Best-first Search for Bounded-depth Trees

Kevin Rose and **Ethan Burns** and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
`kja24`, `eaburns` and `ruml` at `cs.unh.edu`

Abstract

Tree search is a common technique for solving constraint satisfaction and combinatorial optimization problems. The most popular strategies are depth-first search and limited discrepancy search. Aside from pruning or ordering the children of each node, these algorithms do not adapt their search order to take advantage of information that becomes available during search, such as heuristic scores or leaf costs. We present a framework called best-leaf-first search (BLFS) that uses this additional information to estimate the cost of taking discrepancies in the search tree and then attempts to visit leaves in a best-first order. In this way, BLFS brings the idea of best-first search from shortest path problems to the areas of constraint satisfaction and combinatorial optimization. Empirical results demonstrate that this new dynamic approach results in better search performance than previous static search strategies on two very different domains: structured CSPs and the traveling salesman problem.

Introduction

Combinatorial search problems in artificial intelligence can be divided into three classes: constraint satisfaction, combinatorial optimization and shortest path. Informally, in a constraint satisfaction problem (CSP) we must select values for each of a set of predefined variables such that a given set of constraints is not violated. A complete assignment that does not violate any constraints is called a solution. Combinatorial optimization problems (COPs) are like CSPs, however, there is an additional function that assigns a cost value to each solution and we wish to find the cheapest possible solution. In a shortest path problem, we must select a sequence of operations that will reach a desired goal state from a given initial state and we wish to find the cheapest such sequence.

CSPs and COPs are typically solved by searching over the bounded-depth tree of partial variable assignments. This search space is formed by starting with an empty assignment and then assigning values to variables until a solution is found or a constraint is violated. In COPs, a search typically does not stop after finding the first solution, but will continue, pruning subtrees that are more costly than the current incumbent solution.

When solving CSPs, a variable choice heuristic is often used to select a variable to assign, forming a branching node in the search tree. A value ordering heuristic is used to order the values from the variable's current domain, forming arcs from the branching node to its children in the tree. Because traditional tree searches follow a fixed backtracking policy, these orderings are the main way in which heuristic information computed during the search can influence search order. Constraint propagation is used to reduce the size of this search space by taking into account the constraints as well as the current partial variable assignment. Domain values that propagation has determined cannot participate in any solution under the current node are removed from consideration. Constraint propagation can prune away large portions of a search tree, and while it can sometimes influence the heuristics that choose variables and order values, it does not determine search order directly. In this paper, we are concerned with how best to search the tree that remains after constraint propagation, dynamic variable choice, and value ordering have been performed.

Unlike CSPs and COPs, shortest path problems are typically solved with best-first search. This is because the number of decisions is usually not bounded in advance. Best-first search algorithms make use of heuristic lower bound functions on cost-to-go in order to estimate the cost of a solution beneath a given frontier node. Frontier nodes are stored in a priority queue and expanded in order of increasing estimated solution cost. Thus, heuristic information computed at each node directly influences every decision about which node the search should expand next, and the search has the freedom to select a node in any part of the tree.

In this work, we present a search framework called best-leaf-first search (BLFS) that brings the idea of best-first search from shortest path problems to CSPs and COPs. BLFS attempts to visit leaves in a best-first order, where 'best' is defined by using heuristic information calculated during the search. BLFS can be viewed as an adaptation of iterative-deepening A* (IDA*, Korf 1985), a best-first search algorithm that is commonly used in shortest-path problems, to fixed-depth search trees. This adaptation requires two novel techniques. First, a node evaluation function is required to guide the search. Second, iterative deepening must be adapted to handle real-valued costs. In an empirical evaluation on both CSPs and COPs, we find that

BLFS provides excellent search performance on structured CSPs, competitive performance on random CSPs, and excellent performance on the traveling salesman problem, a type of COP. BLFS establishes a useful connection between heuristic search in AI and tree search in CSPs and COPs.

Background

We begin with a brief review of depth-first and discrepancy search. We will follow the convention that values are ordered from left to right so that the left-most successor of a node is the heuristically preferred value for the given variable.

