A Local Monte Carlo Tree Search Approach in Deterministic Planning

Submission 452 #

Abstract
Much recent work in satisficing planning has aimed at striking a balance between coverage - solving as many problems as possible - and plan quality. Current planners achieve near perfect coverage on the latest IPC benchmarks. It is therefore natural to investigate their scaling behavior on more difficult instances. For several recent IPC domains, scalable problem generators, which can create harder instances, are available. Among state of the art planners, LAMA (Richter, Helmert, and Westphal 2008) is able to generate high quality plans, but its coverage drops off rapidly with increasing problem complexity. The Arvand planner (Nakhost and Müller 2009) scales to much harder instances but generates lower quality plans. This paper introduces a new algorithm, Monte Carlo Random Walk-based Local Tree Search (MRW-LTS). Similar to successful game-playing programs based on Monte Carlo tree search, MRW-LTS uses random walks to selectively build a search tree. Unlike game-playing methods, each such tree represents a local, not a global search. Experiments demonstrate that MRW-LTS combines a scaling behavior that is better than LAMA's with a plan quality that is better than Arvand's. In the IPC metric, this planner advances the state of the art for planning problems that are harder than those of IPC-2008, but not so hard that nearly only Arvand can solve them.

Introduction
Recently, Monte Carlo Tree Search (MCTS) brings a remarkable success to the field of difficult game playing domains like Go (Rimmel et al. 2010; Enzenberger et al. 2010) and General Game Playing (Mehta and Cazenave 2010). MCTS selectively grows a single global search tree and performs Monte Carlo random simulations to evaluate leaf nodes. For which actions should be tried in the search tree, there exists a problem of balancing between exploiting the current knowledge to focus on the current best actions and exploring new actions to further increase knowledge. The most famous example of MCTS is UCT (Upper Confidence Bounds applied to trees) (Kocsis, Levente and Szepesvári 2006), and in UCT algorithm, UCB formulas (Auer, Cesa-Bianchi, and Fischer 2002), which contain a exploration term and a exploitation term, are used to handle the balancing problem.

In the field of path-finding planning, the Rapidly-exploring Random Tree (RRT) (LaValle 2006) algorithm balances between exploitation and exploration. RRT gradually builds a tree in the search space until a path to the goal state is found. At each step the tree is either expanded towards the goal, which corresponds to exploitation, or towards a randomly selected point in the search space for exploration. While RRT is an effective method for path-finding, it cannot be directly applied to general domain-independent planning problems since there is no underlying metric space for selecting random points.

Arvand (Nakhost and Müller 2009) was the first planning system that applied Monte Carlo Random Walks (MRW) to deterministic classical planning. In Arvand, random walks are used to explore the neighborhood of a search state before jumping to the best found endpoint according to an heuristic evaluation function. Exploration helps to overcome the problem of local minima and plateaus. Jumping greedily exploits the knowledge gained by the random walks. Using the combination of Monte Carlo random walks and jumping, Arvand can often escape from local minima and plateaus and quickly solve even problems that are hard for other current planners such as LAMA and FF (Hoffmann and Nebel 2001) which use more systematic global search methods. However, one disadvantage of Arvand’s random walk and jumping algorithm compared to more classical planners is that it often produces long, low quality plans.

One previous approach to fixing this problem of MRW planning has been the Aras postprocessor (Nakhost and Müller 2010), which improves a given complete plan by using a systematic local search in the neighborhood of the plan. While successful in finding local improvements, the plan quality of Arvand+Aras can still be inferior in cases where the input plans generated by Arvand deviate too far from a good plan.

The main motivation of this paper is to find an intermediate approach between global systematic search, as in LAMA or FF, which has higher plan quality but poorer scaling, and using local random walks with jumping, as in Arvand, which scales better to larger problems, but often at the cost of poor plan quality. Monte Carlo Random Walks based Local Tree Search (MRW-LTS) is a combination of systematic local tree search with Monte Carlo random walks. Local tree search is used to improve the quality of possible plan continuations.
before jumping to the next endpoint of a random walk as in Arvand. The aim of this local tree search is to prevent the overall search process from veering too far away from a good plan. The method sacrifices much of the planning speed of Arvand for the goal of achieving higher plan quality.

The rest of this paper are organized as follows: Section 2 briefly reviews the framework of Monte Carlo Random Walk (MRW) Planning which is the basis for the current work. Section 3 presents the new local tree search planning algorithm MRW-LTS and a first implementation in a complete planning system. Experimental results are shown and discussed in Section 4, while Section 5, summarizes the approach taken and outlines possible future work.

