A Review and Implementation Of Combinatorial Optimization Algorithms Applied to Workforce Scheduling

By Steven Campbell

2006
# Table of Contents

Abstract.......................................................................................................................3

Chapters
1. Introduction ................................................................................................................4
2. Literature Review .......................................................................................................5
3. Algorithm Details.......................................................................................................10
4. Requirements ............................................................................................................18
5. Software architecture ...............................................................................................25
6. Software design .........................................................................................................27
7. Experimental Environment ......................................................................................37
8. Experimental Results ...............................................................................................41

Bibliography ..................................................................................................................61
Appendix 1: Requirements .............................................................................................63
Appendix 2: Summary of Results for the Aykin Model .................................................67
Appendix 3: Summary of Results for the Bechtold Model ............................................71
Appendix 4: Summary of Results for the Manually Scheduled ..............................75
Appendix 5: Summary of Results for the Algorithm Determined Shifts ..............79
Appendix 6: Daily Schedule Graphs ...........................................................................80
Abstract

This paper looks at combinatorial optimization algorithms as applied to workforce scheduling in the call center industry. Specifically it will look at two integer programming algorithms, one by T. Aykin and another by S. Bechtold and L. Jacobs. Next will be a presentation of a C# based user application built to use each of these two algorithms. Through this program each of these algorithms will be applied to a set of test problems. The two algorithms are evaluated against each other to see which creates the best schedules. They are also contrasted with a manual scheduling process. The goal of a schedule is to place the right number of call center agents at the right times to answer the most telephone calls at the lowest cost. This approach maximizes call center profits.

The analysis of the results show that the schedules produced by the T. Aykin algorithm were the most profitable overall, followed by the schedules produced by the Bechtold and Jacobs algorithm. While the least profitable overall, the schedules produced by the manual process were superior when the calling demand was low.
Chapter 1
Introduction

In the modern service economy personnel costs are often the largest single cost of doing business. It is necessary to have the right number of workers needed to meet demand without over scheduling workers so that the company makes little or no profit. Thus using personnel efficiently is critical to the success of these organizations. In many cases, using personnel efficiently means being adept at workforce scheduling. A small percentage improvement in workforce scheduling can mean millions of dollars in cost savings to the organization which achieves such improvements. To meet this need businesses, governments, and academic institutions have invested tremendous resources into developing computer based scheduling solutions. The purpose of this paper is to investigate two of the linear optimization algorithms being used to address the modern workforce scheduling problem. In particular these algorithms use the implicit approach to address the scheduling of morning, lunch, and afternoon breaks, an aspect of scheduling often ignored by other algorithms. These algorithms use manually determined shift placement. The algorithms are run on an Ilog solver with a custom C# based interface which allows for easy use of these algorithms to create schedules. The data used comes from a La Junta Colorado call center. These schedules are then compared to each other and to manual schedules produced using the same data.

The comparison of the two algorithms to the manually produced schedules shows that the linear optimization algorithms do produce overall more profitable schedules than the manual process, given the same inputs, though not in every individual case. Specifically, the manual process shows a slight advantage when calling demand is low. Additional research may find a way to improve the performance of the algorithms under low demand conditions. In addition to the main experiment, two smaller experimental scenarios were conducted. The first looked at how increasing the revenue per call affected relative profitability of the schedules. It found that at high revenue per call, the manual process went from the least profitable to the most profitable. The second scenario let one of the algorithms determine its own shifts for a small number of schedules. The results were mixed, with better results on some schedules and worse results on others. Additional research into automatic shift placement might proof of value.
Chapter 2
Literature Review

2.1 Introduction to the History of Scheduling Optimization

The history of scheduling optimization algorithms goes back over 50 years. The idea is to use linear programming to find the most efficient schedules to minimize labor cost and meet demand. As problems have grown in size and complexity, different strategies have been devised to deal with this problem such as the branch and price approach and other set reduction techniques. Of particular interest was finding a way to include the scheduling of worker breaks, as well as the shifts themselves, without overloading the systems the algorithms had to run on. This chapter will talk about examples of some of these approaches, include branch and bound, branch and price, constraint reduction, and the implicit approach.

2.2 The Application of Mathematics to Scheduling Optimization

The original application of mathematics to scheduling optimization was in 1954 by L.C. Edie in his ground breaking article *Traffic Delays at Toll Booths* (Edie 1954). In this article Edie proposed using probability tools to better anticipate the demand, so that the supervisors making the schedule would have a more accurate view of the demand they were scheduling to meet. This work inspired a key insight by George Dantzig, that Linear Programming could be used to determine the correct number of workers needed to meet traffic demand, as laid out in his 1954 paper *A Comment on Edie’s ‘Traffic Delays at Toll Booths’* (Dantzig 1954). In that paper he proposed a set covering formula that has formed the basis of much of the subsequent approaches to workforce scheduling.

\[
\begin{align*}
\text{Minimize } & \sum_{j \in J} c_j X_j \\
\text{Subject to } & \sum_{j \in J} a_{jt} X_j \geq b_t \quad \text{for all } t \in T, \\
& X_j \geq 0 \text{ and integer.}
\end{align*}
\]

In this formulation $J$ is the set of all shifts, $T$ is the set of planning periods that the schedule must cover, $b_t$ is the number of employees needed in period $t$, $c_j$ is the cost of assigning an employee to a shift $j$, $a_{jt}$ is equal to one if period $t$ is a work period for shift $j$ and zero otherwise, and $X_j$ is an integer variable defined as the number of employees assigned to shift $j$, $j \in J$. (Dantzig 1954)

This is a basic set covering algorithm. It is recognized as being the basis for much of the subsequent research into the application of linear programming to improve
workforce scheduling. However, it has been found to be very susceptible to the size of the set. There is a serious issue in dealing with problems of size and complexity as the set becomes increasingly large and unwieldy. Since Dantzig’s initial breakthrough other researchers have sought ways to reduce the size of the sets.

2.3 The Branch and Price Method
One of the ways to reduce the size of the data sets is to use the Branch and Price method of column generation as discussed by Cynthia Barnhart et al. in the 1998 paper *Branch and Price Column Generation for Solving Huge Integer Programs* (Barnhart 1998). This paper recognizes that any scheduling problem which seeks to deal with real world constraints tends to be so large and complex as to be unsolvable, at least in a practical time frame. This paper looks at the branch and price approach as way to discern optimal solutions without looking at every possible combination. By reducing the size of the problem it becomes much more feasible.

2.4 Branch and Bound
In most linear programming problems, many of the constraints do not affect the final solution, and thus need not be considered in the calculation. In straight linear programming it is unknown which constraints are relevant and which are not until the solution is found. This means that a great deal of computational complexity and time was wasted on unimportant constraints. Branch and bound seeks to determine which constraints are unnecessary without completing the entire calculation, thus limiting the overall size of the problem. Branch and cut is a generalization of the branch and bound process. It looks at the solution space and focuses on the outer limits of that space because it is known that the optimal solution is to be found at the edge. As Barnhart et. al. demonstrate in their paper, certain classes of these constraining inequalities are left out, thus simplifying the problem (Barnhart 1998). This relaxed linear program is then solved. If it turns out that the relaxed solution is infeasible, a subprogram, called a separation problem, is created and solved looking for the violated inequalities in a class. If such violated inequalities are found, some constraints are added to the linear program to cut off the infeasible solution. The linear program is then reoptimized. If there are no violated inequalities, a branch occurs. In branch and price, the process is similar; but instead of focusing on row generation as branch and cut does, it focuses on column generation. It leaves out sets of columns to simplify the problem.

2.5 Reducing the Number of Constraints
Another approach to the problem of unwieldy size is discussed in the 2004 paper *Multiskilled Workforce Optimization* by Guy Eitzen et. al. (Eitzen 2004). This paper uses the conventional set covering method to find the optimum workforce schedule. It investigates three means of dealing with the large number of constraints. The first is the column expansion method which reduces the size of the problem by focusing on only those tours that do not include multiskilling, instead looking at those schedules for workers that will only involve their core skill level. The second approach is to randomly choose a subset of 100 tours for each employee and optimize those, reasoning that if the quality of the tours is randomly distributed, that a random subset has a good chance of containing a superior, if
not optimal schedule. Thirdly, they use the more traditional branch and bound which does not arbitrarily limit the tours being considered. They conclude that this last method produces superior schedules, but takes much longer to complete.

