Dynamic Programming with Approximation Function for Nurse Scheduling

Peng Shi, Dario Landa-Silva

School of Computer Science, ASAP Research Group
The University of Nottingham, United Kingdom
{psxps4, dario.landasilva}@nottingham.ac.uk

Abstract. Although dynamic programming could ideally solve any combinatorial optimization problem, the curse of dimensionality of the search space seriously limits its application to large optimization problems. For example, only few papers in the literature have reported the application of dynamic programming to workforce scheduling problems. This paper investigates approximate dynamic programming to tackle nurse scheduling problems of size that dynamic programming cannot tackle in practice. Nurse scheduling is one of the problems within workforce scheduling that has been tackled with a considerable number of algorithms particularly meta-heuristics. Experimental results indicate that approximate dynamic programming is a suitable method to solve this problem effectively.

Keywords: Markov Decision Process, Approximate Dynamic Programming, Nurse Scheduling Problem

1 Introduction

The 1956 paper by Richard Bellman [1] described the *principle of optimality* and *dynamic programming*, the algorithm based on this principle. Basically, dynamic programming breaks an optimization problem into simpler sub-problems that can be solved recursively and it has been applied to a variety of optimization problems [2]. The nurse scheduling problem is a complex combinatorial optimization problem that consists in assigning shifts to nurses in each day of a given planning period (usually a number of weeks). This problem has been investigated to a considerable extent and many algorithms have been proposed to solve it [3]. Constructing high-quality schedules for their nurses are crucial for many hospitals because minimizing salary costs and maximizing the overall satisfaction of nurses with their working patterns are key aims. In the context of workforce scheduling, it appears that dynamic programming algorithms have been applied only on small problem instances [4].

The limitation of dynamic programming to tackle larger workforce scheduling problems lies on the curse of dimensionality, which means that the search space grows exponentially as the input size increases. This means that implementations of dynamic programming require too much memory to store the search space and

too much computation time for evaluating all states of the search. To make the search more efficiently, approximate dynamic programming selects only a small part of the search space based on approximation functions [2]. The solution obtained by approximate dynamic programming is close to the optimal solution and the computational time is decreased considerably.

This paper investigates the ability of approximate dynamic programming to solve nurse scheduling problems. There seems to be no previous papers conducting such investigation. The intended contribution of this work is to propose a methodology to solve larger workforce scheduling problems to near-optimality in practical computation time using approximate dynamic programming. The structure of this paper is organized as follows. Section 2 reviews the literature on approximate dynamic programming and nurse scheduling. Section 3 describes how the nurse scheduling problem can be seen as a typical Markov decision model. Section 4 describes the proposed methodology and Section 5 discusses the experimental results. Conclusions are given in Section 6.

2 Literature Review

This section consists of two parts. Some background on the nurse scheduling problem is given firstly and then followed by an overview of the approximate dynamic programming technique.

2.1 Nurse Scheduling Problem

The Nurse Scheduling Problem (NSP) can be grouped into two categories: cyclic scheduling problem and non-cyclic scheduling problem. In cyclic scheduling, nurse shift preferences are not considered so assigned shift patterns can be replicated across planning periods. In non-cyclic scheduling, nurse shift preferences are considered which limits such replication of shift patters. Surveys such as [5, 6] provide detailed reviews of terminology, constraints and objectives. In this paper we tackle the non-cyclic NSP. In this type of NSP, hard constraints are the shift requirements for the whole solution, such as maximum or minimum working shifts, while soft constraints are related to daily requirements on the number of nurses.

There are two widely-used benchmark data sets for non-cyclic nurse scheduling maintained by researchers at the University of Nottingham [7] and the University of Ghent [8] respectively. Benchmark in [7] consists of nurse scheduling problems collected from various real-world scenarios. The number of instances is limited but the problem size ranges from small with 2 weeks, 8 nurses and 1 shift type to larger with 52 weeks, 150 nurses and 32 shifts. On the other hand, benchmark in [8] is artificially generated based on 9 indicators. But overall there are 6 sets of instances according to the number of nurses and the planning period ranges from one week to one month. Both benchmarks also keep a record of the best known solutions.

