My own experience with this dynamic has lead me to pursue the development of a unified scheduling and task management tool for the senior exercise. Where a calendar app, reminders app, and to-do list app have been my go-to tools for time management, I plan to create a tool which, given the user's existing calendar and a set of tasks, imposes them onto the schedule and allocates time for these to be done proactively. Ultimately, the goal is to increase efficiency and free time, while reducing overall stress in the process. I plan to achieve this by framing it as a constraint satisfaction problem (CSP).

In preparation for this project, I've explored the space of CSPs with relation to time management. In this paper, I look to summarize the findings of my exploration into this richly defined space, in which I've found background information, insights on implementation details and algorithmic approaches, related applications (and their successes and failures), as well as a comparable previous attempt which I can learn from as I aim to create a thoughtful, functional result.

The survey begins with a brief introduction of CSPs in general (those familiar with CSPs may skip). I then discuss some general framings of CSPs in regards to planning and scheduling, followed by some common examples of scheduling problems tackled by CSPs. This is to set the scene for the inspiration of my senior project–the gaps in exploration of CSPs surrounding personal scheduling is evident. I delve into the most prominent exception to this, PTIME, in the following section, focusing on what was accomplished and how it relates to what I set out to achieve.

The rest of the paper focuses on the details relating to my own implementation. I begin discussing temporal constraints and how to set them up in our CSP. Next is a detailed introduction to Answer Set Programming (ASP) and the way in which it can be mapped to a CSP. This will serve to be the most concrete in its technical details and provide a clear launching point for next semester's implementation. Finally, I detail some important aspects of what an effecive user interface around the CSP should look like.

2 Background and Literature Review

2.1 CSP Applications in Scheduling

2.1.1 CSP Definition

A CSP [BSR10] is a triple containing sets $\langle X, D, C\rangle$:

- variables $X = \{x_1, ..., x_n\}$
- domains $D = \{D_1, ..., D_n\}$
- constraints $C = \{c_1, ..., c_k\}$

Each variable $x_i \in X$ corresponds with domain $D_i \in D$, where D_i is the set containing all possible values for instantiation of x_i. A given constraint c_j represents some condition involving one or more variables from X. In order for a constraint c_j to be satisfied, the instantiated values of the variables it encapsulates must meet its condition in a way such that it is also possible for all other constraints $\{c_1, c_{j-1}, c_{j+1}, ..., c_k\}$ to also be satisfied.

We solve our poblem by progressively instantiating variables one at a time. This entails assigning a given variable some value from its domain, such that the value satisfies all applicable constraints containing the variable in question, or does not prevent future satisfaction of any constraints. If we find a conflict which prevents one or more remaining variables from being instantiated in a way which satisfies all constraints, we must backtrack and assign a different value to one or more of the prior variable(s). We continue this process until all variables in X are instantiated, and all constraints in C are satisfied.

2.1.2 Applications

With their flexible, finite, and widely applicable nature, it's not surprising that a long history of CSP applications in scheduling has been cultivated by the artificial intelligence community.

Job-Shop Scheduling

The job-shop problem is a popular scheduling application of CSPs. It consists of a set of n jobs $J = \{j_1, ..., j_n\}$ and a set of m machines (which we can denote as R resources) $R = \{r_1, ..., r_m\}$. Each job contains one or more operations $[o_1, ..., o_k]$ which must be completed in a specific order. The goal in working through this problem is to not only find a solution in which we complete all jobs in J using the machines available in R, we also want to minimize the time elapsed from the start of the first job we take on to the end of the last job we complete, known as the *makespan*. In Figure 2.1, the makespan is 13.

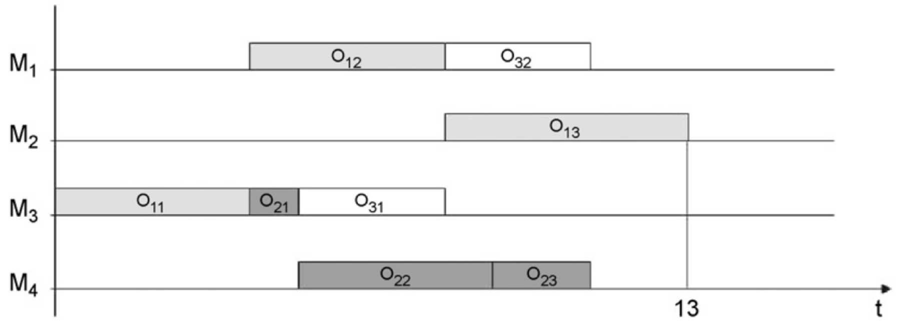
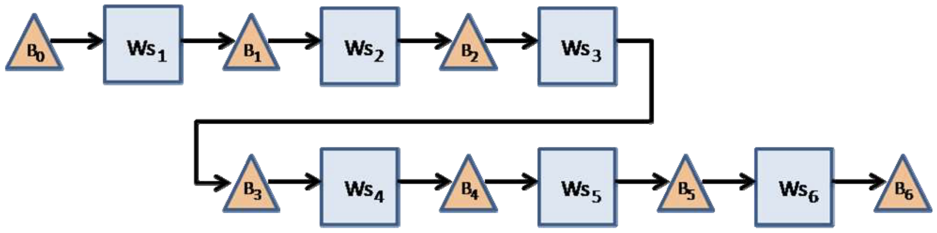


Figure 2.1: "Gantt Chart" depicting three jobs, each containing three operations, distributed over four machines. [PMC08]

Most often, this problem is formulated as a CSP by focusing on the start time of each operation of each job. This leads to a problem with $m \times n$ decision variables, whose domains are comprised of possible start times. However, this problem may also be framed as a disjunctive graph via the PCP approach. In doing so, the problem focuses on setting sequencing constraints which denote which operation is assigned which resource. A decision variable $Ordering_{ij}$ is defined for each pair of operations (O_i, O_j) in which $r_i = r_j$ (the two operations utilize the same resource). In this scenario, O_i may be sequenced either before O_j or after O_j. As decisions are made, they are "posted" to a temporal relations graph, and propagation ripples the consequences through the remaining CSP variables [CS97]. The PCP model utilizes the basic elements of a general temporal constraint network, including qualitative constraints and continuous domains, which is discussed in [DMP91]. Similar processes have also been explored with Multiprocessor Scheduling as a CSP [CGB09], though with the addition of it being real-time.

Flow-Shop Scheduling

Figure 2.2: "Conceptual schema of the six workstations", each padded by buffers between them. An individual worker may only occupy one workstation at a time. [NS09]



Much like in the job-shop scheduling problem, flow-shop scheduling looks to efficiently complete some set of jobs J. However, each of these jobs is identical. And, rather than thinking in terms of operations and machines, we think in terms of workstations along an assembly line, with operators needed for them to function, where there are less operators than workstations. The output of one worker is the input to another, and when all operations of a job are complete, the job is finished. As shown in Figure 2.1, there is a buffer at the beginning of the assembly line, at the end, and between each workstation, to hold work as it's not the focus of an active workstation. Once again, the goal is to minimize the makespan given the jobs, operations (workstations), and workers.

