Integrating Local-Search Advice Into Refinement Search (Or Not)

Alexander Nareyek, Stephen F. Smith, and Christian M. Ohler

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891, USA
`alex@ai-center.com`, `sfs@cs.cmu.edu`, `ohler@cs.cmu.edu`

Abstract. Recent work has shown the promise in using local-search "probes" as a basis for directing a backtracking-based refinement search. In this approach, the decision about the next refinement step is based on an interposed phase of constructing a complete (but not necessarily feasible) variable assignment. This assignment is then used to decide on which refinement to take, i.e., as a kind of variable- and value-ordering strategy.

In this paper, we further investigate this hybrid search approach. First, we evaluate methods for improving probe-based guidance, by basing refinement decisions not only on the final assignment of the probe-construction phase but also on information gathered during the probe-construction process. Second, we consider the relative strengths of probe-based search control and search control that is biased by more classically motivated variable- and value-ordering heuristics (incorporating domain-specific knowledge). The approaches are evaluated on various problems from the job-shop scheduling domain.

Our results indicate that — while probe-based search performs better than an uninformed search — use of domain-specific knowledge proves to be a much more effective basis for search control than information about constraint interactions that is gained by local-search probes, and leads to substantially better performance.

1 Introduction

A broad range of combinatorial problems is naturally formulated as constraint satisfaction problems (CSPs), and as such, the design of efficient and general search techniques for solving CSPs has attracted much attention in recent years.

Generally, these search methods are based on the idea of refinement, i.e., stepwise reduction in the value domains of the decision variables until each variable's domain has exactly one admissible value. Refinement-based search methods are classically formulated within a backtracking search framework. In this framework, constraint propagation techniques are applied at each refinement step, and provide a basis for detection and early pruning of infeasible states. For example, suppose we have two finite variables A and B with domains of $\{1 \dots 10\}$

and a constraint $B > A$. In the case of a chosen refinement of $A \in \{5 \ldots 10\}$, the propagation will entail a domain reduction of B to $\{6 \ldots 10\}$. Refinement decisions and propagation are iteratively repeated until a solution is found. If the refinement that was made turns out to be inconsistent later on, which means that no more possible values are left in a variable's domain, backtracking is applied to choose another refinement option.

Search-control heuristics for refinement search typically exploit measures of flexibility/inflexibility in the evolving partial solution — the sizes of domains of various variables in the most basic case. The effectiveness of these heuristics rests on the ability to properly distinguish more-constrained decisions from less-constrained decisions, which can be more or less evident from propagation results at different stages of the search and at different levels of solution difficulty. Since, in the case of domain-independent heuristics, guidance is usually based directly on (local) constraint propagation results, they can suffer from the lack of a global perspective on decision constrainedness. Domain-specific heuristics (e.g., [2, 3]) offer one approach to providing more globally-motivated search guidance.

Search methods that are based on local search, on the other hand, generate a solution by repeatedly revising a concrete value assignment of the variables. For example, the variables A and B could initially be assigned with values of $A = 1$ and $B = 1$. As long as constraints are inconsistent with this assignment, such as $B > A$, the assignment will iteratively be revised. Changes are typically guided by some inconsistency measurement. For example, if the repair methods allow search to increase or decrease a variable, B might be changed to $B = 2$ in order to lower the inconsistency of the constraint $B > A$. Such repairs are iteratively applied until a solution is found or the search is stopped.

Local-search methods tend to build up a global picture of the "hot spots" currently causing inconsistency, but they lack a more systematic way of resolving these constraint interactions.

Given the complementary strengths of refinement and local-search approaches, it makes sense to combine them. Use of local search can inject a global perspective to refinement search's control decisions; refinement search can provide a systematic basis for exploring tightly constrained regions of the underlying search space. We focus specifically on use of the inconsistency measure of local search as guidance for determining which refinements to apply within a refinement search. Some theoretical justification for this model is given in [9], where it is proved that a backtracking-based refinement search guided by information of a complete assignment has a much better average run-time behavior than previously analyzed algorithms that were not.

A number of researchers have proposed approaches along these lines. [5] and [14] applied the GSAT local-search method [11] to generate complete assignments within a backtracking refinement search in the context of solving propositional satisfiability (SAT) problems. Similarly, [10] studied the use of a linear programming solver as a complete-assignment generator to guide a backtracking refinement search, obtaining results that outperformed all alternative algorithms applied to the problem of minimizing perturbations in dynamic scheduling. A variation of this approach was presented in [7], using a local-search approach instead of a linear programming solver.

