Local Search for AI Planning[⋆]

Oto Mraz[0000−0002−7728−5660]

Department of Computer Science, ETH Zurich, Switzerland
otmraz@student.ethz.ch

Abstract. Recent breakthroughs in the field of AI planning such as the Identidem and Marvin planners support the creation of more advanced and realistic representations of real-world domains. It is well-known that an adequate local search strategy can help to solve increasingly complicated Planning Domain Definition Language problems. Contemporary planners, however, strive to find a balance between the traditional greedy search and a certain degree of randomness. The aim of this work is thus to introduce a new planner that combines applicable local search techniques in a novel way not explored before to enhance the performance of the existing JavaFF planner.
The new proposed planner is based on the principle of local beam search combining different successor selection methods, macros and restarts. Experimental results show that the new planner can solve considerably more problems and often within a shorter time compared to its predecessor JavaFF. Our planner could find its practical utilization in domains such as urban traffic modelling or autonomous robot control.

Keywords: AI Planning · Local Beam Search · Macros · Restarts.

1 Introduction

The field of AI planning experienced dramatic changes in the past two decades. During this time, newly developed planners have greatly increased the scope of applicable AI planning models. Among others, Coles [3] has shown how an efficient local-search technique can find the goal faster. Further research stresses the importance of a customizable planner resilient to changing domain topologies.

This paper presents a new planner on the basis of JavaFF (the Java version of Hoffman's FF planner [4]). Our planner enhances JavaFF with local-beam search, sophisticated successor selection, efficient restarts and macro actions. The paper continues with an overview of the related work in this field and discusses the algorithms used in theory. We will then describe in detail how

[⋆] This research was done in the Informatics Department at King's College London, UK. Special thanks belong to Dr. Andrew Coles for his support and guidance he provided to me while supervising my thesis.
Copyright ©2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

these local search techniques have been incorporated into the novel planner and demonstrate its results in practice. The paper concludes with a summary of the successes achieved and several outlines for future work.

2 Related Work

One of the biggest problems AI planners face are local minima which they are unable to escape. Coles' planner Identidem [3] has successfully tackled this issue using restarts inside a depth-bounded and restart-bounded search. Identidem stores the best state seen so far called *bestClosed* and can continue to explore worse states only for a set amount of time. If the search is fruitless, it restarts from *bestClosed*, explores a different path; hopes to progress further in the search.

An alternative approach is to keep more than one candidate state open. This is the motivation behind local beam search (LBS). In LBS we keep the best k states open rather than just one like in greedy search. For the purposes of AI planning, a beam will represent a state. We proceed by generating the successors of all k beams and will choose the k best ones out of them [7]. The benefit of LBS lies in the ability to share useful information between the beams and focus computational efforts on the most promising areas of the search.

Furthermore, we limit the demand for memory during search since the space complexity remains linear. Meanwhile, we maintain auspicious alternatives in case of failure [10]. When comparing with greedy search algorithms, Wilt [8] has shown, that beam search outperforms its competitors particularly when working with massive search spaces. On the contrary, all search algorithms have also drawbacks. In the case of LBS, all its beams often tend to slide into the same path, which could turn LBS into a slower version of a greedy search.

2.1 Successor Selection

In conjunction with a suitable search algorithm, the planner also requires an effective successor selection procedure. The successors will be chosen based on their Relaxed Planning Graph heuristic values (h-values) (a measure of how far from the goal we are). This work explores five distinct successor selection algorithms that operate with the h-values. The simplest one is always selecting the successor with the lowest h-value called the `BestSuccessorSelector`.

Unfortunately, this approach can lead to being stuck in some local minimum that the planner cannot escape. It is thus necessary to employ some sort of non-determinism into the search process and occasionally allow less advantageous states to be explored. One way to achieve this stochastic behaviour is by adopting certain techniques from genetic algorithms. A commonly used method is a roulette-wheel selection. In a roulette-wheel style selector, an individual is selected randomly. A state s is chosen from the set of all successors $\{s_1, \ldots, s_n\}$ with the probability:

$$p(s) = \frac{F(s)}{\sum_{i=1}^{n} F(s_i)} \tag{1}$$

where the fitness $F(s)$ of the state s is $1/h(s)$, ($h(s)$ being the h-value of s) [9].

