Analyzing and Solving the Nurse Scheduling Problem

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Bachelor of Arts in the Department of Mathematics and Computer Science at The College of Wooster

by
Nicholas Hagopian-Zirkel

The College of Wooster
2018

Advised by:

Dr. Nathan Fox

Dr. Robert Kelvey

Abstract

From large to small, most every workplace requires some sort of scheduling. Nowhere perhaps is this more critical than in hospitals, where, in addition to all the normal labor restrictions for scheduling, it is a matter of life or death that hospitals be sufficiently staffed at any hour of the day. The nurse scheduling problem (NSP) is the problem of creating a schedule that satisfies all the wants and needs of a hospital and its nurses. With current methods, the problem cannot be solved directly in reasonable runtime and its large number of constraints mean that any feasible solution may be difficult or even impossible to find. This problem's difficulty, real-world application, and its unique relationship to its constraints have solidified the NSP's importance in the field of computational complexity theory. In this paper the NSP is analyzed and methods for solving it are discussed. Additionally select methods and constraints are showcased in corresponding software.

Contents

Chapter 1

Introduction

1.1 Summary

The nurse scheduling problem (NSP), sometimes referred to as the nurse rostering problem, is the problem of assigning nurses to a schedule in such a way that both optimal and feasible in terms of the wishes of a given hospital and its nurses. Between any two hospitals there are likely differences of opinion as to what is considered to be the best schedule, but regardless, any hospital opinion can be deconstructed into three parts: an *objective function*, a set of *soft constraints*, and a set of *hard constraints*. In essence, hard (binding) constraints are constraints that cannot be broken and soft (non-binding) constraints are constraints that should not be broken, but may be. More specifically if hard constraints are broken then the schedule is considered infeasible while soft constraints being broken applies a penalty to the value of the schedule. The consequence of infeasibility is that a schedule is "impossible," or, alternatively, it is valued worse than any feasible schedule.

The penalty incurred by defying soft constraints is accounted for in the objective function. An objective function is some function that accounts for anything in a schedule that the hospital considers valuable and applies weights to these values. The objective function is created with the intention of being optimized (maximized or minimized depending on the setup). Since hospital constraints in the NSP can be broken into these three pieces, regardless of the differences between hospitals, all NSPs can be generally set up as the problem of creating a schedule that respects a set of given hard constraints while attempting to optimize an objective function that has a relation to a set of soft constraints [17].

Phrasing the problem is simple but solving for a globally optimal solution is hard. The brute force solution for the problem, in many setups and sizes, can take hours, days, or even years to solve. It is impossible with modern day methods (and believed to be impossible with any method) to solve the NSP in reasonable time (polynomial runtime). The importance and the interest in the nurse scheduling problem thus arises from its difficulty.

An outline for the brute force solution of the NSP starts off by generating all combinations of nurses in all shifts (the problem space). After this step, each schedule is tested for feasibility and the values of all feasible schedules are compared to one another until an optimal schedule is found. However this brute force solution has two problems: both memory space and computational time of this problem grow substantially as problem size increases linearly. The problem of memory space can be reduced easily by comparing each schedule as it is generated to the most optimal schedule generated so far. In doing this only two schedules have to be stored at once. In terms of runtime however, the total value of a schedule cannot be known until it is fully generated. Therefore, this requires an exhaustive search of the space. In other words every combination must be calculated.

For the sake of emphasis, consider the number of schedules that must be generated to solve the NSP for just one day. Also suppose on this day there are three shifts to fill that each require exactly five nurses per shift, there are 30 nurses to choose from, and the problem is constrained by nurses working at most once per day. It is also true that two nurses working in the same shift cannot be swapped to create a new shift (position in the shift does not matter). Thus, the number of ways these shifts can be arranged feasibly to be $\binom{30}{5} \cdot \binom{25}{5} \cdot \binom{20}{5} = \binom{30}{5;5;5}$. In shift 1 there are $\binom{30}{5} = 142,506$ combinations of nurses, for each combination there are $\binom{25}{5} = 53,130$ combinations of the remaining nurses available for shift 2 and shift 3 has $\binom{20}{5} = 15,504$ combinations. Therefore the total number of feasible arrangements for this setup is $15,504 \cdot 53,130 \cdot 142,506 = 117,386,113,965,120$.

The above calculation also simplifies the problem to just consider the feasible solutions however infeasible solutions can be difficult to avoid checking depending on the constraints. When assuming an exact number of nurses per shift, the problem space for the NSP is of the size $\binom{N}{M}^{ds}$ where N is the number of nurses, M is the number of nurses to fill a shift and d and s are the number of days and shifts respectively. If infeasible solutions do exist then while iterating through the problem space, if some feasible schedule already exists, the total number of schedules necessary to compare can be reduced by deciding infeasibility of a schedule during its production. Doing this will allow us to throw out all schedules that have an infeasible prefix. This simplification however still does not prevent the problem from growing on at least a similar scale.

1.2 Applications

The name of the problem specifies nurse scheduling since hospitals are a common place for the problem to arise. This is due to the large size of hospital staff, the 24 hour nature of hospitals, and the critical aspect of having a full, competent staff on duty at all times. Although the title represents the problem as seemingly niche, it can be extended to many types of scheduling. At its most basic level, the NSP is any type of ordered assignment that respects constraints and strives for some form of optimality. As such, it has broader applications, and similar or simpler problems can at least take inspiration from it. It should be no surprise that any problem of scheduling staff can be formulated as a variation on the NSP. Another common application is the extension of ideas from the NSP to course scheduling. More interesting perhaps is that the problem may be used in allocation of computer resources. This can take the form of allocating runtime to processes, or servers allocating computational power between connections [14]. However, some real-time process scheduling requires such frequent scheduling that it is better to use a FIFO (first in first out), or another simple algorithm. Allocating too much runtime to scheduling effectively will end up wasting runtime instead of saving it.

1.3 Scheduling in Practice

The nurse scheduling problem has been discussed in terms of computational approaches for more than 40 years [17]. Until recently, most scheduling was solved manually. One popular method for doing so was called "self-scheduling" in which the job of scheduling was given to the nurses individually [4]. Once nurses were assigned days to work they could negotiate amongst themselves to improve their schedule. Handing over the reigns of scheduling to nurses removed responsibility from the hospital but would very likely result in schedules of very low quality. Automatic scheduling is a relatively new approach to scheduling but has potential to wildly increase efficiency in many aspects of the workplace.

Chapter 2

Computational Complexity Theory

2.1 Introduction

In the 1930s mathematical logicians first began studying the capabilities and limitations of computation [19]. While studying computation two fields arose, computability theory and computational complexity theory. *Computability* answers the questions of whether the solution to a problem is possible to compute while *complexity* deals with the difficultly of problem. Originally, these topics were strictly theoretical, but as technology improved questions of computability and complexity became relevant to solving practical problems. Given that the nurse scheduling problem has a brute force solution, it is clearly computable. So it is the problem's complexity that is more interesting. The question of what makes some problems easy and others hard is central to computational complexity theory, yet the answer is still unknown [19]. In an attempt to define problems as "easy" or "hard," complexity theory may analyze the runtime of algorithms and how they scale with the size of a problem.

2.2 Big-O Notation

A common way to discuss runtime is to use *big-O notation* (or just big-O) which describes the worst-case runtime outcome for an algorithm. In order to determine big-O notation for some algorithm a function that calculates algorithm runtime with respect to input size should be created. The value n is the standard parameter used to denote the input size of a problem. For example, the runtime of sorting algorithms will scale with the n number of elements being sorted.

The formal definition big-O notation states that for a function $f(n)$ has big-O of $O(g(n))$ if and only if there exists some c and some $N > 0$ such that for all $n > N$, $f(n) < cg(n)$. This definition, similar to a limit definition, shows that as $n \to \infty$ the function graph is shaped by the fastest growing term. Big-O notation therefore ignores any term that is not the greatest order (fastest growing term) and ignores any constant factors on this term. Now suppose there are two algorithms with runtime measured as $7n^2$ and one with $n^2 + n$. Considering the fastest growing term and ignoring its constants it is seen that these functions have the same big-O notation $O(n^2)$. There are two main reasons that big-O notation is valuable despite its simplification of runtime: it is used to classify the growth of an algorithm for the sake of comparison and computational complexity mainly cares about large problem scales. The theory is, runtime of an algorithm is eventually determined only by the fastest growing term. Likewise, while comparing the runtime of two algorithms with different big-O classifications, the larger one will eventually always eclipse the other when increasing the input size.

