Guiding Combinatorial Optimization with UCT

Ashish Sabharwal and **Horst Samulowitz**
IBM Watson Research Center, Yorktown Heights, NY, USA
{ashish.sabharwal,samulowitz}@us.ibm.com

Abstract

We propose a new approach for search tree exploration in the context of combinatorial optimization, specifically Mixed Integer Programming (MIP), that is based on UCT, an algorithm for the multi-armed bandit problem designed for balancing exploration and exploitation in an online fashion. UCT has recently been highly successful in game tree search. We discuss the differences that arise when UCT is applied to search trees as opposed to bandits or game trees, and provide initial results demonstrating that the performance of even a highly optimized state-of-the-art MIP solver such as CPLEX can be boosted using UCT's guidance on a range of problem instances.

Introduction

The order in which a search tree is explored can have a dramatic impact on the performance of a solver designed to solve challenging combinatorial search and optimization problems. Various strategies for search tree traversal have been proposed and shown to exhibit different trade-offs. For instance, extensions of depth-first traversal work best in the context of propositional satisfiability (SAT), while best-first, fastest descent, and various heuristic combinations of the above work better in other contexts such as state space search and mixed-integer programming or MIP optimization (cf. Nemhauser and Wolsey, 1999; Wolsey, 1998). In these efforts, the goal is to find a way to balance exploration and exploitation in a manner that is most beneficial to the solver under consideration.

Upper Confidence bounds for Trees (UCT) (Kocsis and Szepesvári, 2006) is an exciting technique for balancing exploration and exploitation in search. It has received much attention during the past few years due to its success in game playing agents, especially for Go (Gelly and Silver, 2007, 2008) and Kriegspiel (Ciancarini and Favini, 2009), as well as for general game playing (Finnsson and Björnsson, 2008). UCT is based on the Upper Confidence Bounds (UCB1) selection strategy introduced by Auer et al. (2002) for the multi-armed bandit problem, which guarantees asymptotically optimal regret. In this work, we address the following question: *Can UCT inspired exploration-exploitation techniques help boost the performance of state-of-the-art combinatorial search and optimization solvers?*

Specifically, we consider optimization in the context of MIP and explore the impact of UCT as a node-selection heuristic for the CPLEX solver (IBM ILOG, 2010). We emphasize that CPLEX is a highly optimized commercial solver for MIP problems, obtaining a consistent improvement upon which on a variety of instances through general, domain-independent heuristic strategies is an extremely challenging task. Nevertheless, we pursue this goal rather than working with a limited set of problem domains or with, e.g., a self-designed branch-and-bound solver.

This agenda raises several interesting challenges due to the inherent differences between combinatorial optimization and game tree search. For instance, while UCT was originally introduced for single-agent tree search, its success and application has mainly been in the context of two-agent adversarial search. Further, UCT's "random playout" based sampling technique for evaluating the utility of a given state has been an appealing strategy in games such as Go where known heuristic functions for state evaluation are still quite weak. This is in stark contrast with tree search in the context of MIP optimization, where not only does the linear programming (LP) relaxation often serves as a very strong heuristic, this heuristic value is in fact a guaranteed upper or lower bound on the true objective value (depending on whether we are working with a maximization or a minimization problem, respectively). Finally, while the UCB1 strategy underlying UCT is designed to exploit (with some balance) a good "branch" once it discovers one, in the context of MIP search, one does not gain anything by revisiting and repeatedly exploiting a "terminal state" even if it always returns the optimal value. UCT must therefore be carefully adapted when applied to our setting.

We show that a UCT-inspired node selection strategy, appropriately modified to take the above mentioned differences into account, can have a positive impact even on sophisticated MIP solvers such as CPLEX. For completeness, we also consider other natural ways of guiding search near the top of the search tree, namely, breadth-first, depth-first, and best-first search (based on LP relaxation values). Given the additional overhead of maintaining our own "shadow" search tree for UCT computations, we find that the most benefit is achieved when UCT is used to provide guidance mostly near the top of the tree. We also find that the overhead of "log" and "square root" computations in the UCB1

formula underlying UCT can be substantial, and that a simpler ϵ-*greedy* version also introduced by Auer et al. (2002) works just as well in this setting. One of our key modifications to UCT is the use of a max-style update rule (the "backup operator") rather than the usual additive update rule when a new node is added to the UCT tree. While previous work in the context of game tree search has found max-style update rules to be too brittle, max-style update has clear benefits in our setting because the heuristic value used, namely the LP relaxation objective value, is a guaranteed upper or lower bound on the true value of the node.