In regards to framing the CSP, Neubert [NS09] creates a dynamics problem, with variables to represent each element in terms of a number of time steps k. For example, a unique variable is assigned for each worker's state for each value of k, one for each piece worked on at a give workstation w

($p_w(k)$), the number of pieces present in the last buffer (denoting the number of jobs that have finished their operations). There also are variables denoting the number of pieces at every intermediary buffer for each value of k.

Finally, constraints give rise to the behavior of the flow-shop itself. We start by constraining the initial conditions, that is, the variables denoting the pieces contained by each buffer at time $k = 0$, which should all be set to empty: $b_s(0) = b_s^0$ for all s buffers. Now, we must denote how jobs advance between workstations. For the first buffer, this is $b_0(k) = b_0(k-1) - p_1(k)$. For the rest of the buffers, we'll also add constraints in the form $b_w(k) = b_w(k-1) - p_{w+1}(k) + p_w(k)$. For the last buffer, given it has no following buffer, we constrain it as: $b_{w_{num}}(k) = b_{w_{num}}(k-1) + p_{w_{num}}(k)$. Finally, we add our buffers' upper bounds to be $b_s(k) \leq b_s^{ub}$. Workers are also constrianed, in a way such that each worker corresponds to at most one station for each k, and that at each workstation, there is at most one worker.

With these two examples, we see a few common threads. Firstly, each has a static backlog of activities to be completed. While there may be dependencies among activities, there will neither be a shift in the number of needed variables nor the number of applied constraints once our problem solving begins. Hence, we are able to frame them as constraint satisfaction problems. Another commonality, however, is the presence of multple entities carrying out the scheduled work. Both job-shop and flow-shop utilize multiple machines or operators to handle applicable jobs.

With that in mind, one could argue these problems are more complex than personal scheduling–they require us to juggle tasks for multiple workers as opposed to just one, and these workers can have different capacities and skills as is the case with job-shop scheduling. However, one might suggest we frame personal scheduling as a single-machine job-shop problem. And, in fact, there are nuances to personal scheduling in practice which the above problems do not fully account for. There is a particular exploration in the literature which sheds light on and looks to address many of these: PTIME.

2.2 Case Study - PTIME

2.2.1 Background

The Personalized Time Manager (PTIME) system was created by the Stanford Research Institute as one of six major functions making up a larger automated assitant endeavor called CALO [BGU$^+$05]. Much like the project

I intend to build, the motivation for PTIME came from the noticeable lack of automated scheduling tools available, in conjunction with the separation of general to-do list and task management tools. The core goal of PTIME was to address this by providing an application where a user could, when scheduling an event or task, find an ideal time for it based on suggestions calculated leveraging temporal constraints. To do this, a dedicated Constraint Reasoner module was implemented, as shown in Figure 3.1. The tool also provided personalized reminder generation, and various degrees of advisability and instruction by the user. Another major objective of PTIME was to learn user preferences over time through passive means, eventually making more autonomous scheduling decisions as it progressively learns and gains the user's trust. The tool was primarily implemented in Java.

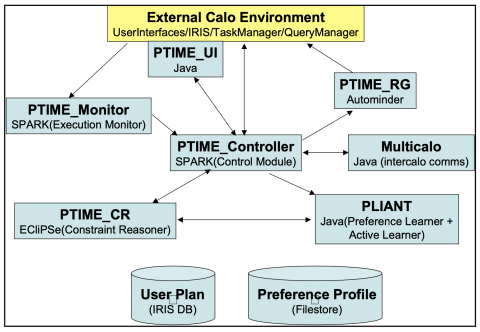


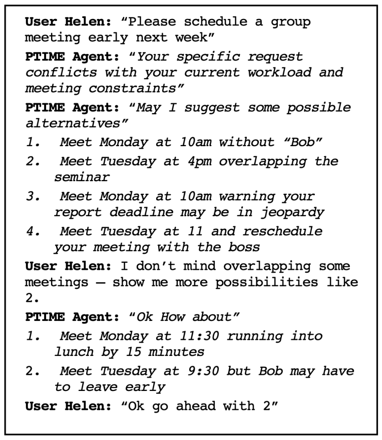
Figure 2.3: PTIME basic architecture within Calo. [BGU+05]

While user preference learning is not one of the goals of my project, the internal events scheduling algorithm built upon temporal constraints is quite similar to what I'm looking to build myself. With this in mind, there are a few things we can observe in PTIME and learn from as I not only look to solve the problems in question, but most clearly and eloquently define them.

2.2.2 What we can learn

As mentioned above, PTIME makes use of a Temporal Constraint Network (TCSP). In this type of CSP, each variable represents a time point, such as the start and end of each event, and each constraint defines ranges of durations and when the events in question may be scheduled. PTIME specifically ensures these constraints are temporally disjuctive–meaning calendar events may not be scheduled overtop to-dos, for example, allowing both to have their necessary uninterrupted time. Above just ensuring these events do not overlap, however, PTIME seeks to understand the "floating" nature of certain events and to-dos; for example, an event might require travel time, and a task relating to exercise might require a shower and change before returning to work.

In light of the complexities of this dynamic, as well as the still underlying need for efficiency, PTIME combines Simple Temporal Problem (STP) representations with Disjunctive Temporal Problems (DTP) representations to create a hybrid approach. STPs utilize simplified, binary constraints which only handle one single, specific time interval. While these allow for solving in polynomial time, they aren't dynamic enough to handle the floating events described above. This is where DTPs come in; their use of non-binary constraints allows for a wider array of time intervals to choose from in scheduling. However, checking the consistency of DTPs is NP-hard [TVP03], incentivizing the use of binary constraints when possible. We can also learn



User Helen: "Please schedule a group meeting early next week"

PTIME Agent: *"Your specific request conflicts with your current workload and meeting constraints"*

PTIME Agent: *"May I suggest some possible alternatives"*

1. Meet Monday at 10am without "Bob"
2. Meet Tuesday at 4pm overlapping the seminar
3. Meet Monday at 10am warning your report deadline may be in jeopardy
4. Meet Tuesday at 11 and reschedule your meeting with the boss

User Helen: I don't mind overlapping some meetings – show me more possibilities like 2.

PTIME Agent: *"Ok How about"*

1. Meet Monday at 11:30 running into lunch by 15 minutes
2. *Meet Tuesday at 9:30 but Bob may have to leave early*

User Helen: "Ok go ahead with 2"

Figure 2.4: An example of the natural language interactions between the PTIME and the user. [BGU+05]

from the ways in which PTIME's user interactions were handled. PTIME utilized natural language to communicate with the user during scheduling, conflict management, and preference learning. For example, the user might input "Please schedule a group meeting early next week." PTIME might then respond "Your specific request conflicts with your current workload and meeting constraints," and suggest several alternatives in plain english, stipulating which sacrifices must be made to accomodate each of the new, numbered meeting options, as shown in Figure 3.2. The user might then say "Ok go ahead with 2," and the meeting is scheduled. This level of transparency both provides the user with valuable information they might not otherwise have known and builds trust between the them and the tool.

2.3 Implementation Research

As we've seen above, CSPs have a rich history of applications to a variety of temporal problems. Let's begin looking at the techniques we will apply in solving our personal scheduling problem.

