A Smart Calendar System Using Multiple Search Techniques

Jake Cowton (Undergraduate student)* and Longzhi Yang†
Department of Computer Science and Digital Technologies,
Faculty of Engineering and Environment, Northumbria University,
Newcastle Upon Tyne, NE1 8ST, United Kingdom
Email: {*jake.cowton@northumbria.ac.uk, †longzhi.yang@northumbria.ac.uk}

***Abstract*—Calendars are essential for professionals working in industry, government, education and many other fields, which play a key role in the planning and scheduling of people's day-to-day events. The majority of existing calendars only provide insight and reminders into what is happening during a certain period of time, but do not offer any actual scheduling functionality that can assist users in creating events to be optimal to their preferences. The burden is on the users to work out when their events should happen, and thus it would be very beneficial to develop a tool to organise personal time to be most efficient based on given tasks, preferences, and constraints, particularly for those people who have generally very busy calendars. This paper proposes a smart calendar system capable of optimising the timing of events to address the limitations of the existing calendar systems. It operates in a tiered format using three search algorithms, namely branch and bound, Hungarian and genetic algorithms, to solve different sized problems with different complexity and features, in an effort to generate a balanced solution between time consumption and optimisation satisfaction. Promising results have shown in the experimentation in personal event planning and scheduling.**

I. INTRODUCTION

Existing calendar systems assume that all the events are fixed and hence they only provide an overview of events and tasks that are happening in a given period of time. They allow the owner to add, delete and edit items, and many are also able to communicate with other people's calendars too, by means of sending activity invitations. However, no mainstream, personal calendar, such as those provided by Google, Apple, Microsoft amongst others, offers functionality to upgrade a standard calendar from a simple collection of reminders to a comprehensive personal planning tool to effectively plan both fixed and flexible events for users.

Although some events, such as lectures, conferences and birthdays, have fixed dates and times that can easily fit into the structure that conventional calendars offer, many events, such as social meetings and studying, are more flexible in when they can be done. The decision-making process used to calculate when is best to do each task can be automated using advances in artificial intelligence, and actually a few such systems have been developed.

PTIME is a business-focused planner that has a strong focus on multi-user calendar compatibility, particularly for room bookings [1]. PLIANT is another event planner that is built on top of PTIME, which also tries to learn the preferences of the user in order to make user-tailored suggestions, besides scheduling events based on preference [2]. PLIANT is developed by applying constraint satisfaction problem (CSP) techniques [2] and is particularly useful when decisions are not easy to make: under-constrained scheduling, where there are many feasible options to chose; and over-constrained scheduling, where there are too few options available.

SELFPLANNER [3] is another smart calendar solution. In this system, everything in a calendar is treated as the same 'type' of object instead of trying to separate things into 'events' and 'tasks' etc. This is because both events and tasks require the user's time, and treating them separately may lead to unnecessary user effort in overall time management. Instead of providing the user with the best calculated plan, it provides several plans and allows the user to select from them based on personal preference. This is implemented through the use of hard and soft constraints to make the calendar more flexible for the user. The software is also capable of taking less traditional soft variables into account such as how much of the user's attention will be required to carry out the event. Plans for SELFPLANNER are created and optimised through the use of squeaky wheel optimisation (SWO) [4].

As only one search algorithm is used the above system may perform well for one type of problem, but they may struggle for others as algorithm's performance varies problem to problem. This paper presents a smart calendar system that is able to optimise personal events using multiple search algorithms, including branch and bound, Hungarian and genetic algorithms. Branch and bound techniques perform an intelligent exploration of the search space but can potentially be exhaustive; Hungarian can have great performance in time, but only works with events of equal length and spacing; and genetic algorithms can handle very complex problems, but do not guarantee perfect solutions. Given an event optimising problem, a specific algorithm is then chosen depending on the search space size and how quick the solution could be generated. Please note that the focus of the proposed system is not on changing how a calendar is used, rather it instead tries to add functionality to it, maintaining its current purpose.

The remainder of this paper is structured as follows. The background of the research topic is introduced in section II. Section III presents the proposed work in personal event planning by globally optimising the preferences and section IV applies the proposed approach to real world problems for demonstration and evaluation. The paper concludes in section V with future research directions recommended.

II. Background

Fundamentally, planning and scheduling is a combinational optimisation problem with symbolic representation of events and various types of constraints on events. Three search algorithms are employed in the proposed system with each targeting a different type of problem, which are introduced as follows.