2.6 Including Breaks In Scheduling

While these papers seek to deal with the issue of increasing problem size, they leave out the issue of worker breaks. Ordinarily, including breaks as part of the shift would greatly increase the problem size and complexity, but in the 1999 paper *Optimal Shift Scheduling: A Branch-and-Price Approach* by Anuj Mehrotra et. al. includes the issue of worker breaks and works to deal with the large number of possible shifts by using branch and price rules to reduce the size of the problem (Mehrotra 2000). In particular they work to develop branching rules that reduce the size of the problem without making the resulting sub problems overly complex. Thus they hope to have maintained the flexibility of the explicit set covering model, but are able to produce solutions at much greater speeds.

2.7 The Implicit Approach

In the paper Mehrotra compares his method to the implicit formulation, which is another linear programming technique that handles breaks. Here is how one of the pioneers of the implicit formulation describes it: The implicit approach includes breaks without overly enlarging the problem size by “….associating break variables with planning periods as opposed to shifts. A particular break variable represents the total number of employees starting a break in its associated planning period. Thus, break assignments are not made until the optimal solution to the implicit mode has been obtained.” (Bechtold 1990). This is from Bechtold and Jacobs 1990 *Implicit Modeling of Flexible Break Assignments in Optimal Shift Scheduling* (Bechtold 1990). This paper recognizes that the inclusion of break placement in shift scheduling dramatically increases the number of shifts to be considered. Here again, the problem is deemed too large for the traditional set covering model. As an alternative this paper looks at the implicit model, as it is believed to achieve optimal result, but uses much less information to achieve them.

In 1996 they were joined by Turgut Aykin who formulated a different but related approach in his paper *Optimal Shift Scheduling With Multiple Break Windows* (Aykin 1996). He achieves the same ends as Bechtold, but with a different linear programming algorithm. Here again, they are creating an algorithm which includes break schedules, but is more efficient. They state that their formulation requires, “…..substantially smaller number of variables than the equitant set covering model in the general shift scheduling problem involving multiple breaks and break windows.”(Aykin 1996, p. 593)

Another paper that uses the implicit approach to deal with the complexity of including breaks in the scheduling process is the 2004 paper *Scheduling a Hierarchical Workforce with Flexible Break Assignments and Variable Demands* by Serap Seckiner et. al. (Seckiner 2004). This paper uses traditional integer programming to optimize the staff schedules. What makes this paper different is that they propose a new constraint on the integer model, the use of hierarchical staff skill levels. This means that higher skilled workers may substitute for lower skilled workers, but a lower skilled worker may not stand in for a higher skilled worker. The authors then use the implicit method from Aykin to handle break placement.
The implicit approach is also used in Irem Ozkarahan et. al.’s paper *Optimization Based Modeling for Tour Scheduling with Flexible Break Windows* (Ozkarahan 1998). This paper looks at a way to perform integer programming with fewer variables than are required by the traditional set covering approach. As previously discussed, such an approach may include millions of different possible shifts. To simplify the problem, this paper looks at the Bechtold implicit approach. This paper goes on to apply this approach to the tour scheduling problem, dealing with the days of the week that are worked as well as the hours worked on a specific day.

Another paper that uses the implicit approach is *Optimal Models for Meal Break and Start Time Flexibility in Continuous Tour Scheduling* by Michal Brusco and Larry Jacobs from 2000 (Brusco 2000). This paper looks at continuous scheduling problems. These are so named because there are people working 24 hours a day, 7 days a week. This paper also simplifies the problem using the implicit model. This paper applies the implicit method from the Bechtold paper to assigning tour schedules, that is the days of the week that the agent works, as well as the hours they work on a particular day. Because it deals in continuous schedules, this paper must also deal with the situation of agent shifts that overlap from one day to the next.

*An Implicit Tour Scheduling Model with Applications in Healthcare* 2004 by Mark Isken uses the implicit approach as well (Isken 2004). This paper looks at the traditional tour scheduling model that simultaneously determines the days to be worked and what hours are to be worked on those days, seeking to minimize total labor costs. In this paper they include both overlapping start times and full and part time tour types. They don’t look at break scheduling, preferring to add that in after the day and hour schedules are completed. This is a particularly valid choice for the healthcare industry, as breaks are often taken as time permits, rather than at a predetermined time. They do mention though that the Bechtold or Aykin approaches could be integrated into their formulation.

### 2.8 Summary and Application

To summarize, this section has touched on a few of the ways that have been employed to automate workforce scheduling, and in particular ways to keep the size of scheduling problems manageable while increasing their complexity. Of special interest is the implicit approach, which has shown promise as a way to include break scheduling without making the problems too large to solve.

While many of the papers employ the implicit approach most have cited one or both of the Bechtold and Aykin formulations. They were formulated some years ago, but the fact that they are still being used and expanded upon points to their continued value to optimizing scheduling problems. As such they are reasonable candidates to be used in testing the proposition that it is possible that linear optimization techniques can create better break schedules than an alternative manual process. Since these two algorithms seek to solve the same problem in different ways, they are good candidates for comparison. In addition, the problem looked at was a reasonably simple one, so it made sense to employ a relatively simple solution, one that is valid, but not overly complex. Thus the Bechtold and Aykin algorithms were the ones chosen to be implemented and compared, both to one another and to the manual process.
2.9 Call Center Applications

The specific problem that the algorithms will be applied to is that of the telephone call center industry. Call centers handle many different kinds of calls, from tech support, to sales, to customer service. Some call centers are dedicated to a particular task like a single company’s customer service, while others handle a mixture of calls for several different clients. What all these call centers have in common is the need to efficiently schedule their workforce. It has been estimated that personnel cost account for 60 to 70% of the total costs of operating a call center (Telephone Call Centers: Tutorial, Review and Research Prospects)2003. With percentages like these the call centers have a great incentive to use their employees as efficiently as possible.

A call center must have enough agents to answer the expected volume of calls, but not too many agents as there won’t be enough calls for them to answer. This is where the scheduling algorithm can help. By applying linear programming algorithms to the scheduling problem it is hoped that such schedules will be more efficient than those that are created by hand. In this paper the two algorithms will be applied to the problem of scheduling agents for a call center. This is a suitable subject as call centers typically have a given set of shifts that need to be populated, have a set number of breaks in a shift and a limited range of when those breaks can be taken. Further, call centers usually have a good idea of what the calling demand is going to be on a given day broken down into discrete periods. All of these factors are exactly the inputs that the Bechtold and Aykin algorithms are designed to accept. Likewise their outputs, the specific number of workers assigned to a shift, the specific break times for a shift, and the number of workers taking a given break are just the information the call center needs for creating its schedules. Thus the call center application is an appropriate test of these two schedules against the standard manually generated schedule and these two algorithms are comparable to one another.
Chapter 3
Algorithm Details

3.1 Introduction
Traditionally a set covering linear programming approach is used for scheduling problems (Bechtold 1990). This consists of sets of shifts with the different shifts having different start and end times. If breaks are introduced, each shift with the same start and end, but different break placement, would be a different shift. Thus a set of shifts with two possible breaks periods would double the number of possible shifts in the set to be covered. Clearly the number of shifts in the set quickly grows unmanageably large as more possible break periods are introduced. For this reason, breaks are commonly left out of set covering models. However, the strategic placement of breaks can greatly improve the quality of the schedule (Bechtold 1990). Thus the challenge was to include break placement, yet keep the model size within workable limits.

One promising way is to represent the break placement implicitly. By implicit, it is meant that the breaks are not directly associated with a particular shift. Rather a particular shift, denoted by its start and end points may have any of a number of breaks. The breaks are denoted by their start period. Thus for a particular shift the algorithm must look at a number of breaks for the shift. In practical terms this means that the set of shifts and the set(s) of breaks that take place during a shift are not directly linked. Thus you may have a set 100 shifts and a set of 100 breaks for a total of 200 variables. Under the older set covering scheme you would have 10,000 variables, one for each shift/break combination. From a model size perspective, the implicit formulation is clearly superior (Bechtold 1990).

3.2 The Algorithms
Many different researchers have formulated algorithms to take advantage of the implicit approach. This paper will look at two of them. The first was formulated by Stephen Bechtold and Larry Jacobs. Their approach is to:

“….define shift variables for each possible combination of shift starting time, shift length, and shift window, and break variables for each period during which a break can take place. The matching between shifts and admissible breaks is done through a set of forward and backward constraints. This set of constraints ensures an eligible break for each shift in the associated break window without specifying its actual starting time. Complete shift schedules with appropriate break assignments are constructed a posteriori using a break allocation algorithm. The allocation procedure arranges shifts in a non decreasing order with respect to the latest period in which a break may occur and assigns shifts to the earliest available breaks.” (Rekik, Monica 2005).