2.3.1 Temporal Constraints and CSP Variables

Temporal constraints are a crucial aspect of our CSP working properly. With temporal constraints, the variables which they constrain can represent either points, intervals, or durations in time. Their domains tend to be integers or rational numbers. With this in mind, intuition tells us that we can see each event in time, which we can refer to as a *proposition*, as corresponding with a time interval containing a start time and an end time. We can denote a proposition and its associated interval as P_i, and $I_i = [s_i, e_i]$ [DMP91].

With a variable representing each s_i and e_i, we can control how our events might relate to one another using *qualitative interval* constraints. The table below, translated directly from [SV98], defines how each of these relations applies to two propositions, $P_i = [s_i, e_i]0$ and $P_j = [s_j, e_j]$:

Relation	Representation	Inverse	Representation
P_i Before P_j	$e_i < s_j$	P_i After P_j	$e_j < s_i$
P_i Equal P_j	$s_i = s_j \wedge e_i = e_j$	P_i Equal P_j	$s_i = s_j \wedge e_i = e_j$
P_i Meets P_j	$e_i = s_j$	P_i Met_by P_j	$s_i = e_j$
P_i Overlaps P_j	$s_i < s_j \wedge e_i < e_j$ $\wedge s_j < e_i$	P_i Overlapped_by P_j	$s_i > s_j \wedge e_i > e_j$ $\wedge s_i < e_j$
P_i During P_j	$s_j = s_i \wedge e_i < e_j$	P_i Contains P_j	$s_i < s_j \wedge e_j < e_i$
P_i Starts P_j	$s_i = s_j \wedge e_i < e_j$	P_i Started_by P_j	$s_j = s_i \wedge e_j < e_i$
P_i Finishes P_j	$s_i > s_j \wedge e_i = e_j$	P_i Finished_by P_j	$s_j > s_i \wedge e_j = e_i$

In embodying our problem as a CSP, these constraint relations, and the form in which the propositons take, will be crucial to realizing a proper structure and behavior, as well as an encoding that is easily interpretable and translated to a front-end.

So, how might our variables and constraints look in practice? What will our domains be? With what we've established above, it only makes sense to structure each proposition as some interval in time. However, rather than the pre-defined ordering of Schwalb's use cases with propositions, determining the order of the propositions is our ultimate task. Thus, we need more to go on, and will utilize a third variable to fix the value of each proposition's duration, allowing our upper and lower interval-defining variables to solely drive the location of the proposition to be scheduled. In addition to this, it's important that we incorporate the "floating" propositions acknowledged by PTIME. in order to generate a schedule which is actually realistic. We can include a fourth and fifth variable to simply hold the padding needed at the beginning and end of the proposition. By default, these will usually be zero.

So, we can represent each proposition as being a quintuple:

$$E = \langle s, e, d, p_s, p_e \rangle$$

Here, s is denotes the start of the proposition, e denotes the end of the proposition, d denotes the duration of the proposition, p_s is the time padding prior to the proposition start, and p_e is the time padding following the end of the proposition. While it might feel bulky to have five variables per proposition, this is a necessity for CSPs and will allow us to properly set up our constraints. We may optionally have a sixth variable, e_{max}, which specifies when the proposition is due.

So, what might a domain look like for our CSP? In the context of our problem, the domains for each s and e will represent some range of time, while the domains for d, p_s, and p_e will be discreet, static values representing time durations. As for the former variables, we might be inclined to follow the classic TCSP definition and utilize continuous domains. However, a finite-domain approach might better suit our needs, as it will ensure all propositions are scheduled on clean, five-minute intervals. We can manifest this by interpreting each increment of 1 in our domain as a 5-minute interval. Assuming we desire to schedule propositions from, say, 8:00 a.m. to 8:00 p.m., we can represent our domain as ranging from 0 - 144. If we desire to schedule a week at a time, we simply set our domain to be 0 - 1008, and constrain each proposition's start and end times to fall within the same multiple of 144 (more on that later). Thus, fd will be our domain-type of choice.

Thanks to the table above, we already have a strong understanding of how to constrain our propositions' start and end boundaries (excluding our padding). However, there is more we must do to ensure each variable in our quintuple is handled properly. We'll discuss constraints in the context of an proposition $E_i = \langle s_i, e_i, d_i, p_{si}, p_{ei} \rangle$ where there are $n-1$ other propositions in our CSP. The following constraints will set our up proposition:

1. $e_i - s_i = d$
   Ensures our start and end bounds match the length of the proposition. Can loosen this to $\geq$ if we desire our d to represent a minimum duration instead.

2. $e_i + p_{ei} \leq s_k - p_{sk} \vee e_k + p_{ek} \leq s_i - p_{si}$, $\forall k \in \{1, ..., i-1, i+1, ..., n\}$
   Essentially using *Before* from the table above, but including padding, this will constrain our proposition to occur before or after all other propositions, prpropositioning any unwanted overlap. We can, of course, utilize any *qualitative interval* constraint for specific cases as well, if we desire an overlap, finish, etc.

3. $s_i \div 144 = e_i \div 144$ *(optional)*
   Constrains the start and end times of the proposition to be on the same day based on relevant 5-minute intervals, assuming an. This constraint is only needed for problems in which the time domain is divided into multiple even subsections, such as 12-hour days in this case.

4. $e_i \leq e_{max}$ *(optional)*
   Constrains the proposition to conclude at or before the time in which it is due. This is not needed for propositions without e_max values.

We now have our variable, domains, and constraints structures defined. Come time to generate our CSP in practice, we can have our program wrapping our ASP program execute follow below steps:

1. For each pre-existing calendar event from the user's personal calendar, create each corresponding variable for the quintuple $\langle s, e, d, p_s, p_e \rangle$. Create variables to hold $s - p_s$ and $e + p_e$ for use in constraint 2 down the line.

   - We consider pre-existing calendar events to be conflicts we must schedule around. Therefore, we add them to our CSP first, setting each variable's domain to be a singleton with its existing value, so that this value does not change.

2. For each event from our front-end to be scheduled, create each corresponding variable in the quintuple $\langle s, e, d, p_s, p_e \rangle$. Create variables to hold $s - p_s$ and $e + p_e$ for use in constraint 2 down the line.

3. For each variable in our CSP, apply constraints 1, 2, and 3 above.

4. Solve via ASP solver.

2.3.2 Answer Set Programming: Basics

I've elected to use Answer Set Programming (ASP) to implement and solve the personal scheduling CSP. ASP is particularly useful as it is a declarative language. Due to this, I will be able to specify the variables, domains, and constraints of the CSP as I normally would. However, unlike more traditional procedural languages, I will not have to implement any kind of search algorithm to solve it, as the language itself requires the user to define *what* to solve, rather than *how* to solve it. Once all rules have been defined, they are fed into a solver. Should the problem be satisfiable, the solver provides an output with stable model (answer set) semantics, specifying all atoms which make up the stable solution. I plan to use Potasco's clingo grounder + solver for this, and thus will utilize their syntax in the explanations below.

Building Blocks

In ASP, our problem takes the form of Prolog-style rules. Let's dive into the basics.

