Chapter 1

CHOOSING SEARCH HEURISTICS BY NON-STATIONARY REINFORCEMENT LEARNING

Alexander Nareyek

Computer Science Department, Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh, PA 15213-3891, USA

alex@ai-center.com

http://www.ai-center.com/home/alex/

Abstract Search decisions are often made using heuristic methods because real-world applications can rarely be tackled without any heuristics. In many cases, multiple heuristics can potentially be chosen, and it is not clear a priori which would perform best. In this article, we propose a procedure that learns, during the search process, how to select promising heuristics. The learning is based on weight adaptation and can even switch between different heuristics during search. Different variants of the approach are evaluated within a constraint-programming environment.

Keywords: Non-Stationary Reinforcement Learning, Optimization, Local Search, Constraint Programming

1. Introduction

All kinds of search techniques include choice points at which decisions must be made between various alternatives. For example, in refinement search, an extension step from a variables' partial assignment toward a complete assignment must be chosen. In local search methods, it must be decided how a complete but suboptimal/infeasible assignment is to be changed toward an optimal/feasible assignment.

However, for large and complex real-world problems, decisions can rarely be made in an optimal way. Especially for local search techniques, this is a very critical issue because they do not normally incorporate backtracking mechanisms. Many different meta-heuristic techniques have therefore been developed to handle the complications involved when choosing a decision alternative.

Figure 1.1 shows a choice point, representing the current state (i.e., the current variable assignment) of local search, and multiple decision alternatives, representing the so-called *neighbor states* that can be reached within an iteration.

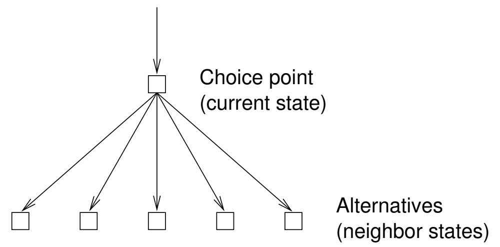


Figure 1.1. A decision point in local search.

Nearly all local search methods evaluate all neighbor states in a kind of look-ahead step in order to choose the most beneficial alternative. However, complex real-world problems – such as action planning including time, resources and optimization – often have utility functions whose computation requires a great deal of computing power. Analyzing large neighborhoods is mostly out of the question, and even smaller neighborhoods are difficult to check. Techniques like simulated annealing (Kirkpatrick et al. 1983) are highly suitable for these tasks because *only one neighbor is analyzed* for a choice decision (though more neighbors may be analyzed if the current neighbor appears to be unsuitable).

For the purposes of this paper, we go one step further, not analyzing any neighbor, but choosing a neighbor according to learned utility values. In addition, we do not choose a specific neighbor state but a *transformation heuristic* that will be applied to create the new state. Unlike other reinforcement learning approaches for learning which heuristics perform well, our approach allows the search to switch between different heuristics during search in order to adapt to specific regions of the search space.

Section 1.2 introduces the constraint-programming environment that is applied in our experiments, and details the use of heuristics. Weights and their adaptation are presented in Sec. 1.3. The scheme is evaluated in Sec. 1.4. Conclusions and related work are discussed in Sec. 1.5.

2. Search Decisions

As an example of local search, we give a brief description of the search method applied in the **DragonBreath** engine. The underlying paradigm is presented in detail in (Nareyek 2001 (a)).

The problem is specified as a so-called constraint satisfaction problem (CSP). A CSP consists of

- a set of variables $x = \{x_1, \ldots, x_n\}$
- where each variable is associated with a domain $d_1, ..., d_n$
- and a set of constraints $c = \{c_1, ..., c_m\}$ over these variables.

The domains can be symbols as well as numbers, continuous or discrete (e.g., "door", "13", "6.5"). Constraints are relations between variables (e.g., "x_a is a friend of x_b", "$x_a < x_b \times x_c$") that restrict the possible value assignments. Constraint satisfaction is the search for a variable assignment that satisfies the given constraints. Constraint optimization requires an additional function that assigns a quality value to a solution and tries to find a solution that maximizes this value.

In our local search approach, a specific cost function is specified for every constraint (so-called *global constraints*), which returns a value that represents the constraint's current inconsistency/optimality with respect to the connected variables. For example, a simple `Sum` constraint with two variables a and b to be added and an s variable for the sum could specify its costs as $\texttt{Sum}_{costs} = |a+b-s|$. The total costs o_{now} of a current variable assignment (which is often also called *objective function value*) is a function of all constraints' costs, e.g., a simple addition.

In addition, a constraint has a number of heuristics to improve its cost function. For example, a heuristic for the `Sum` constraint could randomly choose one of the related variables and change it such that there are no more costs. Another heuristic might resolve the inconsistency by distributing the necessary change such that all variables are changed by the same (minimal) amount. The constraint must make the choice as to which heuristic to apply on its own.

On top of all constraints is a *global search control* which selects, in each iteration of local search, one of the constraints which is to perform a change, i.e., the transition to a neighbor state. Figure 1.2 shows the control flow.

The global search control possesses qualitative and quantitative information from the constraints' cost functions to decide which constraint to choose (e.g., the constraint with the maximal costs), but a constraint itself has little guidance as to which of its heuristics to choose. This choice

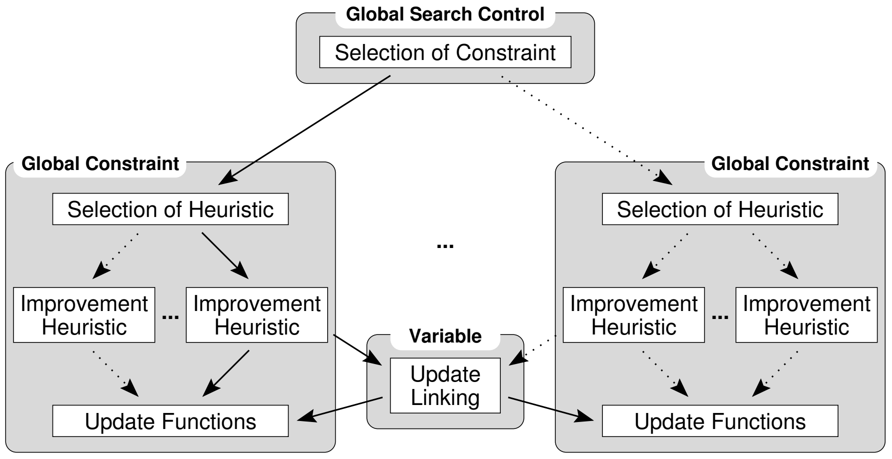


Figure 1.2. Using global constraints for local search.

point — for choosing one of the constraint's heuristics — is investigated below.