A. Branch and Bound

Branch and bound is an algorithm design commonly used to solve combinatorial optimisation problems such as the Job Shop Scheduling Problem (JSSP), which is "based on the idea of intelligently enumerating all feasible solutions" [5]. It is a type of constraint satisfaction problem (CSP) algorithm which uses heuristics to efficiently calculate a feasible solution; the value of this is stored as the upper bound. Branches from this solution have their cost calculated; if their value is better than the upper bound, the branching continues until a full, better solution is found. Once the upper bound is exceeded however, that branch is "pruned" as no solution with this starting combination could possibly be better. A lower bound is also used which is calculated using heuristics to indicate the best likely solution that branch can provide. If the lower bound exceeds the upper bound the branch is pruned.

The branch and bound algorithm is a well-known, versatile solution for many problems, including flow shop scheduling [6], project scheduling [7] and the generalised assignment problem [8]. Often, combinatorial optimisations have the aim of reducing cost or idle time across multiple machines, each carrying out several jobs; whereas this paper presents the use for maximising the preference of events assigned to time slots. However, this becomes much more complex when events vary in length and have multiple preferences.

Somol et al. [9] briefly summarises the problems with branch and bound in saying that there is no possible way to know if branch and bound will prune enough branches to be significantly faster than an exhaustive search. They give two potential reasons for branch and bound having poor performance in some cases; near the root of the search tree, "criterion value computation is usually slower" and "sub-tree cut-offs are less frequent" both of which suggest that there may be problems dealing with deep trees (a large number of events).

In summary, branch and bound search is a significant improvement on a standard depth first search even when backtracking is used, but in some cases is no better than an exhaustive search which, in the worst case scenario, will run in $O(n!)$ time. For this reason, branch and bound is a good search algorithm for a small n but not as useful for larger problems.

B. Hungarian Algorithm

Kuhn [10] proposed an algorithm (see Fig. 1) to solve the general assignment problem, where m jobs must be assigned to m workers in the most cost efficient way. Unlike most of the other algorithms presented so far, the Hungarian algorithm does not use a tree structure for the search space; it instead uses a matrix of cost values which are reduced until a best combination can be found.

The functions used in pseudocode shown below operate as follows:

- minVal - Returns the minimum value in the matrix
- coverZeros - Returns the minimum number of vertical or horizontal lines required to cover all zeros
- lowestUncovered - Returns the lowest not covered value in the matrix
- numCovers - Returns the number of times a value is covered
- assignZeros - Returns a valid assignment of jobs to workers

```
SOLVE(𝕄)
Input:
   𝕄, a preference matrix of m ∗ m
(01) minval ← MINVAL(𝕄)
(02) foreach v ∈ 𝕄
(03)     v ← (v − minval)
(04) while COVERZEROS(𝕋) < m
(05)     low ← LOWESTUNCOVERED (𝕋)
(06)     foreach v ∈ 𝕋
(07)         if NUMCOVERS(v) = 2
(08)             v ← (v + low)
(09)         if NUMCOVERS(v) = 0
(10)             v ← (v − low)
(11) plan ←ASSIGNZEROS(𝕄)
(12) return plan
```

Figure 1. Pseudocode showing the Hungarian algorithm

The problem can be modified to allow n workers by introducing dummy variables to even out m and n which are later ignored [11]. Edmonds & Karp [12] went further with the algorithm by showing that it could be used to solve $m*m$ assignment in $O(n^3)$ time instead of $O(n^4)$ which was the time complexity that the Hungarian algorithm was originally thought to be.

Since the algorithm's original conception, variations of this algorithm have been used to solve specific subsets of the assignment problem such as Mills-Tettey et al. [13] who devised a way to apply the Hungarian algorithm to a matrix of changing costs. This variation was designed with transportation optimisation in mind and would allow calculations to account for unforeseen road closures and other randomly occurring changes to the scenario.

C. Genetic Algorithms

Modelled upon Darwin's theory of evolution by Holland [14], evolutionary algorithms, particularly genetic algorithms (GAs), have become a very powerful tool for optimisation and search problems [15]. This algorithm has 3 main components:

1) Selection
2) Crossover/Mating
3) Mutation

Because a GA must be tested and benchmarked based on its overall performance, a short-list was made of potential functions and values for these components and their pressures. All combinations and variations of these were then tested to find the best performing combination. Each of the three important steps in a GA is introduced as follows.

Selection is the process of determining which solutions will be 'mated' together (crossed over). There are many ways to do this, the simplest of which is where the k best solutions in the pool are selected; however much more advanced methods are also available. A commonly used method is tournament selection. Goldberg & Deb [17] show that the time complexity of this selection method is $O(n)$. It works by selecting n individuals from the population at random, and then selecting one of these individuals with a probability p, which changes based on the fitness score of each individual. The highest fitness score is picked with a probability of $p(1-p)$ the second is picked with probability $p((1-p)2)$ and so on.