In this paper, we extend and further evaluate the idea of integrating local search as a basis for more globally informed control of refinement-based CSP search. First, we consider the added benefit of factoring information related to the local search's assignment history into search-control decisions. The work mentioned above has relied strictly on analysis of the final assignment generated. Second, we consider the relative strengths of probe-based heuristics in relation to the search control bias that is provided by more classical CSP variable- and value-ordering heuristics that incorporate knowledge of the problem domain at hand. We compare the use of probe-based heuristics to an uninformed refinement search and evaluate several solver configurations that combine probe-based and domain-specific refinement heuristics in different ways.

We investigate these issues in the application domain of job-shop scheduling. We concentrate specifically on the problem of generating a feasible job-shop schedule. In brief, a problem consists of n jobs, each of which requires execution of a sequence of m tasks. Each task i of job j has a fixed duration $p_{j,i}$ and requires a specific resource $r \in R$. $|R| = m$, and each job visits all resources in some order (determined by its task sequence). All jobs must be completed by a specified global due date h, and a resource can only perform one task at a time. The goal is to find starting times for the tasks that satisfy all time and resource constraints. Contention for resources by competing tasks is what makes the problem difficult (NP hard). Resource contention is also reflective of global constraint interactions, and hence representative of the sort of domain where local search might be expected to boost refinement-search performance.

We start by describing the assumptions and components of our integrated search framework.

2 Solver Integration

To bring local and refinement search together, [14] argue in favor of a system that uses attributed variables, i.e., which allows the local-search solution's values to be stored directly within the refinement search's solution structures, using the same problem modeling for both search techniques. While this certainly eases problem modeling, it has the major drawback that models cannot be created in a way to enable structure exploitation for both search approaches. Search performance is highly dependent on appropriate models, and we thus instead use an integration of two solvers, both based on constraint programming but specifically designed for local and refinement search respectively.

2.1 The Refinement Solver

The base refinement-search solver is built using the Comirem planning and scheduling framework [13], which promotes a general view of problem solving as an incremental constraint posting process. At its core is a Simple Temporal Problem (STP) constraint-network solver [6]. All input tasks, durations, sequencing constraints, and deadline constraints are translated into a graph of time points and distance constraints, and constraint propagation is performed to establish initial time bounds on the start and end of each task. The STP constraint network solver is then invoked to update these start-time and end-time bounds and to detect constraint conflicts (infeasible states) as constraints corresponding to each new scheduling decision are added to the network.

The process of scheduling in this model is not concerned with assigning task start times, but instead aims to feasibly sequence the set of tasks that are competing for each resource.[1] A basic refinement step in the search involves two decisions: (1) selecting an as yet unsequenced task and (2) selecting where to insert this task into the partial sequence that has been established thus far on the required resource's timeline. We refer generally to a feasible position in a resource timeline as a "slot". In the underlying constraint network, the selection of a slot may result in the posting of one or two precedence constraints between competing tasks, depending on whether the task is inserted at the beginning, at the end, or somewhere in the middle of the resource's timeline. Thus, a final solution will typically designate a set of possible start times for any given task (all feasible) rather than committing to a single point.

At each refinement step, the base refinement procedure recomputes the feasible options (slots) for each uninserted task and applies search control heuristics to determine the next decision. In the event that an infeasible state is detected, the search backtracks chronologically and considers alternative slots for previously inserted tasks. Thus, the basic scheduling procedure is complete if given enough time to execute, and search control heuristics are used to improve average case performance.

In this paper we make use of two different heuristics for task selection, each aimed at selecting the most temporally constrained task for insertion next:

- *Least Options First* (`LOF`) - $Min_{i \in UninsertedTasks}|Slots_i|$. In case of ties, then $Min_{i \in UninsertedTasks}(Min(lft_i - est_i - p_i, \sum_{s \in Slots_i}(lft_s - est_s - p_i)))$, where lft_x is latest-finish-time of x, est_x is the earliest start time, and p_i is the processing time of i.
- *Minimum Slack First* (`MinSlack`) - $Min_{i \in UninsertedTasks}(Min(lft_i - est_i - p_i, \sum_{s \in Slots_i}(lft_s - est_s - p_i)))$, where lft_x, est_x, and p_i are defined as above.

Intuitively, the number of feasible options (slots) remaining provides one basic estimate of temporal constrainedness. However, given the nature of the search space, the insertion of a task into a given resource's schedule (sequence) can sometimes increase the number of options for other pending tasks that require this same resource, and hence LOF decisions may not always be directly correlated to temporal constrainedness. MinSlack provides a simpler but potentially more direct measure of a task's current degrees of freedom[12].

---

[1] Hence, constraint propagation through the STP network in this context implements enforcement of disjunctive resource constraints as described in [1].