Evolutionary algorithms tell us that *roulette-wheel selection* may not work well if the states' fitnesses are large and the differences between them are negligible. Hence we aim to amplify these differences by transforming the fitness function's arguments. *Rank-based selection* confronts this issue by sorting the states according to their h-values and assigning them ranks from the set $\{1, \ldots, n\}$; their fitness is then calculated based on their ranks. Yet another option is *linear scaling*, where the h-values of the successors are scaled down closer to 0 to give the best states a bigger advantage of being selected from the roulette-wheel [1].

The final selection method investigated was a *tournament selection*, which abandons the idea of fitness altogether. Instead, we randomly select a subset of candidate successors and let them compete in pairs in a knockout-style tournament system [1]. In each tournament match the better individual wins with a probability p_t, $0 \leq p_t \leq 1$.

2.2 Macro Actions

Another decent improvement of AI planners was described by Botea [2]. Botea used Solution Abstraction Enhanced Planner (SOL-EP) macro actions in the MacroFF planner. A macro action is essentially a sequence of two or more consecutively applied actions that can assist the planner to reach the goal quicker. A macro action allows us to skip intermediate states in the search space and helps to overcome plateaus in the search space since we can explore deeper states.

SOL-EP macros work as follows: First, the planner solves an easy problem in a given domain. From the found solution, candidate macros are extracted and pruned to keep only the logically helpful ones. Finally, the planner solves the problem once again with the help of a macro. This tests how much the macro improved the planning process. The most helpful macro(s) are then used to solve more complicated instances in that domain. Although MarcoFF successfully enhanced the FF planner with SOL-EP macros, there is still more potential in combining them with a well-guided local search, which this work explores.

3 Design of the Planner

As mentioned above, the JavaFF planner is the basis for this work. The new planner replaces JavaFF's Enforced Hill Climbing (EHC) with LBS. We have a set *open* which contains the states we are currently searching through and a *closed* set containing states visited in the past. The main LBS algorithm is implemented as follows:

1. Begin with the initial state and check if it is the goal state.
2. Add the successors of the initial state to the *open* set.
3. Iteratively search through the state space until we find a solution or reach a dead end:
   (a) Generate the successors of the states in the *open* set.

(b) If a goal state is found, return it.
(c) Discard any previously visited states.
(d) If the *open* set is empty (meaning we have reached a dead end), terminate the search as unsuccessful.
(e) If the *open* set contains more states than k, the number of beams, use a successor selector to pick a subset of size k from them. This subset will form a new generation of the *open* set.
(f) Add any not chosen states to the *closed* set.

In order to make the main LBS algorithm more robust, we decided to enhance it with restarts if the planner reaches a dead end. Like in Identidem, we have a variable *bestClosed* to store the state with the best h-value seen so far. If there are no more successor states we simply start again from *bestClosed* and search with more breadth. Note that LBS already supports a wider search space. Hence, in order to make the restarts truly effective, we perform a breadth-first search of depth 2 from *bestClosed*. This approach gives the planner a decent opportunity to grasp a more promising path in the search space. After that, the search continues once again with k beams only. Similarly to Identidem, there is also a restart bound in order to stop fruitless restarts. Lastly, we only continue restarting if the last restart helped the planner reach a more favourable state.