The crossover function determines how two chromosomes will be mated together. Single-point crossover, for example, being used on individuals of length n, will choose a single bit position from values 1 to n. The offspring is then created using the bits of the first parent individual that are before k and the bits from the second parent individual that are after k, thus making a combination of the two 'parent' individuals.

As in selection, there are more advanced methods of carrying out crossover, some of which are designed to specifically handle individuals whose order is important. Poon & Carter [18] did a multitude of tests on several of these types of crossover algorithms and found that union crossover (UX) is "comparable in applicability and performance to the classical crossover used in binary-string GAs." However it can be very slow to run taking 26 times longer than the faster methods of cycle crossover (CX), partially mapped/matched crossover (PMX) and ordered crossover (OX). In this research, though CX finished quickly, it is one of the worst performing methods used as, unlike PMX, it is not good at preserving substrings of neighbouring elements.

Mutations create small changes in offspring with the purpose of creating diversity within the population of solutions. They allow individuals to avoid getting locked in local optima [19] which, as shown in SAs, can overcome a serious problem in these types of search methods. An example of mutation would be, given mutation probability of 0.2 and an individual component mutation probability of 0.01, as shown in Fig. 2.

MUTATE($\mathbb{Q}$)
Input:
$\mathbb{Q}$, a suggested order of events
(1) **if** rand(0,1) ≤ 0.2
(2) **foreach** $f \in \mathbb{Q}$
(3) **if** rand(0,1) ≤ 0.01
(4) $plan \leftarrow$ BITFLIP(f)
(5) **return** $plan$

Figure 2. Pseudocode showing the function of probabilities in mutation

The probability of mutations is one which needs to be fine tuned for a specific problem; however to summarise several research papers into a short rule of thumb, mutation rates should be between 0.01 (for small populations 30) and 0.001 (for large populations 100) [20], though it is also recommended to take the length of the encoding into account. It also explains in this work that too high a probability can cause the GA to turn into what will essentially be a random search.

Population size plays an important role in the performance of GAs. Too small of a population will increase the chance of peaking at a local optima, due to a lack of diversity. Too large of a population will require a lot more processing time because of the increased number of evaluations that will need to be calculated. However some research [21][22] shows that less generations are needed for larger populations sizes. In most, if not all, cases a larger population size will result in a better final solution [23].

Pressures are used to represent the probability of crossover and mutation taking place. For crossover, pressure is usually set above 0.5, and if set lower than this it would nullify the purpose of GAs, as top-class chromosomes would be unlikely to be mated. For mutation pressure, the value is usually set below 0.4 but over 0.1. This is because a value higher than this would likely alter too many chromosomes thus changing too many individuals; a value lower than this would likely cause the algorithm to arrive at a local optima, as discussed earlier. Essentially, there is a need for an "optimal balance between exploitation and exploration" [21].

Deb et al. [24] proposes a GA which employs elitism, a technique which allows the best solution in the pool to be copied to the next iteration pool unchanged. The fact that it is copied and not moved is important as the elite individual can still be used in the crossover process so long as there is a copy of the original in the next iteration. This is a component that is considered to be a hugely important factor [25] to improve the performance of GAs.

III. THE SMART CALENDAR SYSTEM

There are a good variety of search algorithms available in the literature for computational optimisation problems, each with different strengths and limitations. However the proposed smart calendar system employs three different algorithms, including branch and bound, the Hungarian algorithm, and genetic algorithm. Small search spaces are dealt with by branch and bound algorithm, large search spaces are handled by the genetic algorithm, and Hungarian algorithm is applied to deal with problems when the available time exactly matches the time required for events to schedule.

In particular, branch and bound is employed when the number of slots in the search space is very low (no more than 5 events or 6 time slots regardless of size). The Hungarian algorithm is used for two specific situations: 1) when all the events in the search space, and the empty spaces between them, are of the same length; and 2) when all the events have to be done within a given amount of time as soon as possible without break, and thus the event allocation problem changes to a sequence optimisation problem. This is because this algorithm is only able to deal with problems with the same

number of events and time slots. Finally, GAs are used to solve all other problems as their search spaces can be very large. In theory, a GA could be used to solve all of the problems in reasonable time, but it would not be possible to guarantee that local minima would be avoided in reasonable time. However, both branch and bound and Hungarian algorithms are able to always find the global optimum, which is why they are the first choice when they can solve the problem in a reasonable amount to time.