For option selection, we adopt the following heuristic:

- *Maximum Slack First* (`MaxSlack`) - $Max_{s \in Slots_i}(lft_s - est_s - p_i)$.

Intuitively, the option that retains the most temporal flexibility is selected by `MaxSlack`.

To provide an alternative basis for search control, we also define non-deterministic versions of these task selection and option selection heuristics. Following the principle of calibrating the level of non-determinism to the discriminatory power of the search heuristic in a given decision context [4], we randomize these heuristics in a value-biased manner. Specifically, we modify the heuristic values assigned to each choice by a bias function of the form $value^b$, and use the resulting numbers to define choice probability. We refer to the randomized counterparts of the above heuristics as `LOF(b)`, `MinSlack(b)`, and `MaxSlack(b)` respectively.

2.2 The Local-Search Solver

The local-search solver is a modification of the DragonBreath engine[2] (see [8] for details). The solver is a general constraint-programming system and not specialized to scheduling problems. However, it is easy to express the job-shop scheduling problem by way of the given constraint types.

Two constraint types are used to model the problem: a non-overlap constraint for a machine's tasks and a linear-inequality constraint to describe a temporal relation between tasks. In the search approach of the DragonBreath engine, every constraint calculates an inconsistency value using a constraint-specific measure indicating how far off the involved variables' assignment is from a consistent solution. For example, the non-overlap constraint returns costs related to the time spans during which involved tasks are overlapping.

The sum of the constraints' inconsistencies represents the total inconsistency of the solver's current assignment. In each improvement iteration then, an inconsistent constraint is selected, and the constraint selects one of its constraint-specific heuristics to change its variables' values in order to reduce the constraint's inconsistency. For example, a non-overlap constraint might select a heuristic to shift a task to a time at which it causes less overlaps.

The search landscape does not have local minima because the applicability of the constraints' heuristics is not restricted to greedy improvements of the overall inconsistency. Randomization and learning techniques are applied to promote exploration.

In contrast to the refinement solver, the local-search solver has concrete values assigned for all variables — the tasks' starting times in this case — at any time. The local-search solver therefore also has a complete picture about the currently involved inconsistencies, which makes it easier to identify potential "trouble spots", i.e., regions where it is not easy to establish consistency.

---

[2] The engine is freely available at:
`http://www.ai-center.com/projects/dragonbreath/`

2.3 The Solvers' Interaction

The interaction of the solvers is shown in Figure 1. Refinement search is the master process, using the local-search solver for heuristic guidance. Both solvers have different internal representations of the job-shop scheduling problem and communicate in reference to decision variables, which are the tasks' starting times. For every refinement decision of the refinement solver, the local-search solver is notified to internally add a corresponding constraint `<4>` so that both solvers keep working on the same problem. In case of backtracking, this constraint is removed again `<5>`.

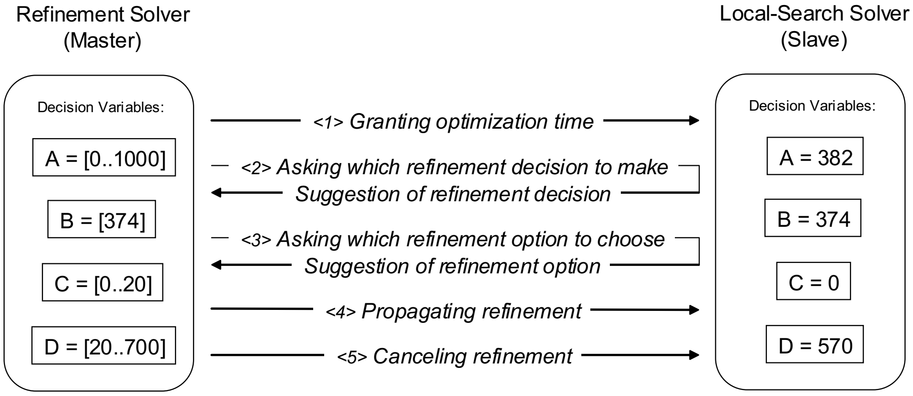
Refinement Solver
(Master)
Local-Search Solver
(Slave)
Decision Variables:
A = [0..1000]
B = [374]
C = [0..20]
D = [20..700]
<1> Granting optimization time
<2> Asking which refinement decision to make
Suggestion of refinement decision
<3> Asking which refinement option to choose
Suggestion of refinement option
<4> Propagating refinement
<5> Canceling refinement
Decision Variables:
A = 382
B = 374
C = 0
D = 570

Fig. 1. Solver integration.