3.1 Successor Selection

Various types of selectors can be supported, below we describe those which we have actually implemented in the planner. The main reason behind choosing LBS as the backbone of our new planner lies in its unique property of selecting multiple successors. This allows us to design a new method incorporating two successor selectors (denoted as primary and secondary) into a single planner thus combining their benefits. The primary selector chooses the first part of the successors (the secondary one chooses the remaining ones). In addition, the user can adjust the *successorSelectorBoundary* (SSB). The variable SSB dictates the proportion of successors chosen by each selector (as a real value in the range from [0, 1]). For beam b_i, $1 \leq i \leq k$, the primary selector is used when

$$(i-1)/k < \textit{successorSelectorBoundary}. \tag{2}$$

Otherwise, b_i is obtained from the secondary selector. This particular design choice for successor selection yields a flexible application, which allows us to maintain some form of elitism in the successor selection process. The user can, namely, ensure that the current best state always proceeds into the next search iteration by opting for the `BestSuccessorSelector`, which picks the state with the lowest h-value. In case of a tie, one of the tied states is chosen at random.

In addition to the traditional `RouletteSuccessorSelector` discussed in Section 2.1, our planner also provides a `RankBasedSuccessorSelector`, which calculates the fitness $F(s)$ of a state s according to the following formula:

$$F(s) = 1/(2^{R(s)}), \tag{3}$$

where $R(s)$ is the rank of the state s. Such a formula yields a geometric distribution for the selection probabilities, the sum of which tends to 1. Moreover, this provides an elegant implementation of the rank-based selection algorithm (the remaining portion of the roulette-wheel is simply given to the best state as an extra advantage). The `ScalingSuccessorSelector`, on the other hand, operates directly with the h-values. It finds the worst h-value among the current set of successors h_{max} and computes the fitness $F(s)$ of each state s as follows:

$$F(s) = h_{max} - h(s) + 1. \tag{4}$$

Lastly, the implemented `TournamentSuccessorSelector` uses tournaments of size 2. It picks 2 states randomly and then returns the better of them with the probability 0.8 (this value worked the best in our preliminary experiments).

3.2 Macros

While the LBS algorithm combined with restarts yields a reasonably robust application capable of dealing with complex domain topologies, another key goal was to make the planner fast. In order to achieve this, we decided to enhance it by means of MacroFF's SOL-EP macros. The newly proposed planner thus comes with a *MacroGenerator*, which is responsible for macro extraction and pruning. From a solution to a training problem the *MacroGenerator* extracts a list of all pairs of consecutive actions (macros consisting of more than 2 simple actions were not considered). We then prune this list to discard macros that:

- consist of a single repeated action,
- are duplicated, i.e., we already have a pair of these actions (albeit possibly using different objects),
- do not satisfy the chaining rule – their actions do not have any parameters in common and, therefore, are unrelated to each other.

The main difference between MacroFF and our macro implementation is how candidate macro actions are evaluated. Instead of considering the number of nodes expanded (which may not always reveal the most efficient macros), we will adopt an alternative proposed by Newton [6]. Newton decided to compare the time taken to find a solution with and without the macro, which provides a more accurate measure of how useful the macro was.

Finally, as a crucial improvement to its predecessor JavaFF, our planner allows the user to adjust its behaviour to the given domain topology. Our preliminary research has shown that every successor selector, macro generation setting or even beam count has its benefits and drawbacks. Therefore, the planner was instead designed to use default values (configurations that performed best during testing) and to parse optional parameters if the user decides to plan otherwise.

Our preliminary research indicates that there does not seem to be an optimal value for the beam count k for all domains. Some domains are solved fastest by means of greedy search ($k = 1$), while others require a more thorough exploration of the search space. Of course, a higher value of k does not stop the planner from solving trivial problems, but it could drastically slow down the entire process and the planner would no longer be competitive against trivial algorithms such as best-first search. Hence, we opted for an iterative approach of increasing k, starting at $k = 1$. Each time a search fails, we start again with one more beam. The user may choose a maximum beam count; if a solution is still not found after reaching it, the planner runs the best-first search for completeness.