A. Problem Representation

The scheduling problem can be formally represented as a CSP problem. Basically, a CSP problem is defined as a finite set of variables, each of which is associated with a domain of finite elements, and a set of constraints that restricts the values the variables can simultaneously take [26], [27]. The task of a CSP problem is to assign a value to each variable such that all constraints are satisfied (and sometime the objective function is maximised or minimised). Formally, A constraint satisfaction problem is a triple (X, D, C), where

- X is a finite set of variables $x_1, x_2, \ldots, x_n$;
- D is a function which maps every variable in X to a set of objectives of arbitrary type. That is, D_i is the domain of variable x_i. Each domain D_i is a set of possible values which variable x_i may take.
- C is a finite set of constraints on an arbitrary subset of variables in X. In other words, C is a set of compound labels.

The time is discretised into a number of time slots in this work such that the event optimisation can be represented as a CSP problem. The number of discretised slots is dependent on the specific problems. Generally speaking, finer discretisation usually requires higher computational effort and thus slower processing, but may lead to fine-tuned optimisation results in the same time; rough discretisation normally leads to fast processing, but the results may not be quite optimal. In this work, slots are created to be the size of the greatest common divisor of all of the event lengths. For example, slot sizes for processing 3 events which are 2, 4 and 6 hours long respectively, the slots size would be 2 hours.

In order to reflect the needs of optimisation, each possible assignment of events to time slots is attached with a preference value. The preference values are ranged from 0 to 9, where 0 means the event must start at this time and 9 indicates the lowest preference. The aim of the proposed system is to minimise the sum of the preference of all events. Suppose that there are m events need to be scheduled in to n time slots. The smart calendar system can be mathematically represented as a CSP problem as follows:

- $X = \{x_1, x_2, ..., x_n\}$, where n is the number of discretised time slots.
- $E = \{e_1, e_2, ..., e_m\}$, where m is the number of events need to be planned/ scheduled.
- $D_{x_i} = \{e_{i1}, e_{i2}, ..., e_{il}, NULL\}$, where $i \in \{1, 2, ..., n\}$, and $e_{ij} \in \{E\}, j = \{1, 2, ..., m\}$. Null represents no event is started at the concerned time slot. The preference value for assigning event e_{ij} to time slot x_i is P_{ij}.
- $C = \{c_1, c_2, c_3\}$.
- c_1: if an event e_p is assigned to x_i, $i \in \{1, 2, ..., n\}$, and if the time required for the event is longer than the discretised time slot, the next p time slots will be assigned as Null, where $p = \lceil |e_P/t\rceil - 1$ and $|e_P|$ is the length of event e_P.
- c_2: each event only can appear once in the calendar.
- c_3: minimising the following objective function:

$$\sum_{j=1}^{m} P_{ij},\ where\ i \in \{1, 2, ..., n, n+1\}. \quad (1)$$

  $P_{(n+1)j}$ is a number greater than 9 representing the penalty when event j is not scheduled. The larger the value of this, the less important the concerned event is.

B. Preprocessing

In order to reduce the search space, two steps of pre-processing are used in this work. When a set of events are submitted by the user for processing, a block of valid time slots are created starting from the earliest preference time to the latest preference time. The database is then checked to see if any existing events are already within this time range. If events are found and they are locked, that is they cannot be modified, then the time slots it occupies are removed from the list of valid slots. If the event is not locked, that is it can be modified, then that event is removed from the calendar and added to the set of events that the user submitted so that it can be re-processed for a global optimal solution. These steps are repeated until no unlocked events exist in the list of valid time slots. Once the above step is completed, partial constraint propagation is carried out to ensure that each event has at least one valid range of time slots that it can be assigned to. This can assist in identifying when a set of events have no valid configuration before using an algorithm.

C. Event Planning by Branch and Bound

As with all problems to be solved using branch and bound, this combinatorial optimisation problem is represented as a rooted tree, as shown in Fig. 3. In this rooted tree, each node represents an event whose position on a branch determines when it is planned for. In order to relate the position on a branch to a specific time, a time period is specified which a branch can be mapped onto so that the position signifies a starting time.

Though the way in which the tree is traversed is the core of this algorithm, the way in which the tree is structured plays an important role. Preferences for events usually increase and/or decrease gradually to/from a main, maximum preference; because of this, trees are generated in an order designed to quickly eliminate branches. This is done by creating branches

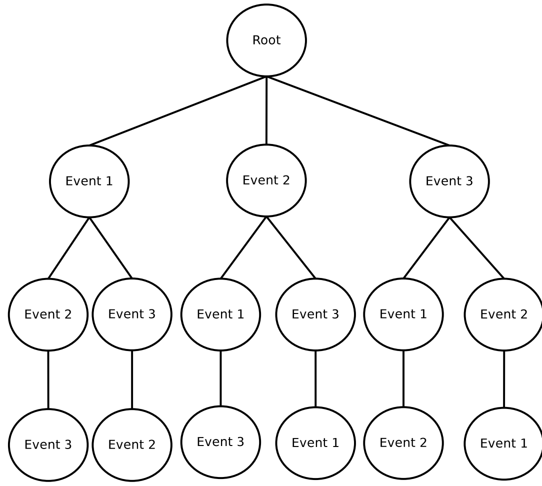

Figure 3. Tree representation of event planning

with as small changes as possible, that is by only changing the position of each event by no more than one position where possible. This makes it easier for the algorithm to prune branches because it is unlikely that preferences change drastically due to small positional changes.