The local-search solver can suggest, at each refinement step, which refinement decision to make and which refinement option to choose. For example, in case of the job-shop scheduling problem, the local-search solver suggests for which task an ordering decision should be made next `<2>` and which ordering constraints (before or after) relative to other tasks on the same machine should be chosen `<3>`. Since we are also interested in exploring the interplay of local search and the base refinement heuristics, we also admit configurations that restrict the use of local-search guidance to either task selection `<2>` or ordering decisions `<3>`.

Refinement recommendations of the local-search solver are generally made in a way to focus on critical regions, i.e., recommending variables for refinement that are related to the highest amount of inconsistency, and to steer the local-search solver's assignment away from inconsistencies, i.e., recommending values that minimize these inconsistencies (see following sections for details).

At each refinement step, the local-search solver is given some time for optimization `<1>` to adapt to the current situation before a recommendation is queried. In our current implementation, this time is split into a time for general optimization (500 iterations for the test runs given below), and a time in which the local-search solver concentrates on satisfying only the refinement decisions already taken (maximally 200 iterations for the test runs given below). There is also some optimization time granted to the local-search solver before the refinement (1000 iterations for the test runs given below). If at any point during the overall refinement search, the local-search solver finds a feasible solution to the problem at hand, then this solution is returned and the refinement search terminates in success.

3 Test Problems

Our sample problems are taken from the OR-library[3]. They include three problems with ten jobs and ten resources ("1000/abz5", "1006/ft10" and "1051/orb04"). We also ran other problems of different sizes to verify our findings; the results confirm our findings but are not included here. Note that the commonly published makespan benchmarks for these problems are not applicable here because we tackle the satisfaction problem — searching for a solution within a fixed horizon — instead of optimizing the makespan.

For each test run, there is a cut-off at 1000 refinement steps. Most variants discussed below involve randomization components. At least 50 test runs are thus run per configuration.

Three different difficulties are identified per problem to define problems of increasing hardness. Problem difficulty is a function of the tightness of the global due date. To determine problem difficulties, the temporal distance between an infinite capacity solution (lower bound) and a due date equal to $\sum_{i \in Tasks} p_i$ (upper bound) was divided into 100 increments. The "hard" difficulty represents the loosest global due date (moving from upper to lower) for which the refinement solver alone (i.e., using only the base refinement heuristics `LOF` and `MaxSlack`) cannot produce a solution. The "medium" difficulty is obtained by setting the global due date one step toward the upper bound and "easy" is obtained by moving 10 steps further in that direction.

4 Involving Local Search's Search History

In previous approaches, a recommendation of the local-search solver was based on the variable assignment (and the involved inconsistency in consequence) after a specific amount of time was given to the local-search solver for improvement[4]. We will refer to this as strategy `CI` (like "Current Inconsistency") in the following. For this strategy, after the local-search phase, the inconsistencies of every constraint that a task is involved in are summed up, and the task with the highest inconsistency sum is recommended for a refinement decision. For the

---

[3] `http://graph.ms.ic.ac.uk/info.html`

[4] Note that there are also some approaches in which local search is run until a local minima is reached, or — using local search only for sub-problems — in which local search is run until a fully satisfying solution is found. Approaches like this, however, provide little temporal control for the search process.

recommendation of the ordering, all current orderings of a task with respect to the selected task are considered, and the reverse of the ordering involving the highest inconsistency is recommended.

However, this approach only exploits the information that is available after the improvement of local search has been completed. If local search is considered to be an intelligent sampler of a search space, this means that only one sampling is returned by this approach. Moreover, the assignments at this time may just represent a short-term anomaly caused by a bad last local-search move. We thus vary the recommendation strategy in the following by additionally involving information gathered during the complete local-search phase, incorporating information of multiple assignment samples thereby.

4.1 Iterations of Inconsistency

The strategy analyzed in this section makes use of demons that record specific features about the constraints and their inconsistency during the local-search improvements.

For every pair of tasks t_a and t_b on a resource, there is a demon d_{ab} recording the number of iterations in which t_a was preceding t_b and constraints involving t_a or t_b were inconsistent. There is also a complement demon d_{ba} for situations in which t_b was preceding t_a. When a recommendation is to be made, the demons propagate their iteration counts to the tasks involved, which sum these counts. The task with the highest number of such “iterations of inconsistency” is recommended for a refinement decision (strategy `II`). For the recommendation of the ordering, the reverse ordering of the demon with the highest iteration count is recommended. The demons’ counts are reset to 0 every time a refinement decision is made or canceled/backtracked by the refinement solver.

4.2 Average Inconsistency

For the following strategy `AI`, we additionally involve the constraints’ quantitative inconsistency values. In contrast to strategy `II`, not only the number of inconsistent iterations is recorded by the demons but also the quantitative inconsistency value that is involved. The average inconsistency during inconsistent iterations is then propagated to the tasks involved and used as recommendation criterion.