3. Utility Weight

For a choice point, a *utility value* $\omega_a \geq 1$ is computed/maintained for every alternative a (an *alternative* stands for a *heuristic* here) that expresses the expected benefit of choosing this alternative.

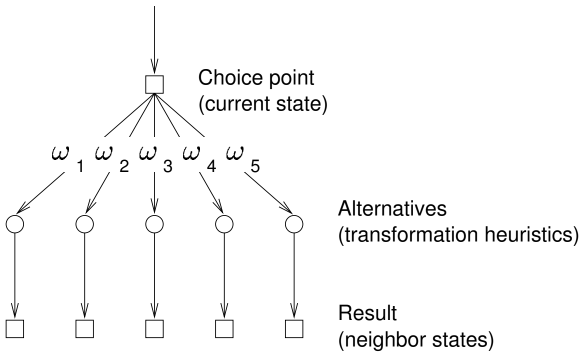


Figure 1.3. A decision point in our approach.

The utility values are subject to learning schemes, which change the values based on past experiences with choosing this alternative. In many cases, an appropriate balance of the utility values will depend on the area of the search space that the search is currently in. We will therefore focus on schemes that dynamically adapt the weights *during* search and not only after a complete run.

Selection Function

Selection between possible alternatives is done based on the alternative's utility values. Here, we look at two simple ones. The first is a *fair random choice* (often called a *softmax* kind of selection rule), referred to below as `M:0`, which selects an alternative a from the choice point's alternatives $\mathcal{A}$ with a choice probability p_a in proportion to the alternative's utility value ω_a:

$$\texttt{M:0}: \quad p_a = \frac{\omega_a}{\sum_{i \in \mathcal{A}} \omega_i}$$

Another possibility is to make a random choice between the alternatives with *maximal* utility values:

$$\texttt{M:1}: \quad p_a = \frac{\begin{cases} 0 & : \exists i \in \mathcal{A} : \omega_a < \omega_i \\ 1 & : \forall i \in \mathcal{A} : \omega_a \geq \omega_i \end{cases}}{\sum_{i \in \mathcal{A} \mid \forall j \in \mathcal{A} : \omega_i \geq \omega_j} 1}$$

Weight Adaptation

All utility weights have integer domains and are initially set to 1. If the choice point is selected, the utility weight of the alternative is changed that was chosen by the choice point when it was called last time. The kind of change depends on the relation of the current cost function value o_{now} to the cost function value when the choice point was called last time o_{before} (i.e., if there is a positive or negative reinforcement). The update schemes below can be combined to give many different strategies, e.g., a simple `P:1-N:1` strategy.

o_{now} `better-than` o_{before}:
(positive reinforcement)

`P:1` (Additive adaptation): $\omega_a \leftarrow \omega_a + 1$

`P:2` (Escalating additive adaptation): $\omega_a \leftarrow \omega_a + m_{promotion}$

`P:3` (Multiplicative adaptation): $\omega_a \leftarrow \omega_a \times 2$

`P:4` (Escalating multiplicative adaptation): $\omega_a \leftarrow \omega_a \times m_{promotion}$

`P:5` (Power adaptation): $\omega_a \leftarrow \begin{cases} \omega_a \times \omega_a & : \omega_a > 1 \\ 2 & : \omega_a = 1 \end{cases}$

o_{now} `worse-than-or-equal-to` o_{before}:
(negative reinforcement)

`N:1` (Subtractive adaptation): $\omega_a \leftarrow \omega_a - 1$

`N:2` (Escalating subtractive adaptation): $\omega_a \leftarrow \omega_a - m_{demotion}$

`N:3` (Divisional adaptation): $\omega_a \leftarrow \frac{\omega_a}{2}$

`N:4` (Escalating divisional adaptation): $\omega_a \leftarrow \frac{\omega_a}{m_{demotion}}$

`N:5` (Root adaptation): $\omega_a \leftarrow \sqrt{\omega_a}$

If a utility value falls below 1, it is reset to 1; if a utility value exceeds a certain max_ω, it is reset to max_ω; if a utility value is assigned a non-integer value, it is rounded down. In the case of an escalating adaptation, each time there is a consecutive improvement/deterioration, the $m_{promotion}/m_{demotion}$ value is doubled. Otherwise, it is reset to 1 (for `P:2` and `N:2`) or 2 (for `P:4` and `N:4`).

A Weight-Adaptation Example

For the illustration of the weight-adaptation mechanism, we follow the development of the weights of a choice point's six heuristics below. The heuristics A to F change the values of the problem definition's variables in different ways, e.g., by additions and subtractions. The chosen weight-adaptation strategy is `P:1-N:2` with a fair random choice `M:0`. The current search situation is to be

ω_A	ω_B	ω_C	ω_D	ω_E	ω_F
1	2	1	3	3	1

last choice: E $m_{demotion} = 1$

$o_{before} = 20$ $o_{now} = 12$

and one of the heuristics is to be selected for execution.

Entering this choice point, the heuristics' weights will be updated at first. The cost function value is now better (assuming that we want to minimize here) than the last time this decision had to be made, and the last choice was heuristic E. According to strategy `P:1`, the weight of heuristic E is rewarded by increasing it by one.

Next, a heuristic is to be selected for execution. The choice probability for the E option is the highest (weight value divided by the sum of all weight values; $4/12 = 33\%$), and we assume that this alternative is chosen by strategy `M:0`. Heuristic E is executed.

Other changes may follow, and after some iterations, our choice point might be called again. The situation is now:

ω_A	ω_B	ω_C	ω_D	ω_E	ω_F	last choice: E	$o_{before} = 12$
1	2	1	3	4	1	$m_{demotion} = 1$	$o_{now} = 17$

The cost function value has deteriorated since the last time the choice point was entered, and — according to strategy `N:2` — the last decision's weight value ω_E is decreased by the $m_{demotion}$ value of 1. The choice probability for heuristic E is now about 27% (3/11) and we assume that this alternative is chosen again.