The most foundational building block of ASP is called a *rule*. A rule is a statement which has one of the following forms:

```
a :- b.
a :- not b.
```

Here, `a` is the *head* of the rule, `:-` is a *turnstile* (pronounced "if"), `b` or `not b` is the body of the rule, `a` and `b` are all literals or *atoms*, and `not` is the *default negation*. Semantically, the first rule specifies that, if `b` is included in our stable model, `a` will also be included. In the second rule, `a` will be included only if `b` is *not*. We also may use the symbols `,` and `;` to represent *and* and *or*, respectively:

```
a :- b, c; d.
```

The body of the rule above is equivalent to $(b \wedge c \vee d)$. The second rule specifies that, if we have no indication that `b` is included in our stable model, `a` is included in it. We may also have *choice rules* surrounded by two braces:

```
{a, b} :- c.
```

In this example, if `c` is included in our stable model, then either `a`, `b`, neither `a` nor `b`, or both `a` and `b` will be arbitrarily included. In other words, our answer set could be one of {`c, a c, b c, a b c`}. Choice rules can also be given heads or tails, denoted by a number to the left or right of the braces, respectively:

```
1{a, b, c}2 :- d.
```

The above will generate an answer set containing at least one and at most two of the atoms inside the choice rule. In this case, all possible answers would be {`a d, b d, c d, a b d, a c d, b c d`}. Thirdly, we may have what's called a *constraint*:

```
:- a, not b.
```

In this constraint, the `a` will not be generated if `b` is not in our stable model. This is helpful the amount of stable models generated. More information on these ASP basics can be found in [Lif19].

*

Variables, Ranges, Operators We've now defined the basic elements of ASP. However, it really becomes more powerful and applicable when more advanced elmenents come into play.

What enables us to make our ASP pograms more powerful? Everything stems from the basic *statement*. A statement is handled the same way as any atom literal, but has a broader meaning to it. Generally, from a semantics standpoint, a statement might be an adjective or category, while an atom might be a specific noun. We wrap an atom in a statement, and in doing so we suggest this statement is true for said atom. Each statement is denoted by a lowercase name with parentheses at the end, and the atom inside. For example, we can say:

```
brown(fox).
brown(bear).
```

Semantically, we've declared that foxes and bears are brown. However, we can't really do much with them yet. This is where variables come in. Variables[1], denoted by a capital first character, allow us to quickly declare multiple statements based on other knowledg we already have. This greatly

[1]Traditionally, ASP solvers do not allow for variables, and instead a grounding tool must first be used before the program is fed into the solver. However, clingo is an exception to this; it combines the gringo's grounding functionality with clasp's ASP solver.

reduces the amount of code needed for setting up larger problems like CSPs. We can also constrain our statements such that an atom must meet some condition in order to be grounded in them. We might have something that looks like this:

```
animal(bear).  animal(tortoise).  animal(beetle).
brown(bear).
green(tortoise).
blue(beetle).

colorful(X) :- animal(X), not brown(X).
```

The above logic defines `X` as our variable, and for each animal, states it as `colorful` as long as the animal is not stated to be `brown`. In solving this, we're given an answer set $\supset$ {`colorful(beetle), colorful(tortoise)`}. Thus, variables are extremely useful as they allow us to replicate multiple statements across many atoms, and greatly reduce our lines of code. Beyond just generation though, they allow us to handle comparisons and set multiple values dynamically, even numerical values.

Numerical values are handled similarly to traditional procedural programming. They can be manipulated and compared, and can be the main focus of our output. However, numbers are always wrapped in some larger statement and thus have their meaning associated with them in that way. We can generate ranges of numerical values quickly using the `..` operator. For example, `item(1..99).` expands to `item(1). item(2). item(3). ... item(99)`. This might be particularly helpful for generating finite, discreet constraint domains. When we're handling numerical values, it's important that we can compare and modify them to ensure we get the most out of them. With clingo, we're provided many of the standard arithmetic and logical comparison operators:

Logic Symbol	Name
`=`	`eq`
`!=`	`neq`
`<`	`lt`
`<=`	`leq`
`>`	`gt`
`>=`	`geq`

Arithmetic Symbol	Name
`+`	`plus`
`-`	`minus`
`*`	`times`
`/`	`divide`
$\backslash$	`modulo`
`|x|`	`abs(x)`
`**`	`power`
`&`	`bitand`
`?`	`bitor`
ˆ	`bitxor`
$\sim$	`bitneg`

More information on clingo can be found in [GKK$^+$08].

2.3.3 Implementing A CSP in ASP

Now that we've laid out the basics of ASP, we can begin to utilize it to create and solve our CSP. See subsection 2 for more information on CSPs.

Given ASP's constraint-centered implementation, translating variables and constraints from CSPs is incredibly intuitive and on its own would be practically indistinguishable in its approach from vanilla ASP programming. However, we also have to handle specific domains for our variables to be instantiated within. This requires quite a bit more thoughtfulness. For some of our basic setup, we'll pull from the structure defined in [Bal09]. To start, we'll plan to implement our CSP domains as finite-domain (fd) domain, meaning each domain contains discreet values within a set range.

We can start by encoding each variable as a `cspvar()` statement containing the triple X (ground term for the variable), L (lower bound on the variable's domain), and U (upper bound on the variable's domain):

```
cspvar(X,L,U)
```

such that the domain of variable X is $[L, U]$. Take the statement:

```
cspvar(v(1), 2, 6)
```

This denotes that we have some variable `v(1)`, which has a domain of {2,3,4,5,6}.

While the crucial parameters of a vaiable's domain are defined in our `cspvar` statement, that alone will not be enough, since we currently don't have any representation of the internal values within the range from `L` to

`U`. It's important to remember that in ASP answer sets, our solution is communicated as a set of the individual atoms whic satisfy the problem, and therefore our problem must be granuular to the atom. If we define domains with just an upper and lower bound, we must provide some mechanism for our ASP grounder to expand it into each possible value within that range, so that our solver can then select and include the relevant value as its own atom. This is where the below statement comes in:

```
1{fix(X, L..U)}1 :- cspvar(X,L,U).
```

There's a fair amount going on here, but we can break it down into components covered in the ASP basics to see why this line is so crucial to implementing our CSP. As we can see, our `cspvar(X,L,U)` statement is now defined as the body of a rule, and will only be defined here.[2] Looking at the head of the rule from the outside in, we first notice a choice which must be between 1 and 1. Inside is a new statement called `fix(X, L..U)`. Each `fix` will contain a CSP vaiable name and a number. Notice that we use the same ASP variable names inside `fix` as `cspvar`. This ensures that, for {X,L,U} in each `cspvar`, there must exist a `fix` with the same {X,L,U}. We might notice, however, that `fix` utilizes a number range. This will fill our choice rule with {`fix(X,L),fix(X,L+1),..,fix(X,U-1),fix(X,U)`}. Since this set is wrapped by a choice, we will only have one `fix` per `cspvar` in our answer set. Thus, if we put it all together, this statement means, *For a given CSP variable* `cspvar(X,L,U)`, *the answer set will contain at least one and at most one* `fix(X,J)`, *wheres* `J` *is some value within* `L` *to* `U`, *inclusive.*

Now that we've covered all the variable and domain setup, let's look at the final element of the CSP triple: constraints. With the operators shown in the above tables, constraints are simple to set up with ASP. A basic structure we can use for constraints is something like this:

```
L [op] R :- [op-name](Lv, Rv), fix(Lv, L), fix(Rv, R).
```

This rule states that, if we declare `[op-name](v(1),v(2))`, and there is some `fix(v(1), L)`, some `fix(v(2), R)`, the L and R in our answer set must satisfy this logical operator. If we replace `[op]` with any of our standard logical operators (e.g. $>$), and `[op-name]` for a name which accurately describes the operation (e.g. `lt`), we're left with a constraint which is clean and simple to use:

```
L > R :- grt(Lv, Rv), fix(Lv, L), fix(Rv, R).
```

[2]Note that when defining what our actual variables are, we will still instantiate each literal `cspvar` value as we did above for $v(1)$.