4.3 Bound Propagation

In addition to notifying local search about the ordering refinements adopted, the refinement-search solver can also inform local search about changes to the time bounds of individual start-time variables, which are updated during constraint propagation when a refinement decision is made. Implicitly, local search already has this information by way of the orderings, but additional explicit constraints may help local search to fulfill these bound constraints. Bounds represent hard constraints in the DragonBreath engine, i.e., a variable's value will immediately be shifted back into the bound if a heuristic tries to assign a value outside the bound. Thus, the propagation of such bounds may have a much larger impact than the implicit information. The extension of strategy `AI` by propagating these bounds to local search will be called `AI-B` in the following.

4.4 Results

Figures 2 to 4 show the results for all strategies. It can be seen that the strategy `AI` outperforms the other two and confirms our thoughts that the additional quantitative information is crucial to improved search performance.

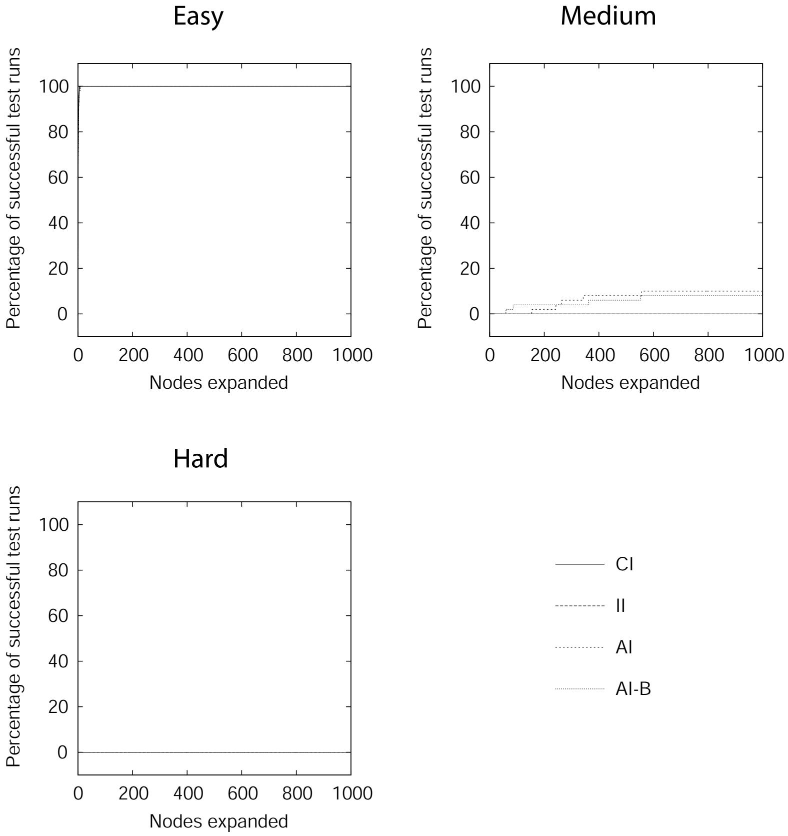

Fig. 2. Comparing variants of local-search recommendations — problem 1000.

Strategy `II` does not work out well. One reason might be that – especially in problems with tight global deadlines – strategy `II` can hardly differentiate