A wide range of methodologies, such as constraint programming and meta-heuristics have been applied to solve nurse scheduling problems and achieving very good solutions [5, 6]. However, it seems that no paper has applied dynamic programming directly to solve this type of problem due to the curse of dimensionality. Only one publication describes the use of dynamic programming as part of a column generation method to solve NSP [7].

2.2 Approximate Dynamic Programming

The Bellman Equation (1) conveys the strength of dynamic programming because of the principle of optimality, but this equation also reflects the issue that prevents applying dynamic programming broadly.

$$V\left(S_t\right) = max\left\{C\left(A\left(S_t, S_{t+1}\right)\right) + V\left(S_{t+1}\right)\right\} \tag{1}$$

In this equation, which refers to a maximisation problem, t indicates a given stage, S_t and S_{t+1} represents the space of sub-problem combinations at stages t and $t+1$ respectively, $A(*)$ is an action space that records all connections from stage t to stage $t+1$, $C(*)$ is the action cost function, $V(*)$ is the value function that being in a particular stage. The purpose of dynamic programming is to evaluate all the combinations terminating in each stage and select the optimal one. However, for a problem with a large action space, it is infeasible for dynamic programming to calculate the optimal solution because of either time or memory consumption. This is known as the *curse of dimensionality* and this issue makes dynamic programming algorithms non-practical to be applied widely to large combinatorial optimization problems.

Hence, developing algorithms that satisfy the principle of optimality but are also practical to solve large combinatorial problems has attracted the attention from researchers in recent decades. Hence, research into Approximate Dynamic Programming (ADP) has increased. Basically, ADP algorithms are divided into two categories. In one category, there are algorithms using linear programming based approaches with dynamic programming to resolve approximations. These approaches have obtained impressive results on solving problems such as backgammon, job shop scheduling and elevator scheduling [9]. The other category is to construct the approximate solution with simulation based procedures [10]. Instead of evaluating the whole action space, the Monte-Carlo simulation mechanism is employed to rank the importance of stage links by processing a number of iterations. The final solution is constructed by selecting the links with minimum penalty cost.

The whole idea of Approximate Dynamic Programming is also known as Reinforcement Learning [11] in Artificial Intelligence or Neuro-Dynamic Programming [10] in control theory. ADP has achieved impressive progress on solving Markov Decision Process (MDP) problems. MDP is a branch of mathematics based on probability theory, optimal control and mathematical analysis [12]. Any MDP model can be summarized in the underlying object collection level with four important factors $\{S, A, Pr(s, a), R(s, a)\}$ where S is a state search

space, A is an action search space that moves to the next possible state, $Pr(s, a)$ is the probability of selecting a particular action a from the current state s to move to the next state, and $R(s, a)$ is the reward function that calculates the reward of taking the action a.

The work presented by Colorini and Dorigo [13] is similar to the idea of simulation based ADP in combinatorial optimisation. The aim of their algorithm is to search an optimal solution based on simulation outcomes of a parallel clients set. This algorithm is called Ant Colony System (ACS) and mimics the behaviour of ants seeking food. They applied ACS to solve a travelling salesman problem. However, the simulation behaviours of ADP and ACS are distinguishable, especially with the new ACS extensions developed.

Approximate dynamic programming algorithms are also being considered for solving stochastic optimization problems. Schuetz and Kolisch [9] applied ADP to solve capacity allocation problems in the service industry. Their algorithm, called λ-SMART, is a simulation based ADP. Their algorithm performed well in terms of solution quality, computational time and memory usage. Koole and Pot [14] applied ADP to solve a stochastic workforce scheduling problem with multi-skills in a call center. Their paper suggested a general structure to apply ADP to this type of problem. It also presented some mathematical proofs on the accuracy of ADP. Instead of comparing with other methods, Koole and Pot investigated the relationship between parameters setting and the quality of the obtained solutions.

Given the successful applications of ADP in the literature, we believe this mechanism can also be applicable to solve the nurse scheduling problem and hence the motivation for the work in this paper as a new research direction on tackling nurse scheduling. This paper uses Q-learning as an ADP approach to solve the nurse scheduling problem. The aim of Q-learning [2] is to solve any problem with a representation based on environmental states S, possible actions A from states, and the value of state-action pairs, called Q value. Basically, Q-learning evaluates state-action pairs and increases or reduces the corresponding Q value depending on the outcome (i.e. state that the state-action led to).

