A Comparison of Approaches to the Nurse Scheduling Problem

Operations Research II Group 4
Yijing Chen, Andrew Liu, Elizabeth Sciannella, Alice Zhang

December 2016

1 Introduction

We were initially interested in the optimization of hospital functioning. Hospitals are often crowded and unable to optimally serve patients, thus nurses and doctors often have to work undesirable shifts. These less-than-ideal working conditions may, in part, contribute to a high nurse turnover rate, which can potentially impact the average hospital's profits by $300,000 per year [1]. One way to improve hospital functioning is to find more optimal schedules of the nurses' shifts.

Typically, a hospital staff member is assigned the task of scheduling the nurses' shifts by hand, or nurses schedule their own shifts, also by hand. The hospital staff may need to frequently create new schedules to account for holidays, changes in nurses' availability, or changes in nurses' preferences. This process is time-consuming to do manually because the Nurse Scheduling Problem (NSP) is NP-hard; there are many thousands of possible schedules, and the staff members may even be unable to determine whether or not there exists a solution at all. An operations research approach to nurse scheduling would allow hospitals to make better use of these staff members' time. A computer-generated schedule may also better satisfy nurses' preferences and more efficiently utilize hospital resources.

Hospitals and other industries are already taking steps toward using computers to improve the scheduling process. The market for scheduling programs has been growing over the past few years. Some of these programs use Knapsack Solving and Tabu Search, e.g. Computer Aided Rostering Environment (CARE) [2], while other programs are collaborative schedulers that ask the nurses to sign onto the program and determine their shifts along with all the other staff members of the hospital department (e.g. NURSEGRID) [3].

These programs apply operations research ideas to the NSP with the goals of minimizing the creation time of the schedules and improving schedules for the nursing staff and the hospital overall. With these goals in mind, in this project, we aim to evaluate two brute-force approaches to the NSP, Integer Programming and Knapsack, and a heuristic approach, Tabu Search, by comparing their run times and proximity to optimality. We found that for problems with varying numbers of nurses, Tabu Search could find near-optimal solutions reasonably quickly, while the brute-force approaches were already inconveniently slow for problems with around 20 nurses.

1.1 Example Nurse Scheduling Problem

Suppose a small hospital is open 9 AM to 6 PM every day and requires the following number of nurses for every hour of operation:

Hour	9 AM	10 AM	11 AM	12 PM	1 PM	2 PM	3 PM	4 PM	5 PM	6 PM
Required	2	2	2	2	3	3	3	3	2	2

and the following nurses are available on staff, with required hours:

Nurse	Minimum Hours	Maximum Hours	Hourly Wage
Nurses 1	6	8	20
Nurses 2	6	8	30
Nurses 3	6	8	35
Nurses 4	6	10	50

One possible solution that minimizes cost can be represented as follows, where a "1" indicates that the nurse is on shift:

Hour	9 AM	10 AM	11 AM	12 PM	1 PM	2 PM	3 PM	4 PM	5 PM	6 PM
Nurse 1	1	1	1	1	1	1	1	1		
Nurse 2					1	1	1	1	1	1
Nurse 3					1	1	1	1	1	1
Nurse 4	1	1	1	1	1	1				

A simple model like this can be hand-computed in little time, but in a real-world example, hospitals are large and have a large number of nurses, and it takes significantly more effort to determine a feasible solution, let alone an optimal solution.

2 Methods

The NSP can be formulated as an assignment problem, where the aim is to find a minimal-cost assignment of nurses to shifts, given hard and soft constraints. Hard constraints are constraints that must be met in order for the solution to be considered feasible. In contrast, soft constraints are not required for feasibility, but are associated with a penalty incurred to the solution. Because soft constraints are merely incorporated into the objective function in the form of a "cost," we largely ignore soft constraints, and instead focus on hard constraints. In the formulation of our model, we also make the assumption that nurses are interchangeable with one another. In most hospitals, nurses' job requirements are typically the same throughout a department, and in the case that a nurse in unavailable, the hospital is set up so that there are other nurses who can take over. So, we believe this is a reasonable assumption.

The hard constraints we focus on are:

1. The hospital requires a certain number of nurses to be on shift for each hour
2. Each nurse must work more than a given minimum number of hours and less than a given maximum number of hours per day
3. No nurse can cover more than one shift at a time

Some definitions:

- A **shift** is a block of time that represents a minimal period of work for a nurse. For example, in a setup with three work periods, the shifts may be from 9 AM - 5 PM, 5PM - 1 AM, and 1 AM - 9PM. In our setup, we assumed that nurse scheduling was very flexible, so we consider a shift to be a 1-hour block, for each hour in a day. (For simplicity, we schedule one day, but our algorithm/code can easily be scaled to schedule a longer time frame.)
- A **pattern** is a vector that represents a proposed work schedule for a single nurse. A pattern has one component for each shift; the ith component of the pattern vector is 1 if the pattern schedules the nurse to work shift i, and 0 otherwise.