Then, to apply this constraint to any two variables `v(i)` and `v(j)`, we simply declare `grt(v(i),v(j))`.

Now that we've covered variables, domains, and constraints, let's look at an example of a full CSP in ASP:

```
% instance
cspvar(v(1), 1, 5).
cspvar(v(2), 4, 8).

lt(v(2),v(1)).

% encoding
1{fix(X, L..U)}1 :- cspvar(X, L, U).
L < R :- lt(Lv, Rv), fix(Lv, L), fix(Rv, R).
```

In the encoding subsection, we create our classic `cspvar` rule, as well as a rule for a less-than constraint (`lt`). In the instance subsection have two variables: `v(1)`, which has domain {1..5}, and `v(2)`, which has domain {4..8}. We then apply the *lt* constraint to our two variables. Given their domains, we know inttuitively that our answer set should contain {`fix(v(1), 5)`, `fix(v2, 4)`}. Sure enough, when running our program in clingo, the answer set is:

```
cspvar(v(1),1,5) cspvar(v(2),4,8) lt(v(2),v(1)) fix(v(2),4)
                         fix(v(1),5)
                         SATISFIABLE
```

2.3.4 Implementing a Personal Scheduling CSP in ASP

At long last, we can implement a basic scheduler in ASP. Let's create a simple scenario with two new events which must be scheduled around one existing conflict to avoid. To begin, we start with our encoding. Note that in the file itelf, encoding will be below the instance, however as we step through in implementation order it will be above. In our encoding, we have our usual rule to generrate one textttfix per `cspvar`. We then define a *lte* rule, a *minus* rule, and a *plus* rule:

```
% encoding
1{fix(X, L..U)}1 :- cspvar(X, L, U).
L <= R :- lte(Lv, Rv), fix(Lv, L), fix(Rv, R).
D = R - L :- minus(Lv, Rv, Dv), fix(Lv, L), fix(Rv, R), fix(Dv,D).
D = R + L :- plus(Lv, Rv, Dv), fix(Lv, L), fix(Rv, R), fix(Dv,D).
```

Now that our CSP encoding is set up, we can utilize its structure to define our problem. To make it more interesting, let's begin by defining our our time domain to be a very tight window which can still fit our events:

```
% instance
csptimedomain(1,35). % 8-hour day
```

Next, we instantiate the first event. It will have a length of 15 minutes, and a padding of 15 minutes following the end. We designate each variable name as `t(x)` to allow us to more easily add constrains to all variables of type `t` and all CSP variables of identifier `x`. Notice we also create variables for `swp` to signify the event's *start with padding*, and `ewp` to signify the event's *end with padding*. We'll late constrain these to equal `s - ps` and `e + pe` respectively.

```
cspvar(s(1), L, R) :- csptimedomain(L,R).
cspvar(e(1), L, R) :- csptimedomain(L,R).
cspvar(d(1), 3, 3). % 15-minute event
cspvar(ps(1), 0, 0).
cspvar(pe(1), 3, 3).
% vars for padded start and end values
cspvar(swp(1),L,R) :- csptimedomain(L,R).
cspvar(ewp(1),L,R) :- csptimedomain(L,R).
```

The second event has a length of 1 hour, and a padding of 10 minutes prior to the start:

```
cspvar(s(2), L, R) :- csptimedomain(L,R).
cspvar(e(2), L, R) :- csptimedomain(L,R).
cspvar(d(2), 12, 12). % hour-long event
cspvar(ps(2), 2, 2).
cspvar(pe(2), 0, 0).
% vars for padded start and end values
cspvar(swp(2),L,R) :- csptimedomain(L,R).
cspvar(ewp(2),L,R) :- csptimedomain(L,R).
```

The third event (our supposed pre-existing "calendar event") has a singleton domain for both its start and end times, and a fixed length of 1 hours. It has no padding:

```
cspvar(s(3), 16, 16).
cspvar(e(3), 27, 27).
cspvar(d(3), 12, 12). % hour-long event
cspvar(ps(3), 0, 0).
cspvar(pe(3), 0, 0).
% vars for padded start and end values
cspvar(swp(3),L,R) :- csptimedomain(L,R).
cspvar(ewp(3),L,R) :- csptimedomain(L,R).
```

With our variables all set up, we can define our constraints. The first constraint, for each variable `X`, imposes Constraint 1 from our constraint list in **subsection 4.1**. The next two constraints prepare the values of `swp` and `ewp`, so that the final constraint can utilize them for Constraint 2 from the aforementioned constraint list[3]:

```
% adding constraints
minus(s(X),e(X),d(X)) :- cspvar(s(X),_,_).

minus(ps(X),s(X),swp(X)) :- cspvar(s(X),_,_).
plus(e(X),pe(X),ewp(X)) :- cspvar(e(X),_,_).

lte(ewp(X1),swp(X2)) | lte(ewp(X2),swp(X1)) :-
     cspvar(s(X1),_,_),cspvar(s(X2),_,_),X2 != X1.
```

Encoding, variables, and constraints set up, all that's left to do is feed it into our grounder and solver. Running clingo on this, we get our final answer (added notes are *italicized*, rearranged for easier reading):

```
csptimedomain(0,35)
fix(s(1),29) fix(e(1),32) (Event 1 start and end time)
fix(s(2),2) fix(e(2),14) (Event 2 start and end time)
fix(s(3),16) fix(e(3),28) (Event 3 start and end time)

Definitions of cspvar, constraints, remaining variables, and everything
else:
fix(d(1),3) fix(pe(1),3) fix(ewp(1),35)
fix(d(2),12) fix(pe(2),2) fix(ewp(2),16)
fix(d(3),12) fix(ps(3),0) fix(pe(3),0) cspvar(d(1),3,3) cspvar(ps(1),0,0)
cspvar(pe(1),3,3) cspvar(d(2),12,12) cspvar(ps(2),2,2) cspvar(pe(2),2,2)
cspvar(s(3),16,16) cspvar(e(3),28,28) cspvar(d(3),12,12) cspvar(ps(3),0,0)
```