3 Nurse Scheduling as a Markov Decision Process

Every MDP model is summarized as $\{S, A, Pr(s, a), R(s, a)\}$ and Figure 1 depicts the decision process for NSP and explained in what follows. The state space S is represented as **STATE** in the figure. There two types of states: pre-condition state and post-condition state. In every MDP iteration, a state is selected as a starting point. This state is called pre-condition state while the post-condition state is the outcome state after making a decision. In the NSP, an example of pre-condition state is the nurse schedule completed until day 4 and the post-condition state is the schedule completed until day 5 as depicted in the figure.

The action space A is represented as **ACTION** in the figure. The daily requirement in NSP is to select w nurses out of the total W nurses with various shift assignment for each nurse. In the consequence, the action space for NSP

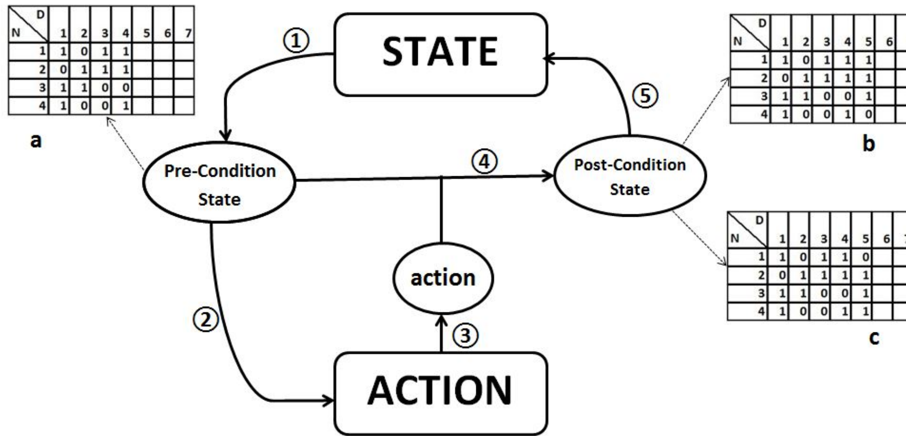

Fig. 1: Transition Process Between State and Action in Markov Decision Process

is modelled as any possible assignment in the next day for any given scheduling period. In order to cover the curse of dimensionality, the action space is decomposed with two parts. The first part is to select a subset of w required nurses (with constraints) or any number (without constraints). With these selected nurses there will be w^{SH} various shift assignment among these nurses in total. In the consequence, the search space is decreased because only a subset need to be evaluated in the iterations. Finally, the selected action will be allocated to different nurses. As shown in figure 1, the allocation process is the outcome to produce a post-condition state represented by schedule b or c.

$Pr(s, a)$ is the transition probability from the state in the current stage with a selected action to the next stage. In most of the MDP models, this value will be updated within the algorithm no matter what is the initial value. In the NSP, the action is the possible shift assignment in the next day. So, the probability value is unchanged in the current implementation.

$R(s, a)$ is the reward function for taking an action from the current stage. The reward should be considered in two parts. In each scheduling day, there are hard constraints and/or soft constraints to be considered. For a produced schedule it can be evaluated if there are any violations of nurse preferences hard constraints. The number of nurses scheduled or on duty is a kind of soft constraint that is evaluated according to the daily nurse requirements. For the next day in the scheduling we should consider the previous working patterns. For example, in most NSP models, there is a hard constraint on the maximum number of working days over a certain period of time. Even though the pre-condition state is feasible, the post-condition state should be tested after the selected action. The value of the reward will be fully described in the next section.

4 The Proposed Approximate Dynamic Programming

This section presents both dynamic programming (DP) and approximate dynamic programming (ADP) procedures applied to solve the NSP. A simple example with 2 nurses, 2 shifts and a 2-days section of the whole planning period is used below to explain the implementation of these two approaches.