The second algorithm is by Turgut Aykin. He takes a slightly different approach within the implicit framework. In his formulation, the placement of relief and lunch breaks are represented by considering a break variable for each shift and each possible starting
time within its break window. A shift is defined as the combination of starting time, a length, the break type it contains and the associated break windows. Equality constraints are used to match each type of shift with the associated type of break and break window. (Rekik, Monica 2005).

Both of these formulations use the following basic terms:
1. Planning Period: the smallest time interval for which labor requirements have been established. In this paper it is 15 minutes.
2. Shift: the detailed specification of the planning periods to be allocated to work and breaks respectively.
3. Work span: the contiguous set of planning periods within which either work or rest must take place for a given shift.
4. Shift Length: The total number of planning periods assigned for breaks as well as work for a given shift. In this paper, all shifts are 8.5 hours long.
5. Shift Type: The set of contiguous planning periods within which a break may begin for one or more work spans.
6. Break Window: The set of contiguous planning periods within which a break may begin for one or more work spans.

A shift type is created by the association of one or more break windows with a specific work span.

The implicit model is based on the following assumptions:
1. The organization operates no more than 24 hours a day.
2. All planning periods are of equal length.
3. The duration of breaks is identical for all shift types modeled.
4. The duration of breaks is one or more planning periods.
5. Each shift type is defined by the assignment of a single break window to an associated work span.
6. The break window for each shift type consists of any selected non-zero subset of periods subject to the restriction that all periods contained within the break interval must be a subset of the work span associated with the shift type.
7. Extraordinary Overlap (EO) does not exist. EO occurs when the start and end time of a break window of a shift are wholly within the start and end time of a break window of another shift.
8. No understaffing is allowed.

(Bechtold 1990)

3.3 The Aykin Formulation

Aykin formulates the general shift scheduling problem as follows:

Minimize \( \sum_{k \in K} c_k X_k \)  

Minimize the total cost of assigning employees to shifts.
Subject to
\[
\sum_{k \in K} c_k x_k - \sum_{k \in L_1} U_{kt} - \sum_{k \in L_2} W_{kt}(-1)
\]
\[
- \sum_{k \in L_1} W_{kt} - \sum_{k \in L_2} V_{kt} \geq b_t \quad \text{for all } t \in T.
\]
For a given period in a working shift, assign it enough workers to meet demand, while accounting for those that are on either their first break, lunch break, or afternoon break.

\[ X_k - \sum_{t \in T_k} U_{kt} = 0 \quad \text{for all } k \in K, \]
All workers must take one first break.

\[ X_k - \sum_{t \in T_k} W_{kt} = 0 \quad \text{for all } k \in K. \]
All workers must take one lunch break.

\[ X_k - \sum_{t \in T_k} V_{kt} = 0 \quad \text{for all } k \in K. \]
All workers must take one second break.

\[ X_k, U_{kt}, W_{kt}, V_{kt} \geq 0 \quad \text{and are integers.} \]
There can be no partial workers or partial breaks.

Definitions:
\[ K = \text{the set of all shifts.} \]
\[ k = \text{a single shift from the set of shifts.} \]
\[ T = \text{the set of planning periods covered by the shift schedule.} \]
\[ t = \text{a single planning period from the set of planning periods.} \]
\[ c_k = \text{the cost of assigning an employee to a shift } k. \]
\[ X_k = \text{the number of employees assigned to shift } k. \]
\[ b_t = \text{the number of employees needed in a period } t \text{ to achieve the desired service level.} \]
\[ a_{kt} = \text{an integer variable that equals one if the period } t \text{ is a work period for shift } k \text{ and zero otherwise.} \]
\[ U_{kt} = \text{an integer variable representing the number of employees assigned to a shift } k \text{ and taking their first relief break in period } t. \]
\[ W_{kt} = \text{an integer variable representing the number of employees assigned to a shift } k \text{ and taking their lunch break in period } t. \]
\[ V_{kt} = \text{an integer variable representing the number of employees assigned to a shift } k \text{ and taking their second relief break in period } t. \]
\[ B_{1k} = \text{the set of planning periods forming the first relief break window of shift } k. \]
\[ B_{Lk} = \text{the set of planning periods forming the lunch break window of shift } k. \]
\[ B_{2k} = \text{the set of planning periods forming the second relief break window of shift } k. \]
\[ T_{1t} = \text{the set of shifts for which period } t \text{ is a break start time within the break window for the first relief break.} \]
\[ T_{Lt} = \text{the set of shifts for which period } t \text{ is a break start time within the break window for the lunch break.} \]
\( T_2_t \) = the set of shifts for which period \( t \) is a break start time within the break window for the second relief break.

The first relief break variable \( U_{kt} \) is defined only for \( t \in B_1 \), the lunch break variable \( W_{kt} \) is defined only for \( t \in B_L \), and the second relief break variable \( V_{kt} \) is defined only for \( t \in B_L \). (Aykin 1996)

### 3.4 The Bechtold and Jacobs formulation

The basic algorithm is as follows:

Minimize \[ \sum_{k \in K} c_k X_k \] Minimize the total cost of assigning employees to shifts.

Subject to \[ \sum_{k \in k_t} a_{kt} X_k - \sum_{j \in TU} u_{jt} U_j - \sum_{j \in TW} \omega_{jt} W_j - \sum_{j \in TV} v_{jt} V_j \geq b_t \]

for all \( t \in T \),

Ensures that for a given period in a working shift, it is assigned enough workers to meet demand, while accounting for those that are on either their first break, lunch break, or afternoon break.

Where

\( X_k, U_t, W_j, V_j \geq 0 \) and are integers, and

\( c_k \) = the cost of assigning an employee to a shift \( k \), \( k \in K \)

\( K \) = the set shifts including all feasible combinations of different shift types and start times.

\( a_{kt} \) = this is equal to one if the period \( t \) is within the workspan (a work or break period) of shift \( k \), and zero otherwise.

\( b_t \) = the labor requirement in period \( t \)

\( t \) = the earlier

\( u_{jt} \) = 1 if the period \( t \) is a first relief break period for the employees starting their breaks in period \( j \), \( j \in TU \)

\( \omega_{jt} \) = 1 if the period \( t \) is a lunch break period for the employees starting their breaks in period \( j \), \( j \in TW \)

\( v_{jt} \) = 1 if the period \( t \) is a second relief break period for the employees starting their breaks in period \( j \), \( j \in TV \)
\( X \) = the number of employees assigned to shift \( k \)

The terms \( U, W, V, TU, TW, \) and \( TV \) are part of the break scheduling constraints, as follows:

First Break Constraints:

\[
\sum_{i \in \mathcal{U}_j} U_i - \sum_{i \in \mathcal{U}_1} X_i \geq 0
\]

Ensures that no first break is scheduled after the end of the first break window.

\[
\text{for all } j \in \mathcal{N}, \quad \sum_{i \in \mathcal{U}_j} U_i - \sum_{i \in \mathcal{U}_2} X_i \geq 0
\]

Ensures that no first break is scheduled before the start of the first break window.

\[
\text{for all } j \in \mathcal{M}, \quad \sum_{i \in \mathcal{U}_k} X_i = \sum_{i \in \mathcal{U}} U_i
\]

Ensures that all employees get a first break.

Definitions:

\( pu \) = the earliest first relief break start time for the shifts in \( KU \).
\( qu \) = the latest first relief break start time for the shifts in \( KU \).
\( TU \) = \( \{pu, ..., qu\} \), the set of all possible first relief break start times for the shifts in \( KU \).
\( SU_j \) = \( \{pu, ..., j\} \), the set of all first relief break start times between periods \( pu \) and \( j \).
\( ZU_j \) = \( \{j, ..., qu\} \), the set of all first relief break start times between periods \( j \) and \( qu \).
\( NU \) = the set of latest first relief break start times (in ascending order) for the shifts in \( KU \).
\( MU \) = the set of earliest first relief break start times (in ascending order) for the shifts in \( KU \).
\( TU_{1j} \) = the set of shifts in \( KU \) with a first relief break window lying completely between periods \( pu \) and \( j \).
\( TU_{2j} \) = the set of shifts in \( KU \) with a first relief break window lying completely between periods \( j \) and \( qu \).