[3]In the current problem, we're dealing with a time range smaller than a day; therefore, we do not need Constraint 3 from the constraint list.

```
cspvar(pe(3),0,0) cspvar(s(1),0,35) cspvar(s(2),0,35) cspvar(e(1),0,35)
cspvar(swp(1),0,35) cspvar(ewp(1),0,35) cspvar(e(2),0,35) cspvar(swp(2),0,35)
cspvar(ewp(2),0,35) cspvar(swp(3),0,35) cspvar(ewp(3),0,35) minus(s(3),e(3),d(3))
minus(s(1),e(1),d(1)) minus(s(2),e(2),d(2)) minus(ps(3),s(3),swp(3))
minus(ps(1),s(1),swp(1)) minus(ps(2),s(2),swp(2)) plus(e(3),pe(3),ewp(3))
plus(e(1),pe(1),ewp(1)) plus(e(2),pe(2),ewp(2)) fix(ps(1),0) fix(swp(1),29)
fix(swp(2),0) fix(ps(2),2) lte(ewp(3),swp(1)) lte(ewp(2),swp(3))
lte(ewp(2),swp(1)) fix(swp(3),16) fix(ewp(3),28)

Result is deemed:

SATISFIABLE
```

2.4 On Scheduling Interfaces

A well-functioning algorithm does not provide much utility to the average user without a clean, intuitive interface in front. In this subsection, we'll discuss some guiding principles for making an intelligent scheduler most interactable for the average person.

Feature 1: Visualizations

In our CSP above, each proposition is made up of a handful of related but ultimately separate variables when instantiated in the answer set output. Even the output shown in **subsection 4.4**, with its manual organization of each variable, is not abundantly clear or effective at communicating a daily schedule to the reader. We can improve this by providing visualizations to the user. Before we make use of them, however, we must decide how to structure them.

There are several types of time and calendar visualizations we can implement; the most common two options are the grid view and the list view.

*

The Grid View

The grid view is often regarded as the most classic calendar representation. Grid views have been a popular choice for most paper calendars as well as calendar and scheduling software systems, making them a familiar option for a calendar interface. They provide the advantage of having a large range of visible dates in one area, and compartmentalize each week into an invididual row which allows for the user to better visualize their weekly load, where they have the most vacancies, and any patterns of recurring events.

Despite these advantages, grid views are characterized by a general lack of precision. Generally speaking, boxes within a grid are too small to fit any rich content due to limitations in the sizes of our screens, and there being multiple fit on each axis. So, while grid views may be more recognizable, this medium would likely not support the functionality needed by a scheduling tool of this nature.

*

The List View

The list view is in fact the oldest known format for calendars, dating back to papyrus rolls in ancient Egypt [HDM14]. While, contemporarily speaking, list views are perhaps less popular than grid views as primary schedule interfaces, they still have a strong presense in scheduling applications, especially available as a secondary option to the grid view. List views provide the advantage of a linear visualization of time, which more closely aligns with how we view it ourselves. They also allow more efficient visual search by the user; it is easier to predict an event's physical location when dealing with just one vertical axis. List views present many other advantages: consistent "today" at the top, flexible and expandable layouts, theoretically infinite range of values, more spatious cells, and so on.

However, list views also lack certain functionalities; while they are great for detailed viewing and manipulation of a schedule, they often fail to effectively show the bigger picture. They require more scrolling work to navigate, and are less effective for dragging and dropping objects over longer time distances. That said, most scheduling done in the tool at hand will generally encompass smaller intervals (between a day and a week) as the focus is on immediate to-do scheduling. Thus, a list view will serve as the best tool for the job. See Figure 5.1 for an example of an effecive list view.

*

The Role of Visualizations

So, what will we use these list-style visualizations for? These will serve to give the user a visual sense of how each proposition relates to one another. Specifically, these will be used in the "before" screen (where the user's current calendar is shown without any new tasks scheduled in), the preview screen (where raw, newly generated solutions are shown), the user adjustments screen (where the user may modify the generated schedule to their preferences), and the completed schedule screen.

Feature 2: Efficient To-Do Setup

Before a schedule can be generated, the user must have to-dos, events, or other propositions needing to be scheduled. To handle these, a simple interface should be provided for the user to input each one. Fields should be provided for the name, frequency, desired or estimated duration, and due date and time. Since this data entry might be a mundane task (especially for new users), the form should be easy to access and quick to complete, with thoughtful (and perhaps configurable) default values.

Feature 3: Pre-Schedule Generation Options

While our ASP solver can be used following the completion of data entry, the user might also want a consistent, default structuring of their day to be incorporated. Options for the start and end of the range of workable hours, amount and duration of breaks throughout the day, and a lunch break range should be configurable. Each of those may be handled as standard, static events in our CSP. Other preferences, such as the grouping of short to-do tasks into one 30-minute productivity sprint, should also be considered.

Feature 4: Previews and User Adjustmenets

As alluded to in the previous subsection, the ability for the user to preview and adjust the generated schedules within the interface will be very important to a positive user experience (UX).

Previews

Following the generation of schedule answer sets, it is critical that the user can see the solution that was generated before confirming. Much like the scheduling options which RhaiCAL proposes in its own interface (see Figure 5.1), allowing two initial list-view options for the user select between empowers them to choose what feels most ideal to them, also providing them with a perhaps necessary hint of agency in determining what they'll be doing during the day.

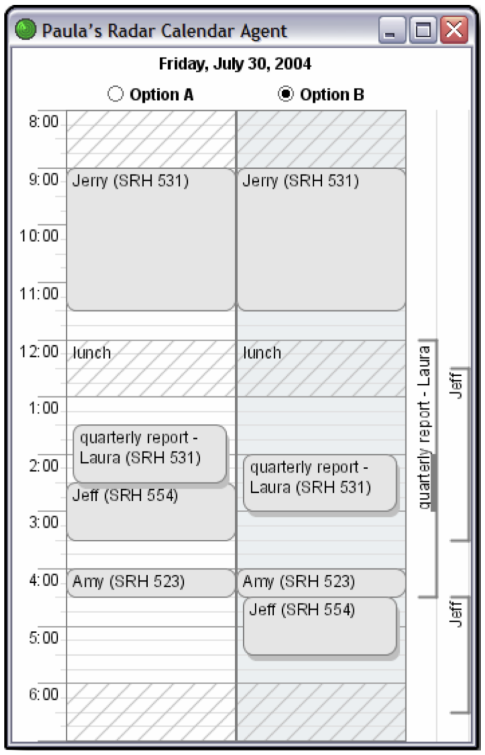


Figure 2.5: "RhaiCAL displays two different options for [user] Paula side-by-side within the context of her calendar". [FM05] This is an example of a preview with two list-view options for the user to choose between.

User Corrections

Once the user has selected between the two starting off points, it's important to allow them the chance to make any specific edits they desire. Perhaps they need a mental health day, or perhaps they're feeling extra productive. Maybe the user prefers exercising in the morning and doing chores in the evening when their favorite television program is on. Corrections allow them to make these necessary personal adjustments.

For these adjustments to be made, another listview visualization will be provided with their selectd starting off point. However, each proposition will be represented by a drag-able, resize-able box to be moved, edited, or deleted as the user pleases. Following the completion of these, the user may confirm and lock their new schedule in.