Depth-first search (DFS, also known as chronological backtracking) visits the successors of each node in a left to right order. When it backtracks, it returns to the most recently visited node with unvisited successors. Because it never generates a node twice, depth-first search is a very efficient algorithm for enumerating a full tree. For large problems, there is no hope of visiting the entire tree in any reasonable amount of time. Instead, we would like to find a solution as fast as possible or the best solutions that we can within the time available. Unfortunately, depth-first search focuses all of its effort deep in the far left subtree and often misses better solutions down non-heuristically-preferred branches. If the heuristic makes a mistake near the top of the tree, DFS won't reconsider this choice until it has explored the entire subtree, which will take an unreasonable amount of time for large trees.

One might suspect that the value ordering heuristic is just as likely, or even more likely, to make a mistake at the top of a search tree as it is at the bottom. In recognition of this, the limited discrepancy search algorithm (LDS, Harvey and Ginsberg 1995) and its improved version (ILDS, Korf 1996) visit leaves in order of the number of deviations from the heuristic ordering (called *discrepancies*) along their paths from the root. ILDS visits all leaves with exactly k discrepancies along their paths during iteration k. Viewed another way, ILDS is counting each edge taking the heuristically preferred child as having a cost of zero and each edge taking a discrepancy as having a cost of one. It then visits leaves in increasing order of cost. ILDS makes the assumption that all discrepancies have equal cost. This may not be the case, and perhaps some discrepancies should be favored well before others.

Depth-bounded Discrepancy Search (DDS) (Walsh 1997) is similar to ILDS, however it makes the assumption that the child ordering heuristic is more accurate at lower levels of the tree. DDS maintains a depth bound that starts at zero and is incremented during search. DDS counts all discrepancies above the current depth bound as free and it takes no discrepancies deeper than the bound. If no solution is found, the bound is increased by one and the search restarts. Due to node re-expansion, DDS has significant overhead for each leaf that it visits and, like ILDS, it makes assumptions in advance about which discrepancies should be preferred.

Best-leaf First Search

Ideally, rather than assuming discrepancy costs, one would estimate the cost of a discrepancy by using information that becomes available during search. The best-leaf-first search (BLFS) framework does just this. First, we will present the generic search framework that assumes that an edge cost model has been provided. Then we will discuss two such cost models. The indecision cost model estimates the cost of discrepancy by using the child ordering heuristic and can be used on both CSPs and COPs. The quadratic cost model uses on-line regression to learn discrepancy costs from the costs of leaf nodes. It requires a function to give the cost of a leaf node, making it well suited for COPs but not directly applicable to CSPs.

At a high-level, BLFS proceeds as iterative-deepening A* (Korf 1985): performing a series of cost bounded depth-first searches. BLFS uses a cost model to define the relative worth of nodes and a bound estimation procedure to select cost thresholds to use across the iterations of search. First, BLFS visits all leaves whose predicted cost is minimal. The main loop of the search estimates a cost threshold that will result in the desired number of expansions for the subsequent iteration. A cost bounded depth-first search is then performed. If the search fails to find a solution, the cost threshold is increased. Because BLFS is based on depth-first search, it uses memory that is only linear in the maximum depth of the tree. Assuming the bound estimation is accurate, it is also possible to geometrically increase the number of nodes visited across iterations, bounding the overhead from node re-expansions (Sarkar et al. 1991). This will also allow BLFS to eventually visit the entire search tree, making it *complete* in that it will find a solution if one exists.

Indecision Cost Model

Because edges in a search tree do not necessarily have costs associated with them, BLFS uses a cost model to estimate the cost of each edge. One simple model is to use information from value ordering. It is often the case in a tree search that the value ordering heuristic is based on a quantitative score. While both DFS and ILDS use this score to order the successors of a node, they ignore the finer distinctions enabled by the score values. Given two backtracking destinations, it seems preferable to select the location where the heuristic was less certain about ranking the successors of a node, i.e., where the heuristic scores were very close to each other. If expanding a node yields a set of successor nodes, S, we define the *indecision* of a node n in S as its heuristic score minus the score of the best node in S:

$$indecision(n) = h(n) - \min_{s \in S} h(s) \qquad (1)$$

where h is the heuristic function being used to order the successors of a node. For example, if the scores of two siblings are 2 and 4, then their indecisions are 0 and 2, respectively. For siblings with scores 5 and 100, the indecisions are 0 and 95, indicating greater decisiveness on the part of the scoring function. We define the cost of a node as the sum of all indecision values along its path from the root:

$$cost(n) = indecision(n) + cost(parent(n)) \qquad (2)$$

a_0+=1
b_0+=d/D=1/2
c_0+=(d/D)^2=1/4

a_1+=1
b_1+=d/D=1
c_1+=(d/D)^2=1

a_0=1, b_0=1/2, c_0=1/4
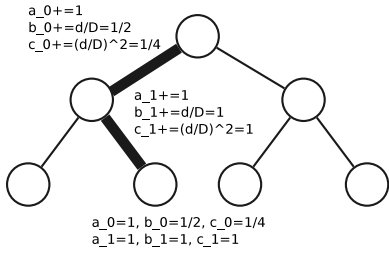a_1=1, b_1=1, c_1=1

Figure 1: Quadratic model training example

Note that the 0-rank sibling will always have an indecision value of zero. This guarantees that a search using indecision values as a proxy for edge cost will visit a leaf beneath each node that it expands because following the 0-ranked children to a leaf will not incur any additional cost. We call BLFS using the indecision cost model *indecision search*. By quantifying the cost of each discrepancy taken along the current path, indecision search can adaptively select which paths to explore based on the actual heuristic values seen during the search. It also generalizes discrepancy search to non-binary trees in a principled way.

Quadratic Cost Model

In optimization problems, additional information is available: leaves have an associated cost. These leaf costs provide information on the accuracy of the heuristic along the path to the given leaf. The quadratic cost model uses linear regression to learn the costs of discrepancies. Although various methods can be used, we present one in which leaf cost is the sum of edge costs along the path from the root. To reduce the number of parameters and enforce smoothness, we model the cost of each edge as a quadratic function of its depth. The cost of selecting a child of rank i at depth d is

$$edgecost(d, i) = a_i + b_i \frac{d}{D} + c_i \left(\frac{d}{D}\right)^2 \quad (3)$$

where D is the maximum depth of the tree and a_i, b_i and c_i are the regression coefficients for successor rank i.

Each time a solution is encountered, its cost and path are used as a training example for on-line LMS regression (Mitchell 1997). To build a training example from a path through the tree, we must create data points for each of the three features for each of the n child ranks. Let $E(i)$ be the set of edges along the current trajectory that correspond to rank i successors and $d(e)$ be the depth of a given edge $e \in E(i)$. The quadratic model estimates the leaf cost as the sum of each edge cost, giving:

$$
\begin{aligned}
leafcost &= \sum_{i=0}^{n-1} \quad edgecost(d(e), i) \\
&= \sum_{i=0}^{n-1} \quad a_i |E(i)| + \\
&\qquad b_i \left(\sum_{e \in E(i)} \frac{d(e)}{D}\right) + \quad (4) \\
&\qquad c_i \sum_{e \in E(i)} \left(\frac{d(e)}{D}\right)^2
\end{aligned}
$$

So, to estimate the coefficients a_i, b_i and c_i, we must compute $|E(i)|$, $\sum_{e \in E(i)} d(e)/D$ and $\sum_{e \in E(i)} (d(e)/D)^2$ for each rank $0 \leq i < n$. These sums can be easily updated each time the search traverses an edge or backtracks

by adding/subtracting from the current sum for the respective coefficient. Figure 1 shows an example trajectory in a search tree with $D = 2$ and $n = 2$. Each of the two edges in the trajectory are labeled with their contribution to the respective data points and the leaf node is labeled with the final set of 6 feature values that are passed to the regression model. Using the learned feature coefficients, Equation 3 may be used to estimate the edge cost for a rank i edge at depth d when searching.

Because the quadratic model updates the coefficients each time it encounters a leaf, a copy of the model is made before each iteration of search. This allows the edge costs to remain constant during an iteration of search while the coefficients that will be used in the next iteration are fluctuating. Also, to ensure that the left-most successor of a node has a cost of zero, the costs of the model are normalized so that the 0-rank successor has a cost of zero.