**Monte Carlo Random Walk (MRW) Planning**

In Monte Carlo Random Walk planning, quick random walks are performed to explore the neighborhood of the current search state $s_0$. A random walk is a sequence of states $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$ starting from $s_0$, in which all transition actions $s_k \rightarrow s_{k+1}$ are randomly selected among all legal actions in $s_k$. Only the end-state $s_n$ of the random walk is evaluated by a heuristic function $h$, such as $h_{FF}$, the FF heuristic (Hoffmann and Nebel 2001). After a number of random walks, the current search state jumps to the most promising end-state of all random walks, which is defined as the first encountered state of minimum $h$-value. The whole process of performing a set of random walks, then jumping to a best endpoint is considered to be one search step. To perform random walks quickly, MRW only evaluates end-states of random walks. Usually, computing heuristics such as $h_{FF}$ is orders of magnitude more expensive than generating and executing random actions. Therefore, MRW can explore a much larger neighborhood of the current search state than systematic planners such as LAMA and FF. MRW chains a series of search steps until either a goal state is reached, or it needs to restart from the initial state when either the minimum $h$-value doesn’t change for a given number of search steps, or the search reaches a dead end.

**Monte Carlo Random Walks based Local Tree Search (MRW-LTS)**

The MRW algorithm emphasizes fast exploration by random walks, while leaving plan improvement to a postprocessor such as Aras. In contrast, MRW-LTS focuses more on quality by performing a local Monte Carlo Tree Search guided by random walks. Figure 1 illustrates the search strategies of MRW and MRW-LTS. Both algorithms use MRW to explore the search space near a starting point $s_0$. After each exploration phase, both algorithms jump and update $s_0$ to be an explored and evaluated end-state with minimum $h$-value and start the next search step from this new $s_0$. Unlike MRW, MRW-LTS grows a local search tree $T$ with root $s_0$ during exploration. All random walks start at leaf nodes of $T$. Each node $n$ in the tree stores the minimum $h$-value encountered by any random walk in the subtree below $n$ as well as a visit count.

![Figure 1: Visualizing search strategies. Left: MRW. Right: MRW-LTS](image)

Algorithm 1 shows an outline of the algorithm. Each execution of the while loop constitutes one search step. A successful search returns the sequence $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$ of states along a plan, with $s_0$ the initial state, $s_n \supseteq G$ a goal state and the partial paths $s_k \rightarrow s_{k+1}$ obtained by search steps starting from $s_k$. MRW-LTS fails if the minimum $h$-value doesn’t improve in one search step, or the current search state ever becomes a dead-end. In this case, the algorithm simply restarts from the initial state $s_0$.

In the algorithm, $\text{RandomWalk}(s, G)$ performs one random walk from a state $s$, as in (Nakhost and Müller 2009). Each random walk returns the end-state reached after either a given maximum number of steps, or earlier in case of a goal or dead-end.

**Local Tree Search (LTS)**

The main motivation of the local tree search is to optimize the paths generated by random walks before each jump. The search tree grows selectively towards states where the $h$-values returned from random walks are more promising. Further random walks are biased towards starting from promising nodes in the search tree. LTS consists of three parts: selection, expansion and backpropagation.

Algorithm 2 selects a leaf node by following an $\epsilon$-greedy strategy in each node. However, new nodes are visited once to gain some initial knowledge. If all children of a node have had one visit, then the algorithm selects the child with minimum $h$-value with probability $1 - \epsilon$, and a random child with probability $\epsilon$. LTS grows a very selective, unbalanced tree from the current search state. To save time and memory, the algorithm only expands nodes when they are visited the second time. After each random walk, the heuristic value reached at its end state is propagated up towards the root node as long as the new value improves the evaluation of those nodes. See Algorithm 3.
Algorithm 1 MRW-LTS

