CSP

CSC384H Fall 2015

Project

University of Toronto

Vincent Tang	998192450	g2tangvi
Justin Djordjevic	997412152	g1djordj
Sandy Tran	996419148	g3transa

Introduction

The problem our group has chosen focuses on scheduling patients in a hospital for different treatments with the goal being a specific schedule for the hospital including times, resources and appointments. The nature of this problem is centered around constraints and there is no need to find a specific path to the solution. We must only ensure that our constraints are satisfied and that appointment requirements are fulfilled. Therefore it was decided that our problem would be modeled as a Constraint Search Problem (CSP). Each of the individual requirements that need to be matched up together can be separated into single constraints and using any form of propagation, a final goal state (a properly scheduled day of appointments) would be found. Viewing this as a search problem is not useful because we do not care about how we arrive at the final goal and a Bayes Net is also unnecessary since there are no probabilities involved with assigning different aspects of the problem like times and resources.

Problem Analysis

This project is a solution to solving the common task of scheduling multiple appointments at a hospital into a given day while considering limited resources and staff. Our scenario rises from the observation that most hospitals still use manual techniques to schedule appointments. Careful analysis shows that we can automate this task by looking at it from the perspective of a CSP. Appointments are almost always scheduled into fixed *slots* within a day and are allocated a set amount of time. Most hospitals run for 24 hours and we reflect this by assuming an appointment can be booked anywhere within a 24-hour period.

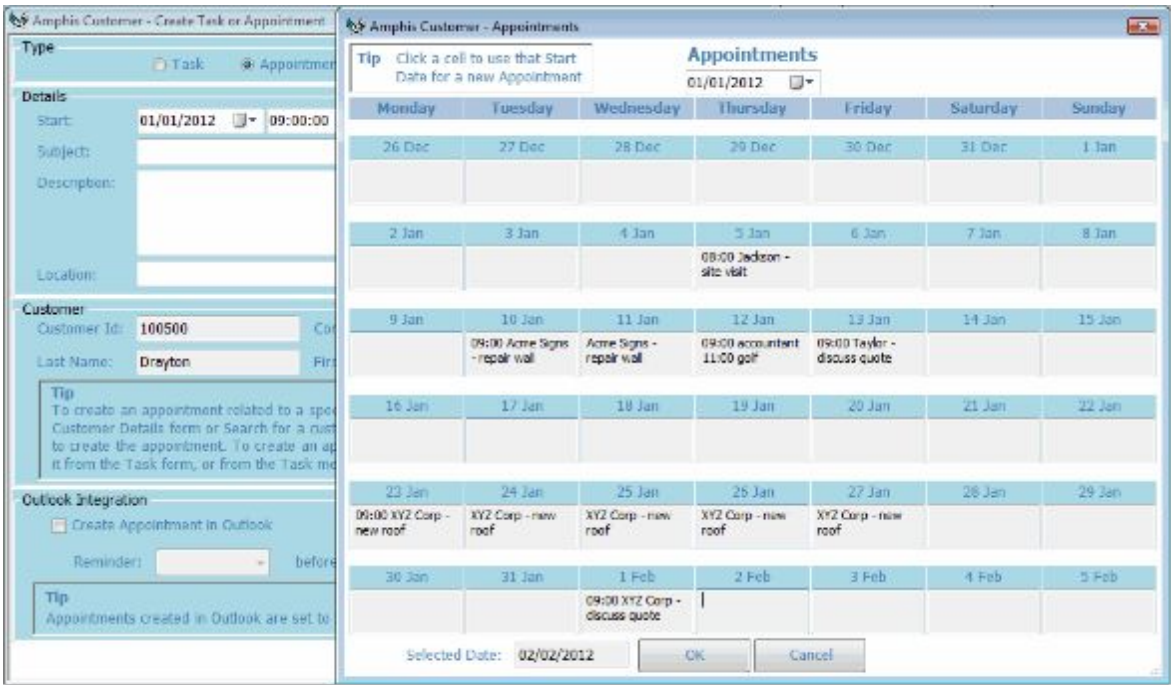

Figure 1: Even with software-based schedules, appointment scheduling is predominantly executed manually

