Balancing Exploration and Exploitation in Classical Planning

Tim Schulte and **Thomas Keller**
University of Freiburg
{schultet, tkeller}@informatik.uni-freiburg.de

Abstract

Successful heuristic search planners for satisficing planning like FF or LAMA are usually based on one or more best first search techniques. Recent research has led to planners like Arvand, Roamer or Probe, where novel techniques like Monte-Carlo Random Walks extend the traditional exploitation-focused best first search by an exploration component. The UCT algorithm balances these contradictory incentives and has shown tremendous success in related areas of sequential decision making but has never been applied to classical planning yet. We make up for this shortcoming by applying the Trial-based Heuristic Tree Search framework to classical planning. We show how to model the best first search techniques Weighted A* and Greedy Best First Search with only three ingredients: action selection, initialization and backup function. Then we use THTS to derive four versions of the UCT algorithm that differ in the used backup functions. The experimental evaluation shows that our main algorithm, GreedyUCT*, outperforms all other algorithms presented in this paper, both in terms of coverage and quality.

Introduction

The sequential satisficing track of the International Planning Competition (IPC) has been dominated by heuristic search (HS) since its introduction. Before the latest installment in 2011, all successful HS planners relied on one or a sequential combination of the best first search (BFS) techniques Weighted A* (WA*), Greedy Best First Search (GBFS) and Enforced Hill Climbing. Prototypical for this kind of planner are both competition winning versions of LAMA (Richter and Westphal 2010) or Fast Downward Stone Soup (Helmert, Röger, and Karpas 2011), which use GBFS for the first plan and WA* thereafter, or FF (Hoffmann and Nebel 2001), which starts with Enforced Hill Climbing and switches to GBFS under certain circumstances.

In contrast to these BFS techniques, UCT (Kocsis and Szepesvári 2006) is a popular Monte-Carlo tree search (Browne et al. 2012) algorithm that addresses the classical exploration-exploitation dilemma. Even though it is not well suited for sequential decision making applications (Bubeck, Munos, and Stoltz 2009), UCT has been applied successfully in General Game Playing (Finnsson and Björnsson 2008), MDPs (Keller and Eyerich 2012), POMDPs (Silver and Veness 2010), or in domain-specific implementations (Gelly and Silver 2007; Eyerich, Keller, and Helmert 2010). Albeit there are several planners that incorporate some way of exploration (Lipovetzky and Geffner 2011; Lu et al. 2011; Xie, Nakhost, and Müller 2012), we found no classical planning system that truly balances exploration and exploitation. We believe that the application of the UCB1 formula (Auer, Cesa-Bianchi, and Fischer 2002) to classical planning is a first step in that direction.

In the context of finite-horizon MDPs, UCT has recently been described as a Trial-based Heuristic Tree Search (THTS) algorithm (Keller and Helmert 2013). The framework allows the definition of algorithms by providing only a few ingredients. As one of them, the backup function, is crucial to derive a competitive UCT variant over the course of this paper, we believe that it is adequate to present our algorithms in terms of the framework as well. After discussing its adaption to classical planning, we show how to model THTS versions of A*, WA* and GBFS. We use these implementations to assess the overhead of using trials and a tree structure. As a side effect, it shows the close relationship among the considered search techniques.

Our main contribution is the definition and evaluation of several UCT variants, which are derived by mixing ingredients of the BFS baseline approaches with UCT. We start with the classical UCT, where backups are computed by extending the current average with the latest sample. While these Monte-Carlo backups have advantages in some scenarios, there is little reason for their usage in deterministic offline planning. We therefore derive UCT* by combining the backup function of WA* with the action selection of UCT. We add the greedy behaviour of GBFS to our algorithms and obtain GreedyUCT and GreedyUCT*, two algorithms that combine properties of GBFS and UCT in a novel way. After applying the well-known enhancements deferred heuristic evaluation and preferred operators to all considered algorithms, we show that GreedyUCT* outperforms all other algorithms both in terms of quality score and coverage. A final experiment on the derived UCT variants reveals that the underlying action selection method is well-suited to balance exploration and exploitation in classical planning.

Algorithm 1 THTS for classical planning

```
1:  procedure THTS()
2:      while time allows and no plan found do
3:          PERFORMTRIAL()
4:      return the plan
5:  procedure PERFORMTRIAL()
6:      $n \leftarrow n_0$
7:      while $n$ is initialized do
8:          $n \leftarrow$ SELECTACTION($n$)
9:      if $s(n) \in S^\star$ then
10:         extract plan and return
11:     INITIALIZENODE($n$)
12:     backupQueue.insert($n$)
13:     while backupQueue is not empty do
14:         $n \leftarrow$ backupQueue.pop()
15:         BACKUP($n$)
16:         if $n \neq n_0$ then
17:             backupQueue.insert(par($n$))
```

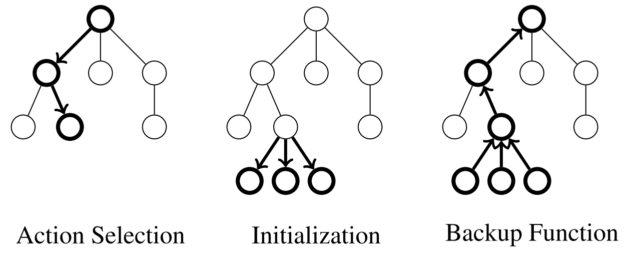

Figure 1: Ingredients of THTS.