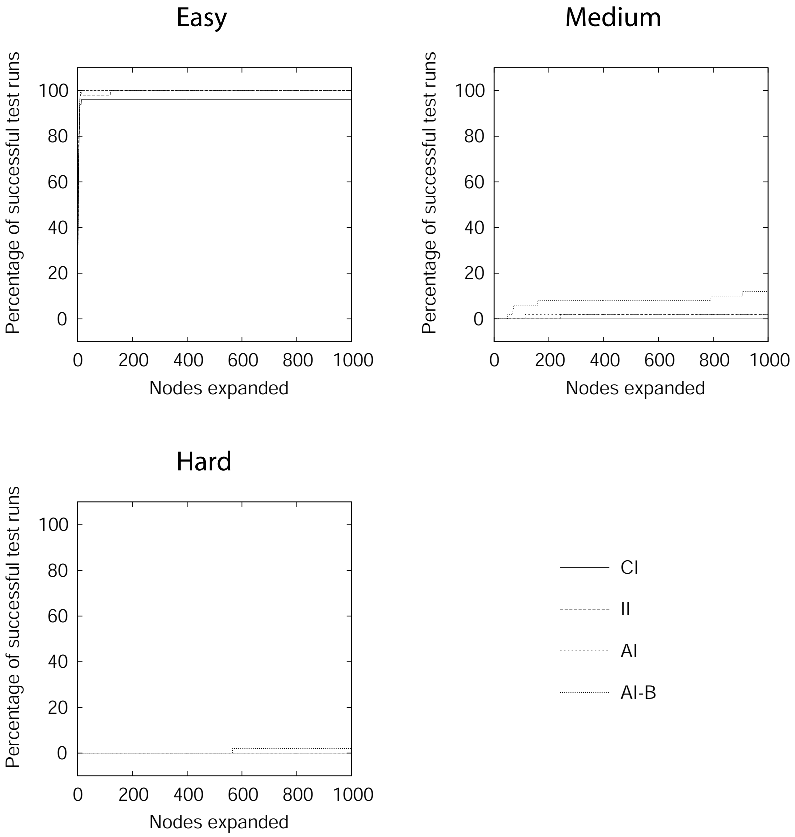

Fig. 3. Comparing variants of local-search recommendations — problem 1006.

between task options because most constraints are unsatisfied during the entire run.

The result of propagating bounds — strategy `AI-B` — does nearly always perform better than strategy `AI`. However, keep in mind that the x-axis shows the number of refinement steps/nodes. Propagating bounds to the local-search solver involves computation costs, and it depends on the applied solvers and communication speed if this additional overhead pays off.

In some rare cases, propagating bounds worsens the results. An explanation for this might be that in certain situations, the application of hard constraints for bounds restricts local search's ability to move to other regions of the search space, sampling less of the relevant assignments thereby.

5 Comparison to Uninformed Recommendations

We were able to improve the use of local search, but have not yet demonstrated that the advice that is extracted by using the inconsistency information of local

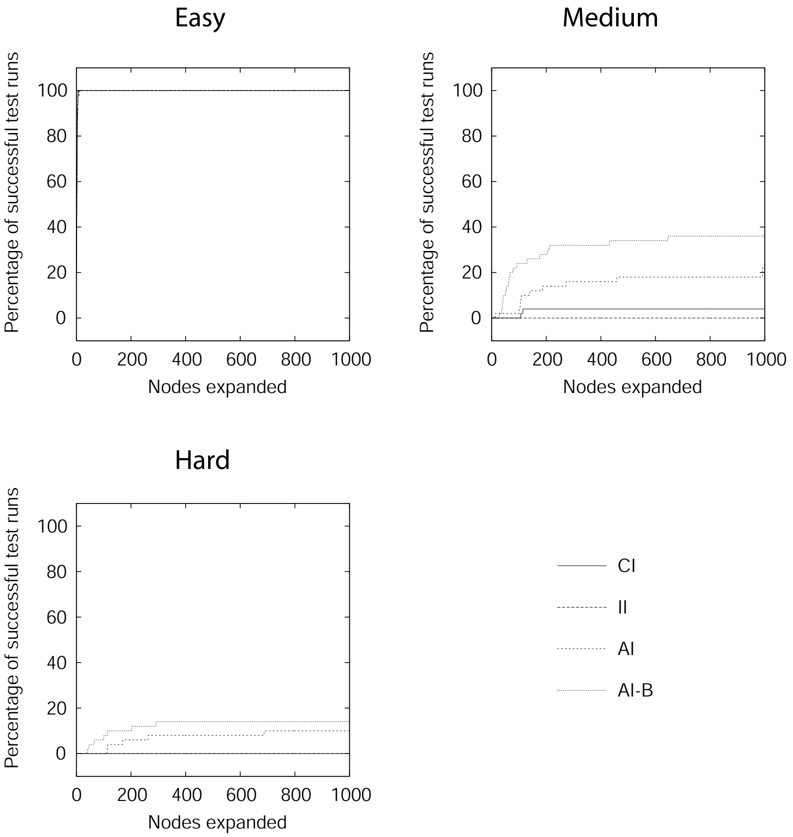

Fig. 4. Comparing variants of local-search recommendations — problem 1051.

search's samplings guides search in a better way than a simple randomized choice for task selection and ordering at each refinement step (`RND`).

Considering only the hard problems (which cannot be solved deterministically), it is difficult to discriminate performance. `AI-B` is able to find solutions for a small number of runs and `RND` never produces a solution on any runs. The utility of `AI-B` over `RND` is more evident if we restrict use of `AI-B` (resp. `RND`) guidance only to task selection and rely on the refinement solver's base `MaxSlack` heuristic for option selection. Figure 5 shows the results for these configurations.[5]

If a local-search solver is used, however, the costs of running the local-search optimization and the overhead for solver communication must be considered. This search overhead may vary depending of the applied solvers and integration, but in our experimental setting, it turned out to be a substantial overhead that hardly justifies the improvement gained in search guidance.

---

[5] Experiments that restricted `AI-B` (resp. `RND`) guidance to option selection were also run, but these configurations were not as effective.