After some time, our choice point is called once more:

ω_A	ω_B	ω_C	ω_D	ω_E	ω_F	last choice: E	$o_{before} = 17$
1	2	1	3	3	1	$m_{demotion} = 1$	$o_{now} = 17$

The cost function value is the same as at the last call. Stagnation is considered to be as bad as deterioration, and because this is a consecutive deterioration, the $m_{demotion}$ value is doubled. Heuristic E's weight value is therefore decreased by 2. We assume that heuristic D is chosen this time (probability of $3/9 = 33\%$) for execution.

At the next call of the choice point, the situation is:

ω_A	ω_B	ω_C	ω_D	ω_E	ω_F	last choice: D	$o_{before} = 17$
1	2	1	3	1	1	$m_{demotion} = 2$	$o_{now} = 16$

The cost function value has improved, and so, the weight value of heuristic D is increased by one, and the $m_{demotion}$ value is set back to 1.

Invalid Alternatives

For some choice points, more than one alternative must be tested. For example, an alternative may turn out to be infeasible (i.e., the corresponding transformation is not applicable). An *applicability flag* f with a value of 0 or 1 is introduced for every alternative, indicating whether the alternative is still a valid option:

$$\omega_a \quad \leftarrow \quad f_a \times \omega_a$$

By the option of setting an applicability flag to 0, alternatives can often be ruled out a priori by simple feasibility tests.

However, in some cases, the infeasibility of an alternative will only become apparent during the state-transformation process of the chosen heuristic, i.e., after the choice has been made. In such a case, all changes in the current state that were made after the choice point are reversed, the corresponding applicability flag is set to 0 and the choice process is repeated. If no alternative remains applicable, the choice point's improvement fails.

If the choice point's selection is subject to the learning scheme, applicability flags are not set to 0 if an alternative fails. The failure may be caused by a bad random decision during the alternative's computations and the alternative may not be *fully* inapplicable. The learning process can handle this situation more appropriately than in a non-learning case, skipping the usual update of the utility weights and *temporarily* dividing the failed alternative's utility weight by two (though no weight may fall below one). If one of the alternatives has been successfully applied, all adaptations of the utility weights that were done for the restarts are undone.

4. Empirical Evaluation

Two optimization problems — the Orc Quest problem and a modification of the Logistics Domain — are evaluated with different learning/selection schemes. The problems are only roughly described here because the actual problems are not important with respect to our following analysis. A detailed presentation of the problems can be found in (Nareyek 2001 (b)).

The Orc Quest problem's solving process involves only three constraints with six heuristics each. Each of these heuristics applies a specific set of additions and subtractions to the problem variables. There is a hierarchical cost function, demanding the minimization/maximization of specific problem variables. The learning scheme is applied to the choice points of all three constraints.

The Logistics Domain is a classical benchmark in the action-planning community. The problem investigated here is enriched so that actions have durations (more specifically, the duration minimization of Problem 6-1a is analyzed). The problem involves a large number of constraints, which are even created and deleted during the solving process. The learning scheme is applied to all constraints of the State Resource type. This constraint type is responsible for projecting a specific state of the environment and has to ensure that all related preconditions of actions are fulfilled. Such a constraint can for example be responsible for the location of a truck, and must ensure that the truck is at the right location when a loading action is to take place. The constraint type includes five alternative heuristics, e.g., to create a new action, to temporally shift an action, and to delete an action.

Results

A strategy is denoted by `P-N-M`, `P` $\in \{1..5\}$ indicating the adaptation scheme that is applied in the case of an improvement, `N` $\in \{1..5\}$ the adaptation scheme for non-improvement, and `M` $\in \{0,1\}$ if the fair random choice is applied or a maximal value is chosen.

The results for some strategies for the Orc Quest problem are shown in Fig. 1.4 as the percentage of test runs (100 % = 100,000 test runs) that found the optimal solution after a specific number of iterations[1]. For example, in case of strategy `1-5-1`, about 98 % of the test runs found the optimal solution after 2,000 iterations while 2 % were still running. The strategies' curves are overlapping in some cases, which means that for these strategies, the strategy dominance is dependent on available computation time.

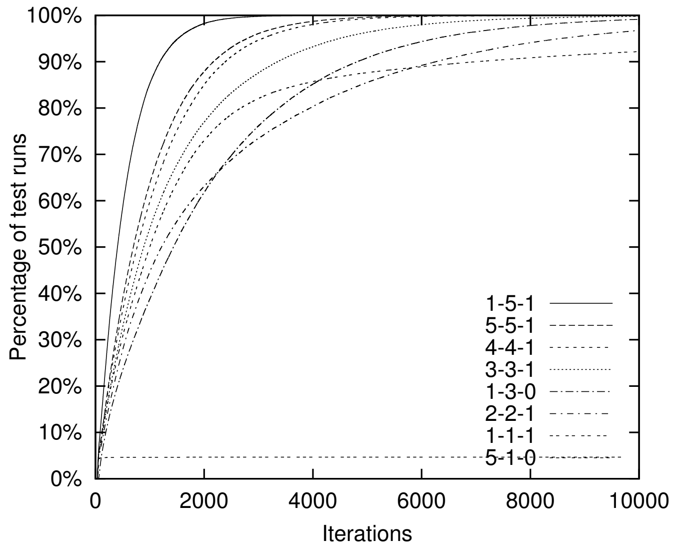


Figure 1.4. Sample strategies for the Orc Quest problem.

[1]While presenting results as so-called *run-time distributions* is not widespread in the Operations Research community, it addresses a number of serious issues related to result presentation and analysis (see (Hoos and Stützle 1998)).

The problem from the Logistics Domain is much harder, so only the best solution (minimal duration) found after 100,000 iterations is shown in Fig. 1.5 (100 % test runs = 1,000 test runs).