Background

We are interested in planning tasks $\Pi = \langle V, s_0, S^\star, O, \rangle$ as defined by Helmert (2009), where V is a finite set of *finite-domain state variables* v, each with a domain D_v. The set of variables induces the set of *states* $S = 2^V$. $s_0 \in S$ is the *initial state*, a variable assignment over V, and $S^\star \subseteq S$ is the set of goal states over V. Each *operator* $\langle \mathit{pre}, \mathit{eff}, c\rangle$ in the set of operators O consists of a partial variable assignments over V, the *precondition*; of a finite set of *effects* $v \leftarrow d \in D_v$; and of a *cost* $c \in \mathbb{R}_0^+$. An operator is *applicable* in a state if the precondition holds in that state. Application of an operator o in state s yields the *successor state* $\mathrm{succ}(s, o)$, which is updated according to the set of effects and induces the operator's cost. The aim of an algorithm is to find a sequence of operators that are applicable in sequence starting from the initial state, and that result in a state fulfilling the goal. Such a sequence is called *plan*. A plan is *optimal* if its incurred cost (i.e., the sum of the costs of each operator in the sequence) is minimal among all plans.

THTS for Classical Planning

The THTS framework (Keller and Helmert 2013) has been introduced as a framework that subsumes a wide variety of different kinds of algorithms for finite horizon MDPs, including approaches from Heuristic Search, Monte-Carlo Tree Search (MCTS) and Dynamic Programming. In the following, we introduce the THTS framework for classical planning as depicted in Algorithm 1. Three functions in the algorithm are not specified, and each corresponds to an ingredient that is necessary to fully specify an THTS algorithm: backup function, action selection, and initialization.

THTS algorithms build an explicit tree of nodes, i.e., tuples that contain an assigned state s, a set of children or successor nodes $\mathcal{N}$, a value estimate f, and any kind of information that is used in the ingredients of the specific algorithm. Since all our algorithms use ingredients that ensure that there is at most one node for each state $s \in S$, we frequently refer to nodes with the name of the associated state and vice versa. In the following, we assume that search trees consist of nodes $n = \langle s, \mathcal{N}, f, v, l\rangle$, where $v \in \mathbb{N}$ is the number of visits and l is a binary attribute that is used to mark a node as locked. We denote the set of applicable actions in state $s(n)$ with $\mathcal{O}(n) \subseteq O$. If the application of operator o leads from state $s(n)$ to $s(n')$, we refer to its cost by $c(n, n')$ rather than $c(o)$. Path costs from the root node n_0 to a node n are denoted with $g(n)$ and the parent of a node n is referred to by par(n). Note that path costs are not stored in search nodes, but are computed on demand. We furthermore assume that the minimum over a set of nodes (the arg min) is an unambiguous node (ties are broken uniformly at random if necessary).

Initially, the explicit tree contains only the root node n_0, a node with $s(n_0) = s_0$. Each trial adds one or more nodes to the tree (it *explicates* them). Trials are performed as long as a time constraint allows and as long as no plan has been found. In the first phase of each trial, the explicit tree is traversed from the root node n_0 to a tip node by applying action selection to each encountered node. Once a tip node has been reached, an initialization procedure adds nodes to the explicit tree and initializes the value estimate with a heuristic function. If a node represents a goal state it is marked as locked and if it represents a dead end it is ignored and not inserted into the tree. In the third phase of the trial, the backup function is called for all nodes in the backup queue. Since the backup queue is such that it orders nodes according to their g-value (states with higher g-values have higher priority), the backup is performed in reverse order, which allows the propagation of collected information through the whole tree. The trial finishes when the backup function is called on the root node. Figure 1 gives an overview how the three ingredients of THTS algorithms are connected.

Relation to Heuristic Search Many HS implementations base their decision which node is expanded next on an open list that is implemented as a priority queue. States are inserted in the open list with a static value estimate that is computed (not necessarily solely) with a heuristic function. This results in a total order of the states where the relative position of each state to each other state in the queue is determined at the moment it is inserted into the open list. In each step, the state with the lowest value estimate is removed from the open list, expanded, and the created successor states are in-

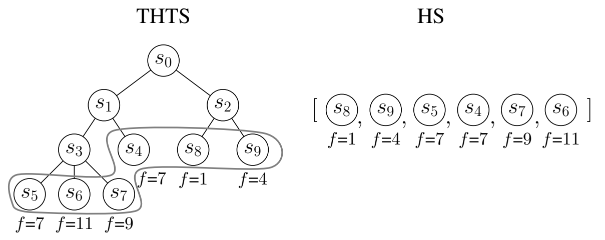

Figure 2: Comparison between THTS algorithms using a tree (left), and HS algorithms using a priority queue (right).