2.3 P, NP, NP-complete, and NP-hard

When discussing a problems difficulty, computational complexity theory gives us four classifications for problems. These classifications contain all decision problems and are P, NP, NP-complete, and NP-hard where some problems can have more than one of these classifications. Using these labels provides a level of abstraction from big-O notation and allows for problems to be placed into only four classes as opposed to the infinite number of potential classifications seen with big-O notation. Additionally, big-O notation discusses specific algorithms while this hardness classification discusses only the most efficient algorithms. This allows for the placement of problems directly into classes but also requires consideration of potentially unknown ”most efficient algorithms”.

P P is the “easiest” classification of problems. It is the set of all problems whose solutions are computable in polynomial runtime. Formally, deterministic computation models are *polynomially equivalent* if any one of them can simulate another with only a polynomially increase in running time [19]. As a result of this, problem Y is in P if for the set X, of problems known to be in P, Y is polynomially equivalent to any (and as a result all) problem X. This idea of turning one problem into another is known as a reduction and is formally discussed in the section 2.5.3.

Every polynomial problem is contained within P. As a result, P may seem as if it is an overly broad classification of problems. Placing $O(n^2)$ and $O(n^30)$ into the same category of difficulty may seem irrational when defining easy and hard problems, however choosing a large problem size we can see why polynomial problems are classified differently from faster growing problems: such as problems that scale exponentially or factorially. Using $n = 1000$ we can see that n^3 is 1 billion, n^30 is 10^90 whereas 2^n is a number larger than the number of atoms in the universe [19]. This example is a dramatic way of emphasizing the importance of separating problems into different classes.

NP NP is the second classification of problems. NP contains all problems whose solutions can be verified to be true in polynomial time. This, of course, includes problems that can be solved in polynomial time (all problems in P) since the method for solving the problem are polynomial and can be used as verification of the problem. Now since P and NP have known problem overlap the question of what separates P from NP should be considered. Alternately, does P=NP or P≠NP (P vs NP)? To answer this question we have two options: we can show every problem in NP has a solution that runs in polynomial time or we can prove the existence of a problem in NP that be solved in polynomial time. This seemingly simple question (P vs NP) remains unsolved and is the most important unanswered question in all of theoretical computer science. Although it is unproven it is generally believed that P≠NP.

NP-hard NP-hard is the hardest classification of problems. It is defined to be the set of all problems that are at least as hard as the hardest problems in NP. Therefore NP-hard and NP intersect at the set containing the hardest problems in NP. If P=NP then this set is all of P. By the definition of NP all problems in NP-hard and not in NP are problems whose solutions cannot be verified in polynomial time. Given that a problem cannot be verified in polynomial time it can either be verified in greater than polynomial time or the problem cannot be verified (the problem is undecidable). One famous undecidable NP-hard problem is the halting problem.

NP-complete NP-complete is the classification that contains problems that are in NP and in NP-hard. Given that NP-hard is the set of problems that are at least as hard as the hardest problems in NP it follows that the intersection of NP-hard and NP must be the set of the hardest problems in NP. Any problem that is not one of the hardest problems in NP is either too hard to be in NP or too easy to be NP-hard. Additionally, if P=NP then it is also true that P=NP=NP-complete. Since P=NP implies that all problems in NP are polynomially equivalent (of equal difficulty), thus each can be reduced to another with only a polynomial increase in running time. This polynomial equivalence would allow us to say all problems in NP are the hardest problems in NP.

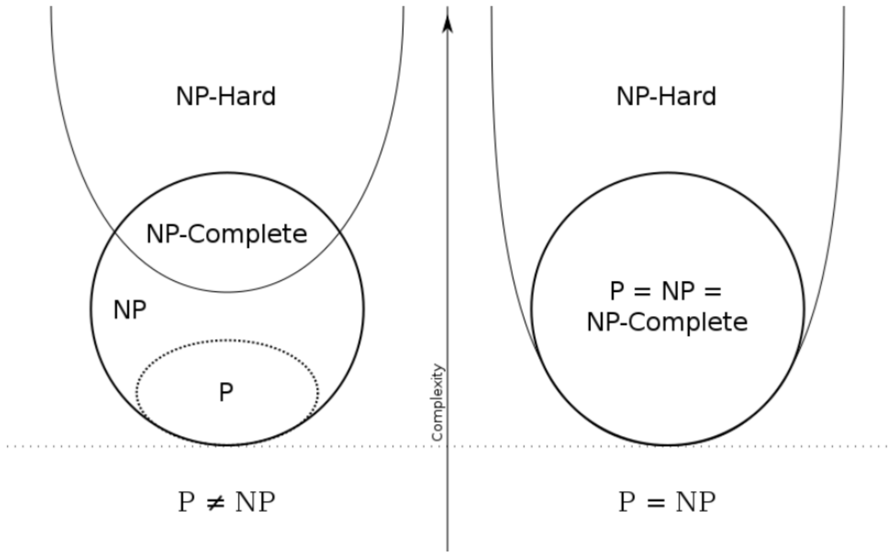


Figure 2.1: Overlap of P, NP, NP-complete and NP-hard given P≠NP and P=NP [12].

2.4 Constraints

Since there is no standard formulation for the NSP, different problem setups and difficulties may arise between hospitals as their specific constraints are considered. The number and complexity of constraints can vary the problem difficulty significantly. Too many constraints and the NSP may completely infeasible, too few and the problem may have obvious greedy solutions. The constraints of the NSP are closely related to its complexity.

Consider the situation where there exists feasible schedules but few of them. Depending on the complexity of the constraints it may be just as hard to find the optimal schedule as it is to find a feasible one. This can be seen with the creation of a variant of the NSP. Consider the NSP that asks to find at least the second most optimal schedule. Doing this is of course on a similar order of difficulty as the normal NSP. This idea can also be extended to the NSP with "few" feasible schedules with the only goal being to find a feasible one.

If few feasible schedule exist the problem may require some modification. In this case, we can simplify schedules such that defied hard constraints are set to $-\infty$. A schedule being set to $-\infty$ has the same effect as any value large enough to more than outweigh the best possible solution. These are equivalent to saying that defying a hard constraint makes any schedule worse than all feasible schedules. Doing this allows us to compare infeasible schedules instead of just feasible ones. Therefore we may also find a near-feasible schedule (few hard constraints are defied) that can be improved into feasibility or to "less" infeasible solutions. Of course, we should not be satisfied with infeasible solutions but in the real world there exist circumstances in which we must be. Additionally, notifying the hospital of this violation of feasibility may allow steps to be taken by the hospital to solve this problem that software cannot consider (like hiring more staff).

Problems with limited feasibility do not strictly make the NSP more difficult. Depending on the method used to get or approximate the optimal NSP solution, we may be able to reduce the computation time of an algorithm by checking for feasibility of schedules at each stage of generation. In some ways this can simplify our problem, but in others it may complicate it. Although we can stop generating a schedule if the prefix is infeasible determining that feasibility with certain constraints may require more complex checks and substantial data storage.

On the other end of the spectrum we may have NSP formations that have too few constraints. The nurse scheduling problem arises as a problem in computational complexity theory only due to the number of constraints and their complexity. This can be shown when considering the most trivial NSP–the NSP with no constraints and a constant objective function–where a solution with greatest possible value can be easily found in polynomial runtime. This task is completed by assigning nurses randomly to every shift. This is always optimal since all assignments are feasible and valued the same by the objective function.

The trivial NSP is not the only setup in which a solution can be found with certainty in polynomial runtime. Changing the trivial NSP by adding a simple objective function or few enough constraints will also allow the NSP to be solved optimally in only polynomial runtime. An NSP with only a few hard constraints for example can be solved by finding one, of perhaps many, feasible solution.