4 Experimental Results and Optimisation

After implementation, the planner was tested on a set of benchmark problems from the International Planning Competition (IPC) [5] using a current lab machine. The planner was run on each problem instance of a given domain ten times with random seeds $1, \ldots, 10$ to make the experiments reproducible. The experiments began with searching for optimal default values for the planner's parameters (tested on the driverlog and rovers domains).

Of course, we cannot feasibly test all possible planner configurations, hence we opted for a sequential approach of optimizing one planner component after another. The found optimal value for a given parameter will then be used for all remaining experiments. After optimisation, the new planner was tested against the original JavaFF that used Enforced Hill Climbing (EHC) on a variety of domains to assess how much the planner's performance improved (additional graphs can be found at `sites.google.com/site/otomrazgee/home/lsaip`).

The pipeline for planner optimization will proceed as follows:

1. First, the performance of each of the successor selectors will be evaluated.
2. Then the most promising selectors will be combined to find a pair, that works best together. Also, the successor selector boundary parameter will be optimised.
3. Next, macro actions will be added to the planner.
4. After that, an optimal number of beams will be found.
5. Lastly, restart options will be added, gradually increasing their amount.
6. Finally, the application will be tested against the original JavaFF planner to evaluate the improvement achieved.

When searching for an ideal combination of successor selectors the planner was run using each selector individually on the base implementation of local beam search and the planning times were recorded. The maximum number of beams was set to three, no macros or restarts were added. Perhaps the most interesting results were observed on the Rovers domain, depicted in Figure 1 below.

The `TournamentSuccessorSelector` clearly outperformed all its competitors on this domain (note the graph's logarithmic scale). The most likely reason behind its success is its rapid decision process. While the other selectors undertake the time-consuming process of calculating the h-values for every state, the tournament selector only needs to compute the h-values for k pairs of states from which it chooses. The remaining four selectors do not always come in the same

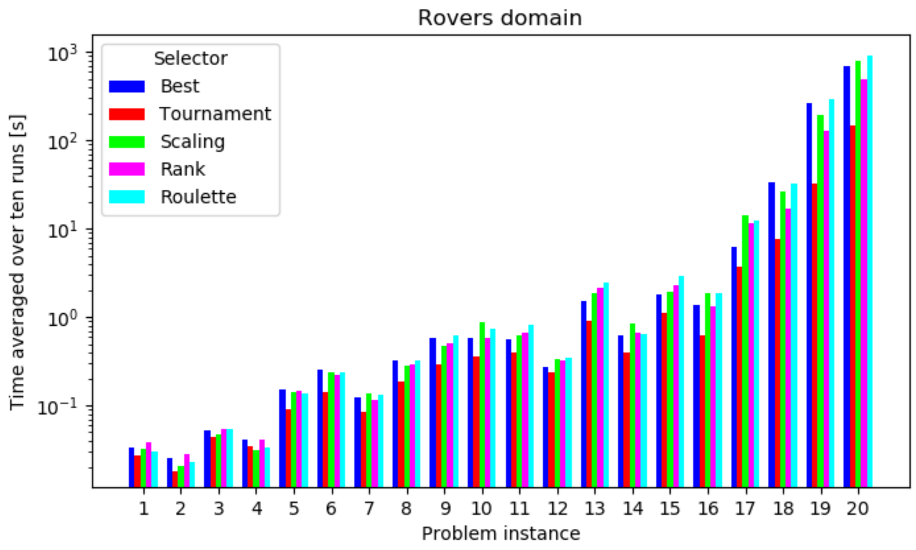

Fig. 1. Comparison of the successor selectors on the rovers domain.

order for various problems, however, the rank-based and best successor selectors seem to be performing a little better in general. Very similar results were seen also on the driverlog domain.

Consequently, we combined the rank-based and best successor selectors with the tournament selector. Furthermore, two values, 0.3 and 0.6, were tested for the *successorSelectorBoundary*. Indeed a combination of selectors yielded a faster planner than any single selector used by itself. For example, the rovers instance 20 took at least 150s with a single selector; using a combination of selectors this time could be cut down to 128s. Out of the tested selector combinations, using the Best and Tournament selectors with a *successorSelectorBoundary* at 0.3 turned out to be triumphant. These observations lead to the conclusion, that elitism (provided by the use of the `BestSuccessorSelector`) is extremely important for obtaining a fast planner.