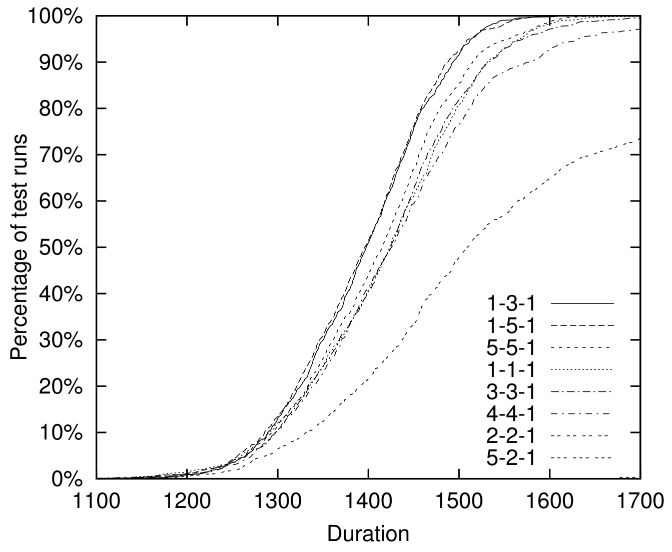


Figure 1.5. Sample strategies for the Logistics Domain.

The following figures will display more detailed results, i.e., for all possible strategy mixes. The amount of data to be shown does however not allow for a complete picture like in the graphs above. Only for specific percentages of test runs, it is shown after how many iterations this percentage of test runs found the optimum (for the Orc Quest problem — Fig. 1.6), and the lowest duration that was found by this percentage of test runs after 100,000 iterations (for the Logistics Domain — Fig. 1.7). This corresponds to vertical slices of the previous figures. The strategies are sorted according to which strategy resulted in the least iterations/duration for a maximal percentage of test runs (not considering the 100 % rate).

The general trend is that a low (e.g., additive/`P:1`) rate of adaptation is good in the case of an improvement, a strong (e.g., root/`N:5`) rate of adaptation is good in the case of deterioration, and a choice of a maximal weight is often better than a fair random choice. The explorative feature of the fair random choice may not be that important because there are very often cases of negative reinforcement that quickly change the weight situation.

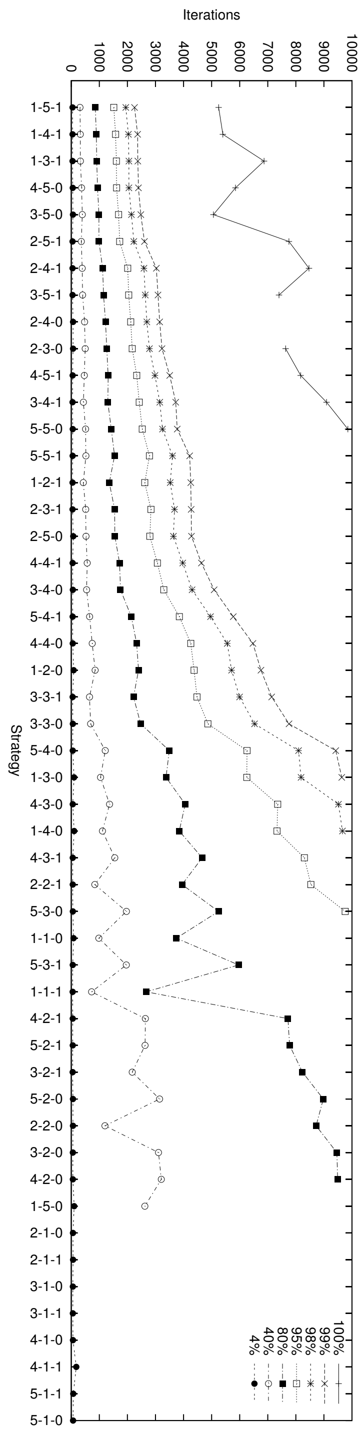
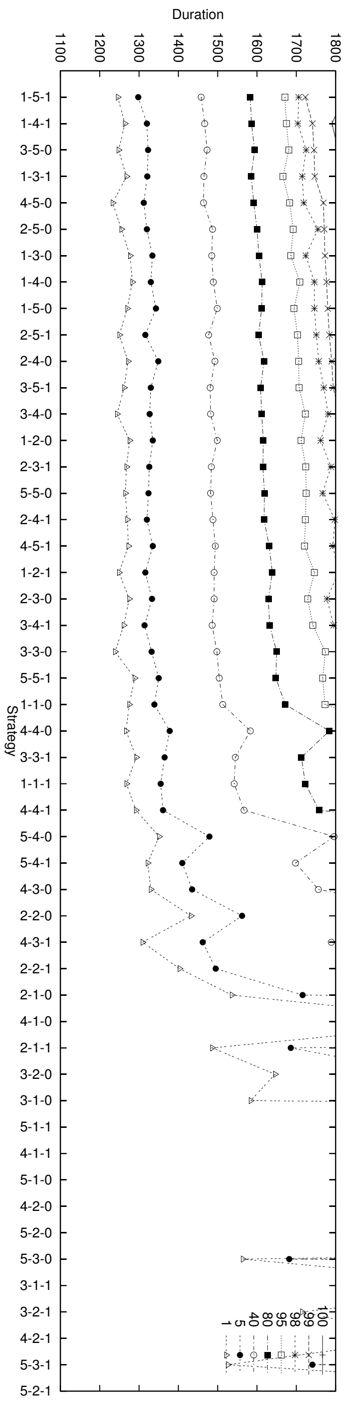


Figure 1.6. Weight-adaptation results for the Orc Quest problem.



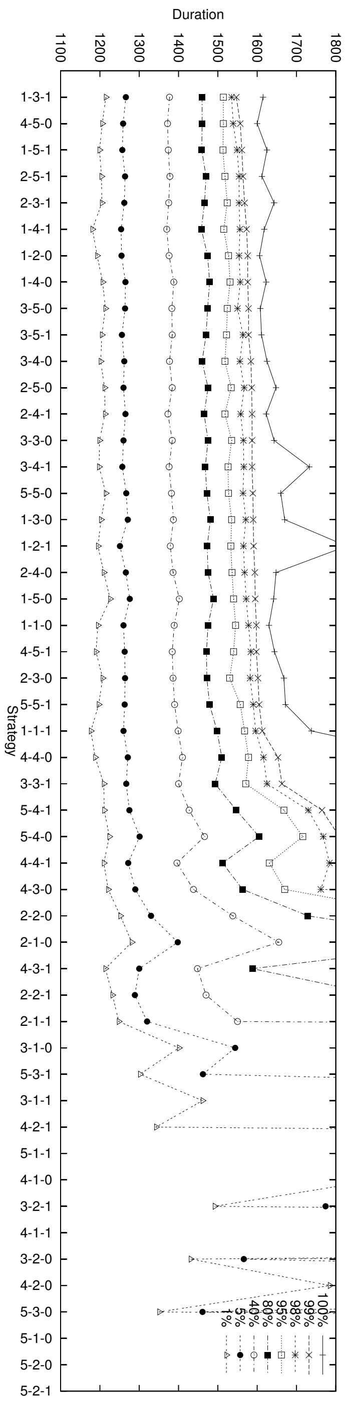


Figure 1.7. Weight-adaptation results for the Logistics Domain after 25,000 (left) and 100,000 (right) iterations.