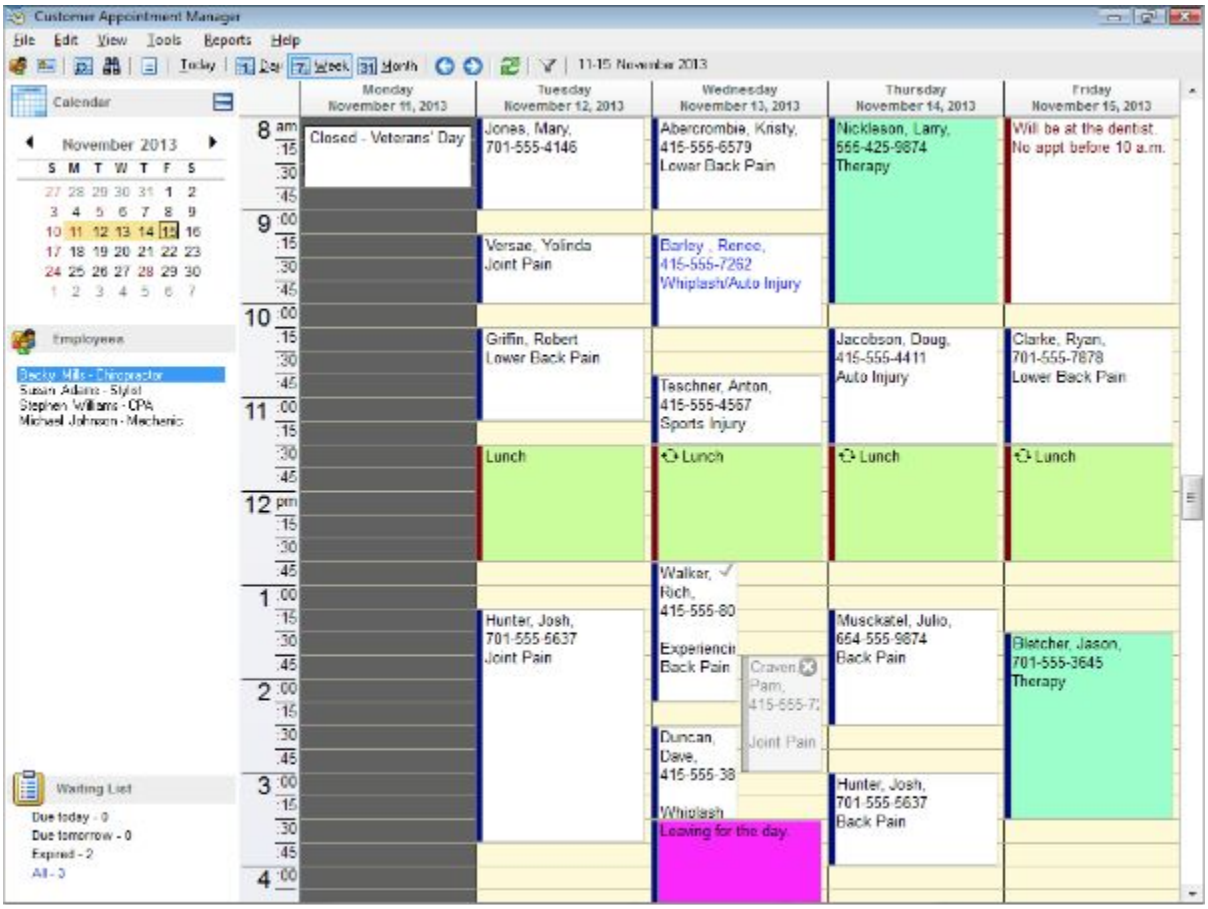

Figure 2: Manual operations make it tedious and time consuming to find conflicting appointments

Design Approach

We began this project by choosing a specific area of focus. After deliberation, we decided that a scheduling problem would be both interesting and practical as it is a task that people manually complete every day in every industry. We brainstormed numerous scenarios of scheduling and eventually arrived at the application of scheduling hospital appointments.

Hospital appointments are a good example of CSP problem as most appointments follow a set list of requirements or criteria. For example, almost every appointment will mandate the presence of a doctor or physician. Additionally, many appointments follow the exact same procedures independently of the patient (i.e. general checkups, or retrieving blood samples). This puts a limit on the list of variables that a schedule can contain which would be too diverse given other scenarios.

After agreeing on our specific application of the scheduling problem, we began to brainstorm more vigorously the exact requirements of an appointment. In real life applications, an appointment always has a start time, a planned procedure, and an estimated

duration. Furthermore, an appointment is typically assigned a professional at an appropriate level of experience. We took these facts and turned them into attributes, giving our appointments a start time, end time, a list of resources required, and a list of staff positions required. We used a similar approach when we designed classes for resources and staff by looking at real life procedures and translating them into classes and attributes. Some key considerations for resources were limited quantities and reusability. Key considerations for staff members were limits on work hours (both minimum and maximums), and positions or professional levels of experience.