4.1 Dynamic Programming Procedure

Algorithm 1 outlines the steps of dynamic programming. The NSP is divided into several sub-problems or stages corresponding to the number of days T in the planning period. In each stage $t \in \{1, \cdots, T\}$, all possible daily assignments (NA) are constructed based on the number of nurses W and the number of shifts SH. Each daily assignment in the stage is treated as a state. In a single step of dynamic programming, every state in the current stage t is added to the input. This input is a partially-built solution until stage $t-1$. Solutions are constructed by recursively repeating this single step from an initial stage until the final stage.

Algorithm 1: Dynamic Programming Procedure

```
1 begin
2     if t = T then
3         return Sol;
4     else
5         NA ← ConstructAssignment(t + 1);
6         for na ∈ NA do
7             ta ← Combine(Sol, na);
8             DP(ta, t + 1);
```

A NSP example is presented in figure 2. The initial input in this example is $\{D, N\}$ in stage T. D is a day shift and N is a night shift. The circle in each stage represents one possible assignment for the two nurses and the set of circles is the value of NA in the algorithm. For instance, the circle with pair $\{D, N\}$ in stage $t+1$ means that a day shift is assigned to the first nurse a night shift is assigned to the second nurse in that day. The solution with the optimal value will be selected based on the objective function (2).

$$f = \min(V + \sum_{i=1}^{W} x_{it} \times c_{it}), t \in [1, \cdots, T] \quad (2)$$

W is the number of nurses and t represents different stages. x_{it} is an individual nurse shift assignment and the cost is c_{it}. V is the value of soft and hard constraints violations that is calculated in (3). V_{hc} is the number of hard constraint violations and c_h is the penalty of hard constraint violations. V_{sc} is the number of soft constraint violations and c_s is the penalty of soft constraint violations.

$$V = c_h \times V_{hc} + c_s \times V_{sc} \quad (3)$$

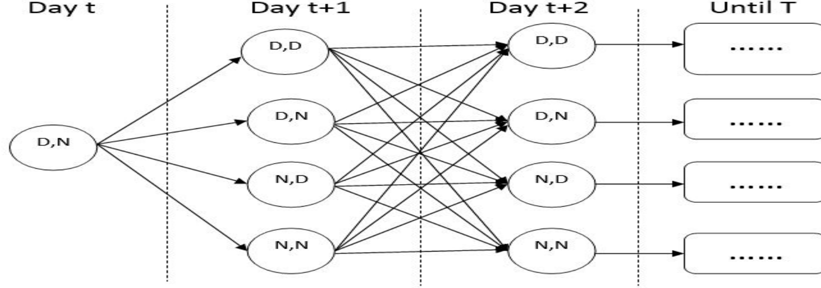

Fig. 2: Structure of NSP for Dynamic Programming

4.2 Simulation based Approximate Dynamic Programming

The approximate dynamic programming implementation in this paper is a modification from algorithm 1 and it uses a simulation process. The objective function and the recursive process is kept as in algorithm 1. Different to dynamic programming, the essence of ADP is to only evaluate one solution per process. With this purpose, the storage space of ADP is changed as recording the importance of a link that connected stages t and $t+1$. Our proposed ADP is an extension of Q-Learning based on the following equations.

$$\begin{aligned} &a_t^n = \begin{cases} RandomSelection, & c < \epsilon \\ \min_{a \in A} Q_{t-1}(s_t^n, a), & otherwise \end{cases} \\ &Q_t(S_t^n, a_t^n) = \alpha_n q_t^n + (1-\alpha_n) Q_{t-1}(S_t^n, a_t^n) \\ &q_t^n = c(S_t^n, a_t^n) + \alpha min_{a' \in A}(Q_{t-1}(S_t^{n+1}, a')) \end{aligned} \tag{4}$$