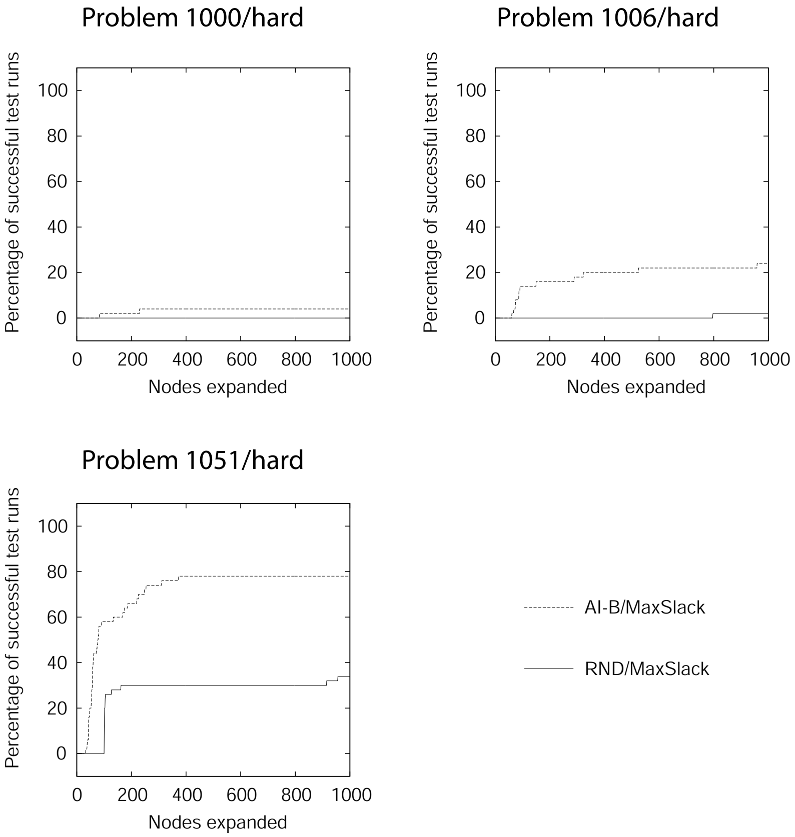

Fig. 5. Comparison to uninformed refinements.

6 Comparison to Domain-Specific Heuristics

We have shown that the inconsistency information of local search's sampling is a well-informed heuristic. However, for many problems, it may be easy to come up with domain-specific refinement heuristics. In the previous section, it already became clear that using the `MaxSlack` refinement heuristic for option selection instead of local search's recommendation drastically improves performance.

Figure 6 shows the impact of the refinement heuristics described in Section 2.1 for task selection. Despite the fact that deterministic `LOF/MaxSlack` is unable to solve any of the hard instances, configurations of `LOF(b)/MaxSlack` for various bias values outperform `AI-B/MaxSlack` in most cases. A second configuration `MinSlack(b)/MaxSlack` that utilizes the simpler `MinSlack` heuristic for task selection was found to consistently perform better than `AI-B/MaxSlack`. Moreover, even the deterministic variant `MinSlack/MaxSlack` is able to solve two of the three hard problems with just one or two backtracks. These results