The constraints for the lunch break are similar to the rest breaks. The only difference is that lunches take two planning periods, not just one period like the rest breaks. This must be reflected in the length of the break window assigned in the data set input into the algorithm.
Lunch Break Constraints:

\[ \sum_{i \in SW_j} W_i - \sum_{k \in TW_{1j}} X_k \geq 0 \]
Ensures that no lunch is scheduled after the end of the lunch window.

\[ \sum_{i \in SW_j} W_i - \sum_{k \in TW_{2j}} X_k \geq 0 \]
Ensures that no lunch is scheduled before the start of the lunch window.

\[ \sum_{k \in \text{all}} X_k = \sum_{i \in \text{all}} W_i \]
Ensures that all employees get a lunch.

Definitions:

\( \text{pw} \) = the earliest lunch break start time for the shifts in KW.
\( \text{qw} \) = the latest lunch break start time for the shifts in KW.
\( \text{TW} \) = \{\text{pw},...,\text{qw}\}, the set of all possible lunch break start times for the shifts in KW.
\( \text{SW}_j \) = \{\text{pw},...,\text{j}\}, the set of all lunch break start times between periods \text{pw} and \text{j}.
\( \text{ZW}_j \) = \{\text{j},...,\text{qw}\}, the set of all lunch break start times between periods \text{j} and \text{qw}.
\( \text{NU} \) = the set of latest lunch break start times (in ascending order) for the shifts in KW.
\( \text{MU} \) = the set of earliest lunch break start times (in ascending order) for the shifts in KW.
\( \text{TW}_{1j} \) = the set of shifts in KW with a lunch break window lying completely between periods \text{pw} and \text{j}.
\( \text{TW}_{2j} \) = the set of shifts in KW with a lunch break window lying completely between periods \text{j} and \text{qw}.

Finally, the constraints for the second break:

\[ \sum_{i \in ZV_j} V_i - \sum_{k \in TV_{1j}} X_k \geq 0 \]
Ensures that no second break is scheduled after the end of the second break window.

\[ \sum_{i \in ZV_j} V_i - \sum_{k \in TV_{2j}} X_k \geq 0 \]
Ensures that no second break is scheduled before the start of the second break window.
for all \( j \in \text{MV} - \{pv\}, \)

\[
\sum_{k \in k} X_k = \sum_{i \in \text{V}} V_i
\]

Ensures that all employees get a second break.

Definitions:

- \( pv \) = the earliest second relief break start time for the shifts in \( KV \).
- \( qv \) = the latest second relief break start time for the shifts in \( KV \).
- \( TV = \{pv, \ldots, qv\}, \) the set of all possible second relief break start times for the shifts in \( KV \).
- \( \text{SV}_j = \{pv, \ldots, j\}, \) the set of all second relief break start times between periods \( pv \) and \( j \).
- \( \text{ZV}_j = \{j, \ldots, qv\}, \) the set of all second relief break start times between periods \( j \) and \( qv \).
- \( \text{NU} \) = the set of latest second relief break start times (in ascending order) for the shifts in \( KV \).
- \( \text{MU} \) = the set of earliest second relief break start times (in ascending order) for the shifts in \( KV \).
- \( \text{TV}_1_j \) = the set of shifts in \( KV \) with a second relief break window lying completely between periods \( pw \) and \( j \).
- \( \text{TV}_2_j \) = the set of shifts in \( KV \) with a second relief break window lying completely between periods \( j \) and \( qv \).

(Bechtold 1990)

With this formulation, these constraints ensure that the number of employees assigned to various shifts and available in period \( t \) minus the number of employees taking their first relief, lunch, and second relief breaks is sufficient to meet demand and provide the desired level of service in that period.

### 3.5 Discussion of the Differences Between the Equations

In respect to the objective functions, the two equations are identical. That is to say, they have the same purpose, which is to assign agents with the minimum cost. Like wise the primary constraints, under the heading ‘Subject To’ are also similar, as both algorithms need to meet the demand for agents while accounting for agent breaks. The big difference between the two approaches is how the break times are matched with the shifts.

Bechtold uses break variables associated with each planning period and then associates those periods with the shift implicitly. The output of the algorithm just indicates how many agents are on break during a given planning period. It is necessary to go back and match up the breaks with a given shift based on when a given shift had a break window. It is even necessary to split up the agents taking a break in a given planning period, as some are from one shift and the rest are from another.

Aykin, on the other hand, defines separate break variables for each shift and then associates the breaks with the shift implicitly. Aykin’s output then differentiates the breaks
in its output, with the number of agents on break in the first position in the break window being in the first entry in the output, the agents in the second position being the next entry, and so on.

3.6 Software Used

Good linear solver software is sophisticated and complex. Fortunately it is also in high demand, a demand that has been met by a variety of software packages. This has been a great boon to researchers, as they can now focus on developing and testing algorithms rather than the spending their time creating the means to test algorithms. For example, Turgut Aykin in his paper “Shift Scheduling with Multiple Break Windows” uses the LINDO linear solver from LINDO Systems, Inc. (Aykin 1996). Other commercially available solvers are AMPL created by Bell Labs and CPLEX from the French company Ilog. There are also free solvers such as lp_solve which was created by Michel Berkelaar at the Eindhoven University of Technology in the Netherlands. This paper uses the CPLEX solver because of its easy to use interface, its English like modeling language, and the ease with which it could be called from another program. Perhaps most important it offered an economical academic license for the use of its product in educational research applications.

CPLEX Ilog solver uses the simplex method to solve this integer programming algorithm. The simplex method tests, “adjacent vertices of the feasible set in sequence so that at each new vertex the objective function improves or is unchanged.” (Carreira-Perpiñán 2006). In most cases it is quite efficient, but in the worst case its complexity becomes exponential (Carreira-Perpiñán 2006).
Chapter 4
Software Requirements Specification

4.1 Background

This document is the result of an analysis of the workforce scheduling needs of Holden Marketing Support Services. They presently schedule their workforce to meet calling demand manually, including the scheduling of rest and lunch breaks during the shifts. It was proposed that an automated system using an optimization algorithm could do a faster and better job of scheduling worker breaks. By better, it is meant that the number of workers available to take calls at any given period in the day, more closely matches the number workers forecasted to be needed during that period. It is outside the scope of this program to improve upon the demand forecasts. It is only concerned with meeting those forecasts as closely as possible. Further the program is not concerned with rostering (the process of assigning actual workers to the given schedules). However, a file containing the schedules is output to assist in the rostering process.

4.2 User Needs

The user needs to create schedules for one week at a time. A week consists of seven days, Sunday through Saturday. A day is a 24 hour period, from midnight to midnight. All days are broken down in to 15 minute periods, starting at midnight, with 96 periods per day.

Each day will contain up to 8 individual calling groups. Each calling group can contain between 1 and 100 individual shifts. A shift is indicated by its start and stop time. All shift start times will coincide with the start of a 15 minute period e.g. 12:00, 12:15, 12:30, etc. All shifts will be for a continuous 8.5 hour block of time. Shifts must start and end within the same 24 hour one day period. Thus shifts may start any time between midnight and 3:30 pm.

The system needs to have a way for the user to input and edit the shifts for a group in a user friendly interface. Once this data is input, the user needs to be able to save it to a file for later use.

Demand data will come in the form of a text file from an external source. The user must be able to find and import the necessary data files.

Once the system has solved for the optimal schedule, it needs to be saved to a file in excel format for additional manual processing, namely rostering.

---

1 Holden Direct marketing manages two Customer Contact Management Centers, one in Denver, the other in LaJunta CO, with over 400 seats, providing a 24/7/365 capability. Redundant T-3s, overflow capacity, and disaster recovery are all built into their call center model. They specialize in programs that are complex and ongoing, including web-chat, email response management, marketing programs management, and inbound/outbound call center.

http://www.holdendirect.com/sub_overview.htm
4.3 System Features

4.3.1 Process to Input the shift schedules

Description and Priority:
The user must be able to input the shifts into the system and save these shifts for later processing. This function is critical.

Functional Requirements:
1. The system shall record what day the shift is to take place.
2. The user shall have the choice of starting with a blank schedule or opening a file with an existing schedule that can be modified.
3. The system shall allow seven days of shifts to be recorded.
4. The system shall require the days to be the days of the week.
5. The system shall record what calling group the shift belongs in.
6. The system shall record the start time and the end time of the shift.
7. The system shall validate the shifts to ensure that all shifts are 8.5 hours long.
8. The system shall validate the shifts to ensure that all shifts begin and end within the same 24 hour period (midnight to midnight).
9. The system shall require exactly 4 separate calling groups.
10. The system shall require that there be at least one shift in each calling group.
11. The user may save the input schedule to a file at any time.