Since some setups clearly allow for polynomial time solutions and the nurse scheduling problem is only significant to discuss due to its difficulty, we can consider the NSP in the context of this paper to be in reference to problem setups that do not have consistent polynomial time solutions. We also, however, want to know at what point there are complex enough or a sufficient number of constraints in problem setup that the problem is computationally difficult. We know this to be true when we need to use any method requires consideration of every configuration. Therefore, it can be generally be said that, the NSP cannot be solved in polynomial time if the value of a configuration is deeply related to all assignments. This is discussed more in section 2.5.

Searching the entire problem space for the NSP will always result in a problem that is of order at least $O(n!)$. However it is not true that we always have to search the entire problem space. Depending on the constraints we may be forced into a specific schedule or we may be able to prove that our current schedule has a score equal to the best possible score. These variations of the problem are of course important, but they should be excluded from most papers as they are solvable in “reasonable” runtime. It should be noted that in this case “reasonable” runtime does not mean polynomial runtime, because the problem still has the same runtime methods for brute force, but the search space is small enough that the order is inconsequential. The NSP as it is canonically discussed relates to only versions of which the problem is NP-hard.

2.5 Classifying the Nurse Scheduling Problem

The previous section states that varying constraints could result in NSP formulations that are not NP-hard. The nurse scheduling problem’s complexity is thus closely related to its constraints. Therefore, in order to classify the nurse scheduling problem each of its constraints must be acknowledged and dealt with. Given there is no standard set of constraints that makes up the NSP, we will only discuss the constraints implemented in this paper’s corresponding software.

2.5.1 Preliminary Work

We would like to classify the NSP as NP-hard and to do this only two constraints are used: strict shift coverage and balanced shifts. Separating these two constraints allows for a simple reduction later but once the NSP is proven to be NP-hard other constraints should be added back to the problem. Thus we use the following lemma provided by Steven Den Hartog [9]

Lemma 1. *Given a nurse scheduling problem already proven to be NP-hard, adding any additional constraints to this problem will result in a problem that is at least NP-hard.*

This is equivalent to saying that adding any of the given constraints should never makes a problem computationally "easy". In order to prove this we need to address each constraint individually. In addressing each constraint our goal is to make them trivial such that we can show that there exists a form of the problem in which the constraint does not change the complexity [9].

(hard) Days in a row The total number of days a nurse can work in a row. Setting this constraint to be such that the number of days in a row allowed is equal to the total number of days per month makes this constraint trivial.

(hard) Nurse requests A nurses ability to request days to work and days not to work. We can make this constraint trivial by considering the situation in which no nurse requests are made.

(hard) Days per period A range in which a nurse can work. Nurses can not work more or less than the range. This can be made trivial by setting the maximum number of days a nurse is working to the number of days of the scheduled period and setting the minimum number of days to 0.

(hard) balanced shifts The balanced shifts constraints states that nurses must work each specific shift (morning shift, day shift, night shift, etc...) a minimum and maximum number of times. This constraint is used in the NP-hardness proof and thus does not need to be made trivial.

(hard) shift coverage The shift coverage constraint comes in two forms: strict shift coverage and range shift coverage. Because the proof for NP-hardness uses range shift coverage this constraint does not need to be made trivial.

(hard) rest between shifts Nurses must have breaks between their shifts this can be made inconsequential by setting the break between shifts to be 0.

Since all of the above constraints have a trivial form, it is clear that if a problem is NP-hard without the constraint it is NP-hard with them as well. This is further discussed by Hartog in his master thesis [9]. When using anything more complicated than the trivial constraints we are just decreasing the scale of the problems feasible search space, not the complexity. The search space of the problem may become quite small, but the problem will still be NP-hard. In the situation of a small enough search space, a brute force method is doable in a short period of time if programmed to be “smart” enough to avoid checking infeasible possibilities. The runtime however will continue to not be polynomial.

Not all relevant constraints are covered by the previous lemma. However with the following lemma helps us further.

Lemma 2. *Consider a NSP with a set of constraints that make it NP-hard. Changing any of the hard constraints to soft constraints creates a new NSP that is also NP-hard.*

We are trying to show that soft constraints are no "easier" than their hard constraint counterpart.

Proof. Consider A to be an NSP that is NP-hard and has some hard constraint. Now let us call another NSP B, where B is equal to A except one hard constraint from A is now soft. Let us call this constraint c. Now, when we are asked to solve B, this is equivalent to asking us to find an optimal schedule in the set of all schedules that do not defy c and all schedules that do defy c. All problems that do not defy c are thus equivalent to A and therefore when we attempt to solve B we must solve A as well. Thus since A is NP-hard, B must be NP-hard. □

2.5.2 Nurse Scheduling Problem in NP

We want to show that the nurse scheduling problem is NP-complete. In order to prove a problem is NP-complete it must be shown that it is both in NP and also NP-hard. The former can be done using the verifier definition of NP. This states that "NP is the class of problems that have polynomial time verifiers [19]." In order to prove that the NSP is in NP then we want to consider its equivalent decision problem that asks us to find a problem that is at least as good as some k. A decision problem is any problem whose answer is either "yes" or "no". All problems that are at least as good as k are considered to be in some set A. Now we set up a verifier with the NSP to be such that, given some schedule of nurses in shifts, we can check this problem to be in A. This can be easily done in polynomial time given that all that needs to be done is the scoring of a schedule and checking whether it is greater than or less than k. Thus the NSP is in NP.

2.5.3 Reduction

Reductions are important for the classification of a problem's hardness. To show a problem is NP-hard, it must be true that every problem in NP reduces to it. What it means for a problem A to reduce to another problem, B, is that for problem A there exists a polynomial-time algorithm converting the inputs of A to the inputs of B such that the result of these inputs from B now solves the problem for A. Reductions can also be used to show that reducing any problem to a problem in P in polynomial time proves that the problems are "polynomial equivalent".

Now, in order to prove NP-hardness there are two possibilities. First, consider the previous statement that says a problem is NP-hard if and only if all other problems in NP reduce to it. This, of course, is quite a difficult task as there are endless problems in NP. Verifying this to be true requires quite a bit of abstraction. The second option for proving NP-hardness is much easier: first, start with a problem that has already been proven to be NP-hard, now, reducing this problem to another shows that the other problem is also NP-hard. In doing this, it is clear that if all problems in NP reduce to some problem Y then if Y reduces to X then all problems reduce to X (at least through Y). Each of these reductions are done in polynomial-time and thus doing both of these reductions together will also be polynomial-time. Of course this method is only possible if there is already a Y proven to be NP-hard. Luckily this has already been done. The first problem proven to be NP-hard was the 3 satisfiability problem (3SAT), proven by Stephen Cook in 1971 [6] and from this many more have followed.

Thus, in order to show this formulation of the NSP is NP-hard the method of reduction must be used. As previously mentioned, the NSPs complexity is closely related to its constraints. Thus the constraints must be chosen carefully. The constraints considered are hard shift coverage and hard balanced shifts. Hard shift coverage states that all shifts must be covered with a number of nurses greater than or equal to some minimum number of nurses and less than or equal to some maximum number of nurses. Hard balanced shifts states that all nurses must work some shift (morning shift, day shift, night shift, etc...) at least some number of times and at most some other number of times. This NSP setup is known to be NP-complete and is discussed by Hartog in his master thesis [9].

The software created adjacent to this paper is modifiable with many aspects that can be changed (number of total shifts, number of nurses working per shift, which constraints are turned on, etc...) and because of this it is hard to know if the software is always dealing with an NP-complete problem. Given the many variations it is incredibly difficult to show all variations are NP-complete (many of them may not be), however from Hartog's proof [9] that the software deals with at least one NP-complete problem. Additionally, with the help of the preliminary work proving one configuration to be NP-complete allows us to say many other "harder" variations are also NP-complete.

Hartog has also proven a number of other combinations of constraints to

Abbreviation	Constraint
SC	shift coverage
SSC	strict shift coverage
BS	balanced shifts
DOR	day-off requests
PR	personal requests
FS	forbidden sequences
SR	skill requirement
IW	identical weekends

Table 2.1: All constraints abbreviated for the following table. These constraints preceded by a lowercase "s" implies soft constraint while lowercase "h" implies hard constraint.

be solvable in polynomial time or he classified them as NP-complete. A complete list of these proofs can be seen in the tables below. This work is proof of the NSPs close relation to its constraints.