Suppose there are m events to be scheduled. The aim of this algorithm is to minimise the preference value based on the preference function:

$$f = \sum_{i=1}^{m} e_i. \tag{2}$$

In most branch and bound algorithms, the starting value will be 0 and the values are summed and added to this to find the total value of the solution. However, the aim of this algorithm is to minimise the preference value, and it is therefore necessary to reverse the way in which the current preference value changes. For this implementation the starting value is calculated as 1000 per event, and the total preference is represented as Equation 3.

$$f_{initial} = \sum_{i=1}^{m} e_i * 1000 \tag{3}$$

As an event is assigned, 1000 is taken from the current value and the preference value is added ($current = current - 1000 + preference$).

As introduced in Section II, two bounds are usually used in Branch and Bound algorithm. The heuristic function used to determine the lower bound of a partially assigned branch is calculated based on what the best possible outcome could be for each of the remaining nodes. This is computed by taking the sum of preference values already assigned and adding the lowest preference value for each remaining node. For example, given the preference matrix as shown in Table I. If event a is assigned to 12:00 and event b is assigned to 13:00 then the heuristic function would take 5 for event c at 14:00 and 1 for event d at 16:00. Therefore, the lower bound of a leaf from the nodes explored so far is 2+3+5+1 = 11.

	12:00	13:00	14:00	15:00	16:00
a	**2**	1	4	5	9
b	3	**3**	1	2	3
c	1	4	**5**	6	7
d	4	1	3	2	**1**

Table I. PREFERENCE MATRIX FOR 4 EVENTS AND 5 TIME SLOTS

Formally, suppose that m events need to be scheduled to n time slots, and i events $e_1, e_2, ..., e_i$ have been tentatively assigned to time slots from x_1 to x_j. The best possible overall preference can be calculated by adding the sum of the preferences so far up to e_i to the best possible remaining preference of e_j, $j = \{i+1, i+2, ..., m\}$, as shown in Equation 4. The value calculated by f constitutes the lower bound of the branch at node x_i, which can be used to determine if a branch is destined to fail to reach an optimal solution, without having to explore the whole branch.

$$f' = f(e_1, e_2, ..., e_i) + g(e_{i+1}, e_{i+2}, ..., e_m), \tag{4}$$

where $g(e_{i+1}, e_{i+2}, ..., e_m)$ is the best possible preference value for the set of unassigned events $\{e_{i+1}, e_{i+2}, ..., e_m\}$:

$$g(e_{i+1}, e_{i+2}, ..., e_m) = \sum_{p=i+1}^{m} \min_{q=j+1}^{n} c_{pq} \tag{5}$$

When the lower bound is found to be greater than or equal to the current, the branch is temporarily pruned at that point.

D. Event Planning by Hungarian Algorithm

As the Hungarian algorithm only works in a one-to-one assignment, meaning assigning a single event to multiple slots would not be possible, it can only be used in very specific circumstances. However, when these requirements are met, it provides a significantly faster way to solve the combinatorial optimisation problem this paper aims to solve. One scenario where these circumstances are able to be taken advantage of is for the attending of a conference and other similar proceedings where events are evenly spaced. At such events there are often multiple sessions on at any one time and these sessions occur in regular intervals meaning an attendee is required to select which sessions they wish to attend based on preference.

For this problem the Hungarian algorithm is implemented using a $m*n$ matrix, where n columns represent n time slots, and m rows represent m events. As in many cases there will be more time slots available than events, extra rows can be added such that $n = m$. These extra events will be dummy events to allow the algorithm to function correctly. In cases where this is required, all rows which are dummy events are then ignored once the algorithm is complete. For example, Table II shows 3 events x, y and z each being one hour long, which are to be assigned to times 12:00, 13:00, 14:00 and 15:00. As $n \neq m$ a dummy event is added to the preference matrix which are given values of 0 for all time slots. The grid is then filled with the preferences for each event starting at a given time. The algorithm is then run as usual for a global optimal solution.

	12:00	13:00	14:00	15:00
x	0	1	4	5
y	3	3	1	2
z	1	4	5	3
-	0	0	0	0

Table II. PADDED PREFERENCE MATRIX FOR THE HUNGARIAN ALGORITHM

E. Event planning by Genetic Algorithm

When a GA is used for combinatorial optimisation, the most common way for a chromosome to be represented is by using a vector of indices which represent events, where each index is the position of the event in the original list. For example, 5 events could be represented as a chromosome shown in Fig. 4. As discussed earlier for the problem presentation, the GA chromosome representation will also include 'NULL' genes to represent empty time slots. For the GA, these values are represented using -1 for the index, as illustrated in Fig. 5.