Instead of updating V as described in the previous section, the whole process of Q-learning is to update the Q value, $Q(s,a)$, which is the value of a pair state s and action a. In equation (4), t is the iteration value, n is the process value, s is a single state. One iteration in ADP is finished by completing the input to a fully constructed solution with a number of processes. In this paper, one iteration is finished until a feasible solution is explored. In every process, only one action will be selected between two stages. The action selection could be random or follow some designed rules. In our proposed algorithm, the selection method follows the idea of ϵ-greedy [2]. The random and greedy selection method will be processed based on the value of ϵ. This selection method is able to guarantee that Q-learning explores a wide search space and converges into a optimum. In the nurse scheduling problem s_t^n is the nurse assignment combination from the initial stage 1 to stage n in current the iteration t, a is the action space, a_t^n is the link between stage $n-1$ and n in iteration t. A is the set of all links between any two continuous stages. The purpose of calculating a_t^n is to select the next action or links from the current stage to the next stage. The structure of this approximate dynamic programming is outlined in algorithm 2.

Algorithm 2: Q-Learning Procedure

```
1  begin
2    Initial value of α, ε and max_iter;
3    i ← 0, M ← Empty;
4    while i < max_iter do
5      Sol ← Empty;
6      for j ← 1 to T do
7        c ← RandomNumberGenerator();
8        NA ← ConstructAssignment(t);
9        if c < ε then
10         a ← RandomSelection(NA);
11       else
12         a ← GreedySelection(NA);
13       q(Sol, a) = FitnessFunction(Sol, a);
14       if (Sol, a) ∈ M then
15         Q(Sol, a) = GetQValue(Sol, a);
16       else
17         Q(Sol, a) = 0;
18       Q(Sol, a) ← α × q(Sol, a) + (1 − α) × Q(Sol, a);
19       Insert(M, (Sol, a));
20       Sol ← Combine(Sol, a);
21       j ← j + 1;
22     Update(α, ε);
23     i ← i + 1;
```

Algorithm 2 outlines the steps of Q-learning on solving NSP instance. These steps are explained with the same previous example. A set of parameters need to be initialized: α is the learning rate and ϵ is the searching rate. Larger values for these parameters guarantee that Q-learning explores a larger portion of the search space. M is a lookup table that record the q value of a link between two stages.

The number of processes per iteration in this example is 2, and its output is moved to the next stage. For any given iteration, the initial input is $\{D, N\}$ at day t. In the first process, there are four actions leading to the next stage. The method to select an action is based on the random value c. If c is less than ϵ then the action will be randomly selected from the four actions. Otherwise, the action with the lowest Q value will be selected. Once an action is selected, for example $\{D, D\}$, then there is a state-action pair $\{\{D, N\}, \{D, D\}\}$. This selection will be evaluated through both the daily preference cost of $\{D, D\}$ and the validity of this pair. In the next step, the Q value of the pair will be updated based on line 18 and added to M. Before starting a new process, part of a temporal solution will be constructed based on the selected action. All the rest

processes are repeated until a solution is fully explored. Q-learning will start a new iteration until this solution is feasible.

One possible solution in the last iteration could be $\{\{D, N\}, \{D, D\}, \{N, N\}\}$. With a higher value of ϵ this solution will not be repeated in the next few iterations. Moreover, a number of state-action pairs will be evaluated until a feasible solution is found. These two points will guarantee the Q-learning process to explore more of the search space.

After a number of iterations, the greedy selection method will be considered for evaluating all selected state-action pairs. With this method, not all new pairs will be added to the lookup table M , but only the pairs that appear promising according to the previous step, i.e. random selection, in the algorithm. At the end of the whole algorithm, there is a temporal solution that records the best solution found during the iterations. While there is also the storage space M that indicates which pair is more important (with lower value) for constructing the optimal solution.

5 Experiments and Results

The Q-learning and dynamic programming (DP) algorithm presented in the previous section were implemented in Java and all computations were performed on an Intel (R) Core (TM) i7 CPU with 3.2 GHz and RAM 6 GB. A sub-group of instances from the NSPLib benchmark was considered in these experiments. There are 7290 instances in this sub-group, each with 25 nurses, 7 scheduling days and 4 shift types. The set of hard and soft constraints were selected from the *Case* 1 file. The detialed descriptions of the instances is given in [15, 8]. Maenhout and Vanhoucke [16] published their mechanisms to solve the NSPLib instances. Here, results from the proposed ADP are compared to the best known solutions from the literature.