**Input** Initial State $s_0$, heuristic $h$, goal condition $G$

**Output** A solution plan

1. $s \leftarrow s_0$
2. $h_{\min} \leftarrow h(s_0)$
3. plan $\leftarrow \emptyset$
4. while $G \not\subseteq s$ do
   5. bestPath $\leftarrow \emptyset$
   6. $s_{\text{best}} \leftarrow s$
   7. $T\text{.clear}()$; $T\text{.root} \leftarrow \text{Node}(s)$ (reset tree)
   8. for $i = 1$ to MAXRANDOMWALKS do
      9. $n_{\text{leaf}} \leftarrow \text{SelectLeaf}(T)$
      10. $s_{\text{end}} \leftarrow \text{RandomWalk}(n_{\text{leaf}}\cdot\text{state}, G)$
      11. UpdateNodeValues($n_{\text{leaf}}, h(s_{\text{end}})$)
      12. if $h(s_{\text{end}}) < h_{\min}$ then
         13. $h_{\min} \leftarrow h(s_{\text{end}})$ (new globally best path)
         14. $s_{\text{best}} \leftarrow s_{\text{end}}$
         15. bestPath $\leftarrow$ path($s, s_{\text{tmp}}$)
      16. end if
   17. end for
   18. if bestPath = $\emptyset$ or $\text{deadend}(s_{\text{best}})$ then
      19. $s \leftarrow s_0$
      20. plan.clear() 
   21. else
      22. $s \leftarrow s_{\text{best}}$
      23. plan $\leftarrow$ plan + bestPath (concatenate)
   24. end if
5. end while
6. return plan

Algorithm 2 SelectLeaf($T$)

**Input** the search tree $T$

**Output** A leaf node $n$

1. $n \leftarrow T\text{.root}$
2. while $n\text{.visited} > 0$ do
   3. if isLeaf($n$) then
      4. createChildren($n$)
   5. end if
6. $n\text{.visited} \leftarrow n\text{.visited} + 1$
7. $n \leftarrow \text{selectByEpsilonGreedy}(n)$
8. end while
9. return $n$

Algorithm 3 UpdateNodeValues($n, v$)

**Input** node $n$, heuristic value $v$

1. while $n \neq \text{NULL}$ and $n\text{.hValue} > v$ do
   2. $n\text{.hValue} \leftarrow v$
   3. $n \leftarrow n\text{.parent}$
4. end while

Jump and Restart Policy

Arvand jumps after at most 2000 random walks, or as soon as acceptable progress (Nakhost and Müller 2009) is made. Jumping after “acceptable progress” makes Arvand faster in finding goal states. In contrast, MRW-LTS never jumps early. There are two reasons for disabling such an early jump: first, since the search tree grows towards more promising states, the probability for random walks to hit even better states often increases over time. The second reason is that the maximum length of random walks is decreased to a small value after a lower $h$-value (Nakhost and Müller 2009) is found. Through the combination of the search tree and short random walks, MRW-LTS very often finds shorter paths to states with the same or better $h$-value.

MRW fails and restarts when the minimum obtained $h$-value does not improve over several search steps, or when it reaches a dead-end state. The restart policy of MRW-LTS is more strict: If the minimum $h$-value does not improve within a single search step, the search restarts from $s_0$.

**Arvand-LTS: A Simple Planner based on MRW-LTS**

The planner Arvand-LTS is a first concrete implementation of the MRW-LTS algorithm. While based on the Arvand framework, it replaces MRW by MRW-LTS. The maximum number of random walks for one search step in Arvand is 2000 and because of acceptable progress jumps the real number can be much less. Arvand-LTS first runs each search step with MAXRANDOMWALKS = 2500 random walks. Once a plan is found, it searches for better quality plans by increasing MAXRANDOMWALKS to 5000, 10000, and finally 20000 for all remaining restarts. For the $\epsilon$-greedy policy in Algorithm 2 $\epsilon = 0.001$ was selected by some quick experiments on IPC-2008 benchmarks. Arvand-LTS uses a random walk length scaling algorithm like Arvand with an initial walk length of 1, an extending rate of 2 and a extending period of MAXRANDOMWALKS/20. Arvand-LTS uses MHA (Nakhost and Müller 2009) for all random walks.

The Aras postprocessor is run alternately with Arvand-LTS after each new plan is found. Compared to the method proposed in (Nakhost and Müller 2010), this applies Aras to a larger variety of input plans, which can be more important than the quality of these plans.

**Experiments**

Experiments include tests on all IPC-2008 benchmarks, and on scaled harder problems generated by IPC-2008 domain generators. Experiments on IPC-2008 were run on a 3 GHz machine and experiments on scaled hard problems were run on a 2.5 GHz machine. The runtime limit was 30 minutes and the memory limit 2 GB for each problem. Results for planners which use randomization are averaged over 10 runs per instance.

**Planners and Aras**

Experiments compare four planners: LAMA, the winner of IPC-2008, as well as LAMA with Aras, Arvand and Arvand-LTS. For improved performance, the latter three programs run alternately with Aras as described above.