0	1	2	3	4

Figure 4. GA chromosome index representation

0	1	2	3	4	-1	-1	-1

Figure 5. Initial GA chromosome index representation with 'NULL' values

The initial population was generated by selecting 300 random permutations of the initial chromosome. For instance, the chromosome shown in Fig. 6 is a random permutation of the initial chromosome shown in Fig. 5. The random initial population guarantees the population has an unbiased starting point to search from.

3	-1	2	0	-1	1	4	-1

Figure 6. Random permutation of Fig. 5

In order to ensure the GA is fine-tuned to the problem, different configurations of methods and values were tested to determine which performs the best when solving an average scheduling problem. Note that tournament selection has a good time complexity and its pressure is simple to modify by altering the size of the tournament, and the larger this is, the less chance a weak chromosome has of going through to crossover. This approach is utilised in this work as the selection algorithm for this GA. In line with the experiment carried out by Miller and Goldberg [28] tournaments of size 2, 3, 4 and 5 were used in a GA configuration test (N.B a tournament size of 1 would be the same as random selection).

Based on the research carried out into crossover methods, PMX and OX were tested in the configuration of the GA as they both perform well and are able to work with ordered chromosomes. This is important as many methods work with whole values rather than ordered individuals. Crossover pressures of values 0.5 to 0.9 (incrementing in values of 0.1) were used in the test.

Index shuffle was used for the final GA configuration as it is specifically designed to work well using vectors of indices. It was tested with pressures of 0.1 to 0.4 (incrementing in values of 0.1). This method also requires that an individual probability be set, which determines how likely it is for a specific index to be shuffled. Individual probabilities are typically significantly lower than normal mutation pressure (under 0.1) as a high value would make it very likely that most of the chromosome would be altered, which would turn the GA into a random search. For this reason, three values will be chosen to be tested for the configuration: 0.02, 0.04, 0.06 and 0.08.

The configuration test checked all combinations of the components mentioned above and revealed that the following functions and values performed the fastest and with the best schedule.

- **Tournament Size**: 2
- **Crossover algorithm**: PMX
- **Crossover Pressure**: 0.5
- **Mutation Pressure**: 0.1
- **Mutation individual pressure**:0.04

As the search space the GA has to cover increases/decreases due to the number of events being processed, it is necessary for the number of iterations to increase also. In order to do this effectively, the number of iterations scales in line with the number of events in the search. The number of iterations used by the GA is calculated using the equation in Equation 6, with a maximum number of 200 in order to ensure the search remains within reasonable time.

$$iteration = (n + 1) * 10. \tag{6}$$

As GAs have the potential to reach a valid solution prior to reaching the maximum number of iterations the algorithm is permitted to terminate early if certain requirements are met. For this algorithm the early termination requirements are that the solution found is acceptable, which is defined as having an average preference of 2, with 'NULL' events are ignored in this average. Once an acceptable solution has been found the chromosome is decoded back into a list of events, maintaining the output order. This is then mapped to a period object which outlines what time slot each position refers to.

IV. EVALUATION

Three sets of testing are conducted in this section to demonstrate and evaluate the proposed smart calendar system. These tests were carried out using a laptop running a dual core i7 processor and 32GB of RAM.

A. Test on Small Number of Events

A scenario was tested using the system which demonstrates how each algorithm would be used. The scenario is as follows:

1) Schedule 3 events of different sizes (1hr, 2hr, 1hr) over one day
2) Add an extra 1hr event which overlaps with the 1hr event from test 1
3) Schedule 6 events over several weeks (Differing in sizes)

The first test would be carried out by the branch and abound algorithm as it is under the threshold. The second test would

be attempted to be carried out by the Hungarian algorithm as it only contains one event (meaning size and spacing is consistent). However, as it overlaps with an existing event which is not locked so this is added to the list of event which need to be processed. As this event is the same size it might be eligible to be processed by the Hungarian algorithm if the spacing between preferences is consistent, if not it would be calculated using branch and bound. In the specific test carried out, Hungarian algorithm was used. The third test is processed by the Hungarian algorithm as it matches the requirements for that algorithm to be used. Finally, the GA is used to process the large set of events.

The times taken for these algorithms are shown in Table III. The times taken for processing of all algorithms have been shown so that a comparative analysis of the results can be carried out to justify the need for all algorithms. Note that the GA results will vary proportionately to the the time taken due to the nature of the algorithm. All algorithms found the global maximum.