3 Goals

In this chapter, I will discuss the specific goals I had for the project itself.

Events and Todos

One of the goals I've cared about most when brainstorming for this project is the ability to handle to-dos and events within the same tool. Incorporating both in a meaningful way is important. Most important is the ability to schedule to-dos amidst existing calendar events. These two tasks oftentimes can only be done in separate tools, hence the motivation for this project in the first place.

Calendar Integration

Calendar integration builds upon event and to-do integration. Even if the resulting project doesn't schedule new events but at least pulls in existing ones, I would consider that to accomplish this goal. In addition, local calendar export (e.g. Apple Calendar) would be considered a success just as much as remote calendar integration via api calls (e.g. Microsoft Graph with Outlook).

Flexibility

Managing one's time is a personal, inexact, and volatile process. A scheduling tool which is too rigid has limited usability in the real world. I intend to include intermediate steps for modification throughout the scheduling process in the final result of the project. For example, there may be an interface to modify the generated schedule before locking it in, allowing for the user to make adjustments for things which a simple constraint satisfaction algorithm wouldn't account for. For example, the user might prefer exercising in the morning before temperatures are too high. Moving an exercise todo earlier in the day should be possible with the result.

Frame the problem as a CSP

I would like to manifest the problem as a CSP, and write and solve it in ASP. The main motivation for this is the fact that, unlike machine learning algorithms which are by nature black boxes, the constraints can be governed directly, and thus the results. This comes at a cost of rigidity and lack of personalization, but mitigates the need for large training datasets and mitigates the risk of just failing to train an effective algorithm. I also considered other artificial intelligence methods such as evolutionary algorithms, however these don't always bring you to an 'exactly correct' solution. ASP also enables optimizations, which I hope to include.

Robust API

In keeping with my interest in more complex software development, I would like to build an API in Node.js to use to interface with the ASP solution. This API should abstract out the more ambiguous aspects of the ASP problem, such as time values for todos. Ideally, this API is secured for the user and requires an authentication token, and in turn stores the user's todos in a database. The API should also utilize objects with schemas which contain all necessary fields for each todo.

Intuitive user interface

Lastly, I hope to build clean user interface to communicate with the API, which is easy to understand and interact with. The goal is to incorporate the list view style from my research above, and I'll likely draw from past scheduling-related interfaces for other aspects besides preview (e.g. todo creation screen layout). I plan to build this for iOS with Xcode and Swift.

4 Results/Conclusion

4.1 Progress Towards Goals

Events and Todos

The project is able to handle todos and technically events, however only with regards to data models and the front-end interface. At present, there is no third-party integration (this required more time than I was willing to pour in given limits of student projects), so "dummy" calendar event data must be added to the database to allow for the other features surrounding events to be shown. With that said,

Calendar Integration

As mentioned above, there is no third-party calendar integration. However, the final schedule can be exported to Apple Calendar on the device in which the front-end app is running.

Flexibility

As suggested in the goals section, the generated schedule can be edited by the user before it is confirmed and exported, allowing them to modify the ordering or included todos to their own preference. However, there is still a lack of convenience for on-the-fly scheduling. A further shortcoming is the lack of ability for the user to add a todo which is not accounted for in the schedule back in. In other words, if a todo has not been included in or has been deleted from a generated schedule, there is no way to insert it without reverse-engineering it with API calls.

Frame the problem as a CSP

The final implementation is in fact a constraint satisfcation problem written in answer set programming.

Robust API

The final API, written in Node.js with Express and MongoDB, abstracts out all processes related to scheduling and CRUD operations on todos. It also requires sensitive requests to contain an authorization token which can be obtained by a successful request to the /api/auth/login endpoint. All passwords are hashed with bcrypt, a secure and slow hashing library, and stored in the user's schema in MongoDB. Todos are also stored and have their own object schema which contains all necessary fields for the algorithm, plus a few extras for featuers like repetitions, should they be expanded to in the future. And, of course, there is a scheduling endpoint from which the user can obtain a candidate schedule.

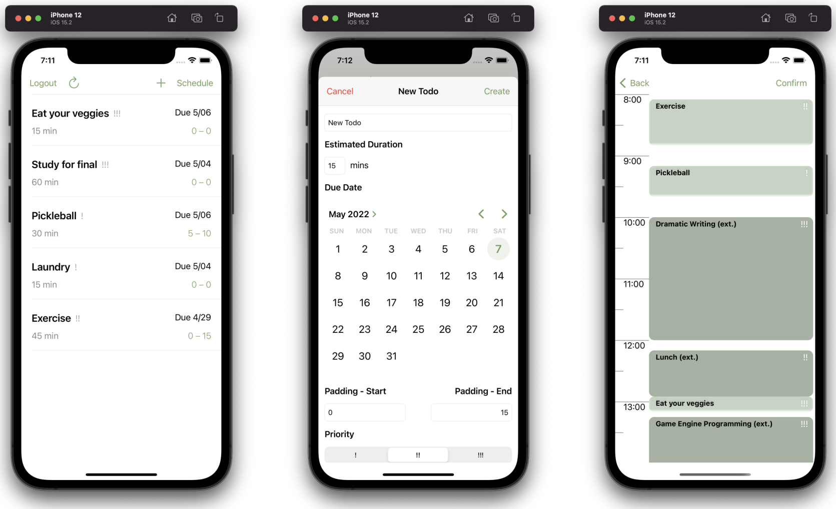


Figure 4.1: Final interface screenshots from the Home, Todo creation, and Schedule Modification screens.

Intuitive user interface

The final interface is simple, yet draws upon many modern conventions for scheduling interfaces. For example, all date-related fields either expand to calendar views or are date spinner views, which prevent any invalid data entry. All numerical fields (e.g. todo duration) are tied to the number pad keyboard in iOS (though this wouldn't be visible in the demonstration, due to iPhone simulators defaulting to hiding the keyboard). There is a simple screen to create new todos prior to scheduling. A screen prior to calling the scheduling algorithm allows for time range preferences to be set. In the schedule view/edit screen, each scheduled todo can be rearranged using familiar gestures. To grab a todo requires a long press, and rearranging it just requires dragging it up and down. Releasing the todo either leaves it in its new spot, or returns it to the start if it conflicts with another scheduled event. Double-tapping deltes the todo.

That said, there are some gaps which might leave a user frustrated. For one, tapping a scheduled todo (as well as a todo on the home screen) does not open any kind of expanded preview of the todo. As well as this, double-tap to delete might be surprising without any warning of this behavior, especially considering the lack of ability to add any todo back in. Loading spinners are also present for feedback, however they can be small and might go unnoticed by some users.

4.2 Conclusion

Reaching the final product was certainly an interesting task. Going in, I felt like the tasks at hand stuck a balance between having some aspects in which I had prior experinece and others which would be new explorations for me. I found that this was in fact the case, however there were some notable reasons why that worked well for me and why that didn't.