After optimising the core components of the local beam search algorithm, the focus turned to macros. The planner was tested using 0–4 macros to find a balance between saving time by taking a shorter path to the goal and the overheads associated with an increased branching factor. We must not forget that macros add additional successor states to the *open* set. If we decide to calculate the h-values of all of them this will slow down the planner considerably.

In general, our research shows, that macro actions seem to cut down the planning times for most problems. They clearly succeed in skipping over intermediate states and thus decreasing the number of steps needed to reach the goal. Moreover, the macros also helped to overcome local minima or plateaus in the search space by leading the planner in the right direction. Especially in the driverlog domain which is known to have many local minima, macros proved to be very beneficial to the search process. We did, however, also encounter cases where the planner was slowed down, e.g., instance 13 of the driverlog domain. It may be the case, that the macro chosen here was helpful in the training problem instances, but in the real instance 13 it in fact increased the branching factor so much that it became counterproductive and slowed the planner down. For this reason, we decided to use only 1 macro by default.

The next step was to optimize k the maximum number of beams used by the planner. The planner was once again asked to solve the first 20 problems on the driverlog and rovers domain gradually increasing k from 1 to 5 (now the planner used a combination of the best and tournament selection methods and one macro was generated). Just as our preliminary research hinted, it was practically impossible to find an ideal, domain-independent value for k. Nevertheless, a compromise value of 3 seems to generalise well.

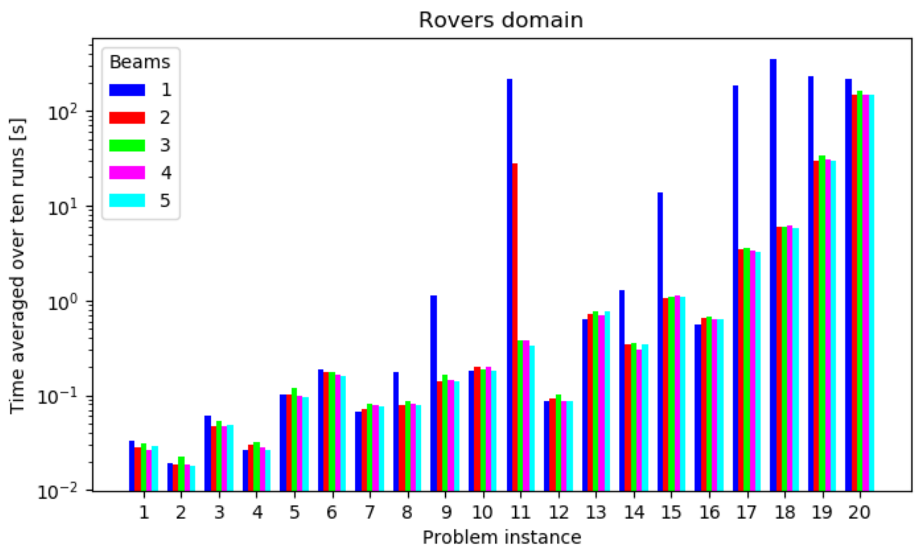

Fig. 2. Comparison of the performance for varying the maximum beam count on the rovers domain.

Figure 2 illustrates the situation in the rovers domain. Clearly, the planner was extremely sensitive to changes in this value. With a single beam the planner experienced severe difficulties particularly in instance 11. For most of the random seeds the planner had to revert to best-first search which was a lot more time consuming. In order to be sure that we can solve this problem with LBS, the planner requires at least three beams were required. The most likely reason for this behaviour is the well-known dead ends in the rovers domain. If the single beam enters such a dead end there is no alternative for the search to recover.