	BB	Hungarian	GA
Test 1	0.012	N/A	0.008
Test 2	0.012	0.001	0.682
Test 3	N/A	N/A	23.164

Table III. PERFORMANCE TESTING RESULTS (TIMES ARE IN SECONDS)

Test 1 represents a general assignment of a days events which is where the branch and bound algorithm is most useful. Although the results show the the GA was able to calculate the solution faster, it cannot guarantee that the solution it produces will always be the global maximum, which is why it is important to use the branch and bound algorithm where possible. Test 2 demonstrates how useful the Hungarian algorithm is when its requirements are met. It was able to process the addition of an event whose preferences overlapped with an existing event in significantly less time than the GA. However, the table also shows that it is not often that the algorithm can be used as it has very strict requirements for it to be able to function properly, but this is acceptable due to the performance that it offers. This algorithm is specifically useful for timetabling for consistent event sizes such as conferences and timetabling. The final test demonstrates how the GA is able to handle a very large search space. Although there are not a large number of events, having a range of preferences over a large period of time for multiple events creates a large number of potential combinations.

B. Test on Medium Number of Events

Though the above tests demonstrate performance with regards to time, the tests themselves do not represent some of the more challenging problems the application can solve and also do not evaluate the quality of the solution provided. Two sets of tests were devised, as illustrated in Tables. IV and V, to evaluate the algorithms performance for solving over and under constrained problems as this is where this software has the best application. These were solved by all three algorithms and the processing time and quality of solution were recorded.

-	09:00	10:00	11:00	12:00	13:00
M	1	2	2	6	6
N	0	-	-	-	-
O	1	1	2	8	8
P	4	3	1	4	5
Q	8	6	3	3	1

Table IV. OVER CONSTRAINED PREFERENCE MATRIX (EVENTS WITH SAME LENGTHS)

-	09:00	10:00	11:00	12:00	13:00
R	1	1	1	2	3
S	2	1	1	2	2
T	3	1	1	1	1
U	1	2	2	2	2
V	5	3	3	3	1

Table V. UNDER CONSTRAINED PREFERENCE MATRIX (EVENT WITH THE SAME SIZE)

	BB	Hungarian	GA
Over Constrained			
Time (Seconds)	0.567	0.001	0.018
Preference (Min 8)	8	8	11
Under Constrained			
Time (Seconds)	0.583	0.001	0.009
Preference (Min 5)	5	5	10

Table VI. PERFORMANCE FOR UNDER AND OVER CONSTRAINED PROBLEMS

Table VI shows the performance of all three algorithms for the test described above. Once again this demonstrates the potential of the Hungarian algorithm when dealing with events of the same size. Alongside this, the branch and bound algorithm is outperformed by the GA with respect to time. However this test shows that the increase in processing time is compensated for as the branch and bound was able to find the global maximum whereas the GA was not.

C. Test on Large Number of Events with Same Length

In order to properly demonstrate the use of the Hungarian algorithm, 600 events with random preferences were generated and set for assignment over a period of four weeks. The times listed in Table VII shows how long the Hungarian algorithm and GA took to find the global maximum solution for this scheduling problem. Branch and bound algorithm was not applied to this problem as it will take unacceptable amount of time to process.

	BB	Hungarian	GA
Time (Seconds)	N/A	21.178	1291

Table VII. STRESS TEST OF HUNGARIAN ALGORITHM AND GA

This stress test demonstrates the full potential of the Hungarian algorithm when used in comparison to the GA. The Hungarian algorithm is able to not only able to find a solution within reasonable time, but also is able to find the best solution possible. For this reason it is able to seriously outperform most other algorithms when its conditions are met.

D. Discussion

Contrary to SELFPLANNER [3], this system does not give the user multiple variations of good solutions for the user to

choose from. It instead relies upon the user to submit accurate data describing their preferences of when an event should be carried out. Also, where it uses hard and soft constraints to represent different sub-problems, this planning system takes a different approach to tackling the same issue. In the software this paper presents, users can specify a preference value of 0 to signify that the event must happen at a certain time. This allows all of the examples brought up by Refanidis et al. to be solvable within this system.

V. Conclusion

This paper has presented a smart, personal calendar that works as a complementary of a regular calendar by adding extra functionality of a comprehensive planning system. In particular, the system effectively employs three search algorithms for different planning tasks in an effort to combine the advantages of these search algorithms while avoiding their drawbacks at the same time. This is done by assigning each algorithm specific types of assignment problems, the allocation of which is based mainly on the size of the search space. The evaluation of the algorithms used shows how branch and bound is used to maximise its ability to always find the global maximum, how the Hungarian algorithm can process data incredibly quickly and how the genetic algorithm is capable of solving large search spaces.