NSP Constraints	Resulting Complexity
hSC	Polynomial
sSC	Polynomial
sSC and hIW	Polynomial
hSC, hBS, and hDOR	NP-complete
hSC and hBS	NP-complete
hSSC and sFS=3	NP-complete
hSC and sFS=2	Polynomial
sSC and sFS=2	Polynomial
sSC, sFS=2, and sIW	Polynomial
hSC, hPR, and hSRs	Polynomial
hSC, hSRs, and (hard) full free weekends	Polynomial
hSC, hPR, and hSRs	Polynomial
hSSC, hFS=2, and hDOR	NP-complete
hSSC, sFS=2, hSRs	NP-complete

Table 2.2: A version of the table found in "The Complexity of Nurse Scheduling Problems", modified to list only the constraint combinations which result in NP-complete complexity [9]. Proofs of each are included in the reference.

Chapter 3

Solving the Nurse Scheduling Problem

3.1 Introduction

The nurse scheduling problem asks the question of "What is the optimal schedule given a set of constraints?" This answer can be computed eventually, but regularly generating schedules is inefficient. To deal with this, when generating the optimal schedule for the NSP, instead the question "What is a nearest-optimal schedule that can be computed in polynomial time?" should be asked. Fortunately, there exist quite a few methods that allow us to generate this near-optimal solution in a way that takes reasonable (polynomial) runtime. These methods can be problem simplifications, approximation algorithms, or most any combination of both. In addition to approximating a solution, the feasibility of a problem should be considered.

3.2 Constraints

There is no “standard NSP,” however all NSPs can have their constraints separated into the wants and needs of hospitals and nurses. Consider *needs* as referring to hard constraints while viewing *wants* as soft constraints. Hard constraints are things that must be satisfied for a solution to be valid. A soft constraint is a constraint that should attempt to be satisfied. If it is not satisfied, the objective function incurs a penalty. An objective function is a function that is meant to be optimized. In the case of the NSP the objective function values a particular schedule with respect to the objective of a hospital. Therefore it makes sense that defying soft constraints will lower the quality of a schedule.

There are certain constraints that appear frequently, however. One such example is of a hard constraint that states a hospital needs to be fully staffed at all times. In terms of a nurse hard constraint there is the need of a nurse to have some days off every month. A common soft constraint for nurses is disallowing them from working shifts too close together or working hours that the nurses previously noted as inconvenient. Additionally many formulations have 3-4 shifts of 6-8 hour shifts per day [17]. Aside from this, the NSP is not well defined. Each version can have any number of various soft and hard constraints in addition to a unique objective functions that could optimize things like staff quality, cost, or nurse satisfaction.

3.3 Problem Simplification

As mentioned in section 2.4, the nurse scheduling problem only arises when there are a significant number of constraints. If the problem is simplified in a way that ignores certain hard or soft constraints, it is possible to find a solution that is optimal to this new problem. Now this can be either accepted as the final solution, knowing it is likely infeasible by the initially given constraints, or constraints can be added one at a time and the solution can be adjusted in response.

For models that value a schedule based on even distribution of some staff characteristic (seniority, specialty) or anything that differentiates nurses (nurses can ask to work together, similar days requested off) the method of grouping nurses together is a viable simplification. Doing so allows shifts to only be filled once with a grouping instead of n times for each nurse required to fill a shift staffing requirement of size at least n. This reduces the number of choices significantly allowing for small brute force solutions or making slow approximation methods more viable.

3.4 Greedy Schedule Generation

Greedy schedule generation is a simple solution to a complex problem. The methods for doing so more often than not will give better results compared to random nurse assignment. Additionally, a well made, greedily produced schedule is more likely than a randomly produced one to create a feasible solution. However, there are better methods for optimizing the nurse scheduling problem. These greedy methods may not provide the best solutions themselves, but they can be used with many other algorithms to produce more optimal results than either by themselves. Two of the more common algorithms that can use greedy methods effectively are genetic algorithms for an initial generation seed and swap algorithms.

3.5 Dynamic Programming

At first this problem may seem like a perfect candidate for dynamic programming; the problem has the hallmark features of overlapping subproblems. If the particular setup of the NSP does not have soft constraints related to sequences of shifts then there exists overlapping subproblems. That is, each shift (or set of shifts) with a set of nurses only needs to have its value computed once and then the value can be stored. If all permutations of schedules are being calculated then specific periods of shifts in a schedule must be calculated multiple times or in the case of dynamic programming, saved. Saving schedules is done with the intent of finding an optimal partial schedule to avoid future recalculation.

This cannot be done in the NSP however, since partial schedules are too related to the full schedule they are placed in. Therefore the NSP does not have an optimal substructure. Dynamic programming is useful so as to not repeatedly compute the same subtrees, however it still has to check all the possible permutations of schedules since a value is only known to be optimal once all points are connected. Thus a locally optimal schedule is not necessarily globally optimal. In addition the space complexity is now also exponential which creates new issues.

3.6 Genetic Algorithms

The model used in all genetic algorithms (GAs) was constructed using concepts from the process of evolution. The goal of genetic algorithms is to "breed" solutions to problems. In evolution and genetic algorithms alike when a generation is presented with a problem, due to random genetic variation, some of the members of the generation will have features that allow them to handle the problem more successfully. Conversely other members have features that make them especially unsuccessful at completing the problem. If the problem is life threatening then those who are the most suitable to handle a problem are the most likely to survive and pass on their superior traits to the next generation. As this process repeats, all members of a generation should improve in their ability to solve a given problem.

In genetic algorithms work more specifically by first choosing a problem to be solved. Then the first generation is initialized either randomly or from some other method that may attempt to approximate a solution to the problem. Next the members of a generation have their ability to solve the given problem evaluated with a fitness function. Those who do this worst are killed off, and the fittest are bred together. Their children share traits with the parents and also have the chance to mutate, which allows for the entrance of random variations and new possibilities. New generations are bred and selected until some stopping condition is reached.

A stopping condition is implemented in a GA to use as a point where an acceptable solution has been considered reached. A stopping condition can be implemented in a number of ways depending on the GA. The most simplistic way of setting a stopping condition is to iterate through a set number of generations before ending. The next way is to stop breeding generations if the solution has reached a certain predefined threshold. This is difficult to do effectively in problems that are NP-hard but not NP-complete. This is because solutions to problems harder than NP-complete cannot be verified in polynomial time. Given this is true setting a threshold cannot be scaled with the solution to the problem in polynomial time. The final way to create is a stopping condition is by way of stagnation. Stagnation occurs if,over a set number of generations,the solution has not improved substantially enough. Any of these methods for choosing stopping conditions may be combined as well. The steps for a genetic algorithm can be seen below.

Step 1 - Choose how to randomly generate the initial population.

Step 2 - Evaluate fittest members of the population using a fitness function.

Step 3 - Breed next generation using crossover from fittest members of parent generation.

Step 4 - Mutate members of child generation.

Step 5 - If the genetic algorithm has not reached any stopping condition, go to Step 2.

Step 6 - Return the most successful member of the final generation.

3.6.1 Initial generation

The members of the first generation can either be randomly varied or they can be mutated from a given initial seed. The initial generation is not expected to have very successful solutions. It may however have some good “ideas”, meaning some of its members may have optimal substructures. Unfortunately, depending on the initial generation, the quality of a final solution could change substantially. This would likely be a combination of not enough mutation introduced between generations and the genetic algorithm instance getting caught in locally but not globally optimal solutions.

3.6.2 Fitness Function

The fitness function is a necessary component of all genetic algorithms. This is the part of the algorithm that answers the question of whether a member of a generation should survive to pass on its traits. The fitness function evaluates how well a member of a generation solves a problem. In the case of the NSP the fitness function is the objective function that should be optimized. The term “fitness” refers to the evolutionary idea of “survival of the fittest” in which those who are more “fit” are more likely to survive long enough to reproduce. The more successful members are awarded a higher chance of survival.