Algorithm 2 Initialization

```
1:  procedure INITIALIZENODE(node $n$)
2:      for $o \in \mathcal{O}(n)$ do
3:          $s' \leftarrow \text{succ}(s(n), o)$
4:          if $s'$ not in $TT$ and not isDeadend($s'$) then
5:              $n' \leftarrow \langle s', \emptyset, w \cdot h(s'), 1, \text{isGoal}(s')\rangle$
6:              $TT[s'] \leftarrow n'$
7:              $\mathcal{N}(n) \leftarrow \mathcal{N}(n) \cup \{n'\}$
8:          else if $g(n) + c(o) < g(TT[s'])$ then
9:              backupQueue.insert(par($TT[s']$))
10:             $\mathcal{N}(\text{par}(TT[s'])) \leftarrow \mathcal{N}(\text{par}(TT[s'])) \setminus \{n'\}$
11:             $g(TT[s']) = g(n) + c(o)$
12:             $\mathcal{N}(n) \leftarrow \mathcal{N}(n) \cup \{TT[s']\}$
```

serted in the queue with a value estimate. THTS algorithms do not maintain an open list, but determine the node that is expanded next by traversing the tree with repeated application of the action selection ingredient until a tip node is encountered. The open list of HS algorithms therefore corresponds to the tip nodes in THTS. This is shown with a small example in Figure 2, where the six tip nodes of the THTS tree on the left side are shown ordered according to their value estimate in the priority queue on the right side.

The next part of this paper shows how action selection and backup functions must be designed to obtain THTS algorithms that are equivalent to the well-known best first search algorithms (W)A* and GBFS. It is, of course, not our only aim to simulate already known algorithms with THTS, though. We are also seeking advantages that are impossible or prohibitively costly in an implementation that is based on an open list: our main contribution, the application of several UCT variants that balance exploration and exploitation, cannot be implemented competitively with a priority queue based open list. This is because it is crucial that the node ordering is *dynamic*, i.e., that it allows a potentially large number of states to change their relative position in each trial. Note that the possibilities that arise from this property are not restricted to the application of UCT to classical planning. In fact, we believe they are manifold, and ongoing work, e.g., on dynamic combinations of search strategies and heuristics, shows promising results.

The second important data structure of HS is the closed list, a set that contains all states that have been expanded in previous iterations. The closed list is used to make sure that states are not inserted in the open list more than once. The

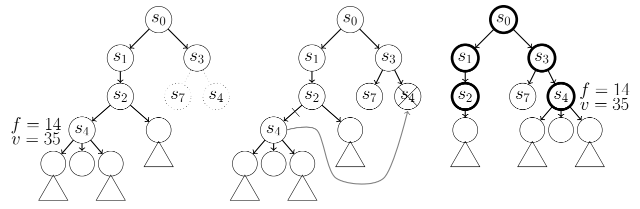

Figure 3: Handling of transposition nodes. Each node is denoted with a state and we assume uniform action costs. The backup function is applied to all bold nodes.

idea of closed lists can be ported to THTS algorithms in a straightforward manner: each tip node that is encountered and expanded at the end of the action selection phase is inserted into the closed list, and child nodes are only created for those applicable operators that lead to states that have not been encountered before. Removing duplicates this way leads to a smaller branching-factor and decreases the size of the tree considerably.

The tree structure of THTS even allows an improved version of a closed list, which is defined in the initialization procedure depicted in Algorithm 2: before adding a child to the tree in the expansion phase, we look up its state in a transposition table (TT). If the state has an entry in TT, the g-values of the hashed node and the new node are compared. If the new node's g-value is greater than the one of the existing node, it will not be added to the tree. Otherwise the subtree of the existing node is moved to the node that is currently expanded. Since we maintain the tree structure, moving a subtree can be achieved by adapting the parent and child pointers of the involved nodes. We have to be careful in the backup phase, though: whenever we remove a transposition node from its original parent, we have to update not only the trace of nodes that has been visited in the current trial, but also all previous ancestors of the node that was moved. This is achieved by adding the previous parent to the backup queue as well. (All parent nodes of relocated nodes and the encountered tip node are inserted into the backup queue during the action selection phase, and the backup phase is such that parents of updated nodes are inserted as well.) Since g-values of nodes are computed on demand rather than stored in the nodes, moved subtrees never contain obsolete data.

An example of this procedure is depicted in Figure 3. During the initialization of node s_3, a transposition is detected. State s_4 is a successor both of node s_3 and of node s_2. Since s_4 can be reached by a cheaper path via s_3, the subtree rooted at s_4 is moved to s_3, and all nodes on the paths between s_2 and the root and s_4 and the root (the bold nodes in the rightmost figure) are updated once the backup starts. Note that the same behavior is also desired in some implementations with priority queues (e.g., in A* with admissible heuristics). This is usually achieved by *re-opening* nodes that are encountered on a cheaper path, which causes the algorithm to expand large parts of the search space again. This kind of bottleneck is circumvented with the tree structure that is presented here.

Best First Search with THTS

In the following, we explain how the common BFS algorithms A*, WA* and GBFS can be modeled within THTS by choosing suitable ingredients. This allows us to show the potential of the framework, which subsumes a huge variety of algorithms. Moreover (and more importantly), it allows us to measure the computational overhead of the THTS versions compared to the open list based BFS implementations that come with the Fast Downward (Helmert 2006) code base. And lastly, we use these ingredients later to derive new algorithms by mixing them with parts of UCT. We give a short introduction to the algorithms below, followed by an experiment where the overhead of using trials and the tree structure is measured by comparing the number of solved problems of the different implementations.