4.3.2 Process to Import Shift Schedules

Description and Priority:
This program process shall take shift schedule data from a file and prepares it for processing. This feature is critical.

Functional Requirements:
1. The process shall allow users to select the shift file by name.
2. The process shall verify that there is shift data for exactly eight calling groups present in the file.
3. The process shall verify that there is seven days worth of demand data in the file.
4. The demand data may be imported at any time the program is running, except when another process is running.

4.3.3 Process to Import Worker Demand

Description and Priority:
This program process shall take group worker demand data from a file and prepare it for processing. This feature is critical.

Functional Requirements:
1. The process shall allow users to select the demand file by name.
2. The process shall verify that there is demand data for exactly eight calling groups present in the file.
3. The Process shall verify that there are seven days worth of demand data in the file.
4. The process shall verify that each of the 96 daily periods is accounted for in the file.
5. The process must verify that all demand data is greater or equal to zero. No negative demand data is allowed.
6. The Demand data may be imported at any time the program is running, except when another process is running.

4.3.4 Process to create break schedules for the input shifts

Description and Priority:
This program process shall take the demand data and the shift data and determine what period breaks should take place and how many workers should be taking their break during a particular break period. This feature is critical.

Functional Requirements:
1. This process is only available after the user has input/imported and saved the shift schedule and imported the worker demand file.
2. If the user has not done these things, the system indicates that it cannot create break schedules until they are completed.
3. The process will allow the user to indicate the name of the output file, and its location.
4. When finished the process indicates that it is complete.
4.3.5 Process to determine number of workers on a shift

Description and Priority:
The program schedules how many workers should be in each shift. Feature is critical.

Functional Requirements:
The process will use linear optimization to determine the number of people who should work a particular shift. The goal is to have at least as many people working during a shift period as needed to meet worker demand for that period.

4.3.6 Process to Schedule Breaks

Description and Priority:
The program determines when breaks should occur on a shift, and how many of the scheduled workers should take their breaks on that shift. Feature is critical.

Functional Requirements:
1. The process will use linear optimization to determine when workers on a shift should take their breaks, and the number of workers who should take their break during a given period.
2. The process must determine the number of workers for three separate breaks: two rest breaks and a meal break.
3. The first rest break must be taken no sooner than the 7th period of the shift and no later than the 12th period of the shift. This break shall last one period.
4. The meal break must start not sooner than the 15th period of the shift, and must start not later than the 19th period in the shift. This break shall last two periods.
5. The first rest break must be taken no sooner than the 25th period of the shift and no later than the 30th period of the shift. This break shall last one period.

4.3.7 Process to Shift/break schedule output file

Description and Priority:
This takes the created schedules and outputs them to a file. Feature is critical.

Functional Requirements:
1. The output file shall consist of an Excel file.
2. The output file shall be output to the C:\BreakSchedule\Output directory.

3. The report file shall be named BreakSchedMMDDYY.

4. Each excel file shall contain one work book. This work book will consist of 7 worksheets, one for each day of the week. They will be labeled mm/dd/yy.

5. Each worksheet will contain all the shifts for all the groups for that sheet’s day and the number of people on that shift. It will also include: the times for the breaks for each shift and the number of people on break at that time.

6. Example of the Excel layout is in the Requirement Appendix.

4.3.8 Display Internal Error

Description and Priority:

General system error, not handled by other error messages. This feature is important.

Functional Requirements:

If the system is unable to complete a task due to a problem not related to user input, it should provide the user with a message indicating that this is the case.

4.4 Environmental Requirements

The program will run on any PC running any Microsoft Windows, version 2000 or later. The PC can be networked or stand alone. It is recommended that the user have a text editor to create/modify the input files and to print the output files, if desired.

4.5 Interfaces

4.5.1 User Interfaces

The user Interface is left to the discretion of the programmer.

4.5.2 Hardware Interfaces

All hardware interfaces issues are handled by the Operating System.
4.5.3 Software Interfaces

1. The software interacts with two external programs, Ilog Cplex and Microsoft Excel. BuildSchedule is the program doing the calling, so it must be able to create instances of these programs and pass data to them.

2. These external programs take data in integer and string form, so the main program must be able to pass these data types and to receive these data types.

3. The chosen language for this project is C#, which is supported by both Ilog Cplex and Microsoft Excel.

4. This program will not be called by any program so it does not need an external interface of its own.

4.5.4 Input file Specifications

The system requires one type of input file, the workgroup demand file. This is a file containing group demand information, grouped by day of the week and the eight calling groups for that day. For each group there are a series of 96 numbers indicating the number of workers needed for each of the 96 periods in a day.

The first line of the file is the word “Sunday”, to indicate that the following groups’ demand occurs on Sunday.

The next line is “group1”, to indicate that the following demand information is for group 1.

The third line consists of 96 numbers, separated by a tab.

Example:
Sunday
Group1
1 12 0 0 0 1 1 0 0 0 1 0 0
0 0 0 0 0 1 0 0 1 2 2
1 4 2 7 6 3 11 6 7 9 6
11 4 8 10 13 8 15 10 8 4 6 8
8 10 11 9 12 6 10 9 7 9 5
18 10 3 8 6 3 8 5 6 7 11 5
6 3 5 9 2 7 2 2 3 3 5 4
1 5 0 4 0 7 4 3 2 4 2 2
1

Every period must be accounted for. If there are no workers needed in a period, the period should contain a zero.

The next line is group 2, followed by its worker demands.
This continues until eight group worker demands are in the file; then the next line is Monday, followed by its eight groups and demand. This pattern repeats for each day of the week, ending with the eight Saturday groups.

A more complete example is located in Appendix 4.

4.6 Other Nonfunctional Requirements

4.6.1 Performance Requirements

The system is expected to complete its scheduling in under 3 minutes, unless an error is detected.

4.6.2 Security Requirements

None. All security issues must be handled by the user. The program makes no provisions for securing data.

4.6.3 Software Quality Attributes

Correctness
This is defined as all shifts having the minimum number of workers assigned that will still meet the demands of the work group.

Reliability
The code needs to be reliable in that it produces useable results every time, with little or no intervention by the user. The task this code is meant to automate is tedious and time consuming, so the program must be neither.

Robustness
The environment the code is designed to work under is a lenient one. The program may assume that the physical environment is one conducive to computing, and that the software environment (state of the operating system, etc) is ideal for the program’s needs.

4.7 Other Requirements

None.
Chapter 5
Software Architecture

Figure 1: Software Architecture

To meet the stated requirements a reasonably compact program was designed. It has a straight forward Object Oriented Architecture. Each of the various functions is encapsulated within their own classes and instantiated as needed.

There are 3 primary components as shown in Figure 1:

1. User Interface.
2. Model.
3. External solver.

The UI allows the user to input shift information, through its input grid. It also allows some manipulation of the grid by the user, such as adding a new line to the grid or deleting a current line. The UI also allows the user to call for the importation of a shift file and the importation of a demand file. Finally it allows the user to initiate the process to solve for the optimal break schedule. The code libraries which were used to create the grid
control are open source software, written by Davide Icardi, and available for free download at the website http://www.devage.com.

The UI is described as ‘calling for’ particular actions because these actions are actually performed by the second component of the program, the Model class. All actions having to do with processing the data are handled by the Model class, as called by the UI. The Model Class does not do the processing itself, but rather instantiates the utility classes needed to perform its functions. These functions consist of preparing the data files used by the solver, calling the solver, receiving back the solution, and outputting it to Microsoft Excel.

The third major piece is the external linear solver. It is the Cplex solver by the Ilog software company which is being used under an academic license. This program has no direct interface with C#. Rather it is executed as an external process. It reads the pre-positioned data and algorithm files and sends output to Standard Output. This output is then read by the program and passed to the output function. This function reformats the solver’s output and writes it out in Excel based output files.
Chapter 6
Software Design

6.1 Introduction

This program was designed using object orient principles. The language chosen was C#. In addition to C# being an OO language, it is also the language used by the author’s employer. It was felt that using C# for this project would afford an opportunity to gain proficiency in a language that will be used in future projects.

6.2 Modules

6.2.1 BreakSchedMain
This module is the initiation module. All it does is to instantiate the user interface as soon as the program is started.
It takes no input and produces no output.

6.2.2 frmMain
This is the main user interface for the program. It is where the user can input and edit the group shifts for the days of the week. It receives all user input, relaying these instructions to the processing modules.
This module takes user input to initiate the importation of saved schedules, as well as the command to import weekly workgroup demand. In addition it receives the user instruction to initiate the algorithm solver. Finally, this form is where the user can add new schedules to a group, remove old ones, or edit current ones. All inputs are made via drop down lists, so data validation is not necessary.
This module outputs the import shift information to the display grid for the user to see. It passes the user inputs to the Model class for further processing.