2.1 Equations

Given n nurses, let $N_1, ..., N_n$ represent the nurses from 1 to n.

Given m shifts, let R be a vector of length m representing requirements such that $R_1, ..., R_m$ represent the required nurses per shift.

Given k_i patterns for nurse N_i:

- Let $p_{i1}, ..., p_{ik}$ represent the feasible patterns for that nurse, such that each p_{ij} is a vector of length m.
- Let $c_{i1}, ..., c_{ik}$ represent the penalty associated with the corresponding pattern.

Then our system can be expressed as follows:

Optimize

$$\sum_{i=1}^{n} \sum_{j=1}^{k} c_{ij} p_{ij}$$

such that

$$\sum_{i=1}^{n} \sum_{j=1}^{k} p_{ij} = R$$

$$\sum_{j=1}^{k} p_{ij} = 1$$

p_{ij} is integral

Thus, this formulation incorporates the hard constraints outlined on the previous page. Hard constraint (1.) is represented by the first constraint in this system, hard constraint (2.) is satisfied because only feasible patterns p_{ij} are considered for each nurse, and hard constraint (3.) is represented by the second constraint in this system.

2.2 Algorithms

2.2.1 Integer Programming

The model can directly represent a mixed-integer programming problem, with p_{ij} being the problem's indicator variables. Once the model has been formulated, it can be submitted into a linear programming solver and solved to optimality.

2.2.2 Knapsack

The model can also be formulated as a Multiple-Choice Knapsack Problem, where the knapsack represents the requirements, and it can be "filled" by selecting one pattern from each nurse. The model can then be solved to optimality using dynamic programming like the 0-1 Knapsack Problem.

2.2.3 Tabu Search

Instead of optimizing, Tabu Search can be used as a "satisficing" method for determining solutions. That is, in exchange for not solving to optimality, Tabu Search finds *decent* solutions much more quickly than a to-optimality method for larger-scale problems.

Tabu Search is an incremental search algorithm, using the same concept as Local Neighborhood Search to find feasible solutions. Given an initial solution, the algorithm looks for solutions that differ only by a single value, and evaluates whether or not these solutions are an improvement upon the original. The search then accepts the best solution out of these neighboring solutions and repeats the process.

The primary drawback of standard incremental search algorithms is their tendency to get stuck at local optima; a local neighborhood search cannot identify better solutions if they exist beyond its line of sight. In contrast, Tabu Search is resilient to this near-sightedness. If a search iteration finds a solution at a local optimum, the Tabu Search algorithm is permitted to continue searching beyond the local optimum by using a list of allowed and disallowed options (called *tabus*).
Pseudocode:

```
Create a tabu list T.
Formulate an initial solution S.
While the stopping criterion has not been reached:
    If S is a local optimum:
        Add S to T.
    Collect the neighboring solutions of S acceptable to T.
    Set S to be the best solution out of the collected solutions.
Return S.
```

Tabu Search comes with several shortcomings, however. For instance, Tabu Search requires an initial solution. It is not hard to determine that a system has a feasible solution, but it can be impractical to find an initial feasible solution. Some utilizations of Tabu Search either have a randomized algorithm to determine an initial feasible solution or start Tabu Search using a known initial feasible solution. However, when these options are impractical or impossible, we can start from an infeasible solution. If Tabu Search is given a well-defined heuristic, it can find a feasible solution regardless.

The heuristic itself can be problematic. The penalty defined in optimization algorithms must be considered as part of the heuristic in Tabu Search, and an alternative heuristic must be used if a pattern is infeasible. The heuristic must carefully weigh the penalty of feasible solutions so as not to accept infeasible solutions that would otherwise produce a low penalty. In our implementation, we reward infeasible patterns if they are close to a feasible solution, and we ensure that the best infeasible pattern is worse than the worst feasible solution.

2.3 Code

We used the Java programming language to implement solutions to the Nurse Scheduling Problem, using the following interfaces to represent our data:

```
public interface Pattern extends Iterable<Integer> {
    String UNIT = "Hour";
    int SIZE = 24; // Can be changed; 24 here represents hours per day
    int get(int index);
    void set(int index, int value);

    void add(Pattern addend);
    Pattern plus(Pattern addend);
    void subtract(Pattern subtrahend);
    Pattern minus(Pattern subtrahend);
}

public interface Nurse {
    boolean hasFeasible(Pattern pattern);
    double getPenalty(Pattern pattern);
    public Iterable<Pattern> patterns();
}

public interface Solver {
    Map<Nurse, Pattern> solve(List<Nurse> nurses, Pattern requirements);
}
```

Extending the definition of "pattern" given earlier, here we define a `Pattern` as a vector-like data structure to represent an allocation of nurses to shifts. Each index of the `Pattern` represents a shift, and the value at each index represents the number of nurses working that shift. A `Pattern` can be used to represent either an individual nurse or the total nurse requirement. For a `Pattern` that represents an individual nurse's feasible pattern, we expect that pattern to have values of only 1 or 0, but for a `Pattern` representing the requirement R, we expect the `Pattern` to have many nonzero entries.