Because the Orc Quest problem involves only three constraints, we can easily visualize some further properties of the search process. Figure 1.8 shows how many times a constraint's highest weight changes, i.e., how many times a weight is assigned a value above a certain percentage of the total values of the choice point's weights, and the last time this percentage threshold was reached, it was reached by another weight. Strategies that perform many changes in the configuration appear to perform better. This might be an indication of why strategies with a low rate of adaptation in the case of an improvement and a strong rate of adaptation in the case of a deterioration are likely to perform better — such strategies facilitate a reconfiguration of the weight situation.

An exception here are strategies with a very low positive reaction (`P:1`). Because of the slow growth of the weights, a smaller number of clear reconfigurations are performed. However, this simmering situation would appear to have its advantages as well.

Figure 1.9 shows how many times the highest weight is re-established, i.e., how many times a weight is assigned a value above a certain percentage of the total values of the choice point's weights, and the last time this percentage was reached, it was reached by the *same* weight. In general, one would expect strategies that re-establish old configurations to do needless work and thus, possibly perform worse. However, the figures do not show many differences here. The reason for this is probably that the better strategies perform many configuration changes in general, and, are thus also more likely to re-establish configurations more often. Even though the absolute numbers of re-established configurations are similar for all strategies, the better strategies have a much lower ratio of re-established configurations to all reconfigurations.

Extended Experiments

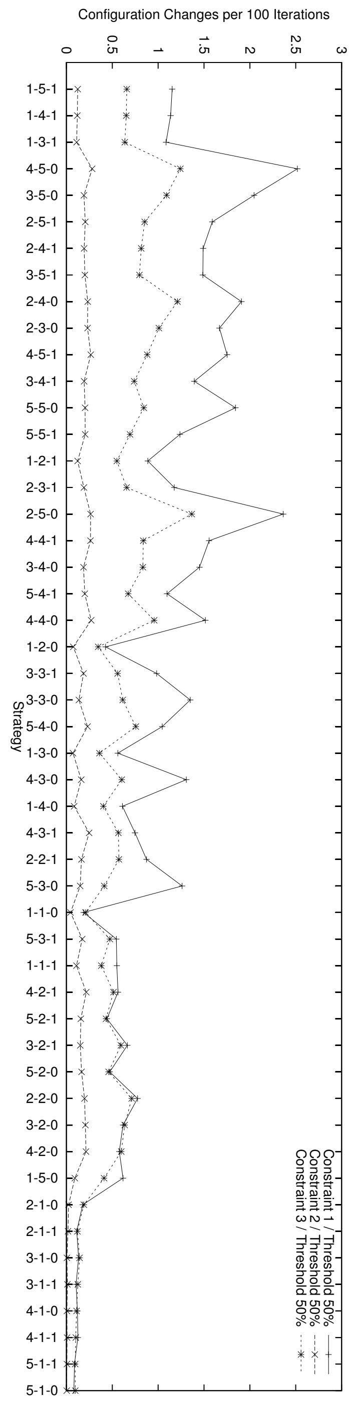
Following the observed trend, we can extend our experiments by more extreme options in this direction:

`P:0` (No adaptation): $\omega_a \leftarrow \omega_a$

To enable weight increases for this option at all, in the case of a negative change, the decrease of ω_a is distributed as an increase to all $\omega_{i \neq a}$ (starting with high initial weights).

`N:6` (Total loss adaptation): $\omega_a \leftarrow 1$

Figure 1.10 shows that these options do not improve performance for the Orc Quest problem. However, as shown in Fig. 1.11, strategies with an `N:6` option appear to work well for the early phase of search, i.e., for less constrained problems.



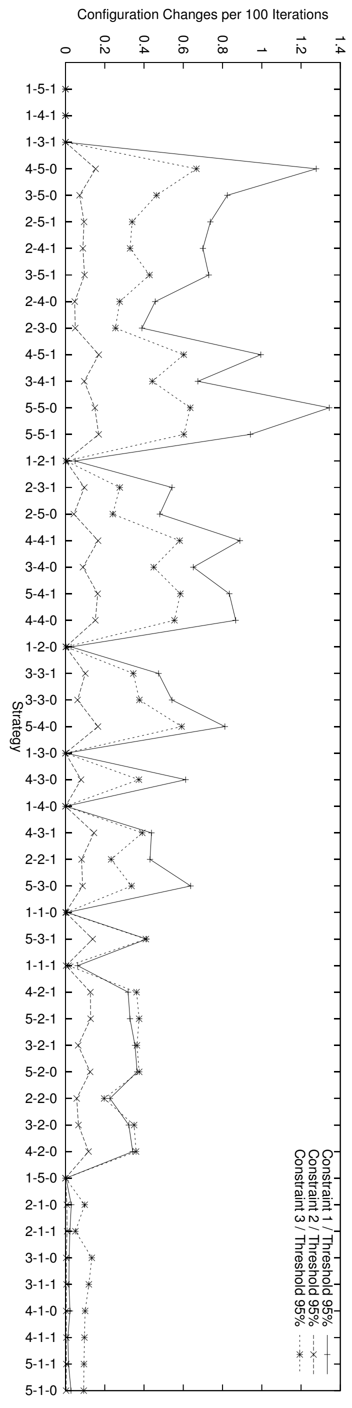


Figure 1.8. Number of configuration changes for the Orc Quest problem.