The purpose of only making more fit members more likely to survive and not simply eliminating a flat percent of the least fit members is so that generations do not become homogeneous. Although a very high chance of survival should certainly be awarded to the top members of a generation, a high chance of survival can be awarded to any members who are relatively successful while also have drastically different genes then the most successful members. This variance should eventually lead to stronger genes. Simply killing a flat percent of a generation by the weakest members will likely hurt the final solution. But genetic variance is also implemented through mutation, so it will not ruin a GA.

The amount of members to be killed is important to fine tune in any given GA. Killing too many members of a generation is okay. It just minimizes genetic variance, which can be introduced still through mutation. Likewise killing too few members is also a only a minimal problem. The only negative effect comes in the form of more necessary generations. Ensuring variation between generations sets apart good GAs from great ones.

3.6.3 Crossover and Mutation

Randomness is one of the most important factors of evolution and genetic algorithms alike, this is why mutation is so important. A central philosophy of genetic algorithms in using randomness is the idea that eventually a member of a generation will get lucky. The purpose of the rest of the GA is to notice when a member of a generation got lucky.

It is helpful to think of each new generation as a two-step process. The first step is where the unfit members of a generation are killed in order to generate an intermediate population of fit members. Then, recombination of genes and mutation are applied to the intermediate population to create the next generation. In all genetic algorithms, member of a generation have their own “genes” (typically represented by a string of bits) that can be used in crossover or mutation.

It may seem confusing why crossover and mutation are both used in genetic algorithms. The purpose for crossover is to pass on the successful genes so as to get the best out of every generation. On the other hand the goal of mutation is to introduce new opportunities to succeed. Without mutation the only variance would be whatever was created in the initial generation. It is also important not to over-do crossover, one solution that was successful may need many of its components to maintain its success. Likewise too much mutation may lead to a generation too random to get the successful characteristics of the parent generation. Crossover is incredibly important and tricky. If just single points are crossed over then the success of a generation is not shared well. Instead whole substructures must be traded. This is because optimal solutions are relevant to a collection of nurses in shifts, not just the placement of a nurse.

3.7 Swap Heuristics

Swap heuristics are useful since they can be used to try and improve any schedule that already exists. This allows the swap heuristic to be used with any initial schedule constructed randomly, constructed using any greedy assignment or a schedule produced by another method. The simplest forms of swap heuristics are 2-swap or 3-swap local searches. An n-swap algorithm starts by selecting n different shifts and their related nurses in the schedule. Now, a brute force method is used to check every formation of these n nurses in the initially chosen n shifts and the formation that is the “most” valid and has the highest value is stored. Once this process is completed the stored formation is assigned to these shifts and the swap heuristic moves onto another set of n shifts until all possible sets have been checked or until stagnation is reached. Stagnation is determined by a timeout and threshold constant. A n-swap reaches stagnation if a counter is equal to the timeout constant, and the timeout counter resets if an improvement to the schedule has been made recently. Essentially an n swap using stagnation will stop if it has not improved recently enough. Since runtime of each n-swap increases factorially with the size of n, 2-swap and 3 swap are typically used. Larger values of n not only take more time but also provide diminishing returns. Additionally, since runtime grows factorially with the size of n, swaps with a high value of n will almost always use stagnation.

Chapter 4

Work completed

Creating software for the nurse scheduling problem can help aid in understanding and analyzing the nurse scheduling problem and its complications. The purpose of the complimentary software is also to compare the viability of multiple NSP methods and to analyze how constraints affect and change the problem. Given the nurse scheduling problem is not strictly theoretical, the software should also be created in a way that can be used by anyone.

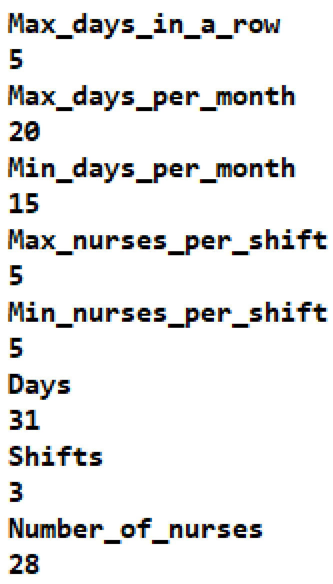
4.1 Class Overview

The congruent software for this paper is written in C++, this is largely due to the language being known for its speed. Speed is an important goal, considering the NSP is only relevant given its difficult complexity. Although none of the approximation methods have exponential runtime, the search space is still quite large for the problem and, as a result, runtime can be noticeably slower in other languages. In addition the runtime for the genetic algorithm and 3-swap function can both take over a few minutes to run even written in C++.

The software has four classes, a `main` file, and a configuration file. The classes are named `Optimize_Schedule`, `Nurse`, `Schedule` and `Hospital`. The main file contains one `Optimize_Schedule` object. Inside `Optimize_Schedule` there are `Nurse` objects equal to the number of nurses specified in the configuration file. `Optimize_Schedule` also contains one `Hospital` object which carries all the hospital constraints and a dynamic number of `Schedule` objects that are used to save schedules for future comparison. Each piece of this software has unique uses.

`config.txt` The configuration file stores features of the specific nurse scheduling problem being analyzed by the software. It meant to be modified or at least reviewed by the hospital prior to runtime. The modifiable features are less constraints about the problem and more decisions about the problem space. The modifiable variables include the number of nurses working at the hospital, the maximum and minimum number of nurses working per shift, how many days in a row a nurse can work and how many days a nurse can work per time period. Additionally the software and the default values for the configuration file assume that the scheduling period is a month, however this time period can also be changed in the configuration file along with the number of shifts that occur per day.

`Hospital` `Hospital` is a class has the job of being aware of any constraints that



```
Max_days_in_a_row
5
Max_days_per_month
20
Min_days_per_month
15
Max_nurses_per_shift
5
Min_nurses_per_shift
5
Days
31
Shifts
3
Number_of_nurses
28
```

Figure 4.1: A sample setup for the configuration file, `config.txt`.

are specific to the hospital. A constraint specific to the hospital is any constraint that occurs due to some sequence of nurses and cannot be detected by viewing a single nurse's schedule. These constraints are used to check the validity of a schedule and the score of a schedule. The purpose of the `Hospital` class is also informing `Optimize_Schedule` of the values the hospital chose in `config.txt`.

`Nurse` The `Nurse` class is used mainly to store all the constraints that are specific to nurses. The `Nurse` class has two schedules of size equal to the number of shifts over a period of days specified in `config.txt`. The first schedule stores all the times that a nurse has made a request, where a request by a nurse is a time where a nurse informed the hospital that they want to work, not to work or cannot work. The second array contains the shifts in which a nurse is assigned to work. With both of these arrays combined, a score can be calculated. Calculating a score compares all the times a nurse has made a request with whether they are working during that time.

The exact method for finding a nurse's score begins by counting the number of requests a nurse makes. That is, the number of days a nurse has requested to work or not to work is equal to their total requests. Then, once all assignments are made a count is collected of how many requests are satisfied. The score is now calculated using the following ratio:

$$\frac{\texttt{nurse requests satisfied}}{\texttt{total nurse requests}}.$$

This will always produce a number between 0 and 1. The reason for this is that all nurses should be able produce equivalent scores to each other. In other words, nurses should be given even priority to each other. A nurse in this format will have diminishing returns for each additional request made. This results in nurses with fewer requests being more likely to have their requests satisfied than a nurse who has many requests. If the `nurse soft requests` constraint is disabled, nurses will default to outputting a score of 0 automatically.

A nurse's score is between 0 and 1 only if their schedule is feasible. If a nurse is assigned to an invalid time then the score receives a penalty of $-10,000$ for each of these assignments. Total schedule scoring is discussed more in Section 4.3.2.

Another key feature of the `Nurse` class is checking a potential shift assignment against hard constraints. This is done using the `validshift` method, which returns a boolean value. This method takes as an input a given shift and then, using this information, checks this time against all the nurse-specific hard constraints. In checking constraints, the function also implements what it means to check these constraints. Constraints are checked only if they are currently turned on. The state of constraints is also reflected in the function that calculates scores.