CSP Approach

Approaching this problem from the perspective of a CSP, we focused on each variable to see how they would affect the scheduling of appointments. In this section we will discuss numerous constraints that we came up with and why we decided to enforce the constraints that were chosen.

Appointments have no constraints independent of other variables. We assume that the appointments are booked for their corresponding start and end times. We use constraints to see if the schedule is appropriate given the resources and staff available.

Resources present potential problems with scheduling appointments. If an appointment is scheduled and there are no resources available for a given procedure, then the appointment may need to be rescheduled. It is much more efficient to determine whether or not the appropriate resources are available before the appointment is booked. This brings us to the first resource constraint: an appointment cannot be made if the resources required for the appointment are not available. We also consider reusable and non-reusable resources. For example, a doctor may require a stethoscope. The doctor can use the stethoscope on one patient, and then use the same stethoscope again on another patient. However, a resource such as a bandage cannot be reused. Once a bandage is used on one patient, the total quantity of bandages available decreases. We take this into account by having a *reusable* attribute in our resource class. A reusable resource's available quantity decreases like a non-reusable resource, except once the appointment is complete, the reusable resource becomes available again. To ensure that reusable resources are only used when available, our last constraint on resources ensures that the number of reusable resources used in overlapping appointments does not exceed the total quantity available.

When it comes to appointments, hospital staff a major concern. Not only do they present limitations and potential problems in a similar manner of resources, but labour laws put the addition of limitations on how much a staff member can work. To represent the potential limits on work hours of staff, we gave staff attributes representing the minimum and maximum number of appointments they should cover. These limitations translate into constraints for our CSP. The first constraint requires that staff members work a specified

minimum number of appointments. The second constraint requires that staff members work no more than a specified maximum number of appointments. We also have to note that if a staff member is busy with one appointment, we must not book them for another appointment at the same time. We embody this in our third constraint, which specifies that no staff members are booked for overlapping appointments. Lastly, we must ensure that appointments have positional requirements fulfilled by only allowing appointments to be scheduled if staff with the appropriate qualifications are available.

The result of our approach can be summarised by the 6 constraints listed in the next section.

Constraints

1. All appointments are fulfilled by staff with correct positions/qualifications
2. No staff member is attending more than one appointment that is overlapping
3. Every staff member is working no more than their max number of appointments
4. Every staff member is working at least their min number of appointments
5. Overlapping appointments do not use more of one type of reusable resource than what is available
6. All appointments do not use more than the available number of each non-reusable resource

Input & Output

The main components of our CSP are the appointments, staff, and resources. These three components make up the input of our CSP program. More specifically, our input is comprised of the following components:

Inputs

- A list of appointment objects, each with a predetermined start time, end time as well as a list of required resources and a list of required staff positions
- A list of resource objects, each with a name and quantity, as well as an attribute that states whether or not the resource is reusable
- A list of staff objects, each with a name and position, as well as their minimum and maximum appointment hours to be assigned

As the goal of our CSP application is to determine a valid schedule of appointments with the proper distribution of resources and staff, our output will consist of one component. Our goal, assuming one exists, is to have an arrangement of staff and resources to fulfill the appointments that were given as inputs. The output is summarized below:

Output

- If a solution is possible, the program outputs each appointment with associated lists of required staff members and resources that fit the schedule given the inputs
- If a solution does not exist, the output contains a notification that there are insufficient resources or staff members to accommodate all the appointments given the inputs. The output is printed to the console as a formatted string.

Formal Definition of the CSP

In our CSP, we manage multiple variables.

Appointment Variable: $[a_1, a_2, a_3, \ldots, a_n]$
Domain: A list of appointment objects
Constraints:

- All scheduled appointments are fulfilled by staff with the correct positions

Resource Variable: $[rv_1, rv_2, rv_3, \ldots, rv_n]$
Domain: A list of resource objects e.g. $[r_1, r_2, r_3, \ldots, r_n]$
Constraints:

- Overlapping appointments do not use more of one type of reusable resource than available
- The total number of resources used in all appointments must be less than the number of available resources for reusable resources

Staff Variable: $[sv_1, sv_2, sv_3, \ldots, sv_n]$
Domain: A list of staff objects e.g. $[s_1, s_2, s_3, \ldots, s_n]$
Constraints:

- No staff member is attending more than one appointment that is overlapping
- Every staff member is scheduled for at most their max number of appointments
- Every staff member is scheduled for at least their min number of appointments

Project Code

Our code is based off of cspbase.py (and propogators.py) which was provided in assignment 2. Our solution is implemented in schedule_csp.py. We have also included our test file, schedule_test.py. The files included in this project are summarized below:

Files

- *cspbase.py*: Provided in A2, our implementation uses various methods and classes from this file
- *prpogators.py*: Contains different constraint propogators for BT Search/GAC
- *schedule_csp.py*: The implementation of our hospital scheduling CSP
- *schedule_test.py*: The test file containing run examples used to test our schedule implementation

Input

- A list of Appointment objects
- A list of Resource objects
- A list of Staff objects

Note: Appointments only use resources and staff listed in the Resource and Staff objects lists.