On the other hand, in the driverlog domain, there were a couple of problems that could not be solved using LBS at all and the fastest solution was to revert to best first search as soon as possible. In this case, adding more beams was actually counterproductive and increased the time need to find a plan.

Finally, a default value of five restarts was chosen. Interestingly, tests regarding restarts found little correlation between the maximum number of restarts and the time taken to find a solution. Yet, when investigating the planner's behaviour more closely, we found that the planner generally needs fewer beams when aided by restarts. Therefore, we can say that restarts do take credit in making the planner more resilient to changing domain conditions.

4.1 Comparison with the Original JavaFF Planner

After the optimisation, the planner was tested against its predecessor JavaFF in practice. Figure 3 compares the performance of the new planner based on LBS and traditional JavaFF using EHC on the rovers domain. With the exception of the first four problems, LBS (red bars) evidently performs a lot better than EHC. The hardest problem, instance 20, took 148 s to complete with LBS compared to, roughly double, namely, 330 s with EHC.

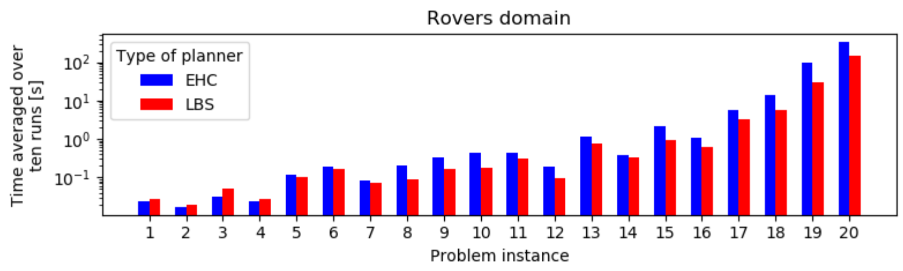

Fig. 3. Comparison of Enforced Hill Climbing (EHC) and Local Beam Search (LBS) on rovers domain.

A different picture can be seen in the driverlog domain, shown in Figure 4. Here the LBS algorithm can be considered more successful, too. Although EHC was slightly faster for some problems, the key improvement comes in problem instances 14, 15 which LBS could solve, but EHC could not. These results confirm that the parameter optimisations were fruitful in generating a robust planner.

In order to examine how universal the planner is, we will now investigate the planner's behaviour also on the pipes non-tankage and freecell IPC domains. In the pipes domain, once again, certain problems are solved faster with LBS, others with EHC. Crucially, however, LBS proved to be much more robust than EHC by solving four more problems out of 23. Lastly, the freecell domain is known to have many unrecognizable dead-ends and local minima that are hard to escape. Indeed both LBS and EHC could only solve problems 1–10. Moreover, LBS was

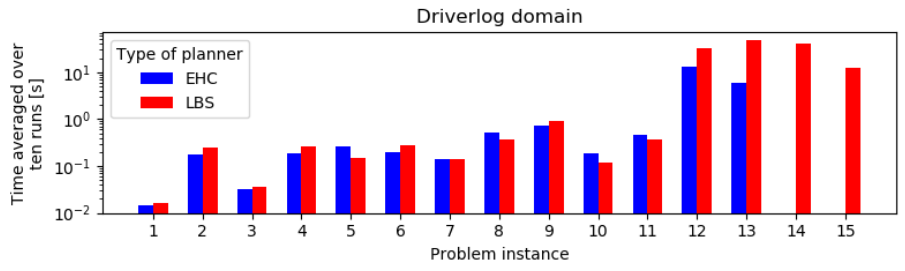

Fig. 4. Comparison of EHC and LBS on driverlog domain. The missing bars indicate a failure to find a solution.

unable to solve problem 10 using the default planner configurations unlike EHC. Furthermore, LBS in general took more time to find a solution than EHC in problems 1–9. We hence began to search what caused such a poor performance.