`Schedule` The `Schedule` class stores schedules of assigned nurses using vectors. Vectors allow for the schedule to be of any size and up to three dimensions. In `Schedule` the number of dimensions is hard coded to be three simply because there is no need for a more dynamic schedule. The `Schedule` is used substantially in this software's genetic algorithm where each member of a generation is some schedule configuration. Additionally, it can be used to save and load schedules in the menu. This can be useful in comparing a few different methods for producing a solution.

The schedule vector is really a *one* dimensional vector stored in a pseudo one, two, or three dimensional vector. This is done for the sake of efficiency and at very little cost. It also allows for the schedule to be iterated through linearly. Functions in the class allow for indexing using the true one dimensional location or functions that translate two or three dimensions into one. The functions of this class are all used to get and put data to and from the schedule. There is also a variable in the schedule that stores the score of the inputted schedule for easy retrieval.

`Optimize_Schedule` The `Optimize_Schedule` class is the most substantial class in the software. `Optimize_Schedule` provides functionality for all the other classes. It links together the constraints from the nurses and the hospitals, and it attempts to create a schedule that is optimal. It mostly houses methods for creating or improving given schedules. The class interacts with three important data structures. One is a vector of `Schedule` objects which house any saved schedules. These `Schedule` objects are loaded to and from a special array inside the `Optimize_Schedule` class. It is an array of size `days` · `shifts` · `shift size`, the second important data structure. This array is a special schedule in `Optimize_Schedule` that notifies nurses to update their parallel schedules. Although `Schedule` objects hold schedules, `Optimize_Schedule` can communicate directly with `Nurse` objects and quickly assign and unassign nurses. Keeping one primary schedule allows only loading in the schedule to `Nurse` objects once as opposed to every time the schedule is modified and a score or the validity of an assignment must be calculated.

The third data structure is a vector of `Nurse` objects. Given that a `Nurse` object exists for every nurse this vector is of size equal to the total number of nurses. Additionally nurses can be referred to by their IDs which is equivalent to the index into the vector of nurses. This nurse ID is an alias for nurses since mnemonic names do not transfer well into programming. These aliases are used in schedules to denote where a nurse is assigned to work. However zero in the schedule is used to store empty positions so, since IDs are equivalent to indices and indices start at zero they are increased in the schedule by one for the purpose of visualizing the schedule.

As `Optimize_Schedule` provides communication between classes, it is also used to display schedules to the user and to calculate the objective function. Displaying a schedule is done using the `printSchedule` function, which nicely formats and displays the schedule in a way that is dynamic to the number of days, number of shifts, and shift size. The objective function in `Optimize_Schedule` is a function that sums the scores of all nurses and weights them against the score the hospital provides. This function is simple yet important for determining the quality of any given schedule and whether optimization is working.

4.2 Constraints

The NSP can vary in constraints significantly, but there are some constraints that appear more frequently than others. This software chooses a few of these more common and simple constraints to keep track of since just a few are needed for a formulation that is NP-hard and the problem does not have to adhere to the (possibly strange) requirements of a real hospital. Hard constraints for this problem are the number of days nurses can work in a row, the minimum and maximum number of days nurses can work per month, the minimum and maximum number of nurses that can work per shift, minimum rest time between shifts, balanced shifts, and days that nurses have already approved to take off. The balanced shifts constraint states that nurses must work each given shift a minimum and maximum number of times. The soft constraints used are satisfying any nurse requests to work or not work on specific days.

As a default all constraints except for the balanced shifts constraint are turned on, however all can be toggled within the main menu. In the software, constraints are used for two purposes: in calculating the value of a schedule and evaluating the validity of a nurse assignment to a specific day and shift. Constraints are stored and considered by the `Hospital` and `Nurse` classes where each class houses its own constraints respective of the other. All constraints are either relevant to the hospital as a whole or to nurses individually.

There are certain hard coded constraints that did not make sense allowing the ability to turn them off. For example certain impossibilities like the fact that a nurse cannot work more than once per shift. Specific values for certain hard constraints, such as the range of days all nurses have to work within, can be found and modified in the `config.txt` file.

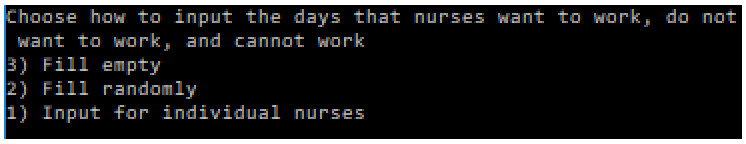
4.3 User Interface

Given the nurse scheduling problem is a real-world problem it is less abstract than many other problems in complexity theory. Because of this, it is important that software provided for the problem contain a semi-intuitive user-interface.

4.3.1 Menuing

This software allows for interaction with the problem through a simple menuing system. The system accepts numeric input that precedes the menu option that the user would like to select.

Setup Menu: Upon running the program the user must first input the days that nurses have made requests about their work. Requests by nurses are days in which nurses have previously requested to, or not to, work and the importance of fulfilling these requests. Nurse requests are the soft constraint for the problem and what is attempting to be optimized once a feasible schedule is found. There are three options for this constraint: a manual input for each nurse, a random input and giving all nurses no constraints.

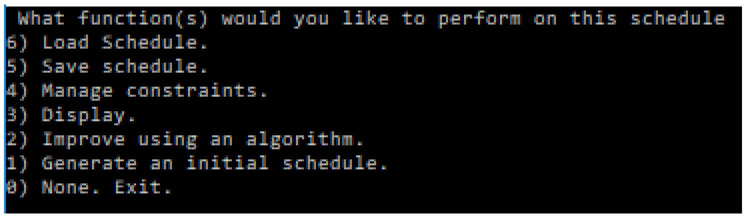


```
Choose how to input the days that nurses want to work, do not
 want to work, and cannot work
3) Fill empty
2) Fill randomly
1) Input for individual nurses
```

Figure 4.2: Setup Menu

Manual input (Option 1 shown in Figure 4.2) is the most likely to be used in a real world situation. Each nurse individually has their constraints added. This continues until either all nurses have been given their respective constraints or the user has decided to end the input early. Ending the input early can be done by filling the remaining nurses with random constraints or filling the remaining nurses with no constraints. Options 2 and 3 in the main menu (Figure 4.2) are less likely to be used than Option 1. Option 3 will only be used if the nurses are not allowed individual constraints and Option 2 is useful in testing how the software works.

Main Menu: Once this initial setup is complete the user will be presented with the main menu of the software. This menu, unlike the initial menu, is created with the intent of being returned to. It contains six menu options that can be seen in Figure 4.3, not including the option to exit.



```
 What function(s) would you like to perform on this schedule
6) Load Schedule.
5) Save schedule.
4) Manage constraints.
3) Display.
2) Improve using an algorithm.
1) Generate an initial schedule.
0) None. Exit.
```

Figure 4.3: Main Menu

Generating Schedules: This option in the main menu uses the methods that are referred to in Section 4.4 to create a schedule. Depending on the method these schedules use the schedule generated may have a good score or a score that is incredibly infeasible.

Improving Schedules: Once a schedule is created, it may be improved using any of the methods in the menu obtained by entering 2 while on the main menu (seen in Figure 4.3). These methods can only be run provided that the current schedule is not empty.

Display Options: This menu is used to store all the display choices for the current schedule. This includes **Display score** which prints the score of the current schedule. This allows for users to compare the quality of a schedule or allows users to see how a schedule has improved. This score is 0-100 if feasible and negative if infeasible.
Another function is **Display schedule** which prints and formats the schedule in a way that is easy for users to decipher. The schedule

displays the nurse ID in shift positions that a nurse is working and displays the "-" symbol when no nurse is working.

Toggle Debug, toggles print statements that may appear when other menu options are chosen. This text is called debug text and it is output that describes what is happening while a function is running. It is useful for fixing problems in the code or showing how the code is working. By default this is set to off.

Constraint Options: This menu allows the user to view whether constraints are turned on or off. It also allows the user to toggle any of these constraints. By default all constraints are set to be on except the balanced shifts constraint.

Saving/Loading Schedules: In order to test different methods for creating schedules this software allows for schedules to be saved. Selecting Option 5 in the main menu will automatically save the current schedule and notify you of the current schedule's ID which can be used to retrieve it in the future.