Output

- If a solution exists:
  Each appointment is printed along with the staff member required and that staff member's position
- If no solution exists:
  A notification is printed informing the user that there are insufficient resources or staff to accommodate all appointments within the day

Class Definitions

Appointment Class

Appointment objects represent appointments that we want to schedule in a day. They are initialized with start_time, end_time, [resources], [position]. These attributes are defined as follows:

start_time (integer [0-24]): The hour this appointment starts
end_time (integer [0-24]): The hour this appointment ends
resources (list[objects]): A list of resource objects required for this appointment
positions (list[objects]): A list of staff positions required for this appointment

Resource Class

A resource is an item needed for an appointment. Resources are limited which means some appointments may not be able to be scheduled because there are no resources available. Every resource has a fixed quantity, *qty_total*, which represents the total number of that

resource available. Resources who have the *reusable* attribute set to *False* are consumed once used, reducing the total quantity. Resources with *reusable* set to *True* can be reused and the total quantity is not affected after being used. Resources have the following attributes:

name (string): The name of the resource (e.g. 'needle', 'swab', 'bandage')
qty_total (integer): The total number of this resource available
reusable (boolean): True if the resource is reusable, meaning the qty_total does not decrement after being used. False if the resource is not reusable, meaning qty_total decrements with every use

Staff Class

A staff object represents a worker at the hospital. Staff have a set position, such as a nurse or a doctor, which may be specifically required for an appointment. Every staff member also has a minimum and maximum number of appointments they have to work in a day. Staff objects have the following attributes:

name (string): The name of the staff member
pos (string): The position of the staff member (i.e. nurse, doctor)
minh (integer): The minimum number of appointments that need to be assigned to this worker
maxh (integer): The maximum number of appointments that can be assigned to this worker

Methods

We use a variety of methods to define constraints, as well as help with determining whether a constraint is satisfied. The methods used in this project are outlined below:

assign_resource*(r, r_list):*

Input:

r: A resource variable
r_list: A list of resource objects

Output:

A list containing the resource object

Description:

This method is a helper function for csp_setup. It takes a resource variable, r, and assigns to it a list of objects, r_list. Returns the resource object in a list.

csp_setup*(name, app, res, staff):*

Input:

name: The name of the CSP
app: A list of appointment objects
res: A list of resource objects
staff: A list of staff objects

Output:

csp, app_vars, res_vars, staff_vars (see description)

Description:

This method first creates Variable objects out of the given lists. First we create variables with the appointments list. For each appointment variable, resources are made into resource variables whose domain contains the resources needed for the appointment. We also do the same for staff by creating staff variables whose domain is the staff required for that appointment. This method returns the csp (*csp*), appointment variables (*app_vars*), resource variables (*res_vars*), and staff variables (*staff_vars*).

add_overlapping_staff_constraints*(csp, overlaps, staff_list):*

Input:

csp: A CSP object which contains variables and constraints that specify the scheduling problem and utility routines for accessing the problem.

overlaps: A nested list of overlapping appointments. Each entry represents an hour for which multiple appointments are scheduled. Each entry is a list containing Appointment objects for that hour.

staff_list: A list of Staff objects

Output:

None

Description:

This method creates a constraint and adds it to the CSP object. This constraint ensures that staff members are not scheduled for more than one appointment at any given time. This is done by creating a list combining all positions required at the overlapping appointments and requiring that no staff member appears more than once in that time period. Satisfying tuples for this constraint are any permutation of the set staff members. Since we are taking permutations of a set, no one staff member will be allowed to appear more than once.

add_correct_staff_constraints*(csp, app_list, app_vars, staff_list):*

Input:

csp: A CSP object

app_list: A list of Appointment objects

app_vars: A copy of list of Appointment objects

staff_list: A list of Staff objects

Output:

None

Description:

This method creates a constraint and adds it to the CSP object. This constraint ensures that positions at each appointment are filled by appropriate staff. For each appointment, we create a satisfying set of staff members for an appointment by going through the list of staff members and looking for members who have positions that

match those required for the appointment. Staff members with positions that match those needed for the appointment in question is added to the satisfying tuple.

add_maxh_staff_constraints*(csp, app_list, staff_vars, staff_list):*

Input:

csp: A CSP object

app_list: A list of Appointment objects

staff_vars: A list of Variable objects that have Staff objects in their domain

staff_list: A list of Staff objects for entire CSP

Output:

None

Description:

This method creates a constraint and adds it to the CSP object. This constraint ensures that staff members are not scheduled for more appointments than allowed.

add_minh_staff_constraints*(csp, app_list, staff_vars, staff_list):*

Input:

csp: A CSP object

app_list: A list of Appointment objects

staff_vars: A list of Variable objects that have Staff objects in their domain

staff_list: A list of Staff objects for entire CSP

Output:

None

Description:

This method creates a constraint and adds it to the CSP object. This constraint ensures that staff members are scheduled for at least their established minimum number of appointments.

get_reusable_resources*(r):*

Input:

r: List of resources

Output:

result: List of reusable resources

Description:

This method is a helper function. It creates and returns a subset of the Resource objects which contains only reusable Resource objects.

add_reusable_resource_constraints*(csp, overlap_vars, resource_list):*

Input:

csp: A CSP object

overlap_vars: A nested list of overlapping appointments. Each entry represents an hour for which multiple appointments are scheduled. Each entry is a list containing Appointment objects for that hour.

resource_list: List of Resource objects in

Output:

None

Description:

This method creates a constraint over reusable resources. For each time slot with overlapping appointments, we create a list of all resources used for that time slot. We then go through all reusable resources and ensure that the number of reusable resources being used at any given time does not exceed the amount available at any one time. A satisfying tuple is a group of resources in which the number of times each reusable resource occurs does not exceed the amount of which the reusable resource is available for a given time slot.

get_nonreusable_resources*(r):*

Input:

r: List of resources

Output:

result: List of non-reusable resources

Description:

This method is a helper function. It creates and returns a subset of the Resource objects which contains only non-reusable Resource objects.

is_nonreuseable*(name, nonreusables):*

Input:

name: Name of a resource

nonreusables: List of non-reusable resources

Output:

Bool. Returns True if name is the name of a non-reusable resource. Returns False otherwise.

Description:

This method is a helper function that takes the name of a resource and checks if it is the name of non-reusable resource.

add_nonreusable_resource_constraints*(csp, app_list, app_vars, resource_list):*

Input:

csp: A CSP object

app_list: A list of all Appointment objects in CSP

app_vars: List of Appointment objects

resource_list: List of Resource objects in

Output:

None

Description:

This method creates a constraint over non-reusable resources. We create a list of all resources used across all appointments in the problem. A satisfying tuple is an arrangement of resources in which the number of times each non-reusable resource occurs does not exceed the amount of which the reusable resource is available for the entire duration of the CSP.

get_overlapping_appointments*(app_vars):*

Input:

app_vars: A list of Appointment objects

Output:

overlaps: A nested list of overlapping appointments. Each entry represents an hour for which multiple appointments are scheduled. Each entry is a list containing Appointment objects for that hour.

Description:

This method groups appointments which overlap in time together. The method creates a list with 24 entries. Each entry represents one hour of the day. Each hour entry is a list that contains all appointments that occur within that hour.

print_soln*(l):*

Input:

(l): List of Appointment objects

Output:

None

Description:

Print the final arrangement of staff to appointments. For each appointment, the appointment number, start and end times, staff member names and positions are displayed.

schedule_model*(a,r,s):*

Input:

a: List of Appointment objects

r: List of Resource objects

s: List of Staff objects

Output:

csp: CSP object

app_vars: List of Appointment objects

Description:

Create the CSP, all the Variable objects and all the Constraints. Return the final CSP and all the appointments to print the final output if the problem has a solution.

solve_schedule*(a,r,s):*

Input:

a: List of Appointment objects

r: List of Resource objects

s: List of Staff objects

Output:

Bool

Description:

This method takes lists for Appointment objects, Resource objects and Staff objects and converts them into a CSP object representing the scheduling problem. It then applies a GAC routine to the CSP to solve the scheduling problem. If the problem can be solved, the method prints the solution and returns True. Otherwise the method returns False.

References

1. http://www.saimgs.com/imglib/products/screenshots/amphis-customer-5-large.jpg?v=48840
2. http://www.abs-usa.com/files/3613/4091/2307/Day_View.png