BFS algorithms usually expand nodes from a priority queue that is ordered by the heuristic estimates of its nodes. These are computed by an evaluation function $f(n)$ when new nodes are generated. In each step the node that is on top of the queue, i.e. that minimizes $f(n)$, is expanded, its successor nodes are generated and inserted in the queue. The order of expansion therefore depends on the evaluation function used. GBFS always expands the node with the best heuristic score ($f(n) = h(n)$), whereas A* additionally considers path costs ($f(n) = g(n) + h(n)$) and WA* adds a weight to encourage greedy behaviour ($f(n) = g(n) + w \cdot h(n)$).

To achieve equivalent behavior, we have to make sure that nodes are expanded in the same order. Since GBFS, A* and WA* only differ in the evaluation function used, we can model all of them as shown in Algorithm 3 and achieve each algorithms unique behaviour by choosing the right k value ($k = 0$ for GBFS, $k = 1$ for A*/WA*). For WA* the weight parameter w of the basic initialization function (Algorithm 2) can be altered and equals one in all other algorithms considered in this paper. Beginning with the root node, always the child with the lowest value estimate is selected until a tip node is reached. For the GBFS version the value estimate corresponds to the heuristic value as k is set to be zero. To mimic A* and WA* we have to incorporate path costs into the value estimates, which is achieved by setting $k = 1$. The chosen node is expanded by generating all its successor nodes, computing their heuristic score and explicating them in the tree. In the backup phase that follows, all nodes that are in the backup queue are updated from the tip nodes to the root by updating the value estimate of each node with the minimal value estimate of their children (GBFS) or the minimal sum of the childrens value estimates and the cost of their creating operator (A*/WA*). It is easy to see that our action selection method therefore always ends up in the tip node with minimal $f(n)$. The order of expanded nodes remains the same as in the classic BFS implementations, except for tie breaking.

Algorithm 3 THTS-BFS

```
1: procedure SELECTACTION(node $n$)
2:     return $\arg\min_{n' \in \mathcal{N}(n):\neg l(n')} \{f(n') + k \cdot c(n,n')\}$

3: procedure BACKUP($n$)
4:     $f(n) \leftarrow \min_{n' \in \mathcal{N}(n)} f(n') + k \cdot c(n,n')$
5:     $v(n) \leftarrow \sum_{n' \in \mathcal{N}(n)} v(n')$
6:     $l(n) \leftarrow \bigwedge_{n' \in \mathcal{N}(n)} l(n')$
```

Assessing the overhead of THTS While THTS allows us to explore the search space in novel and innovative ways, runtime suffers from repeatedly traversing the tree from the root node to a tip node before a state is expanded, and by updating all nodes in the backup queue. Therefore we conduct an experiment that measures the overhead of THTS by comparing our implementations of A*, WA* and GBFS with the classic versions of the same algorithm that are implemented in FD. The setup we use for all experiments presented in this paper is as follows: all experiments have been performed on 2.6 GHz Eight Core Intel Xeon computers, with one task per core simultaneously, a memory limit of 6 GB and a time limit of 30 minutes per benchmark instance. If a plan is found before the time limit is reached, we stop the search. We used the planning problems from the deterministic satisficing tracks of IPC 2008 and 2011, which consist of 520 problem instances from 14 domains. All THTS algorithms have been implemented in the FD planning system and are therefore identical to the compared search methods apart from using trials and a tree structure. Furthermore, all tested planners were set to break ties uniformly at random[1] If search enhancements, such as deferred evaluation or preferred operators are used, this is explicitly stated. Whenever quality scores are provided, they correspond to the IPC score, which is obtained by dividing the cost of the best found plan among all planners by the cost of the plan found by the current planner. This yields a value between zero and one for each problem instance, where zero means no plan was found, and one means the found solution was the best among the evaluated planners. The scores for each problem instance are summed up, resulting in the total quality score. When we present the quality over mutually solved instances (in Table 2 and Figure 4), the score is computed by comparing only the involved planners; otherwise, all planner configurations that are presented in this paper are considered.

Table 1 shows the total number of solved problems of the open list based BFS algorithms, followed by their respective THTS equivalent. Each configuration has been tested, using three different heuristics. The results reflect that, for the context enhanced additive (CEA) heuristic (Helmert and Geffner 2008) and the FF heuristic (Hoffmann and Nebel 2001), there is almost no difference between the THTS implementations of GBFS and WA* compared to the original versions in terms of solved problem instances. However, when heuristics are used that are comparably cheap to compute, like the goal count (GC) heuristic, the overhead of the THTS framework becomes evident. Because CEA seems to be most effective it is used in all other experiments.

[1]Random tie breaking in THTS algorithms and priority queue based algorithms are not equivalent, but the differences are negligible for this work.

	Coverage		
Planner	CEA	FF	GC
GBFS	328	297	315
THTS-GBFS	330	291	276
A*	282	238	148
THTS-A*	263	222	145
WA$^*_{w=5}$	342	307	234
THTS-WA$^*_{w=5}$	344	307	218

Table 1: Comparing original BFS algorithms to their respective THTS counterpart on the benchmark instances from the IPC 2008 and 2011 using CEA, GC and FF heuristics.