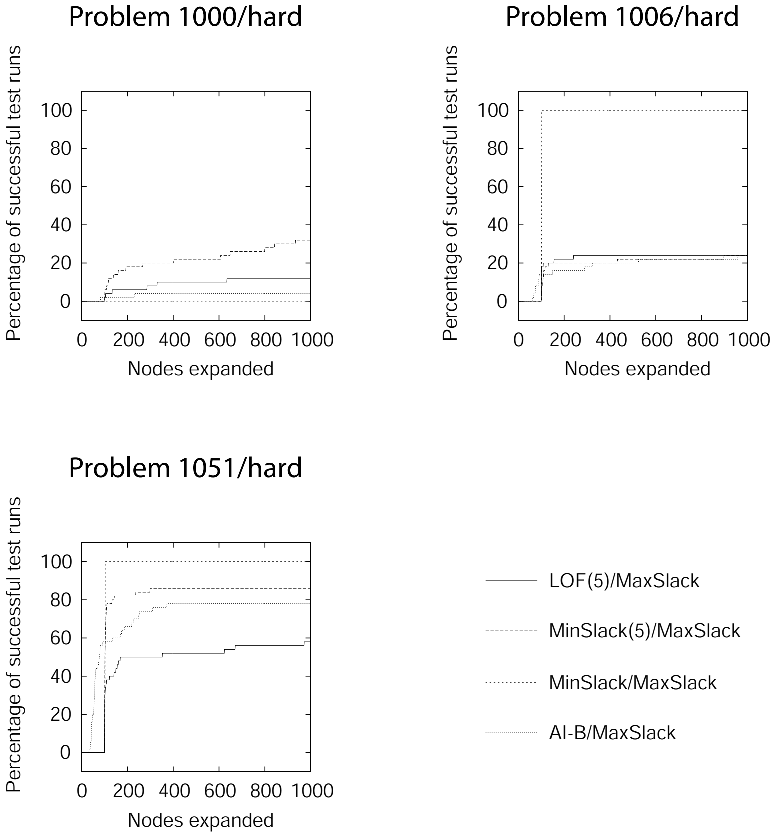

Fig. 6. Comparison to domain-specific heuristics.

are all the more significant when coupled with the fact that use of these refinement heuristics is *substantially* cheaper than calling the local-search solver.

7 Conclusion

We have studied the integration of local-search advice into a refinement search. The advice was provided by suggesting refinements that guide refinement search away from the inconsistency of local search's assignment samplings.

In the approaches of previous work, local search was allowed to run for a while, and only the final assignment was used for recommendations. The local-search assignment at this time may be of high quality in terms of local search's cost metric, but we have shown that integrating the information of intermediate search steps results in better advice. Intuitively, the advice includes more possible assignments that were sampled by local search this way.

We have shown that the inconsistency of local search's assignment can provide productive heuristic guidance. However, the computational costs for the integration overhead and local-search optimization make this a rather dubious result. Moreover, even simple domain-specific heuristics seem to result in equal or better performance than the "mushy" information of inconsistency of local-search samplings. This seems to be plausible, but we were surprised how easy it is to outperform the probe-based heuristic.

An interesting question for the future is if we can develop heuristics that can classify their recommendation value, and possibly switch to the probe-based heuristic in decision contexts where they can provide only little guidance. Preliminary experiments that decrease the probability of using the probe-based heuristic with increasing refinement depth, however, did not look highly promising. The rationale for this approach was that refinement search potentially gets more informed with increasing refinement depth because the variables' domains get smaller, enabling predictions that are more precise.

The presented emipircal results only cover job-shop scheduling examples, and for other domains, where less efficient refinement heuristics are known, probe-based search may still be a viable option. Our results, however, suggest that one should be careful in investing too much efforts in improving the probe-based approach instead of thinking about more appropriate refinement heuristics.

References

1. Baptiste, P., LePape, C. and Nuijten, W. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems.* Kluwer Academic Publishers, Boston, 2001.
2. Beck, J. C. Texture Measurements as a Basis for Heuristic Commitment Techniques in Constraint-Directed Scheduling. Ph.D. Thesis, Department of Computer Science, University of Toronto, 1999.
3. Cheng, C., and Smith, S.F. Applying Constraint Satisfaction Techniques to Job shop Scheduling. *Annals of Operations Research* 70: 327–357, 1997.
4. Cicirello, V., and Smith, S. F. Amplification of search performance through randomization of heuristics. In Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002), 2002.
5. Crawford, J. M. Solving Satisfiability Problems Using a Combination of Systematic and Local Search. Second DIMACS Challenge: Cliques, Coloring, and Satisfiability, Rutgers University, NJ, 1993.
6. Dechter, R.; Meiri, I.; and Pearl, J. Temporal Constraint Networks. *Artificial Intelligence* 49(1): 61–95, 1991.
7. Kamarainen, O., and El Sakkout, H. Local Probing Applied to Scheduling. In Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP 2002), 155–171, 2002.
8. Nareyek, 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.
9. Purdom Jr., P. W., and Haven, G. N. Probe Order Backtracking. *SIAM Journal on Computing* 26(2): 456–483, 1997.
10. El Sakkout, H., and Wallace, M. Probe Backtrack Search for Minimal Pertubation in Dynamic Scheduling. *Constraints* 5(4): 359–388, 2000.

11. Selman, B.; Levesque, H.; and Mitchell, D. A New Method for Solving Hard Satisfiability Problems. In Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), 440–446, 1992.
12. Smith, S.F.; and Cheng, C. Slack-Based Heuristics for Constraint Satisfaction Scheduling. *Proceedings 11th National Conference on Artificial Intelligence* Washington D.C., July 1993.
13. Smith, S.F.; Hildum, D.; and Crimm, D. Interactive Resource Management in the Comirem Planner. *Proceedings AAAI Workshop on Mixed-Initiative Intelligent Systems*, Acapulco Mexico, August 2003.
14. Schimpf, J., and Wallace, M. Finding the Right Hybrid Algorithm – A Combinatorial Meta-Problem. *Annals of Mathematics and Artificial Intelligence* 34(4): 259–269, 2002.