Loading in a schedule overwrites the current schedule and, unless saved, will delete it forever. Choosing a schedule to load is done through the use of ID numbers that are assigned to schedules. These ID numbers are displayed to the user when a schedule is saved but will also be listed alongside the scores of each currently saved schedule immediately prior to loading a schedule.

4.3.2 Scoring a Schedule

When evaluating the quality of a schedule an objective function is used to provide a score. This score is somewhat abstract in the sense that a score does not have any tangible equivalent. It is however still useful in describing the value of schedules in relation to each other. The score nonetheless is presented to the user in order for them to get an understanding of schedule variability and relativity. Deciphering a score can be challenging for any users who are new to the software. In order to get a better understanding of what a score means, it may help to understand how it is created.

When a score is positive, it is between 0 to 100, and the schedule is feasible. This is true no matter the number of nurses in the schedule. First note that each nurse will return to the objective function a sum of 0 to 1. The specifics for doing this is in section 4.1. Therefore, the sum of all the nurse scores divided by the sum of best possible scores of all nurses (equal to the total number of nurses) will produce a number between 0 and 1. Multiplying this by 100 now produces a number between 0 and 100

Provided all constraints are being used it is likely that no schedule exists that has a score of 100. But the exact highest score can not be known for sure without a thorough check using the brute force method. This is a result of the NSP being NP-hard. The score of 100 is achieved if all soft constraints are satisfied, but there are known setups in which soft constraints sometimes cannot be satisfied. For example, if more nurses want to work a shift than are allowed to then least one nurse will not get what they want.

If the score is negative, then the schedule is infeasible. This means at least one hard constraint is defied by the schedule. By the definition of hard constraints, any schedules that violate them should be rejected. In practice however, there are benefits from thinking of infeasibility as a spectrum. There are two cases where this is useful. The first is comparing two infeasible schedules to choose. It is ideal if there is a way to to select the one which is the “least” infeasible. This might need to be done in cases where no feasible schedule exists or at least no feasible schedule can be found by the software. The second case occurs when trying to improve an infeasible schedule. The methods implemented in the software benefit from knowing whether actions preformed on a schedule are moving it towards violating fewer hard constraints.

In order to make infeasibility a spectrum, hard constraints are set in a way that they have large enough negative scores such that no infeasible schedule can compare to the quality of a feasible schedule. The penalty in this software is a value of $-10,000$ for each hard constraint defied. Using this information, if a score is rounded to the nearest $10,000$, dividing by $-10,000$ gives the total number of hard constraints defied.

4.4 Implemented Methods

Implemented methods include a genetic algorithm, four different types of swaps and three greedy assignment methods. These algorithms, although described in Chapter 3, have traits that are specific to the nurse scheduling problem and this software. For a more general view of these methods see Chapter 3.

4.4.1 Swaps

2-swap: This algorithm that chooses two positions in the schedule and sees if swapping the nurses in those positions will improve the schedule. The algorithm then chooses two new positions in the schedule until all swaps have been checked. If a swap occurs then there is a chance that new opportunities for the schedule to improve have opened up. Therefore the 2-swap function will run again until it has ran through completely without any improvement to the schedule

3-swap: The 3-swap algorithm chooses three positions in the schedule and tries all new combinations of the swapping the nurses in these positions. It works a little differently than the 2-swap. In this software, it is made to be preformed after the 2-swap is complete. This is because the standard 3-swap includes swaps where three objects are chosen but one remains in the same place. This is equivalent to a 2-swap and thus the 3-swap function in this software only tests swaps in which all three nurses end up in a new position in the schedule. There are only *two* orderings of *three* objects such that all objects are not in their initial position. Now, given the nurses a, b and c is order a, b, c, the 3-swap only has to test the orderings b, c, a and c, a, b since all other orderings keep a, b or c in their original positions and would thus be covered by a two-swap.

Swap to feasible: This swapping algorithm that is only looking to make swaps if they remove hard constraints. The appeal of doing this is to create a schedule that is feasible for functions that are not the two- or three-swap. It is run like a two-swap by first choosing a position in the

Shift	Nurse 1 Scores	Nurse 2 Scores	Nurse 3 Scores
1	1	0	0
2	2	1	0
3	1	1	0

Table 4.1: Shown above is a table in which three nurses are assigned to three shifts (assignment is denoted by blue boxes). Whichever shift a nurse is assigned to produces a score equal to the "Nurse Score" column. This table shows a setup in which which swapping any two nurses will not improve the schedule (2-swap) but swapping all three nurses will (3-swap). This is certainly not the only example of this situation.

schedule in which a hard constraint is defied and then asking all other nurses to swap with the nurse in that position. In case no feasible schedule exists there exists, a timeout counter that will ensure the method does not run forever.

Swap for minimum: This swap should be used prior to other swaps. It attempts to solve any violation of the hard constraints related to nurses working the minimum or maximum number of days. The other swap methods will never be able to solve this hard constraint since the other swap methods will always result in nurses working the same number of total days. This function first finds all nurses that have not worked enough or that have worked too many days. Then, if a nurse worked too much then all other nurses are asked to take each shift from that nurse until the nurse no longer exceeds the maximum number of days allowed to work. On the other hand, if a nurse worked too little then that nurse asks if it can take every shift in the schedule until it works the minimum requirement. For similar reasons as the swap to feasible method, swap for minimum has a timeout counter.

4.4.2 Initial Assignment Methods

Three types of simple assignments are implemented in the software with the purpose of creating an initial assignment for other methods such as swaps and genetic algorithms to improve on. There are three initial assignments in this software. All these assignments try and assign nurses only if the assignment does not defy hard constraints. If no nurses can be validly assigned then a random one is chosen to be assigned. The order in which nurses are placed in the schedule is what separates the three assignments.

The first assignment method is greedy, and it attempts to assign nurses in order of their nurse IDs. For example, at every position the function will first try and assign Nurse 1; If Nurse 1 cannot work then the function will try Nurse 2. This iterates through all nurses until one is found that can work or until all nurses cannot work. If all nurses cannot work then one is assigned to work anyway. This assignment unfortunately will often create infeasible schedules and schedules in which nurse 1 is far more likely to work than nurses with large nurse IDs.

The second assignment function is created virtually identically to the first except for one key difference. The function keeps track of the last nurse that was successfully assigned, and the next nurse that the function will try to assign has a nurse ID 1 greater than the previous. In doing this nurses will be more evenly assigned. This assignment can more frequently produce a feasible schedule than the previous one.

Although the previous assignment method described creates a schedule that is unlikely to defy constraints, one more assignment method should be used for the genetic algorithm. Genetic algorithms exploit randomness to improve schedules. Therefore the third initial assignment function is a pure random assignment. This function is used to seed the initial generation in the genetic algorithm.

4.4.3 Genetic Algorithm

The genetic algorithm (GA) is the most complicated of the algorithms. For the theory behind GAs, view its respective section in Chapter 3. This specific algorithm begins by calling a main GA function which has the purpose of containing all the calls to iterate generations and considering stopping conditions. Upon calling the main GA function, a call is immediately made to an intialize function that creates two "generations" for future use. These generations are vectors of `Schedule` objects, and the GA initialization function also formats the `Schedule` objects to be the appropriate size. Additionally, one of these two generations is filled with randomly assigned schedules. This filled generation is currently active generation, while the unfilled generation will be filled with children.

Initially only two generations are stored, this is because there are only ever two generations that are in use at any time, a parent and a child. Once the creation of a child generation is finished then the parent generation is no longer in use. Therefore when the child generation becomes the parent to another generation this newest generation can be placed where the original "grandparent" generation was stored as it is no longer in use.

When a child generation is being created all genetic algorithms use parent crossover. Parent crossover chooses two parents and creates a child that has features from both parents. This is done for every member of a generation. Let us call the two parents `parent 1` and `parent 2`. `parent 1` is the primary parent and because of this will give more traits of their schedule than `parent 2`. `parent 2` can give up to 50% of their traits to their children but no more. If `parent 2` could give more than 50% of their traits then they could equivalently be called `parent 1` since both parents were chosen the same way.