We define a `Nurse` as a data structure that operates on `Pattern` instances. Given a particular `Pattern`, the `Nurse` can identify whether the `Pattern` is feasible for itself and/or what penalty the `Pattern` has. In addition, the `Nurse` can generate all of the `Patterns` feasible for itself, which the brute-force solvers use. In our implementation, we have types of `Nurses` that determine a `Pattern`'s feasibility if the `Pattern`:

- is a single interval of time (e.g., "1 PM - 6 PM," but not "1 PM - 3 PM, 4 PM - 6 PM")
- sums to a particular value (quota)
- is in an certain `Set` of `Patterns` (in the real world, these will depend on what hours a nurse can work)

By using this interface, we can create `Nurse` instances that satisfy arbitrary constraints that can be used to solve the model.

Then, we can define a `Solver` as a data structure that operates on a `List` of `Nurse` instances and a `Pattern` representing requirements, which solves the problem. We implemented a `LinearSolver`, `KnapsackSolver`, and `TabuSolver`, matching the approaches to the problem as described above.

3 Results

The algorithm implementation was executed in Eclipse version Neon using single-threaded versions of the algorithms on a computer running Windows 10 with an Intel Core i5-3230M processor. We ran the algorithms on several cases with varying numbers of nurses: 4 nurses as a simple test for correctness, and 20 nurses as a standard-difficulty NSP problem. We ran the Integer Programming and Knapsack algorithms to completion 10 times per problem, using the mean of the time taken per algorithm. Meanwhile, Tabu Search was run using a stopping criterion of 30 seconds of run time.

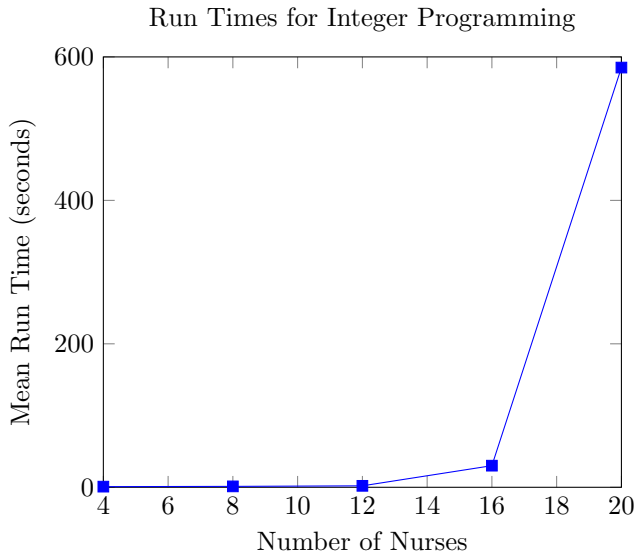
We found that for the four-nurse case, both the Integer Programming and Knapsack methods generated the optimal solution in less than one second. The Tabu Search algorithm returned a suboptimal solution, although the solution was only slightly worse than the optimal one. However, in the twenty-nurse case, the Knapsack method failed to finish within an hour, while the Integer Programming method took approximately 10 minutes on average. The Tabu Search again returned a slightly suboptimal solution.

These results are summarized in the figures below:

Case: Four Nurses		
Method	Value attained	Time taken
Integer Programming	850 (optimal)	0.90sec
Knapsack	850 (optimal)	0.35sec
Tabu Search	880	[30 sec]

Case: Twenty Nurses		
Method	Value attained	Time taken
Integer Programming	4670 (optimal)	585.1sec
Knapsack	Did not finish	Did not finish
Tabu Search	4945	[30 sec]

We further tested the Integer Programming method to see how the run times grew with the number of nurses, and the results follow:



It is not surprising that for larger cases (i.e., more nurses), the Tabu Search method is the only method among the three that can consistently produce near-optimal solutions in a reasonably short

amount of time. Each n nurse has potentially 2^m patterns, so the asymptotic complexity of brute-force algorithms (Integer Programming and Knapsack) is $O(2^{mn})$. In other words, the problem becomes exponentially longer to process with each new nurse. However, this is not to say that the other methods should be disregarded. In small cases with few nurses, using the Integer Programming or Knapsack methods may yield better solutions more quickly than Tabu Search. In addition, some facilities may opt to choose a slower method if the method can present a better solution than a faster method can.