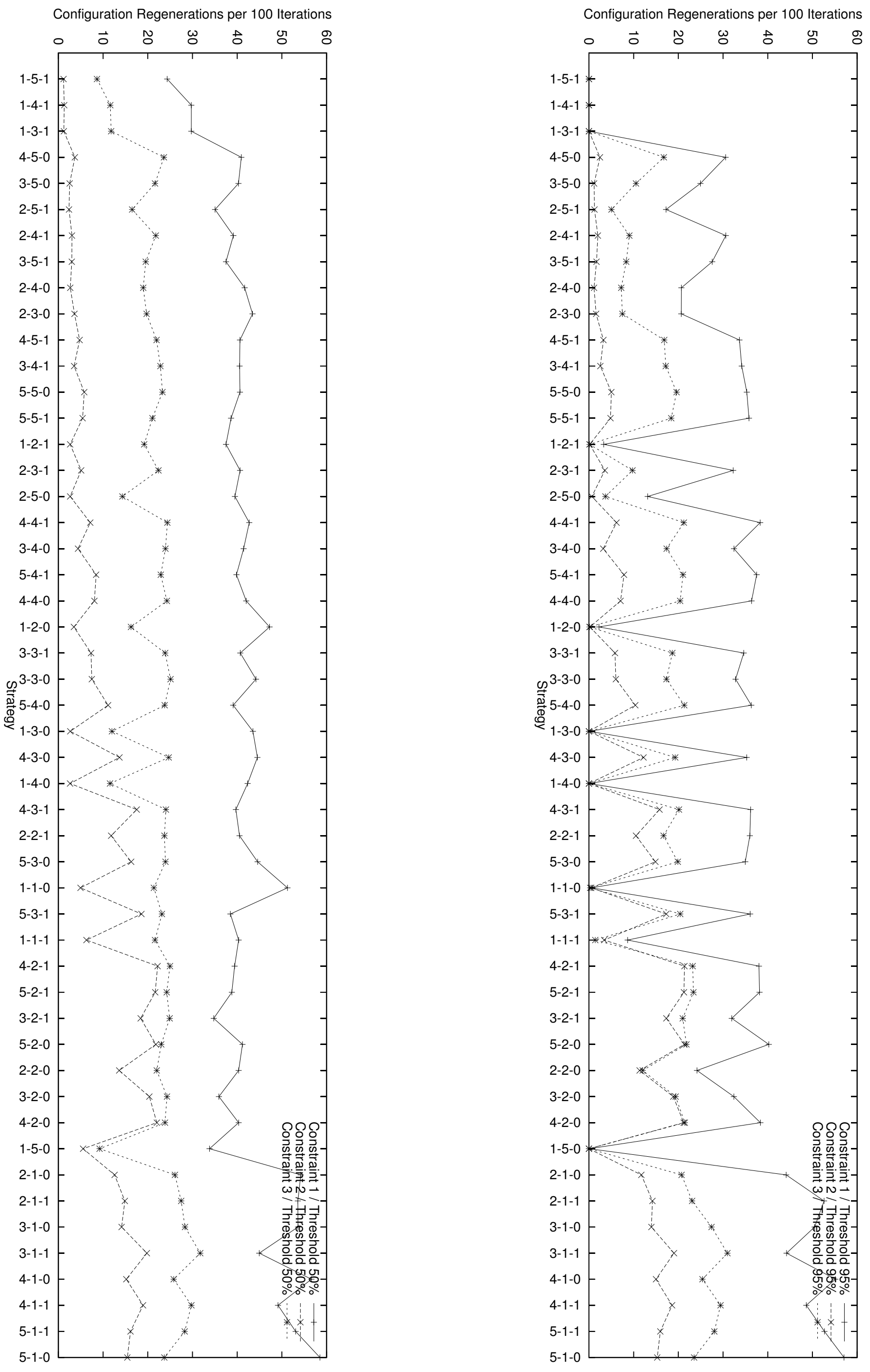


Figure 1.9. Number of configuration regenerations for the Orc Quest problem.

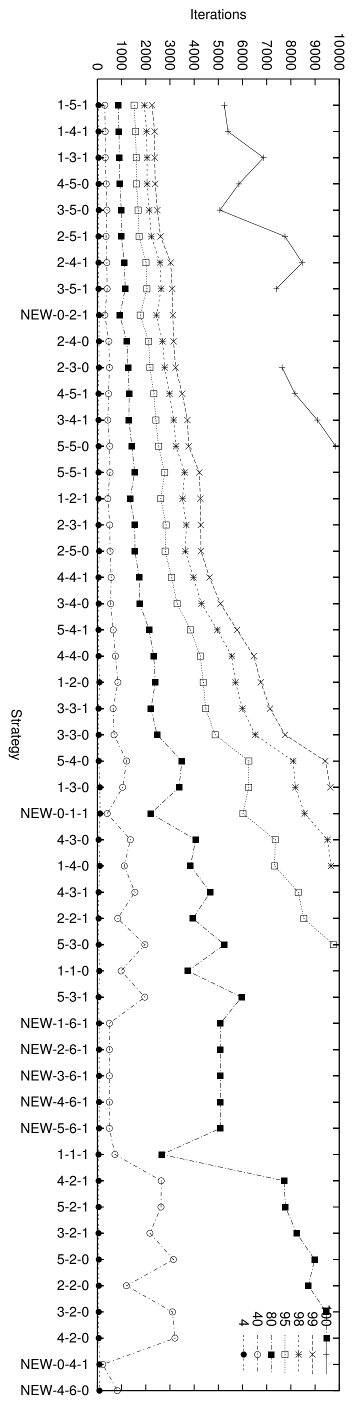
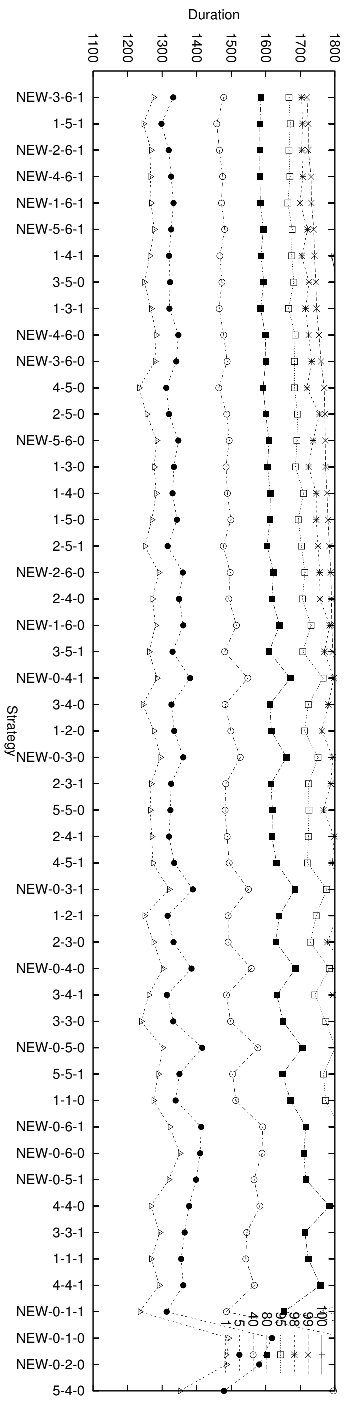