Lessons Learned

Time Management

As is well known by all software engineers, whatever the maximum is that we *think* we can accomplish in a span of time, whether or not conditions are perfect, rarely ends up being truly doable in the given timeframe. In the process of setting my goals for the semester, I tried to be conservative yet leave room for bells and whistles if time allowed. However, I of course could not have predicted how the rest of my commitments would go, and as a result ended up prioritizing other, more immediate deadlines. My project by no means fell by the wayside, but I only was able to dedicate two or three hours per week on it.

As a result, I found progress was slower than originally anticipated. And, in the end, I spent the week prior to the presentation building the a large part of the front-end, fixing inevitable bugs, and ultimately having to "trim the fat" and remove some of the important but not critical third-party calendar integration feature. That said, I'm still proud that I was able to accomplish what I did given the inflexible timeframe of a semester.

Avoiding Stubbornness

Some of my challenges with time management were also amplified by some stubbornness I had with specific aspects of the project. For example, I was passionate about creating a clean user interface, especially in the schedule editing screen. I spent some extra time tweaking positions of lines and time labels and colors that I could have spent on more functionalities or integrations. Of course, a clean interface is important too, but it is considerably less meaningful to the overall impact the tool can make, and leans more towards gold plating than usefulness.

I was also a bit stubborn with the data schemas that I first drew up, only adding fields rather than removing or optimizing. The limitations of this forced me to find workarounds which added some unnecessary complexity to the implementation.

Utilize Available Support Networks

This is something I've historically struggled with doing, and definitely shone through in this project. Professor Osborn as an advisor was a wonderful resource available to me, however I did not make use of his time and expertise as much as I could have. My initial implementation of the constraint satisfaction problem in ASP, for example, was a first stab at solving any scheduling problem algorithmically, my first stab at setting up a CSP on my own, and my first time working with ASP. The odds that inefficiencies due to my inexperience existed are extremely high, and a more trained eye might be able to pick those out and advise more efficient routes. This really showed when it came to the speed of the Clingo solver when solving generated scheduling problems over scheduling time ranges longer than a day or so; I'm sure optimizations were possible.

4.3 Successes

Incorporating New Things

On a more positive note, I learned (or perhaps, was reminded) that it's always a joy to learn something new and apply it to a project in some way. In this instance, I learned about answer set programming and applied it to an app in platforms I'd already had experience with, which allowed me to see new angles on familiar development stacks.

Building a Useable Product

While it fell short of my goals in certain areas, I truly believe that this app is a useable minimum viable product. Responses from friends I've shown it to have been positive and in some instances I was asked when it might be available for download. In that sense, I'm glad that I focused enough time on the interface to at least support the basic scheduling algorithm. As a result, I have a great base scheduling app which I could add as many bells and whistles to as I'd like, which alone makes it more valuable to me.

4.4 Future Work

Third-Party Integrations

Perhaps the highest priority addition I would like to make is the ability to pull events from, and schedule into, the user's Outlook calendar. An app which is able to integrate with the tools that are core to users' existing workflows, in this case with regards to scheduling, is much more useful than one that lives in a vacuum.

Subdividing Todos

Specifically, this involves breaking up a time-consuming task (e.g. writing an essay with an estimated duration of 180 minutes). Having the ability to break it up into multiple shorter parts, spread throughout the week or scheduling period, would be very useful. Additionally, being dynamic enough to still schedule just one subsection of the broken up event in the event of a single-day scheduling range would be necessary.

Recurring Tasks

Support for recurring tasks would be relatively straightforward to implement and save the user a lot of time in the long-run. The schedulable data models are already set up to hold recurrence-related fields; business logic and support in the front-end are all that's needed to bring recurrence in.

Grouping Small Tasks

There are often chores which may only take 5-10 minutes that we put off (e.g. calling the dentist to set up an appointment, making the bed, drinking water). These could be grouped into a 15-minute ”task block” which would help the user accomplish more as well as reduce the number of 5-minute blocks in the scheduling interface, which are thin due to their shortness and difficult to present effectively.

Optimizations

Optimizations are possible on all fronts. As mentioned above, recurring events would prevent the need for repeated data entry. Some data models have unnecessary fields (e.g. schedulables have a start time, end time, and duration, when duration can be calculated from the former two fields).

The setup of the problem in ASP could also likely benefit from optimizations of some kind; there is currently a high standard deviation of runtimes depending on certain parameters going into solving.

5 Acknowledgements

I would like to thank Professor Osborn, whose strong support, productive feedback, unwavering kindness, and engaging artificial intelligence teachings inspired this project and supported its completion. I would also like to thank the Pomona College Computer Science Department staff and students, for fostering the strong and creative community I've gotten to be a part of during my four years at the college. Finally, I would like to thank my parents, two brothers, and dog for their support through a challenging four years and through my entire journey learning software development, and Olivia Gay for continually inspiring me and keeping me sane through even the most difficult moments of the develpment process.

Bibliography

[Bal09] Marcello Balduccini. Representing constraint satisfaction problems in answer set programming. In *Proceedings of ICLP*, volume 9, pages 61–70. Citeseer, 2009.

[BGU$^+$05] Pauline Berry, Melinda Gervasio, Tomas Uribe, Martha Pollack, and Michael Moffitt. A personalized time management assistant: Research directions. pages 1–6, 01 2005.

[BSR10] Roman Barták, Miguel A Salido, and Francesca Rossi. Constraint satisfaction techniques in planning and scheduling. *Journal of Intelligent Manufacturing*, 21(1):5–15, 2010.

[CGB09] Liliana Cucu-Grosjean and Olivier Buffet. Global multiprocessor real-time scheduling as a constraint satisfaction problem. In *2009 International Conference on Parallel Processing Workshops*, pages 42–49. IEEE, 2009.

[CS97] Cheng-Chung Cheng and Stephen F Smith. Applying constraint satisfaction techniques to job shop scheduling. *Annals of Operations Research*, 70:327–357, 1997.

[DMP91] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1):61–95, 1991.

[FM05] Andrew Faulring and Brad A Myers. Enabling rich human-agent interaction for a calendar scheduling agent. In *CHI'05 Extended Abstracts on Human Factors in Computing Systems*, pages 1367–1370, 2005.

[GKK$^+$08] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo. 2008.

[HDM14] Philipp M Hund, John Dowell, and Karsten Mueller. Representation of time in digital calendars: An argument for a unified, continuous and multi-granular calendar view. *International journal of human-computer studies*, 72(1):1–11, 2014.

[Lif19] Vladimir Lifschitz. *Answer set programming*. Springer Heidelberg, 2019.

[NS09] Gilles Neubert and Matteo M Savino. Flow shop operator scheduling through constraint satisfaction and constraint optimisation techniques. *International Journal of Productivity and Quality Management*, 4(5-6):549–568, 2009.

[PMC08] Ferdinando Pezzella, Gianluca Morganti, and Giampiero Ciaschetti. A genetic algorithm for the flexible job-shop scheduling problem. *Computers & operations research*, 35(10):3202–3212, 2008.

[SV98] Eddie Schwalb and Lluís Vila. Temporal constraints: A survey. *Constraints*, 3(2):129–149, 1998.

[TVP03] Ioannis Tsamardinos, Thierry Vidal, and Martha E Pollack. Ctp: A new constraint-based formalism for conditional, temporal planning. *Constraints*, 8(4):365–388, 2003.