4 Discussion

We learned that the best algorithm to use may depend on the staff size of the hospital—smaller staff sizes would benefit from utilizing the integer programming method, whereas larger staff sizes would benefit from utilizing the Tabu Search method. The integer programming method will produce an optimal solution, however, the long run times would be disadvantageous for large hospitals. Using the Tabu Search method for a short amount of time may not produce the optimal solution, but it is still reasonable compared to the long run-time of the integer programming algorithm.

We have only run our code with the hard constraints that the pattern must be an interval with a length between a given minimum and maximum. However, there are many other hard constraints we could easily incorporate into our code framework by adding more specifications for which `Patterns` are feasible and infeasible for each `Nurse`. For example, we could incorporate times during the day when certain nurses would be unavailable to work. If we scheduled over a longer period of time, we could also add the constraint that a nurse cannot work a night shift followed by a morning shift, for example, since this would lead to the nurse being overworked. Hard constraints such as these could be easily incorporated into our code. There are also some logistical changes we could make to adapt to how different hospitals' scheduling works. This may include incorporating an optional hour-long break to the nurses' shifts. Or, we could add a classification of nurses by their type and/or seniority at the hospital, with different requirements for different classes of nurses. Finally, we could also take greater consideration of soft constraints, such as how to determine how much penalty to assign for violations of different nurse preferences.

In the future, this project could be extended to include the testing of other incremental algorithms, such as the genetic algorithm, another common algorithm used to approach the Nurse Scheduling Problem. In particular, it may be interesting to compare a genetic algorithm with Tabu Search for large (50+ nurse) problems because for significantly large problems, Tabu Search may fail to find a global optima. Tabu Search does a fair job of keeping solutions away from local optima, but with the sheer size of the Nurse Scheduling Problem's domain and the limits to the length of the Tabu List, it is still possible to *cycle* through a set of solutions without improving, since the algorithm accepts poorer solutions at local optima. However, it might be possible to avoid this issue by developing a better set of conditions for the Tabu List. So, that may be an area in which to improve our Tabu Search code for larger problems. We may also look into finding an initial feasible solution non-incrementally for Tabu Search.

In summary, this project tested the effectiveness of Integer Programming, Knapsack, and Tabu Search in optimally scheduling nurses. We found that the nurse staff size is a heavy factor in determining which algorithm is best to use to solve for a schedule. With an increasing nurse staff size, Tabu Search will provide a near optimal solution in a reasonable time frame. Looking forward, we designed our code to be highly adaptable to solving many variations of the Nurse Scheduling Problem.

5 Acknowledgements

We were kindly sent a data set from Jen Brunner, a Health Care Operations Professor at Universität Augsburg, who co-wrote a paper on Physician Scheduling with Jonathan F. Bard and Rainer Kolisch in 2009 titled "Flexible Shift Scheduling of Physicians." They were happy to share the data that they used, which came from a German hospital [5].

References

[1] Hunt, Steven. "Nursing Turnover: Costs, Causes, & Solutions." *SuccessFactors, Inc.* 2009. https://www.nmlegis.gov/lcs/handouts/LHHS%20081312%20NursingTurnover.pdf.

[2] "Care - Computer Aided Rostering Environment." *goweralg.co.uk.* Accessed November 22, 2016. http://www.goweralg.co.uk/care/caredb.htm.

[3] "NurseGrid - Nurse Scheduling, Staffing, and Communication Tools" *nursegrid.com.* Accessed November 22, 2016.

[4] Dowsland, Kathryn Anne, and Jonathan Mark Thompson. "Solving a nurse scheduling problem with knapsacks, networks and tabu search." *Journal of the Operational Research Society* 51, no. 7 (2000): 825-833.

[5] Brunner, Jens O., Jonathan F. Bard, and Rainer Kolisch. "Flexible shift scheduling of physicians." *Health Care Management Science* 12, no. 3 (2009): 285-305.

[6] Curtois, Tim. "Shift Scheduling Benchmark Instances." *Nottingham.ac.uk.* Accessed November 18, 2016. http://www.cs.nott.ac.uk/ psztc/NRP/.

[7] Augustine, Elizabeth, Morgan Faer, Andreas Kavountzis, and Remma Patel. "A Brief Study of the *Nurse Scheduling Problem (NSP).*" (2009).

[8] Gondane, Mudra S., and D. R. Zanwar. "'Staff Scheduling in Health Care Systems." *IOSR Journal of Mechanical and Civil Engineering* 1 (2012): 28-40.

[9] Shure, Loren. "Generating an Optimal Employee Work Schedule Using Integer Linear Programming". *MATLAB Central Blogs.* 2016. https://blogs.mathworks.com/loren/2016/01/06/generating-an-optimal-employee-work-schedule-using-integer-linear-programming/.