As discussed above, no planner configuration is ideal for all domains. Depending on the topology of a given domain, we have encountered cases, where the increased branching factor associated with macros can prove to be detrimental in our attempt to find a solution. Hence, our first shot was to plan without macros. Not only was the planner now successful in finding a solution without the macros, but it also managed to improve on the time of EHC. EHC took 37 minutes on average to complete the problem, LBS reduced this time to 32 minutes.

The experiments in the freecell domain underline the need for a customizable planner that this work presents. This approach of having a reservoir of alternative search techniques could also be easily applied to many other contemporary planners. For instance, a planner successful in overcoming local minima might not necessarily be good at dealing with dead ends. While such a planner would be apt to quickly solve problems in the driverlog domain, this planner could struggle in the freecell domain. Hence, the planner customization that this work proposes could serve as a solution and make the planner more resilient to changing environments.

5 Conclusions and Future Work

To conclude, this project has produced a novel PDDL planner based on a local beam search. This work has highlighted the unique properties of LBS that make it suitable for many different planning domains. On the contrary to greedy search, local beam search keeps a reservoir of alternative candidate paths to a solution and thus constitutes a much more robust planner. The new planner has also enabled us to combine the benefits of different successor selection procedures.

Additionally, the planner employed restarts and macro actions, which help it to combat local minima and dead ends in the search space. The conducted experiments provided useful default parameters for our planner, nevertheless, they have also emphasized the importance of creating a customizable application.

More importantly, the methods for combining various search techniques together used in this work could serve as inspiration for many other planners, too. Not only can this approach make the planners suitable to a wider range of domains, but also make them more robust. For instance, we have observed that adding restarts decreases the number of beams needed to find a solution.

Even with all the wonderful breakthroughs in recent years, the area of AI Planning still remains widely underdeveloped with lots of potential to offer. A promising direction to explore would be to combine LBS with another search algorithm. If LBS fails to find a solution, its work could be exploited by another search technique that would finish the search. Once again we could combine the speed of one search method with the robustness of another.

Furthermore, in AI planning we are interested not only in finding a valid solution but rather in finding the best solution. We have demonstrated the qualities of LBS, but we must remember, that LBS is not complete on its own, therefore it might not always provide us with the optimal solution. Zhou [10] outlines, how completeness can be achieved by transforming LBS into beam-stack search. Beam-stack search makes use of systematic backtracking using which we can continue to refine the existing solution until the optimal solution is found.

References

1. Baeck, T., Fogel, D., Michalewicz, Z.: Evolutionary Computation 1: Basic Algorithms and Operators, vol. 1. CRC Press (2018)
2. Botea, A.: Improving AI Planning and Search with Automatic Abstraction. PhD thesis, University of Alberta (2006)
3. Coles, A., Fox, M., Smith, A.: A new local-search algorithm for forward-chaining planning. In: Proc. of ICAPS'07. pp. 89–96. AAAI (2007)
4. Hoffmann, J.: Where 'ignoring delete lists' works: Local search topology in planning benchmarks. Journal of Artificial Intelligence Research **24**, 685–758 (2005)
5. Long, D., Fox, M.: The 3rd international planning competition: Results and analysis. JAIR **20**, 1–59 (2003)
6. Newton, M.A.H., Levine, J., Fox, M., Long, D.: Learning macro-actions for arbitrary planners and domains. In: Proc. of ICAPS'07. pp. 256–263. AAAI (2007)
7. Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited (2016)
8. Wilt, C.M., Thayer, J.T., Ruml, W.: A comparison of greedy search algorithms. In: Third Annual Symposium on Combinatorial Search. pp. 129–136 (2010)
9. Wong, C., Guo, Z.X., Leung, S.: Optimizing decision making in the apparel supply chain using artificial intelligence (AI): from production to retail. Elsevier (2013)
10. Zhou, R., Hansen, E.A.: Beam-stack search: Integrating backtracking with beam search. In: Proc. of ICAPS'05. pp. 90–98. AAAI (2005)