5.1 Comparing with Dynamic Programming Procedure

The parameter values for the Q-learning method to produce the results shown in the table are the best after tuning. The initial value of α and ϵ are both set to 0.9 and the maximum number of iterations is set upto 1000. Table 1 shows a summary of the experimental results from applying Q-learning and DP to solve the selected NSP instances. The performance of the proposed Q-learning and DP methods is compared using four aspects. *No. Solutions* is the number of solutions that each algorithm found. *Avg. Cost* is the average objective cost of the solutions. *Avg. Violation* is the average soft constraint violations in the solutions. *Avg. Time* is the average time to generate the solutions.

The contribution of approximate dynamic programming is to extend the usage of dynamic programming on larger instances. To give an illustration, none of the selected instances could be solved by dynamic programming due to the state space exceeding the memory. These experimental results serve as evidence that approximate dynamic programming is an improvement from the standard

	No. Solutions	Avg. Cost	Avg. Violation	Avg Time
Q-Learning	7290	306.0798	0.471056241	9.946227709
DP	Could not be solved			

Table 1: Comparison of Q-learning with Dynamic Programming

dynamic programming approach that is able to tackle the nurse scheduling problem. Not only the computational speed but also the algorithm effectiveness is improved by considering only the useful partial state space.

In our experiments, the outcome of one iteration is to construct and evaluate a feasible solution, as described in the previous section. Constructing such feasible solution might cause a large number of processes because of the infeasible solutions. The value state-action pairs that construct an infeasible solution are updated within the same iteration. In the consequence, the number of evaluations within one iteration is larger than those required for a single feasible solution. This is also the reason why the average computation time is longer when comparing to other mechanisms in the next section. Figure 3 shows the solution convergence for a problem instance. The number of iterations is set to 2000 but the objective function value settles at around 817 in iteration 453 for this problem instance. Experimental results show similar results for other instances.

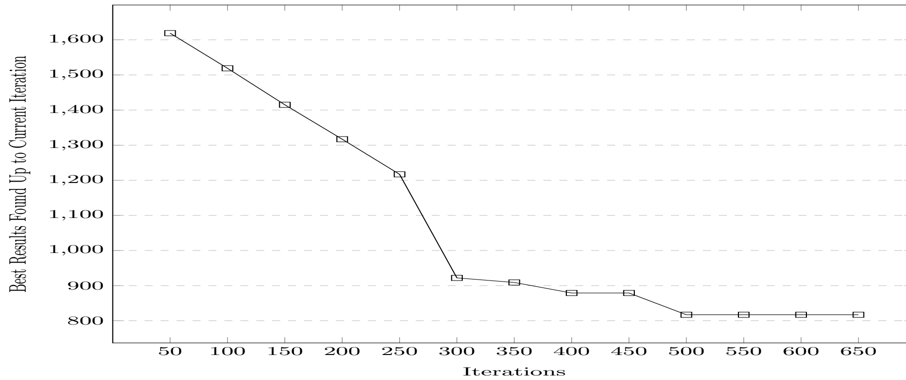

Fig. 3: The Convergence of Objective Value Updating for Sample Instance

5.2 Comparing with Meta-Heuristic Approaches

A number of published papers describe approaches to solve the NSPLib instances. For comparison in this paper we use the results from [16] because they provide detailed results for every instance. That paper combined two meta-

heuristics, an Electromagnetism (EM) Approach and Scatter Search (SS). Comparison is carried out using the 5 aspects shown in table 2.

	No. Solutions	Avg. Cost	Avg. Violation	Avg Time	Achievements
Q-Learning	7290	306.0798	0.471056241	9.946227709	84.79%
EM and SS	7290	305.776	0.53	0.532	88.27%

Table 2: Comparison of Q-Learning with Electromagnetism (EM) and Scatter Search (SS) [16]

The achievement is calculated using equation (5). $V(Approach)$ is the number of instances for which the given approach achieves a solution that is not worse than the best solution reported [16]. The average objective cost, violation and achievements for both the proposed Q-learning and meta-heuristics are very close, a difference within 5%. From this point of view, the quality of our solutions is comparable to published results. Hence, we argue that approximate dynamic programming could make contributions to the state of art for solving the Nurse Scheduling Problem. It should be noted that the computation time of solving instances by meta-heuristics approaches are much faster than the proposed algorithm. This shortcoming means that future research can be diverted at improving Q-learning or approximate dynamic programming to make it more efficient.