Figure 1.10. Extended weight-adaptation results for the Orc Quest problem; showing only the 50 best strategies.



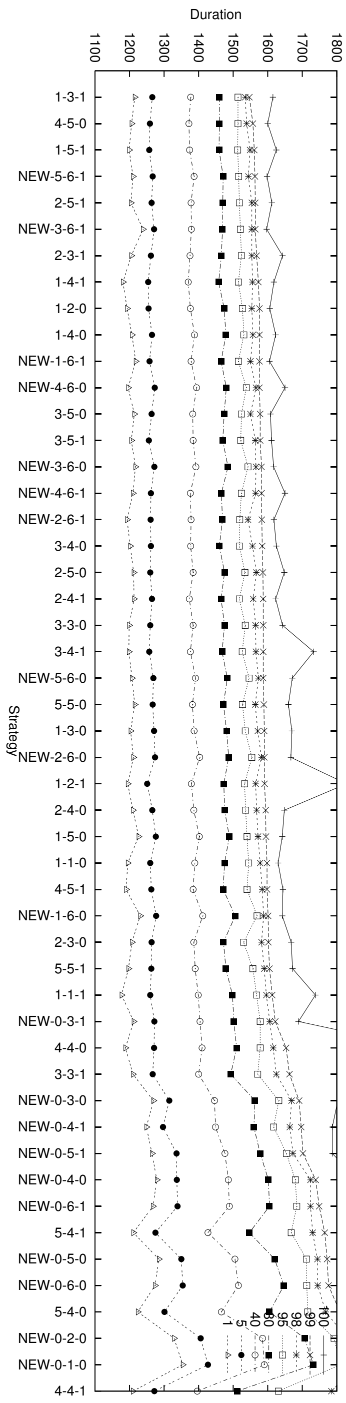


Figure 1.11. Extended weight-adaptation results for the Logistics Domain after 25,000 (left) and 100,000 (right) iterations; showing only the 50 best strategies.

Stationary Reinforcement Learning

So far, we have looked at different methods to adapt the weights during search, assuming that different areas of the search space can be handled more efficiently using different search strategies. Although this assumption seems to be intuitively correct, it remains to be shown to be true. This section, then, compares adaptive non-stationary learning with stationary approaches.

Previous approaches adapted learning parameters *after a complete run* or *when a local minimum was reached*. Of these two options, only an adaptation after a complete run (with an upper bound of a specific number of iterations) is applicable here because we do not evaluate the whole neighborhood and cannot therefore tell if we are in a local minimum. The time taken by the learning process to find an optimal stationary weight distribution is not measured because the results may be very different for different learning techniques. Thus, the adaptive learning strategies are compared with the optimal stationary distribution.

For the Orc Quest problem, we can actually find an optimal static weight distribution such that all test runs find the optimum in about **35** iterations. With the most simple, non-stationary `1-1-0` strategy, 50 % of the test runs found the optimum after **1,305** iterations, and after **405** iterations for strategy `1-5-1`. Thus, an adaptive strategy would seem to perform very poorly for the simple Orc Quest problem. This is not completely true, however, given the time that would be required to learn the optimal stationary distribution. For example, using a simple static distribution such that every heuristic is chosen equally often, **none** of the 100,000 test runs found the optimum within 100,000 iterations. We conclude that, if the problem (or very "similar" problems) is solved very often, a stationary reinforcement learning approach will ultimately perform much better; but for a short time-frame, the non-stationary approach is probably much superior.

For the more complex Logistics Domain, our findings are different. The performance of even the most simple, non-stationary `1-1-0` strategy is similar to that of a carefully hand-tailored static weight distribution, i.e., a static distribution does not work well even disregarding the learning time for it (see Fig. 1.12). Our assumption that it is useful to switch between different heuristics during search in order to adapt to specific regions of the search space proves valid for this more complicated problem.

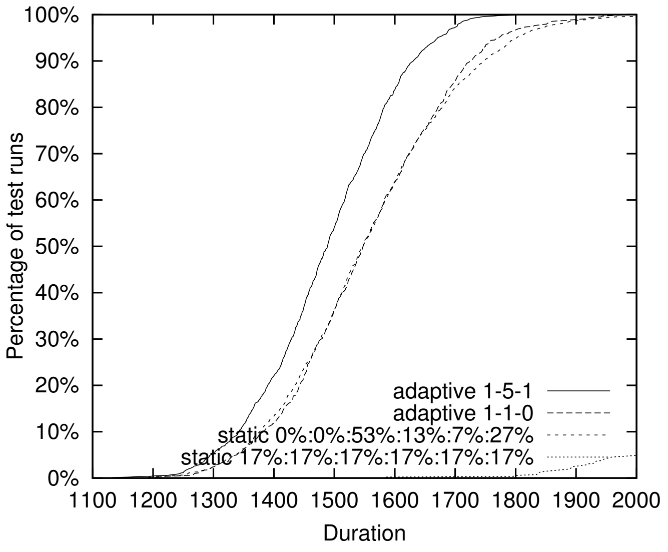


Figure 1.12. Adaptive vs. static strategies for the Logistics Domain (after 25,000 iterations), disregarding the learning time for the static distribution.