While UCT is generally thought of as being tied to stochastic sampling of the space via random playouts, a good heuristic function, when available, can in fact work better for UCT. For example, Ramanujan et al. (2010) recently demonstrated this in the game of Chess where, unlike Go, very strong heuristic functions are available. We observe the same trend when applying UCT to CPLEX.

MIP Search, Node Selection, and UCT

We begin with a brief discussion of the basic mechanisms underlying search tree exploration by a MIP solver, specifically, CPLEX 12.2.0.0. The search starts with an empty root node, marked as *open*. It proceeds in general by selecting an open node N for expansion using a *node selection heuristic* $\mathcal{H}$. At this point, the solver tests the sub-problem associated with N for being infeasible, being worse than current best solution (the incumbent), or resulting in a new incumbent; it processes these cases appropriately and marks N as *closed*. If the test fails, assuming binary branching (e.g., bisection domain splitting), node N is split into two open nodes N_{left} and N_{right} by branching on some variable x and restricting its value to a subset of its domain, using a *branching heuristic*; N is marked as closed and N_{left} and N_{right} are marked as open. The search now continues by selecting another open node using $\mathcal{H}$. While the solver usually maintains only the list of open nodes, there is clearly an underlying *search tree* T that is being explored, with all internal nodes and some leaves marked as closed.

In this work, we explore the use of UCT operating on the underlying search tree T as the node selection heuristic $\mathcal{H}$. There are several natural candidates for $\mathcal{H}$ besides UCT, such as *best-first*, *breadth-first*, and *depth-first* selection. Best-first search would always greedily expand the node with the highest "quality" value (e.g., objective value of the LP relaxation) while breadth-first or depth-first would always expand an open node at the shallowest or deepest level, respectively. Combinations of these basic approaches, such as best-first mixed with depth-first "diving", often work well for MIP. On its own, best-first guides the search towards a solution and proof of optimality in the minimum possible number of explored nodes, but its greedy nature often results in a substantial overhead due to rapid context switches for the solver. Breadth-first search, on the other hand, is purely exploratory and ignores node quality information. Here we consider UCT as a promising candidate for balancing such exploration and exploitation.

At a high level, the UCT algorithm works as follows on an underlying tree T (see (Kocsis and Szepesvári, 2006) for details). It alternates between a node selection phase and a tree update phase. *Node selection phase*: Traverse T from the root to a leaf by following, at each node N, the child N' whose *UCT score* is higher (breaking ties arbitrarily). The UCT score of a node N with parent P is defined by the UCB1 formula: $\text{estimate}(N)+\Gamma\cdot\sqrt{\log \text{visits}(P)/\text{visits}(N)}$, where Γ is a fixed constant balancing exploration and exploitation, $\text{visits}(N)$ indicates the number of times N has been visited by UCT so far (similarly for $\text{visits}(P)$), and $\text{estimate}(N)$ is an estimate of the "quality" of N if N is currently a leaf node of T and is otherwise the value resulting from previous tree update phases. *Tree update phase*: Once node selection reaches a leaf L of T, the estimate for L is computed and propagated upwards in T towards the root so that each node N now on the path from L to the root has a value that equals the *average* value seen in the entire subtree rooted at N, and visits is incremented by 1; this is known as the *backup operator* for UCT. L is now further expanded by branching, if possible.

Guiding MIP Optimization with UCT

In order to perform UCT-based node selection within CPLEX, additional infrastructure must be put in place. We maintain a "shadow" search tree T' whose open leaves coincide with the open nodes list maintained internally by CPLEX.[1] Each node maintains a counter for the number of UCT visits to it so far, and a measure of quality — which for a newly created node is taken to be its LP objective value normalized by the root LP value.[2]

Assume for simplicity that we have a maximization problem. In contrast to the common averaging backup operator used in UCT, we propagate the *maximum* of the current estimates of the two children when updating the UCT tree. This is motivated by the fact that we do not perform sampling to obtain a node quality estimate, but can instead rely on a guaranteed bound. Hence, averaging would simply blur the knowledge that one has at a given (internal or leaf) node in T'. Further, for computational efficiency, we replace the UCB1 selection criteria with the following simpler ϵ-*greedy* version (Auer et al., 2002) for a node N with parent P:

$$\text{score}(N) = \text{estimate}(N) + \Gamma \cdot \frac{\epsilon \times \text{visits}(P)}{\text{visits}(N)} \qquad (1)$$

As mentioned above, $\text{estimate}(N)$ is the normalized LP objective value when N is a leaf node. Based on a small manual search, the value 0.7 for Γ worked well in our setting . We use a fixed value 0.01 for ϵ (which would normally slowly decrease over time in order to obtain theoretical guarantees of UCT converging to the true optimal value (Auer et al., 2002)). Equation (1) aims at balancing exploration

[1] Maintaining a search tree that properly mimics CPLEX's open nodes is somewhat more complex than one might expect because of issues related to capturing every event that may cause CPLEX to close nodes in-between node- and branch-callbacks.