Choosing parents should be done in such a way that all parents can be chosen but those valued higher should be more likely to be chosen. This weighting is done by first checking all the parent scores and if they are negative dividing them by 2000. This is because if a score is negative then for each hard constraint it has defied it receives a penalty of $-10,000$. Thus for each hard constraint defied the score now only has a negative score of -5. In doing this scores that are feasible can be differentiated in quality. The difference between negative and positive scores previously made it so that the difference in a score of 0 and a score of 100 was dwarfed by the difference in a score of $-10,000$. Once the negative scores have been modified the lowest score is found and a new variable is made which is the lowest score made positive and increased by 10 units. This new variable is then added to all of the scores previously modified. This ensures that all scores are now positive, all differences in score are maintained and the increase of 10 units means that the modified lowest score is 10 units instead of 0. Now, sum all the modified scores and randomly choose a number from 0 to the sum. The modified scores now are made into partitions of size equal to the modified score.

The next step in this genetic algorithm is called mutation. In this step certain nurses in a schedule are changed into something else randomly. This genetic algorithm uses two kinds of mutation: random swaps and random changes. Random swaps choose two different nurses in the schedule and attempt to switch their positions. Random changes chooses a nurse in the schedule and attempt to randomly choose another nurse and place it where the original nurse was stored. The random swap is more useful in later generations when the hard constraint of the number of days working is satisfied. However if this is not satisfied then a random swap could never fix this. Thus a random change fixes this problem. Although pure randomness is a trait of most GAs, once the worst schedule is feasible this specific GA will not mutate members of its schedule if this mutation would lead to an infeasible schedule.

Once the children are all created the generation replaces the worst children with the best parents. This will assure that the best member of a generation is at least as good as the best member of the previous generation. Moreover the overall quality of a generation will also only improve.

The next generation will then be created until a stopping condition is found. This software uses number of generations as the stopping condition. If the stopping condition is met then the best member of a generation is loaded into `Optimize_Schedule` and the rest are deleted.

4.5 Evaluation of Work

This software provides a few select methods for approximating an optimal solution the nurse scheduling problem. These methods are varied enough that they show a few different processes. There are many more algorithms that could be applied.

4.5.1 Evaluation of Methods

Assignment algorithms These algorithms are incredibly fast as they assign once and in a linear day by day manner. They thus have runtime $O(DSP)$, where $D =$ `days`, $S =$ `shifts per day`, and $P =$ `nurses per shift`. This typically can be done almost instantly for $D = 30$, $S = 3$, $P = 5$.

2**-swap algorithm:** The 2-swap algorithm chooses every combination of nurses held in every combination of two shift positions. As such it loops through the schedule twice, once for each shift position. Thus the runtime for this algorithm is $O((DSP)^2)$, where $D =$ `days`, $S =$ `shifts per day`, and $P =$ `nurses per shift`. This typically takes just under a second to run for $D = 30, S = 3, P = 5$.

3**-swap algorithm:** The 3-swap algorithm chooses every combination of nurses held in every combination of three shift positions. As such it loops through the schedule three times, once for each shift position. Thus the runtime for this algorithm is $O((DSP)^3)$, where $D =$ `days`, $S =$ `shifts per day`, and $P =$ `nurses per shift`. This typically takes a minute and a half to run for $D = 30, S = 3, P = 5$. This is expected since

2-swap takes less than a second and 3-swap runs for $D \cdot S \cdot P = 450$ times longer ($(DSP)^3$ versus $(DSP)^2$). The 3-swap produces the best score out of all methods in this software. The improvement from 2-swap to 3-swap is minimal but consistent.

Genetic algorithm: The complexity of the genetic algorithm is quite complicated to calculate, but for 75 generations and a population size of 75 the genetic algorithm takes about a minute to run. This algorithm produces a score that tends to be worse than the score of either the two or three swap but is better than the scores produced with greedy initial assignment heuristics. Through tweaks in this genetic algorithm, I believe it is possible to get score up to or equal to the 2-swap. In addition although the genetic algorithm has sub par performance, unlike swap algorithms, it can be ran repeatedly and always have a chance to continually improve the schedule, provided the global optimum has not yet been reached.

4.6 Future Work

Currently the software provides a sufficiently well-established base of code such that constraints and optimization methods can be added easily on top of it. In terms of constraints, many more can be added. Currently the constraints chosen are just enough for the problem to be NP-complete. This allows for an analysis of certain solution methods but not a robust analysis of differing constraints. Future software should have many more constraints such that complexity of the nurse scheduling problem can be analyzed as it relates to constraints. One example of a common constraint not implemented in the software is some constraint that involves nurse attributes. An attribute could be something like salary, special skills, or work experience. Salary may affect the schedule score through minimization where experience could increase a scores value if it is evenly spread out between shifts.

In addition to adding more common constraints it would also be interesting to allow for users to "create" constraints. This could allow the software to be significantly more versatile. An example of these constraints could be disallowing or encouraging specific "patterns" in the schedule, where a pattern is a sequence of nurses over a set of shifts. Some patterns that could be created from this are disallowing nurses to work weekends more than once in a row, or making it so the schedule receives a bonus to its score if nurses tend to work the same shift(s) every week. Attributes could also be abstracted to allow users to input their own attributes and decide how they want them to affect the schedule score. This addition to software would have the upside of customizability at the cost of the software being quite complicated. This complication arises both in creating the software and also in users needing to thoroughly understand the software to use it to its full potential.

New variations on constraints can be added virtually without end. Likewise, the same can be done for methods that solve the nurse scheduling problem. There are over a few dozen methods discussed in this paper and its references, and each of these has a substantial number of variations and combinations with each other. In addition to these methods, hundreds more exist for optimization and NP-hard problems in general that can all be applied to the NSP in a relatively successful manner.

Although this software is largely for the purpose of understanding the nurse scheduling problem it could be further developed to have applications in real hospitals. In doing this it would need a better user interface. There are obvious limits to a full text based user interface, and future software could do away with these problems. Although hospitals could use a menuing system that accepts numeric input, it is much more intuitive and much nicer to view through some graphical interface. This is especially true in the case of printing schedules in a way that is easy to understand. This software can be built substantially with both the goal of creating a software for real hospitals or to create.

Bibliography

[1] M. Affenzeller and R. Mayrhofer. Generic heuristics for combinatorial optimization problems. 2002.

[2] L. Bottaci. *A Genetic Algorithm Fitness Function for Mutation Testing*. 2001.

[3] E. Burke, P. D. Causmaeker, and G. V. Berghe. A hybrid tabu search algorithm for the nurse rostering problem. pages 187–194, 2003.

[4] E. K. Burke, P. D. Causmaecker, G. V. Berghe, and H. V. Landeghem. The state of the art of nurse rostering. *Journal of Scheduling*, 7, pages =, 2004.

[5] M. Choy and M. Cheong. A greedy double swap heuristic for nurse scheduling. *CoRR*, abs/1205.2200, 2012.

[6] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[7] T. Cooper and J. Kingston. *The Complexity of Timetable Construction Problems*. Basser Department of Computer Science, 1995.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. 1989.

[9] S. Hartog. On the complexity of nurse scheduling problems. 2016.

[10] H. Jafari and N. Salmasi. Maximizing the nurses' preferences in nurse scheduling problem: mathematical modeing and a meta-heuristic algorithm. pages 439–458, 2015.

[11] R. Karp. Reducibility among combinatorial problems. 1972.

[12] R.E. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22:155–171, 1975.

[13] H. C. Lau. Manpower scheduling with shift change constraints. *Algorithms and computations*, pages 616–624, 1995.

[14] J. Y-T. Leung. *Handbook of Scheduling*. CRC Press, 2004.

[15] L. A. Levin. Universal sequential search problems. *Probl. Peredachi Inf*, 9(3):115–116, 1973.

[16] B. Maenhout and M. Vanhoucke. *An electromagnetic meta-heuristic for the nurse scheduling problem*. pages 358–385, 2007.

[17] H. Miller, W. Pierskalla, and G. Rath. Nurse scheduling using mathematical programming. *Operations Research*, 24(5):857–870, 1976.

[18] M. Mitchell. *An Introduction to Genetic Algorithms*. 1999.

[19] M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2006.