Figure 2, GUI.
Internal Functions:

1. Initialize Components
   This function is a form designer generated function that creates and initializes the graphical interface of the module. It takes no arguments and returns nothing.

2. frmGrid1_Load
   This function creates the user input/display grid. It is called by the initialization function. While it takes event arguments, this is a formality as it does not actually need to handle any events after its initial instantiation. What this function does currently is to load the default schedule file. This could be changed so that it loads blank, but allows the user to choose a file and load it if they like.

3. Selection_CellLostFocus
   This function traps the user event of moving their mouse out of a cell. It then reads the cell for the shift start time and updates the adjacent cell with the correct shift end time. The input parameter is the event object and it has no return parameters.

4. brAddRow_Click
   This function handles the click event for the ‘Add Row’ button. It inserts a new row into the input grid, allowing users to add additional shifts to a workgroup. While it accepts the click event it does not use it. It knows where to insert the row by reading the selected row attribute from the grid object. It returns nothing.

5. RemoveRow_Click
   This function handles the click event for the Remove Row button. It takes the even arguments, but does not use them. It determines which row to delete by reading the selected row attribute from the grid object. It returns nothing.

6. Solve_Click: This function handles the click event from the solve schedule button. It accepts the even arguments, but does not use them. It returns nothing. It calls the solver function from the model class.

7. SaveShifts_Click
   This function handles the click event to call the GridToFile function from the model class. It does not use the event arguments and returns nothing.

8. ImportShifts_Click
   This function handles the click event from the import shifts button. It calls the import shift function from the model class. It does not use the event arguments; it is passed and returns nothing.
9. **ImportDemand_Click:**
   This function handles the click event from the import demand button. It calls the import demand function from the model class. It does not use the event arguments; it is passed and returns nothing.

### 6.2.3 Model

This function is the main data processing function. It holds the data objects that are filled from the input grid and the data files, passes them to the solver, and accepts back the solution. It then passes the solution to the output function.

**Internal Functions:**
- It has four functions that are called from frmMain:
  1. **GridToFile**
     This function accepts the input grid object as a parameter. It then unwraps this object and writes the grid contents to an xml file. It returns nothing.
  2. **ImportDemand**
     This function takes no parameter and returns none. It does however instantiate the ImportDemand class, and uses it to populate an array of Day objects with workgroup demand from the demand file. This array is declared globally in the Model class and need not be passed around.
  3. **Import Shifts**
     This function takes no parameters and returns none. It does however instantiate the ImportShift class, and uses it to populate an array of Day Objects with workgroup shifts from the shift file. This array is declared globally in the Model Class and need not be passed around.
  4. **Solve**
     This function takes no parameters and returns none. It does however instantiate the Solver class, and pass in the two Day arrays, with the demand and shifts. The solver class then returns the solution to a third ‘solution’ Day array. This array is then passed to an instantiation of the export Results class.

### 6.2.4 ImportDemand

This module is responsible for opening up the work group demand file and reading it into an array of Day objects.
Internal Functions:

1. ReadFile
   This function begins the process of reading a work group weekly demand file. It takes no parameters and returns none. It does instantiate and use an instance of OpenDialogBox, which is used to allow the user to choose a demand file. The name of this file is then passed to the readFileIntoArray function.

2. readFileIntoArray
   This function takes the name of a file as a parameter and returns nothing. It first passes the file name to the readFile function, which determines the number of groups listed in the file. This number is used to initialize the array of Day objects. Next the readFileIntoArray function creates an instance of the StreamReader object using the passed in file name, and reads the file into the new array of Day objects.

3. getWeekArray
   This function takes no parameters, but returns the modules Day array object.

6.2.5 ImportShift

   This module reads the group shift schedule information from an xml file and writes it into an array of Day objects. An XML reader is instantiated for global use in this module, as well as a global array of Day objects.

   Internal Functions:

   1. openFile
      This function takes no parameters, but returns a file name string.

   2. ReadXMLDocument
      This function takes a string filename as a parameter and returns none. The global XML reader is given the file name and then the ReadSection Function is called.

   3. getWeekArray
      This function takes no parameters and returns an array object when called.

6.2.6 ExportSched

   This module takes the solution array and writes it to an MS Excel workbook.
Internal Function:

writeFile
This function takes a file name string as a parameter, along with a Day Array object. It returns nothing. It opens an excel file with the passed in name and populates it with the solution from the array of day objects.

6.2.7 Solver

This module takes the input data objects, sends them to the external solver and reads the solution.

Internal Functions:

1. checkData
   This function verifies that there are shifts scheduled for a particular group for a particular day. If there are none, then no calculations are necessary.

2. solveSchedule
   This function calls the external solver process and tells it where to find the problem file to solve. Once the solution is found, the solution is captured from the external process.

3. translateEmployeesPerShift
   Since the external process does not return the solution in the form required by the program, it calls this function to parse out the number of employees assigned per shift by the solver.

4. translateEmployeesPerBK1
   Since the external process does not return the solution in the form required by the program, it calls this function to parse out the number of employees assigned to a particular first break period.

5. translateEmployeesPerBKL
   Since the external process does not return the solution in the form required by the program, it calls this function to parse out the number of employees assigned to a particular lunch break period.

6. translateEmployeesPerBK2
   Since the external process does not return the solution in the form required by the program, it calls this function to parse out the number of employees assigned to a particular second break period.
6.2.8 timeUtil

Users view shifts and breaks in terms of various times of day, whereas the program views them in terms of the 96 distinct 15 minute periods. The methods of this class are used to translate between the two.

Internal Functions:

1. parseTime
   This function takes the input time and converts it to the appropriate period number, taking into account if it the start of shift or the end.

2. calcBreaksPeriod
   The periods during which a break can take place during a shift varies with the start and stop times of the shift itself. Given the starting and ending times of a shift, this function returns the appropriate break periods for that shift. This function sets the break periods as part of the data sent into the solver.

3. calcBreaksPeriodSolve
   Once the solver is finished, the solution is fed into a third set of shift objects. This function is called to recreate the breaks for the solution to be fed into.

4. initPeriodList
   Initializes the periodList array list of times. This list is used in the next two methods.

5. timeToPeriod
   This function returns the period a given input time (int HH:MM meridian) form.

6. periodToTime
   This function returns the time(int HH:MM meridian form) for the input period.

7. periodToTimeOutput
   This function is the same as periodToTime, except the results are adjusted for the user’s viewing. To a user, a shift end time represents the last minute worked, not the first minute of the last period, as the system views it. This function makes this adjustment.

6.2.9 fileUtil.

The program needs to be able to allow the user to select files names and paths to open and save files. This Class provides these capabilities.
Internal Functions:

1. **openFile**
   This method calls up the open file dialog box. This allows the user to select the name and path of the file they want to open.

2. **saveFile**
   This method calls up the save file dialog box. This allows the user to input or select the name and path of the file they want to save to.

### 6.2.10 createXMLFile

Input schedules are saved in XML format files. This class creates those files.

Internal Functions:

1. **openFile**
   This method takes an input filename and path and opens it for writing as an xml file.

2. **StartDoc**
   This starts the xml file document.

3. **writeStartElem**
   This writes the starting xml element to the file.

4. **writeEndElem**
   This writes the ending xml element to the file.

5. **End Doc**
   This ends the xml file document.

6. **closeFile**
   This closes the xml file.

7. **writeSection**
   This writes the contents of the xml file.

### 6.2.11 importPath

The program does not know where it may be installed, but it must know where its data files are, so this file allows the user (or the set up program) to specify where certain files will be located.
Internal Functions:

1. getSchedPath
   This method returns the path to the schedule file that was read from the path file.

2. getOutPutPath
   This method returns the path to the where the output file should go, that was read from the path file.

3. readPathFile
   This method opens the path file and reads its contents into the set path variables so they can be returned by the two ‘get’ methods.

6.2.12 writeDataFile

The external solver requires data to be in an exacting format that is very unlike the format in which the data is input into the system. It is the purpose of this class to arrange the data in the necessary format.

Internal Functions:

1. writeFile
   This method takes in the shift information, the demand information, the number of the day and the number of the group that is being processed. Then, using the methods of the class, it converts these inputs to the correct format and writes out the necessary files in the preset directory housing the solver.

2. processShiftArray
   This method accepts the array containing the shift information and returns the correctly formatted string containing the information.