[2] We also experimented with more refined measures combining the LP objective value with the number of integer infeasibilities as a "confidence" guide or with pseudo-costs, but did not observe a clear improvement in performance.

Table 1: Summary of results: Comparison of various node selection strategies in terms the number of unsolved instances within 600 seconds, the number of instances where they had the best runtime, and their average runtime, across a set of 170 benchmarks.

	default CPLEX	***UCT***	*best-first*	*breadth-first*	*depth-first*
# time outs	10	**8**	16	17	14
instances fastest on	31	39	27	29	**41**
average runtime (sec)	158.1	**155.7**	167.6	177.1	177.7

Table 2: Direct comparison of node selection strategies. Shown are the number of instances on which the row approach outperforms the column approach in terms of runtime by at least 10%.

	default CPLEX	*UCT*	*best-first*	*breadth-first*	*depth-first*
default CPLEX	–	52	44	**61**	51
UCT	**64**	–	**62**	**62**	**63**
best-first	**47**	49	–	**55**	52
breadth-first	57	49	50	–	56
depth-first	**55**	60	52	**57**	–

and exploitation. While nodes with very promising objective values are pursued because of the high "estimate" term in the expression, sub-optimal nodes begin to get priority if they have been visited roughly only an ϵ fraction of the time their sibling has been visited.

Nodes that fail or are pruned by CPLEX are removed from T' (without any objective estimate "penalty" propagated upwards) and never visited again by UCT. For nodes that do yield a feasible solution, we do not treat their resulting objective value in any special fashion when back propagating and we remove them from further consideration in T' because, unlike the usual multi-armed bandit setting of UCT, the optimization process doesn't have any incentive to revisit this node ever again.

Experimental Evaluation

We compare the performance of our UCT based node selection strategy with CPLEX's default heuristic as well as alternative approaches. The experiments were conducted on Intel Xeon CPU E5410 machines, 2.33GHz with 8 cores and 32GB of memory, running Ubuntu. In our evaluation we always executed only one run per machine to prevent interference between runs. This is critical when experimenting with CPLEX as its runtime is known to vary by as much as 30-40% when multiple runs are performed on a single machine, even when using different CPUs. We use CPLEX 12.2.0.0 (IBM ILOG, 2010) with node and branch "callbacks" turned on (using empty callbacks) as our baseline CPLEX solver ("default CPLEX").[3] Starting with a wide selection of publicly available benchmarks comprising 1,024 instances, we kept the 170 instances (see Appendix) that required between 10 and 900 seconds to be solved by default CPLEX. These instances represent a challenging benchmark spanning a variety of problem domains.

It turns out that UCT-inspired node selection often does not pay off if applied *throughout* the search, for various reasons. Maintaining the infrastructure for UCT in addition to CPLEX's computations at each node is simply too expensive. In addition, UCT's guidance deep down in the tree appears not to be particularly accurate, which we tried to alleviate by searching for better nodes from a local neighborhood, based on CPLEX's node ranking, after reaching a certain depth threshold. While this improved our strategy significantly, it still did not outperform default CPLEX (mainly in terms of runtime). Since we believe that the largest impact lies in the choice of the nodes close to the top of the search tree, in the experiments considered here, we only apply our strategy at the very beginning of search. In fact, we turn off custom node selection after 31 nodes (equivalent to the first 5 levels of a complete binary tree), and then apply CPLEX's default node selection heuristic.

The results, with a 600 second timeout, are summarized in Tables 1 and 2. We compare default CPLEX with our UCT based node selection strategy, and with best-first, breadth-first, and depth-first strategies. In all settings we revert back to CPLEX's default node selection after processing 31 nodes.

Table 1 reports, for each of the node selection strategies, the number of instances that could not be solved in 600 seconds, the number of instances a given strategy was the fastest on, and the average runtime across all 170 instances. Although the runtimes were often comparable, our approach was the fastest on 39 instances and had the fewest timeouts (8) with the lowest average runtime (155.7 sec). To our surprise, depth-first selection with a 31 node cutoff was the fastest on a couple of more instances than even UCT. However, it did not excel in other metrics.