Balancing Exploration and Exploitation

Having introduced the THTS framework for classical planning and shown how well-known BFS algorithms can be modeled within the framework, we proceed by describing algorithms that balance exploration and exploitation during search. We show how the popular UCT algorithm can be applied to classical planning and evaluate on the usefulness of a balanced action selection strategy. We then derive two new algorithms, UCT* and GreedyUCT*, by mixing ingredients of UCT and the algorithms that were presented in the previous section. The naming convention is, that an algorithm is preceded by the term "Greedy", whenever path costs are not considered during backup and selection phases and an asterisk is attached whenever best first search backups are used, i.e. the minimum is propagated rather than the average.

UCT Let us start by summarizing the basic principles of UCT and by providing an implementation of the algorithm applied to classical planning and based on THTS ingredients. UCT (Kocsis and Szepesvári 2006) is a MCTS algorithm which bases its action selection on the UCB1 formula (Auer, Cesa-Bianchi, and Fischer 2002) for multi-armed bandit problems. It has been applied successfully to several sequential decision making scenarios like MDP planning or General Game Playing. We briefly explain the general idea of UCT in its original setting before we show the modifications that are necessary to apply it to classical planning. The value of a node in UCT corresponds to the expected reward of the associated state. UCT approximates this value by performing a series of Monte-Carlo simulations and treating the rewards as random variables with unknown distributions.

The selection of a successor of node n is therefore treated as a multi-armed bandit problem. The successor node n' that maximizes

$$\overline{X}(n') + C\sqrt{\frac{\ln v(n)}{v(n')}}$$

is selected, where $\overline{X}(n') \in [0,1]$ is the expected reward estimate in n' and C is the *exploration parameter* that is used to adjust the exploration rate (we compare different values for the bias parameter later in the paper; it is set to $\sqrt{2}$ in all other experiments, the value that is proposed in the original

Algorithm 4 UCT

```
1: procedure SELECTACTION(node $n$)
2:     return $\underset{n' \in \mathcal{N}(n):\neg l(n')}{\arg\min}\ \overline{f}(n') - C \cdot \sqrt{\frac{\log v(n)}{v(n')}}$
3: procedure BACKUP($n$)
4:     $f(n) \leftarrow \frac{\sum_{n' \in \mathcal{N}(n)} \left(v(n') \cdot (f(n') + k \cdot c(n,n'))\right)}{\sum_{n' \in \mathcal{N}(n)} v(n')}$
5:     $v(n) \leftarrow \sum_{n' \in \mathcal{N}(n)} v(n')$
6:     $l(n) \leftarrow \bigwedge_{n' \in \mathcal{N}(n)} l(n')$
```

paper). This selection mechanism addresses the exploration-exploitation dilemma by favoring nodes that led to high rewards in previous trials (where $\overline{X}(n')$ is high) and nodes that have been rarely tried in previous trials (where $v(n')$ is low and $\sqrt{\frac{\ln v(n)}{v(n')}}$ hence high). UCT as a THTS algorithm is shown in Algorithm 4.

As we are not interested in maximal rewards but minimal costs, we adapt the UCB1 formula as shown in line 2 of Algorithm 4, where $\overline{f}(n')$ is the sum $f(n') + k \cdot c(n, n')$ normalized to a value in $[0, 1]$ as required by UCB1. The normalization is such that the sibling with the lowest sum value gets an $\overline{f}$-value of 0, whereas the sibling with the highest sum value gets an $\overline{f}$-value of 1; all other values are interpolated in between accordingly. Since operator costs are considered in the original UCT algorithm, k is set to be 1 (line 2 and 4). Note that the presence of $v(n)$ in the action selection formula makes UCT an algorithm where the ordering of all nodes can change implicitly in each trial, a functionality that can only be achieved under prohibitively high cost with a priority queue.

GreedyUCT By setting k to be 0 in Algorithm 4 we obtain a path cost insensitive variant of UCT called GreedyUCT (following our naming conventions). We expect the difference between UCT and GreedyUCT to be similar to the difference between A* and GBFS, i.e. goals are found quicker but plan quality suffers from not considering path costs.

UCT* The backup function of UCT calculates Monte-Carlo backups, which seems natural for problems under uncertainty or in sequential decision making in large state spaces. It appears rather odd for an algorithm for classical planning. By retaining the balanced action selection of UCT, but replacing Monte-Carlo backups with the backup function of THTS-A* we attempt to combine properties of MCTS and HS. By applying A* backups we update each node based on the value of its best child rather than aggregating over all children. A potential pitfall of Monte-Carlo backups that arises when a node n has a child n' with a very high value compared to an optimal sibling is thereby avoided. (Propagating the average value estimates can bias $f(n)$ disproportionately for many trials.) Furthermore, we expect to find higher quality plans because path costs are

Planner	coverage	quality
THTS-GBFS	330	143.5
THTS-A*	263	162.81
THTS-WA*, $w = 5$	**344**	147.46
UCT	207	155.05
UCT*	234	**166.8**
GreedyUCT	250	143.57
GreedyUCT*	253	152.03

Table 2: Coverage and plan quality of classic BFS algorithms compared to UCT variants.

considered in the value estimates of A*. An implementation of this algorithm can easily be derived by combining the backup function of A* as given in Algorithm 3 with the remaining ingredients of UCT as shown in Algorithm 4. The name of this algorithm, UCT*, stems from a recently released algorithm for planning under uncertainty (Keller and Helmert 2013). It was employed as an attempt to combine properties of Dynamic Programming, MCTS and HS, by replacing Monte-Carlo backups with a partial version of Full Bellman backups, and it is equivalent to our algorithm when applied to deterministic MDPs.