3. findGapsInShiftArray
   The solver is unable to handle situations where there is demand for an employee, but there is no available schedule to cover it. This method identifies these periods and sets the demand to zero.

4. processDemandArray
   This method accepts the array containing the demand information and returns the correctly formatted string containing the information.

5. processBreakArray
   Like the schedules and the demand, the exact periods the breaks may take place on must be in a specially formatted string. This function does this.
6. translateShift
   This method is used by the processShiftArray method to convert the shift arrays to strings.

7. translateBreaksShift
   This method is used by the processBreakArray method to convert the shift arrays to strings.

6.2.13 Day

   This class is a data object. It holds an array of Group objects

6.2.14 Group

   This is a data object used to hold two arrays:

   The Demand Array, which holds the demand for agents for each of the 96 periods in a day.

   The Shift Array, which holds each of the Shift Objects created to hold the shifts assigned to a Group.

6.2.15 Shift

   This class is a data object. It holds the various shifts. It has the following methods.

   Internal Functions:

   1. setBreaks
      This method instantiates the timeUtil object, using the calcBreakPeriod function to place the range of possible break times for the newly created shift, based on the start and end periods of the shift. These break ranges are later used in creating the data file for the solver.

   2. setBreakSolve
      This method instantiates the timeUtil object, using the calcBreakPeriodSolve function to place the range of possible break times for the newly created shift, based on the start and end periods of the shift. Unlike the setBreaks method, this method uses a different data structure for the break information. The set breaks method only stores the range of possible periods for a break, where as this method stores the number of employees assigned to those
breaks as well. Two different methods were used because the shift objects used to capture the solver results are not the same as those used to construct its inputs. It was far easier to create new objects rather than try to gather up the input objects and update them with the solver results.
Chapter 7  
Experimental Environment

7.1 The Scheduling Environment

7.1.1 Variables  
There are many variables which go into creating the break schedules. Scheduling is done on a weekly basis; each week consists of 7 days, Sunday through Saturday. The length of a calling day is $H$ hours the call center is open, out of the 24 hours of a solar day. The number of clients the call center provides service for may vary. Each client’s calling demand is independent of the others and must be scheduled separately. To accommodate these separate schedules, each calling day is divided into $G$ calling groups. Calling demand for these groups is organized into Demand Sets. Demand Sets are created by an external process independent of the scheduling process. While independent, its characteristics are however important in judging the quality of the scheduling processes. The demand sets creation process takes the number of calls expected in a given period $P$ and calculates the number of agents needed in that period to answer them. The length in minutes of a period $P$ is determined by the number of periods and the length of a calling day. For a 10 hour calling day with 10 periods, $P$ would be 60 minutes. To determine the number of agents needed the process must know how many calls an agent can answer in a given period. This is agent efficiency $E$. $E$ is based on a number of factors, such as the average length of the expected call, the interval between calls, the time an agent spends actually taking calls (as opposed to chatting, doing non call paper work, or visiting the rest room). Likewise the demand set creation process engages in a certain amount of rounding to arrive at its predicted numbers. This does not affect the value of the scheduling research, as all processes under examination use the same demand sets. The study is only making claims about how well these processes deal with the given demand inputs, not the quality of those inputs themselves. Such issues are outside the scope of this paper.

Given then that an agent can answer $E$ calls in a $P$ period requires one agent to be present and available in that period to answer the calls, a single period with $E$ calls expected can be termed an ‘agent demand unit’ or simply a demand unit. If that period has twice $E$ calls expected, it is said to require 2 demand units. The input demand sets thus consist of the number of demand units required in each of the periods in a day.

The call center is paid $V$ dollars for each call answered. Thus for a single demand unit, the possible revenue to the company is $V \times E$, or the value of the call times the number of calls in a demand unit. Likewise, anytime the demand set called for an agent in a period and the agent was not scheduled, those calls would be lost, forfeiting the company the revenue of the lost calls.

On the cost side each agent receives an hourly wage of $W$ dollars per hour. An agent’s hours are organized into shifts of $L$ hours in length. Shifts typically include an unpaid lunch break and two paid rest breaks. An agent’s daily wage is then $W \times L$, or their hourly wage times the number of hours in a shift. So for every period an agent is correctly scheduled for, the revenue of calls minus the cost of the agents equals the profit to the company. Likewise, anytime an agent is scheduled for a period in excess of the expected
demand, the company has a loss of the agent’s wages for that period with no compensating revenue.

For example, Table 1 presents a simplified 10 hour day. Two agents have been scheduled to answer calls, each on a 9 hour shift with a 1 hour (unpaid) lunch. Their hourly wage is $5.55 per hour so the total cost of the agents is 8*5.55*2 = $88.80.

Table 1: Example Schedule

<table>
<thead>
<tr>
<th>Hours</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demand</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>Agents Scheduled</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>Demand Met</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>Demand Unmet</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Mis-Scheduled</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

In the example shown in Table 1 there were 19 demand units for the 10 periods. In this example each demand unit represents 10 calls, so there were 190 calls expected during the 10 periods. At $.95 per call this example has potential revenue of $180.50. However, there were only 13 demand units that were met, so (13*10 * .95) only $123.50 in revenue was actually realized. Since 6 demand units went unmet, those calls were lost and their revenue forfeit. Thus $57.00 in revenue was forfeit due to lost calls. There were 3 demand units mis-scheduled. In this situation the call center still has to pay the hourly cost of the agents, but takes in no corresponding revenue to offset the cost of the agents’ wages. Thus 3*$5.55 or $16.65 in wages were wasted because 3 of the periods had too many agents assigned. Overall though, the day was profitable, as the $123.50 in revenue more than covered the $88.80 in wages, leaving $34.70 in profit for the call center for the day.

7.2 The Shifts

The second scheduling input to consider is the collection of the shifts themselves. In this study the available start and stop times for the shifts are essentially fixed. To attract workers, the call center seeks to keep agent shifts consistent. They are drawn up by the scheduler ahead of time based on general calling demand and worker preferences. All the scheduling processes must work within these shift constraints. With the number of shifts and their start and end times fixed, the scheduling processes works to meet calling demand by varying the number of agents assigned to a particular shift as well as when the agents must take their morning, lunch, and afternoon breaks.
7.3 Actual Inputs

The experimental calling data which this paper uses are two weeks of predicted demand from the Holden Marketing call center in La Junta, Colorado. To study high demand periods the week of 12/06/2004 to 12/12/2004 was used as this was a high call volume period with 69,787 calls. To study low demand, the week of 6/21/2004 to 6/27/2004 was used as this was a low demand period with 29,568 calls.

Scheduling is done in 1 week at a time. Each week consists of 7 days, Sunday through Saturday. Since each calling day contains 8 individual calling groups, there are a total of 112 unique calling demand sets which require schedules. As this is real world data, 9 of the demand sets are empty, as no calls were predicted for that group for that day. Three others contain only one or two calls a piece for the day. As this is too small to be scheduled for, it is ignored. This leaves 99 demand sets that are used to create schedules.

Table 2: System Parameter Values

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length of Day</td>
<td>H</td>
<td>24</td>
</tr>
<tr>
<td>Calling Groups</td>
<td>G</td>
<td>8</td>
</tr>
<tr>
<td>Calling Periods</td>
<td>P</td>
<td>96</td>
</tr>
<tr>
<td>Agent Efficiency per Period</td>
<td>E</td>
<td>3.5</td>
</tr>
<tr>
<td>Length of calling periods</td>
<td>H/P</td>
<td>15 minutes</td>
</tr>
<tr>
<td>Value of Individual Call</td>
<td>V</td>
<td>$0.95</td>
</tr>
<tr>
<td>Agent hourly wage</td>
<td>W</td>
<td>$5.55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1/2 hour unpaid lunch)</td>
</tr>
<tr>
<td>Length of Shift</td>
<td>L</td>
<td>8.5</td>
</tr>
<tr>
<td>Agent cost per shift</td>
<td>L*W</td>
<td>$44.40</td>
</tr>
</tbody>
</table>

Table 2 summarizes the current system parameter values. Since agent efficiency is 3.5 calls per period, and each call is valued at $.95, then revenue to the company is $.95 * 3.5 or $3.325 per agent demand per period (demand unit). Likewise, anytime the demand set called for an agent in a period and the agent was not scheduled, those calls would be lost, forfeiting the company that $3.325 in revenue.

Since shifts are 8.5 hours long and a calling day is equal to one solar day (24 hours) a shift can start any time between midnight and 3:30 pm. Shifts may not start later than that because then they would run over into the next day.