$$Achievement(\%) = V(Approach)/7290 \times 100\% \tag{5}$$

6 Conclusion

This paper proposes an approximate dynamic programming (ADP) algorithm to solve the nurse scheduling problem (NSP). For this, the NSP is modelled as a Markov Decision Process. Then, a typical ADP algorithm, Q-Learning, is applied to generate solutions. Instead of evaluating the whole state-action space like in standard dynamic programming, ADP only works on a subset of the space, the most promising state-actions according to the objective function. The experimental results here support the idea of modelling NSP as a Markov Decision Process and provide evidence that ADP exhibits very good performance by achieving as good solutions as heuristics proposed in the literature.

This paper has shown that ADP is able to solve effectively a range of NSP instances that dynamic programming is not able to solve. Although the instances solved are of realistic size, it is still a challenge to solve larger instances with ADP. Hence, future work will focus on speeding up ADP and make improvements on the procedure so that applying it to solve more challenging problem instances becomes practical in terms of computational time. This paper demonstrates that applying ADP to solve difficult combinatorial optimization problems like workforce scheduling is a viable and interesting research direction.

References

1. R. Bellman, "Dynamic programming and lagrange multipliers," *Proceedings of the National Academy of Sciences*, vol. 42, no. 10, pp. 767–769, 1956.
2. W. B. Powell, *Approximate Dynamic Programming: Solving the curses of dimensionality.* John Wiley & Sons, 2007, vol. 703.
3. J. Van den Bergh, J. Beliën, P. De Bruecker, E. Demeulemeester, and L. De Boeck, "Personnel scheduling: A literature review," *European Journal of Operational Research*, vol. 226, no. 3, pp. 367–385, 2013.
4. M. Elshafei and H. K. Alfares, "A dynamic programming algorithm for days-off scheduling with sequence dependent labor costs," *Journal of Scheduling*, vol. 11, no. 2, pp. 85–93, 2008.
5. B. Cheang, H. Li, A. Lim, and B. Rodrigues, "Nurse rostering problems—a bibliographic survey," *European Journal of Operational Research*, vol. 151, no. 3, pp. 447–460, 2003.
6. P. De Causmaecker and G. V. Berghe, "A categorisation of nurse rostering problems," *Journal of Scheduling*, vol. 14, no. 1, pp. 3–16, 2011.
7. T. Curtois, "Employee shift scheduling benchmark data sets," School of Computer Science, The University of Nottingham, Nottingham, UK, Tech. Rep., September 2014.
8. M. Vanhoucke and B. Maenhout, "Characterisation and generation of nurse scheduling problem instances," Ghent University, Faculty of Economics and Business Administration, Tech. Rep., 2005.
9. H.-J. Schuetz and R. Kolisch, "Approximate dynamic programming for capacity allocation in the service industry," *European Journal of Operational Research*, vol. 218, no. 1, pp. 239–250, 2012.
10. D. P. Bertsekas, *Dynamic programming and optimal control.* Athena Scientific Belmont, MA, 1995, vol. 1, no. 2.
11. R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 1998, vol. 1, no. 1.
12. M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons, 2014.
13. M. Dorigo and L. M. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *Evolutionary Computation, IEEE Transactions on*, vol. 1, no. 1, pp. 53–66, 1997.
14. G. Koole and A. Pot, "Approximate dynamic programming in multi-skill call centers," in *Simulation Conference, 2005 Proceedings of the Winter.* IEEE, 2005, pp. 576–583.
15. B. Maenhout and M. Vanhoucke, "Nsplib – a nurse scheduling problem library: a tool to evaluate (meta-)heuristic procedures," in *O.R. in health.* Elsevier, 2005, pp. 151–165.
16. B. Maenhout and M. Vanhouck, "New computational results for the nurse scheduling problem: a scatter search algorithm," in *Evolutionary Computation in Combinatorial Optimization.* Springer, 2006, pp. 159–170.