GreedyUCT* Since GBFS usually outperforms A* with regards to the total number of solved problem instances and with regards to the IPC score, it seems natural to implement another variant of UCT which is equivalent to UCT* but replaces the backup function of A* with the backup function of GBFS. GreedyUCT* is an algorithm that aims to find a plan greedily (the search tree is built with a stronger focus towards the goal), but still mixes this with decisions that lead to exploration which will help in some situations to overcome local minima, i.e., it balances exploration and exploitation but does not consider path costs. We hope to get plans of higher quality compared to GBFS while still maintaining a high coverage. Compared to UCT*, we expect to find solutions faster (and hence also more often within a given time limit), but of lower quality.

Performance Comparison Table 2 shows the total coverage as well as the quality of mutually solved instances, i.e. instances where all planners found a solution. Unfortunately, the UCT based algorithms do not live up to the expectation and disappoint especially in terms of coverage: the best, GreedyUCT*, solves 91 instances less than THTS-WA* (and, not depicted but implied by Table 1, 89 less than the priority queue based version WA*). The quality of the mutually solved instances is promising, though, since UCT* has the overall best quality (even better than THTS-A*, which is only possible since THTS-A* is not optimal due to the inadmissibility of the used h^{cea} heuristic). Regarding GreedyUCT*, we find our expectations confirmed, i.e. UCT* is superior to GreedyUCT* in terms of plan quality, while GreedyUCT* solves a significant larger number of problem instances.

Enhancements

Deferred evaluation of heuristic functions (DHE) and the use of preferred operators (PO) (Richter and Helmert 2009) are commonly used enhancements in modern HS algorithms. In this section, we show how to apply them when search is performed with trials on a tree, before we evaluate the impact of the enhancements on each of the proposed algorithms.

Deferred Heuristic Evaluation According to the initialization procedure in Algorithm 2, the successors that are created when a node is expanded are evaluated heuristically when they are generated. Since it is usually the case that more successors are generated than needed until a path to the goal is found, a lot of time is wasted on evaluating heuristics for these states. Deferred heuristic evaluation is a technique that decreases the number of state evaluations substantially; it has been incorporated in FD and other successful planning systems. The idea is to insert generated nodes into the open list with the heuristic estimate of their parent. Only upon expansion, i.e. when a node is removed from the open list, its heuristic function is evaluated. The successors are then generated and again enqueued with the heuristic value of their parent that has just been evaluated. On the downside, deferred heuristic evaluation also leads to an increased number of state expansions, because heuristic values are less informative, since they stem from the parent of a state instead of the state itself (Richter and Helmert 2009). DHE can easily be implemented in THTS by replacing the initialization method that is depicted in Algorithm 2 with a version where $h(s(n))$ is used in line 5 instead of $h(s')$. Instead of computing $h(s')$ for each generated child, $h(s(n))$ has to be computed only once, before the loop in line 2.

Performance Comparison with DHE When heuristic estimates are inaccurate, GBFS is often led astray on suboptimal paths, while A* can easily degenerate to an algorithm that resembles breadth first search. In the following experiment we aim to use the imprecision generated by using deferred heuristic evaluation to see whether the balanced search approaches UCT, GreedyUCT, UCT* and GreedyUCT* overcome this more easily than classic BFS algorithms.

Table 3 shows the coverage and the quality of the baseline planners that were introduced earlier compared to those using deferred heuristic evaluation. (Note that the quality entries of the baseline planners differ from Table 2 since they were based on mutually solved instances before and on all instances and planners here). The number of solved problem instances and quality scores increases significantly for all UCT based planners when deferred evaluation is used, whereas a significant quality decay is noticeable for the BFS configurations. The latter results are not surprising and coincide with observations made by Richter and Helmert (2009), but the former exceed our expectations by far: they clearly show that balanced methods overcome heuristic inaccuracy a lot better than classic BFS algorithms. Especially the results of GreedyUCT and GreedyUCT* with 84 and 55 more solved instances and a 58.78 and 42.41 points higher IPC

	Base		DHE		DHE+PO	
Config (CEA)	coverage	ipc	coverage	ipc	coverage	ipc
GBFS	328	258.71	325	236.08	347	248.56
A*	282	251.73	258	234.54	271	244.07
WA$^\star_{w=5}$	342	278.99	313	242.31	339	259.08
THTS-GBFS	330	256.95	322	224.54	369	213.43
THTS-A*	263	241.42	259	233.56	314	208.37
THTS-WA$^\star_{w=5}$	**344**	**281.89**	314	240.52	364	220.30
UCT	207	178.80	239	210.48	315	256.34
UCT*	234	216.55	269	244.64	322	256.07
GreedyUCT	250	193.84	**334**	252.63	357	250.67
GreedyUCT*	253	210.92	308	**253.33**	**389**	**297.98**

Table 3: Coverage and quality scores for the baseline planners (Base), the enhanced versions using deferred heuristic evaluation (DHE), and the final planners, using DHE and preferred operators (DHE+PO).

score compared to the baseline versions exceed the expectations.