Table 2 shows a direct, pair-wise comparison of all approaches. We say that an approach outperforms another ap-

[3] Note that this causes some features of CPLEX to be turned off (e.g., dynamic search) but is the only way to enhance CPLEX with a custom node selection strategy without access to the internals of CPLEX.

proach on an instance if there is a difference of runtimes of at least 10%. According to this measure, our approach again appears to be the best with 62 to 64 pair-wise wins (notice that UCT's row is the only one in which all numbers are in bold), albeit by not a very high margin.

UCT guides the search in a manner that appears to be complementary to CPLEX's default node selection heuristic, as evidenced by the fact that when taken together, the best of the two for each instance was able to solve all but 3 out of the 170 instances within the time limit, in an average of only 130 seconds.

In conclusion, these results suggest that the UCT method for balancing exploration and exploitation, used typically in adversarial game trees and stochastic settings, holds promise also as a node selection strategy in the context of MIP solvers.

References

P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

P. Ciancarini and G. P. Favini. Monte Carlo tree search techniques in the game of Kriegspiel. In *21st IJCAI*, pp. 474–479, Pasadena, CA, July 2009.

H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *23rd AAAI*, pp. 259–264, Chicago, IL, July 2008.

S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *24th ICML*, pp. 273–280, Corvallis, OR, June 2007.

S. Gelly and D. Silver. Achieving master level play in 9×9 computer Go. In *23rd AAAI*, pp. 1537–1540, Chicago, IL, July 2008.

IBM ILOG. IBM CPLEX Optimization Studio 12.2.0.0, 2010.

L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *17th ECML*, vol. 4212 of *LNCS*, pp. 282–293, Berlin, Germany, Sept. 2006.

G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1999.

R. Ramanujan, A. Sabharwal, and B. Selman. Understanding sampling style adversarial search methods. In *26th UAI*, Catalina Island, CA, July 2010.

L. A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.

Appendix: Benchmark Set Used in Experiments

10teams ab51.40.100 ab71.20.100 acc-tight3 acc-tight4 acc-tight5 acc-tight6 air04 air05 aligninq arki001 atlanta-UUM bc1 berlin bienst1 binkar10_1 bley_xl1 bley_xs2 brasil dano3_3 dano3_4 dano3_5 dfn-gwin-DBE dfn-gwin-DBM dfn-gwin-UUE di-yuan-DBE eil33.2 eilB101 exp.1.1000.20.2 exp.1.500.20.1 exp.1.500.20.5 exp.1.500.50.2 exp.1.500.50.4 exp.1.500.50.5 exp.1.5000.5.2 exp.1.5000.5.3 fc.60.20.2 fc.60.20.6 france-DBM france-UUM g200x740 g200x740b g55x188 harp2 ic97_tension k20x380 l451x885b markshare_4_0 mas76 mik.250-20-75.1 mik.250-20-75.2 mik.250-20-75.3 mik.250-20-75.4 mik.250-20-75.5 misc07 mkc1 mod011 mzzv11 mzzv42z n12-3 n5-3 n7-3 neos-1109824 neos-1112782 neos-1112787 neos-1171737 neos-1200887 neos-1211578 neos-1215259 neos-1228986 neos-1337489 neos-1440225 neos-1440447 neos-1445738 neos-1445743 neos-1445755 neos-1445765 neos-1480121 neos-1582420 neos-1597104 neos-1620807 neos-430149 neos-476283 neos-480878 neos-503737 neos-504674 neos-504815 neos-512201 neos-522351 neos-530627 neos-538867 neos-547911 neos-555424 neos-570431 neos-584851 neos-585192 neos-593853 neos-595925 neos-686190 neos-785899 neos-801834 neos-803219 neos-803220 neos-806323 neos-807639 neos-807705 neos-808072 neos-810326 neos-820879 neos-825075 neos-827015 neos-829552 neos-839859 neos-860300 neos-862348 neos-906865 neos-912023 neos-916173 neos-935627 neos-935769 neos-936660 neos-937446 neos-937511 neos-941313 neos-941698 neos-960392 neos1 neos11 neos12 neos14 neos17 neos18 neos20 neos21 neos22 neos23 neos6 neos7 nexp.50.20.4.1 nexp.50.20.4.3 nexp.50.20.8.2 nexp.50.20.8.3 ns4-pr4 ns60-pr9 nu25-pr4 nu60-pr4 p50x288b p80x400 pdh-DBE pdh-DBM pdh-UUE pdh-UUM pk1 prod1 r20x200 r50x360 ran10x26 ran12x21 ran13x13 ran16x16 rout seymour1 sp98ir stein45 swath1 swath2 ta1-DBE ta1-DBM ta2-UUE ta2-UUM