In Arvand and LAMA, the cost of the best known plan before using Aras is used for pruning the search. For Arvand-
LTS increased diversity is more useful, so it uses a different bounding strategy. Arvand-LTS can run either with or without pruning based on the cost of the best plan. Arvand-LTS first runs 3 times with and without pruning respectively, and prefers the version which computed the smallest cost plan after Aras postprocessing. However, with probability 0.2 it randomly chooses among these two versions in each run.

Results on IPC 2008 Benchmarks

The experimental results of LAMA, ARV AND and MRW-L Ts are shown in Table 1. Overall, the performance of based planner Arvand and Arvand-LTS are competitive with LAMA in IPC-2008 benchmarks. However, LAMA still has the best coverage and highest total score1.

The overall coverage in these four domains is around 90% for all planners. LAMA and LAMA+Aras solve all problems except in Elevator (87%), Parcprinter (77%) and Sokoban (77%). Arvand fails on some problems in Scanalyzer (91.3%) and Sokoban (15.66%), while Arvand-LTS' coverage is incomplete in Pegsol (90.03%), Scanalyzer (94%) and Sokoban (15.18%).

The bad performance of Arvand and Arvand-LTS in Sokoban is striking. There are several reasons why Monte Carlo random walks are weak in this domain. First, the percentage of paths to goal states among all paths is very low. Second, the heuristic is also misleading sometimes in this domain. The search space of the Sokoban problems selected for the planning competition is relatively small. In 50% of problems solved by LAMA, the search space is completely explored. All these reasons make A* based planners work much better in this domain.

In Elevator, LAMA failed on four hard problems with average branching factors between 40 to 50 and shortest known plans of between 100 and 210 steps. Even though Arvand solved all problems here, its total score is below LAMA because of low plan quality. Comparing Arvand and Arvand-LTS, Arvand-LTS generated much better plans in the 15 hardest problems than Arvand. For large problems, both Arvand and Arvand-LTS perform more jumps to find the goal states. The deviation from a good overall plan is much larger for Arvand since it jumps along random walks while MRW-LTS jumps along locally optimized plans

Scaling Up Problems

The overall performance of Arvand-LTS is better than Arvand, however, in some simple2 problems, Arvand has even better performance than Arvand-LTS. Because of the post-processing system Aras, for small problems, the diversity of initial plans before using Aras is more important than the quality of initial plans. The plans generated by Arvand are more diverse than Arvand-LTS because no systematic search tree is built. The number of plans generated by Arvand is

Table 1: Score (S) of LAMA, LAMA with Aras(LAMA-A), Arvand and Arvand-LTS(LTS) on IPC 2008 benchmarks

<table>
<thead>
<tr>
<th>Domains</th>
<th>LAMA</th>
<th>LAMA-A</th>
<th>Arvand</th>
<th>LTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cybersec</td>
<td>29.11</td>
<td>29.11</td>
<td>28.891</td>
<td>28.92</td>
</tr>
<tr>
<td>Elevator</td>
<td>26.53</td>
<td>28.94</td>
<td>26.7</td>
<td>33.38</td>
</tr>
<tr>
<td>Openstacks</td>
<td>26.89</td>
<td>26.89</td>
<td>26.9</td>
<td>26.28</td>
</tr>
<tr>
<td>Parcprinter</td>
<td>21.93</td>
<td>22.14</td>
<td>29.8</td>
<td>29.18</td>
</tr>
<tr>
<td>Pegsol</td>
<td>28.97</td>
<td>28.97</td>
<td>29.00</td>
<td>26.226</td>
</tr>
<tr>
<td>Scanalyzer</td>
<td>25.47</td>
<td>26.27</td>
<td>25.22</td>
<td>23.694</td>
</tr>
<tr>
<td>Sokoban</td>
<td>21.62</td>
<td>22.75</td>
<td>4.732</td>
<td>5.48</td>
</tr>
<tr>
<td>Transport</td>
<td>28.7</td>
<td>30.51</td>
<td>28.32</td>
<td>29.15</td>
</tr>
<tr>
<td>Woodworking</td>
<td>24.74</td>
<td>25.54</td>
<td>28.823</td>
<td>28.92</td>
</tr>
<tr>
<td>Total</td>
<td>233.96</td>
<td>240.92</td>
<td>228.4</td>
<td>234.2</td>
</tr>
</tbody>
</table>

Figure 2: Scores of MRW-LTS and Arvand in Elevator with increasing problem difficulty.