Preferred Operators Some heuristics, including the CEA heuristic that is used in our experiments, provide additional information about whether an operator is promising or not. A popular extension to heuristic search planners is to use this information to improve search guidance. For classic BFS algorithms which are based on an open list, the most successful implementation of *preferred operators* is using an additional open list for the nodes that are reached by preferred operators, but this is not the only possible implementation.

There are also many ways to incorporate preferred operators in THTS. Since we believe that the usage of multiple open lists (which can be modeled in THTS by storing multiple value estimates in the nodes) is such a deep topic that we'd like to dedicate more than just a paragraph on it in future work, we decided not to implement preferred operators in the same way for the THTS algorithms. Instead, we alter the action selection such that as long as there is an unlocked successor that was created by a preferred operator, we select the next action only among that kind of successor node. Only if no such child exists we choose among all unlocked successor nodes. This implementation also allows us to guide the search quickly towards promising states with preferred operators while maintaining a complete algorithm. (Although we have to admit that completeness is rather theoretical since it takes prohibitively long in practice until all preferred siblings of a non-preferred state are locked.)

Performance Comparison with DHE and PO In our main experiment, we compare all implemented algorithms in terms of plan quality and coverage. Both deferred evaluation and preferred operators are used to enhance search performance. All THTS algorithms use preferred operators as described above, while the implementations of GBFS, A* and WA* that come with the FD code base use the dual-queue approach that is also incorporated in the LAMA planner. Results are given in Table 3.

If we compare the THTS version of BFS with their priority queue based counterparts, we can see that the THTS versions have closed the gap in terms of coverage, but that it widened significantly in terms of quality. We see two possible explanations: first, it might be the result of the different usage of preferred operators; and second, it might be that the structure of the tree biases tie breaking of the THTS algorithms in favor of nodes nearer to the root. We believe that the results nevertheless indicate that the overhead that is caused by using trials and the tree structure is mostly negligible.

The algorithm that profits the most from the introduction of preferred operators is, again, GreedyUCT*, which solves another 81 instances more than before, which leads to a 44.65 points better IPC score. It is therefore the best configuration in both categories among all considered ones. A comparison among the seven THTS configurations is also interesting since they are implemented in the same way and use exactly the same enhancements. The quality increase clearly shows that balancing exploration and exploitation, e.g. with UCT, improves overall plan quality.

Adjusting the amount of Exploration In our last experiment we determine how adjusting the amount of exploration of GreedyUCT* affects coverage and quality scores. Figure 4 shows coverage and quality of several GreedyUCT* configurations as a function of the exploration parameter C. Quality scores of problems that have been solved by all configurations are shown as well as quality scores over all benchmark instances. Some observations are as expected: the lowest C, i.e. the configuration behaving most greedily and performing the least exploration, yields the lowest plan quality, and the plan quality among the mutually solved problems increases consistently with increasing C.

Nevertheless, there are also some results that are not completely obvious: it is, for example, not the case that the greediest algorithm (the one with the lowest C) solves the highest number of instances. In fact, the coverage plot shows that the coverage rises with increasing C until the maximum

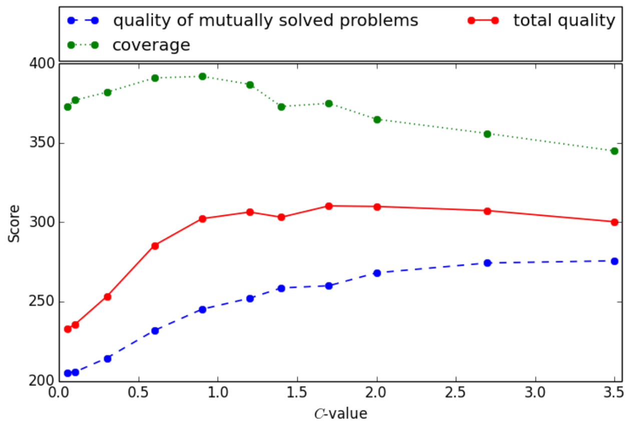

Figure 4: Coverage and quality scores for different GreedyUCT* settings.

is reached somewhere between 0.6 and 0.9 with 398 solved problems – another 9 more than in the configuration that was used in the previous experiment, where C was set to $\sqrt{2}$. Similarly, the total quality rises and has its maxima between $C = 1.2$ and $C = 2.0$, where 310.35 points are reached. Both results show that too much exploitation is as bad as too much exploration – the former get stuck in plateaus (i.e., regions in the search space where the heuristic value does not decrease between successive states), while the latter behaves too much like breadth first search. The logical implication is that balancing exploration and exploitation in search is clearly worth the effort.

Towards a better Planner There are a lot of enhancements that are used by classical planning systems like LAMA, that were not considered in this paper. This includes using multiple heuristics or open lists, and performing iterative searches to find a solution quickly and then improving plan quality by, e.g., conducting succeeding WA* searches with decreasing weights. Most can easily be incorporated in our framework, and sometimes even improved: most planners that use iterative searches restart the search from scratch whenever a solution is found. This is not necessary in THTS algorithms, since the dynamic evaluation function allows to maintain the existing structure and alter the behavior by applying a different weight in the action selection.