The work presented herein is a summary of an undergraduate final year project developed in limited amount of time, and the work can be improved in a number of ways. The current system is a single objective optimisation based on event preference. Sometimes, planning may need to be made based on other important factors besides preference, such as event importance and potential gains led by the event. Therefore, it would be worthwhile to allow the system to optimise event planning based on multiple objectives. Also, preferences are represented as natural numbers between 0 and 9 in the current work, which limits the user's preference to 10 levels. This limitation can be addressed by introducing fuzzy logic to allow rich representation and inference on various personal preferences. In this case, flexible CSP [29] may help. In addition, the work needs to be further evaluated by more complex real world cases, as the current case study is mainly for demonstration and validation.

References

[1] PM Berry, T Donneau-Golencer, K Duong, M Gervasio, B Peintner, and N Yorke-Smith. Emma: An event management assistant. *ICAPS08 System Demos*, 2008.

[2] Melinda T Gervasio, Michael D Moffitt, Martha E Pollack, Joseph M Taylor, and Tomas E Uribe. Active preference learning for personalized calendar scheduling assistance. In *Proceedings of the 10th international conference on Intelligent user interfaces*, pages 90–97. ACM, 2005.

[3] Ioannis Refanidis and Neil Yorke-Smith. On scheduling events and tasks by an intelligent calendar assistant. In *Proceedings of the ICAPS Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems. MA Salido and R. Bartak Eds*, 2009.

[4] David E Joslin and David P Clements. Squeaky wheel optimization. *Journal of Artificial Intelligence Research*, pages 353–373, 1999.

[5] Peter Brucker and P Brucker. *Scheduling algorithms*, volume 3. Springer, 2007.

[6] Edward Ignall and Linus Schrage. Application of the branch and bound technique to some flow-shop scheduling problems. *Operations research*, 13(3):400–412, 1965.

[7] Erik Demeulemeester and Willy Herroelen. A branch-and-bound procedure for the multiple resource-constrained project scheduling problem. *Management science*, 38(12):1803–1818, 1992.

[8] G Terry Ross and Richard M Soland. A branch and bound algorithm for the generalized assignment problem. *Mathematical programming*, 8(1):91–103, 1975.

[9] Petr Somol, Pavel Pudil, and Josef Kittler. Fast branch & bound algorithms for optimal feature selection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(7):900–912, 2004.

[10] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[11] A. Kumar. A modified method for solving the unbalanced assignment problems. *Applied mathematics and computation*, 176(1):76–82, 2006.

[12] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19(2):248–264, 1972.

[13] G Ayorkor Mills-Tettey, Anthony Stentz, and M Bernardine Dias. The dynamic hungarian algorithm for the assignment problem with changing costs. 2007.

[14] John Holland. Adaptation in natural and artificial systems. *Ann Arbor, University of Michigan Press*, 1975.

[15] Carlos A Coello Coello and Efrén Mezura Montes. Constraint-handling in genetic algorithms through the use of dominance-based tournament selection. *Advanced Engineering Informatics*, 16(3):193–203, 2002.

[16] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[17] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, 1:69–93, 1991.

[18] Pui Wah Poon and Jonathan Neil Carter. Genetic algorithm crossover operators for ordering applications. *Computers & Operations Research*, 22(1):135–147, 1995.

[19] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[20] Kim-Fung Man, Kit-Sang Tang, and Sam Kwong. Genetic algorithms: concepts and applications. *IEEE Transactions on Industrial Electronics*, 43(5):519–534, 1996.

[21] Gabriela Ochoa, Inman Harvey, and Hilary Buxton. Optimal mutation rates and selection pressure in genetic algorithms. In *GECCO*, pages 315–322. Citeseer, 2000.

[22] John J Grefenstette. Optimization of control parameters for genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, 16(1):122–128, 1986.

[23] Di-Wei Huang and Jimmy Lin. Scaling populations of a genetic algorithm for job shop scheduling problems using mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 780–785. IEEE, 2010.

[24] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. *Lecture notes in computer science*, 1917:849–858, 2000.

[25] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary computation*, 8(2):173–195, 2000.

[26] Ian Miguel and Qiang Shen. Solution techniques for constraint satisfaction problems: Advanced approaches. *Artificial Intelligence Review*, 15(4):269–293, 2001.

[27] Ian Miguel and Qiang Shen. Solution techniques for constraint satisfaction problems: Foundations. *Artif. Intell. Rev.*, 15(4):243–267, June 2001.

[28] Brad L Miller and David E Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9(3):193–212, 1995.

[29] Ian Miguel and Qiang Shen. Fuzzy rrdfcsp and planning. *Artif. Intell.*, 148(1-2):11–52, August 2003.