Given the above characteristics, the aforementioned 99 demand sets contain 99,354.5 calls to be answered with potential ideal revenue of $94,386.78. This ideal revenue is not attainable in the real world, but it does give a baseline figure for comparison with the totals generated by the actual schedules.

While shifts are 8.5 hours long, there are two 15 minute breaks and a half hour lunch as noted in Table 2. Lunch is unpaid, but the breaks are paid. Thus each agents works for 7.5 hours, but is paid for 8. Thus for the $44.40 paid for a shift, each agents works for only 30 of 34 possible 15 minute periods. This means that each period costs the company $44.40 divided by 30 or $1.48. This is important because there are a total of 28,387 demand units in the entire experimental data set, which at $1.48 to cover a single demand unit, give a lowest theoretical cost of $42,012.76. Of course this is not possible because
that would mean paying people only for those calls they answered, not hourly for 8 hour shifts. Like the maximum revenues, this cost figure is not possible to meet in the real world, but it gives a baseline figure to use when comparing the costs of the actual schedules.
Chapter 8
Experimental Results

8.1 Criteria for Success

The most important aspect on which to judge a schedule is its profitability; that is the amount of money left from revenues after costs have been deducted. This is the criteria upon which the three scheduling processes are judged. However, there are several components which contribute to profitability.

The first component is the number of Calls Taken as a percentage of calls offered. As previously noted, the 99 demand sets are made up of 99,354.5 calls. The manual process answered the most calls with 93,299.5. The Aykin process answered 87,220 and Bechtold answered 83,947.5. For ease of comparison, these numbers are shown in Chart 1 as a percentage of the calls offered.

Chart 1: Calls Taken as percentage of all Possible Calls

<table>
<thead>
<tr>
<th>% of Calls Offered</th>
<th>Calls Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>90%</td>
<td>87.97%</td>
</tr>
<tr>
<td>80%</td>
<td>84.42%</td>
</tr>
<tr>
<td>70%</td>
<td>94.33%</td>
</tr>
</tbody>
</table>

Calls Offered: 99354.50
Aykin: 87405.50
Bechtold: 83874.00
Manual: 93719.50

Chart 1 shows that Aykin is the best of the two automated processes. At 87.97% it is 3 percentage points better than Bechtold. The Manual process is best in terms of Calls Taken, at 94.33% of the possible.

Of course what matters about the calls taken is the revenue they represent as seen in Chart 2.
Chart 2: Revenue.

Chart 2 shows that the calling revenue closely tracks calls answered as expected. The inverse of the calls taken and their revenue is the calls lost, and the revenue not earned. This is displayed in Chart 3. Just as the Manual process had the most calls taken it had the fewest calls lost at 6,635.0 Aykin had the second fewest with 111949, and Bechtold had the most calls lost with 15,480.5.

Chart 3: Calls Lost as percentage of all Possible Calls
As Chart 3 shows manual had only 5.67% of the possible calls lost, which is more than 6 percentage points better than Aykin and nearly 10 percentage points better than Bechtold. Aykin was second best, with 12.03%, beating Bechtold by 3%.

Chart 4: Revenue Lost

<table>
<thead>
<tr>
<th>%</th>
<th>Revenue Lost</th>
</tr>
</thead>
<tbody>
<tr>
<td>18%</td>
<td></td>
</tr>
<tr>
<td>16%</td>
<td></td>
</tr>
<tr>
<td>14%</td>
<td></td>
</tr>
<tr>
<td>12%</td>
<td>15.58%</td>
</tr>
<tr>
<td>10%</td>
<td>12.03%</td>
</tr>
<tr>
<td>8%</td>
<td>0.00%</td>
</tr>
<tr>
<td>6%</td>
<td>5.67%</td>
</tr>
<tr>
<td>4%</td>
<td></td>
</tr>
<tr>
<td>2%</td>
<td></td>
</tr>
<tr>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>

Possible Revenue
Aykin       $11,351.55  Bechtold $14,706.48  Manual $5,353.25

The calls lost translates to revenue lost, as seen in chart 4.

The third component of profitability is cost. Naturally there will be some incurred cost in answering calls, but the goal must be to incur those costs efficiently and not waste resources. In this study resources are wasted when agents are mis-scheduled and are incurring costs with no chance of earning revenue. The Chart 5 shows the amount of mis-scheduling done by each process.

Chart 5: Calls Mis-Scheduled

<table>
<thead>
<tr>
<th>%</th>
<th>Calls Mis-Scheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>120%</td>
<td></td>
</tr>
<tr>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>60%</td>
<td></td>
</tr>
<tr>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
</tr>
<tr>
<td>0%</td>
<td></td>
</tr>
</tbody>
</table>

Offered Calls
Aykin     66734.50  Bechtold $66871.00  Manual $96866.00
What is striking is that in all cases, as seen in Chart 5, there is massive mis-scheduling by all processes. Both Aykin and Bechtold mis-scheduled 67% and the manual mis-scheduled 97%. This is a tremendous waste of resources.

Chart 6 makes this waste even more explicit, by showing that the majority of the mis-scheduled calls were in excess of the number of actual calls and were not needed altogether. The point is the scheduling process did not simply put needed people in the wrong place, but it put in people that were not needed at all. Some of this is unavoidable, as all shifts must be eight hours long, but it can and should be minimized.

Chart 6: Total Calls Scheduled

As shown in Chart 6, the Manual process was the worst in terms of scheduling unneeded agents, 191.82% of what was needed. Aykin is the second worst, with 155.14% of what was needed, and Bechtold did the best in this area, with 151.72% of what was needed.

These patterns of the number of calls the processes scheduled for is reflected in the cost incurred by the processes. Chart 7 shows the cost of the three. Just as the manual scheduled to answer the most calls, and Bechtold the least, so was the manual’s cost the highest, Bechtold the lowest, and Aykin in the middle.
Comparing the cost with the revenue there is symmetry as can be seen in Table 3. The manual, which has the highest revenue, also has the highest cost. Likewise Bechtold, with the lowest revenue, has the lowest cost, and Aykin with the middle revenue has the middle cost.

Table 3: Comparing Cost with Revenue

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Aykin</th>
<th>Bechtold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revenue</td>
<td>$89,033.53</td>
<td>$83,035.23</td>
<td>$79,680.30</td>
</tr>
<tr>
<td>Cost</td>
<td>$80,763.60</td>
<td>$65,179.20</td>
<td>$63,447.60</td>
</tr>
</tbody>
</table>

These costs are a direct result of the number of workers scheduled by each algorithm, as seen in chart 8. The more agents the higher the cost.
Chart 8: Agents Scheduled

The revenue and cost figures are combined to give profit figures in Chart 9.

Chart 9: Profit

<table>
<thead>
<tr>
<th>% of Possible</th>
<th>Profit Possible</th>
<th>Aykin</th>
<th>Bechtold</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>120%</td>
<td>$52,374.02</td>
<td>$17,856.03</td>
<td>$16,232.70</td>
<td>$8,269.92</td>
</tr>
<tr>
<td>100%</td>
<td></td>
<td>100.00%</td>
<td>34.09%</td>
<td>30.99%</td>
</tr>
<tr>
<td>80%</td>
<td></td>
<td></td>
<td>15.79%</td>
<td></td>
</tr>
<tr>
<td>60%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In Chart 9 one can see that manual schedule’s advantage in calls taken and calls lost comes at the price of much lower profitability: $7962.78 less than Bechtold and $9586.11 less than Aykin.

As for the $1623.33 difference between Aykin and Bechtold, it is the result of Aykin’s better accuracy, i.e. more calls taken, fewer lost, with the same amount mis-scheduled. So for these reasons, the aggregate profits of Aykin are superior to Manual and Bechtold, as it is more efficient.

A look at the individual schedules shows that the superiority of Aykin over Bechtold is not the case of a few extreme cases skewing the overall picture. Of the 99 schedules, Aykin was more profitable in 62 of the schedules and equally profitable in another 13. Bechtold was the more profitable in the remaining 24. In those 24 cases Bechtold had fewer workers, which kept costs down. But this approach failed in 11 cases where Aykin was able to use additional agents to earn additional revenue, giving it a higher profit margin overall.

Further, these occasions of Bechtold superiority are evenly distributed throughout the demand range. Thus it is not a case where Bechtold was superior in a particular level of demand and was later overtaken as Aykin high demand profits covered earlier low demand losses. This is seen in Chart 10. The curves for both Aykin and Bechtold are very close, with Bechtold occasionally being the more profitable, but Aykin is generally the winner.