The UCT based algorithms that were presented in this paper can even use another parameter to adjust exploration during search: it can start with a low value of C, and increase it over the course of time. And it might even be beneficial to combine decreasing weights and an increasing exploration parameter with the right mix of THTS ingredients. Even though there is still much room for improvement, we believe that we have successfully established a plain foundation that forms the basis for future work with this paper, which is only possible if more sophisticated techniques are added to the mix step by step.

Related Work

There has been quite some work recently that follows a similar direction: several planners that incorporate some way of exploration in their search strategy competed successfully in IPC 2011: Nakhost and Müller (2009) introduced the idea of Monte-Carlo Random Walks (MCRW) to deterministic planning, which are used by Roamer (Lu et al. 2011) and Arvand-LS (Xie, Nakhost, and Müller 2012) in combination with a BFS strategy. Both use MCRW for deep exploration of the search space, where states are investigated that would be expanded by the basic BFS routine much later. The Probe planner (Lipovetzky and Geffner 2011) incorporates the same idea, but differs in the fact that the exploration traces are not random but follow a carefully chosen strategy that aims to fulfill landmarks quickly. All three planners have in common that they are able to overcome plateaus more quickly.

Another approach worth mentioning is diverse best first search (DBFS) (Imai and Kishimoto 2011). In each search step, a node is selected from the open list according to a distribution which favors nodes with a low g-cost and a low heuristic value. Then a local GBFS is applied, which expands a limited number of nodes. This process is repeated until a solution is found. Although all these search techniques are robust alternatives to BFS or enhancing it, this is mostly in regards to coverage. On the other hand a balancing between exploration and exploitation as found in the UCT based algorithms does also improve plan quality, as our benchmark results reflect.

Conclusion

We have examined the influence of balanced exploration and exploitation on coverage and plan quality in satisficing classical planning. We did so by presenting the Trial-based Heuristic Tree Search framework, and describing the ingredients that lead to the well-known BFS algorithms (Weighted) A* and GBFS within THTS. We showed how to adapt the well-known UCT algorithm to classical planning and showed that it is competitive with GBFS and WA* if all are equipped with search enhancements like deferred heuristic evaluation and preferred operators.

By combining parts of UCT and the mentioned BFS methods, we came up with the variants UCT* and GreedyUCT*. Especially the latter is able to outperform all other algorithms that are considered in this paper. Our experimental evaluation emphasizes an increase in plan quality and coverage between the balanced methods, clearly indicating that balancing exploration and exploitation is a promising technique for classical planning.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47:235–256.

Browne, C.; Powley, E. J.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte

Carlo Tree Search Methods. *IEEE Transactions Computational Intelligence and AI in Games* 4(1):1–43.

Bubeck, S.; Munos, R.; and Stoltz, G. 2009. Pure Exploration in Multi-armed Bandits Problems. In *Proceedings of the 20th International Conference on Algorithmic Learning Theory (ALT)*, 23–37.

Eyerich, P.; Keller, T.; and Helmert, M. 2010. High-Quality Policies for the Canadian Traveler's Problem. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI)*, 51–58.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based Approach to General Game Playing. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press.

Gelly, S., and Silver, D. 2007. Combining Online and Offline Knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning (ICML)*, 273–280.

Helmert, M., and Geffner, H. 2008. Unifying the Causal Graph and Additive Heuristics. In *ICAPS*, 140–147.

Helmert, M.; Röger, G.; and Karpas, E. 2011. Fast Downward Stone Soup: A Baseline for Building Planner Portfoilios. In *Proceedings of the ICAPS-2011 Workshop on Planning and Learning (PAL)*, 13–20.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence* 173(5–6):503–535.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)* 14:253–302.

Imai, T., and Kishimoto, A. 2011. A Novel Technique for Avoiding Plateaus of Greedy Best-First Search in Satisficing Planning. In *AAAI*.

Keller, T., and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 119–127. AAAI Press.

Keller, T., and Helmert, M. 2013. Trial-based Heuristic Tree Search for Finite Horizon MDPs. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 135–143. AAAI Press.

Kocsis, L., and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning (ECML)*, 282–293.

Lipovetzky, N., and Geffner, H. 2011. Searching for Plans with Carefully Designed Probes. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*, 154–161.

Lu, Q.; Xu, Y.; Huang, R.; and Chen, Y. 2011. The Roamer Planner – Random-Walk Assisted Best-First Search. In *Seventh International Planning Competition (IPC 2011), Deterministic Part*.

Nakhost, H., and Müller, M. 2009. Monte-Carlo Exploration for Deterministic Planning. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, 1766–1771.

Richter, S., and Helmert, M. 2009. Preferred Operators and Deferred Evaluation in Satisficing Planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 273–280.

Richter, S., and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research (JAIR)* 39:127–177.

Silver, D., and Veness, J. 2010. Monte-Carlo Planning in Large POMDPs. In *In Advances in Neural Information Processing Systems 23 (NIPS)*, 2164–2172.

Xie, F.; Nakhost, H.; and Müller, M. 2012. Planning Via Random Walk-Driven Local Search. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 315–322.