5. Conclusion

The use of a neighborhood of repair heuristics is a promising way to implement a local search — especially for complex real-world problems in which the computation of a state's cost function value is often very costly. Using the repair heuristics, domain-dependent knowledge to guide the search can easily be incorporated into the search process. The approach used here is based on (Nareyek 2001 (a)). Similar techniques were applied in (Rabideau et al. 1999, Smith 1994, Zweben et al. 1994).

However, finding an appropriate strategy that guides when to apply which heuristics is not easy. This article has presented a promising approach to learn a selection strategy by modifying weights. Other approaches that make use of weights for re-structuring the neighborhood include (Boyan and Moore 1997, Schuurmans and Southey 2000, Voudouris and Tsang 1995). Unlike these approaches, we do not only change the weights when a local optimum or solution is reached. Search usually undergoes different phases (i.e., search-space regions) in which different search heuristics work best. Thus, even during search, the heuristics' configuration is constantly updated. For this purpose, a less-carrot-more-stick strategy seems to be most adequate, allowing for quick configuration changes and preventing the old configuration from being re-established too quickly.

WGSAT Frank (1997) follows a similar strategy, even though weights are connected to problem features (i.e., SAT clauses) instead of improvement heuristics. The approach is not really comparable, but it is close to a `P:0-N:1` strategy with an additional decay factor.

In reinforcement learning, non-stationary environments (such as the search-space region) are only rarely considered. Examples include approaches based on supervised techniques (Schmidhuber 1990), evolutionary learning (Littman and Ackley 1991) and model-based learning (Michaud and Matarić 1998). Unlike these approaches, we have used a modification of standard action-value methods (Sutton and Barto 1998), applying functional updates instead of cumulative value additions in order to influence the impact of the already learned reinforcements. This simple method enables the search to compute weight updates very quickly – which is very important for a local search environment because a single iteration should consume only very little computing power.

Adaptive weighs are not restricted to local search; they can also be used for (esp. restart-based) refinement search. Examples include the pheromone trails in ant colony optimization (Dorigo et al. 1999), the use of domain-specific prioritizers (Joslin and Clements 1999) and action costs in adaptive probing (Ruml 2001). The results obtained in this study may be transferred to these areas, and techniques like pheromone evaporation are worth studying for neighborhoods of heuristics as well.

So far, we have not considered quantitative cost-function effects of decisions. Improvement or non-improvement was the only criterion for learning. However, "good" heuristics may not be equally good on the quantitative level and incorporating mechanisms to exploit the quantitative differences is a promising idea for future work. An interesting approach in this direction was presented at this conference by Cowling et al. (2001).

The presented techniques are integrated into the **DragonBreath** engine, which is a free optimization engine based on constraint programming and local search. It can be obtained via:
`http://www.ai-center.com/projects/dragonbreath/`

Acknowledgments

Thanks to Michael Littman for his feedback.

Bibliography

Boyan, J. A., and Moore, A. W. Using Prediction to Improve Combinatorial Optimization Search. In Proceedings of the Sixth International Workshop on Artificial Intelligence and Statistics (AISTATS-

97), 1997.

Cowling, P.; Kendall, G.; and Soubeiga, E. A Parameter-Free Hyperheuristic for Scheduling a Sales Summit. In Proceedings of the Fourth Metaheuristics International Conference (MIC'2001), 127–131, 2001.

Dorigo, M.; Di Caro, G.; and Gambardella, L. M. Ant Algorithms for Discrete Optimization. *Artificial Life* 5(3): 137–172, 1999.

Frank, J. Learning Short-Term Weights for GSAT. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97), 384–391, 1997.

Hoos, H. H., and Stützle, T. Evaluating Las Vegas Algorithms — Pitfalls and Remedies. In Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-98), 238–245, 1998.

Joslin, D. E., and Clements, D. P. Squeaky Wheel Optimization. *Journal of Artificial Intelligence Research* 10: 353–373, 1999.

Kirkpatrick, S.; Gelatt, C. D.; and Vecchi, M. P. Optimization by Simulated Annealing. *Science* 220(4598): 671–680, 1983.

Littman, M. L., and Ackley, D. H. Adaptation in constant utility nonstationary environments. In Proceedings of the Fourth International Conference on Genetic Algorithms, 136–142, 1991.

Michaud, F., and Matarić, M. J. Learning from History for Behavior-Based Mobile Robots in Non-Stationary Environments. *Machine Learning* 31, Joint Special Issue on Learning in Autonomous Robots, 141–167, 1998.

Nareyek, A. (a) Using Global Constraints for Local Search. In Freuder, E. C., and Wallace, R. J. (eds.), *Constraint Programming and Large Scale Discrete Optimization*, American Mathematical Society Publications, DIMACS Volume 57, 9–28, 2001.

Nareyek, A. (b) *Constraint-Based Agents – An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds.* Reading, Springer LNAI 2062, 2001.

Rabideau, G.; Knight, R.; Chien, S.; Fukunaga, A.; and Govindjee, A. Iterative Repair Planning for Spacecraft Operations in the ASPEN System. International Symposium on Artificial Intelligence Robotics and Automation in Space (iSAIRAS 99), 1999.

Ruml, W. Incomplete Tree Search using Adaptive Probing. In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01), 235–241, 2001.

Schmidhuber, J. Making the World Differentiable: On Using Self-Supervised Fully Recurrent Neural Networks for Dynamic Reinforcement Learning and Planning in Non-Stationary Environments. Technical Report, TR FKI-126-90, Department of Computer Science, Technical University of Munich, 1990.

Schuurmans, D., and Southey, F. Local search characteristics of incomplete SAT procedures. In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), 297-302, 2000.

Smith, S. F. OPIS: A Methodology and Architecture for Reactive Scheduling. In Zweben, M., and Fox, M. S. (eds.), *Intelligent Scheduling*, Morgan Kaufmann, 29–66, 1994.

Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction.* Reading, MIT Press, 1998.

Voudouris, C., and Tsang, E. Guided Local Search. Technical Report CSM-247, University of Essex, Department of Computer Science, Colchester, United Kingdom, 1995.

Zweben, M.; Daun, B.; Davis, E.; and Deale, M. Scheduling and Rescheduling with Iterative Repair. In Zweben, M., and Fox, M. S. (eds.), *Intelligent Scheduling*, Morgan Kaufmann, 241–255, 1994.