Table 1 also larger than for Arvand-LTS since local tree search takes much more time per search step. For harder problems, the very long plans generated by Arvand deviate too much from good plans, and the local optimization of Aras is not enough here. Figure 2 plots the scores and score difference between Arvand and Arvand-LTS in the domain of Elevator with increasing problem size. The scores are close in the first 15 problems, but for problems 16 to 30, which have longer plans, Arvand-LTS is much better.

Most of the original problems in IPC-2008 are a bit too easy for these planners, which solve around 90%. The following four domains of IPC-2008 have scalable problem generators which allow the creation of harder instances: Elevator, Openstacks, Transport and Woodworking.

In Elevator, passengers need to be transported using both fast and slow elevators. Planners should minimize the time usage. Openstacks is a combinatorial optimization problem. Products are produced and temporarily stacked. Planners need to minimize the maximum number of stacks that are in use simultaneously. In Transport, planners must move packages to their destinations while minimizing costs generated by driving, pick-up and dropping. In Woodworking, parts must be processed in order to make woodworks. Planners should minimize the cost. To scale problems, the following parameters were varied: Elevator - number of passengers, Openstacks - number of products, Transport - number of packages, Woodworking - number of parts. For each prob-

---

1The score for a plan equals to the value of the baseline plan cost of IPC-2008 divides the cost of this plan. If not solved, the score is 0, and because the IPC-2008 baseline plans are not optimal, it is possible for some plans to get scores bigger than 1.

2simple: plans are not very long and the average branching factor is also not very big.
lem size, 5 different instances were created using different random seeds.

Figure 3 shows the coverage of FF, LAMA, Arvand and Arvand-LTS on these new generated problems. Openstacks is omitted since solving these problems (ignoring cost) is trivial, and all these planners solved all Openstacks problems. The coverage is mostly in the order Arvand > Arvand-LTS > FF > LAMA.

For large problems, scores are computed similarly to IPC-2008, by assigning the cost of the best plan found by these three planners a score of 1. Figure 4 shows the average scores of different problem sizes in the 4 domains over only solved runs, and the results of FF is not shown because FF doesn’t consider action costs in search. For example, one problem size of Elevator contains 5 problems, Arvand runs 10 runs on these 5 problems and finally solves the problems 32 times, then we get the score by averaging these 32 scores (the score is 0 if never solved).

Unsolved runs are ignored here to highlight plan quality. LAMA usually generates the best plan except in Woodworking. Arvand-LTS dominates Arvand in Openstacks and Elevator. In Transport and Woodworking, Arvand-LTS generates better plans for problem sizes which it can always solve, but becomes weaker when the size of problems increases further. One possible reason for why Arvand creates better plans for large problems here is again its increased speed leading to larger diversity. Even if Arvand-LTS solves such a problem, it may only generate one or two plans, while Arvand can generate many more plans. Combined with Aras optimization, Arvand has a better chance to generate at least one good plan. For problem sizes which MRW-LTS can solve all the time, although Arvand still creates many more plans, Arvand-LTS can generates enough plans to secure its advantage.

For large problems, Monte-Carlo Random Walk based approaches have much better coverage than LAMA. As local search algorithms, random walk methods scale much better than systematic search planners such as LAMA. Systematic search can be slow to escape from extensive local minima/plateaus, while random walks lead to bigger jumps that help get away from such traps. However, when it comes to plan quality, LAMA, if solves the problem, generally creates better quality plans. The plan produced by local search in Arvand-LTS alternates between sequences generated systematically by local tree search, and random walks. Arvand-LTS tries to balance coverage and plan quality by combining the benefit of Monte Carlo random walks in solving problems with the ability of systematic search algorithms of producing shorter paths.

**Conclusion**

In practical planning problems, there exists an important tradeoff between coverage and plan quality. Current state of the art planners seem to emphasize either one or the other. The latest International Planning Competition, IPC-2008, focused on plan quality while the Arvand system focused on coverage. Arvand-LTS represents a balanced approach by combining random walks for increased coverage with systematic local search for plan quality.

Important directions for future work are: 1. to introduce a measure such as “acceptable progress” for MRW-LTS, to make it faster in solving problems. Unlike the definition used in MRW, this measure should not only consider progress regarding the $h$-value, but also the cost in terms of increased $g$-value. 2. To generate plans with better quality, plan cost should also be considered in the child selection function of local tree search. For example, LTS could use a weighted combination of $g$ and $h$-values for selection, and gradually decrease the weight of $h$ while searching for better quality plans.

**References**


Figure 4: Average scores among solved instances in Elevator (top left), Transport (bottom right), Woodworking (bottom left) and Openstacks (bottom right).