Bound Estimation

When the number of nodes between cost bounds does not grow geometrically, IDA* may perform $\mathcal{O}(n^2)$ expansions where n is the number of nodes expanded in its final iteration (Sarkar et al. 1991). The cost functions used by BLFS tend to have many distinct values, giving rise to non-geometric growth of iterations. In order to prevent this worst-case behavior, a model of the search space can be learned on previous iterations of search to estimate a threshold that will cause geometric growth. In our implementation, we try to double the number of nodes visited across iterations. If this is achieved then the total overhead due to node re-expansion is bounded by a constant factor of three in the worst case.[1]

If we were given a distribution over the cost of the nodes in the search tree, it would be a simple matter to select a bound that allows the desired number of expansions: choose a bound such that the weight to the left in the distribution is the desired number. Since we are not given this distribution, we estimate it using the distribution of cost values seen during the previous iteration and simulating the growth of the tree using a recurrence relation. The recurrence estimates, for each depth, the cost of the nodes that will be generated, based on the estimated distribution of costs at the previous depth and the distribution of edge-costs at the current depth:

$$\hat{C}(d + 1) = \hat{C}(d) * \Delta(d) \quad (5)$$

where $\hat{C}(d)$ is an estimation of the distribution of costs at depth d, $\Delta(d)$ is a distribution of the cost of the edges from depth d to $d + 1$ and $*$ is the convolution operator (defined in detail below) which adds the $\Delta(\cdot)$ distribution to each of the $\hat{C}(\cdot)$ values forming the new distribution of the costs over the newly generated nodes. Equation 5 builds the $\hat{C}$ distribution recursively starting with the root node that has a distribution of 1 node with 0 cost and the Δ distributions

---

[1]If n is the number of nodes visited during the last iteration of search, then in the worse case, the previous iteration will have visited $n - 1$ nodes. In the case of doubling, the combined total of all the nodes visited on iterations prior to the last two is $(n - 1)/2 + (n - 1)/4 + ... < n$. So the total number of nodes visited by BLFS in the worse case is approximately $3n$.

GET-BOUND (*desired*)
1. $total \leftarrow$ an empty histogram
2. $\hat{C} \leftarrow$ an empty histogram
3. add $\{\text{cost} = 0, \text{weight} = 1\}$ to $\hat{C}$ and $total$
4. for $i = 1$ to $d_{\max}$ do
5. $\Delta \leftarrow$ edge-cost histogram for depth i
6. $\hat{C} \leftarrow \hat{C} * \Delta$
7. $total \leftarrow total + \hat{C}$
8. return cost_for_weight ($total$, $desired$)

Figure 2: Pseudo-code sketch of bound selection.

for each depth that are either recorded during the previous iteration of search (in the case of indecision cost model) or are extracted directly from the cost model (for the quadratic cost model). As the computation of the recurrence proceeds, we also accumulate a total distribution containing all of the costs for each depth d. When the recurrence reaches the final depth, the accumulated distribution can be used to find the desired cost threshold.

Figure 2 shows high-level pseudo code for this estimation technique. Distributions are represented using fixed-size histogram data structures. The $total$ histogram contains the distribution of the costs of all nodes at all depths that have been computed thus far. The $\hat{C}$ histogram contains the distribution of the cost of all of the nodes at the current depth in the recurrence. When the maximum depth is reached, the bound for the desired number of nodes can be found in the $total$ histogram (line 8).

The convolution operation (line 6) used to combine cost distribution from one depth with the edge cost distribution works by adding all values in the input histograms and multiplying the weights. So, the convolution of two histograms $\omega_a * \omega_b = \omega_c$, where ω_a and ω_b are functions from values to weights, and their convolution is $\omega_c(k) = \sum_{i \in Domain(\omega_a)} \omega_a(i) \cdot \omega_b(k-i)$. The resulting histogram is a distribution of costs after summing the costs of the nodes at one depth with the cost of the edges that generate the successors at the next depth.

One technique for increasing accuracy and speed of the simulation is pruning. While simulating the number of nodes at each depth of the tree, if the $total$ histogram ever contains more than the number of desired nodes, the excess nodes can be pruned away from the right-hand side of the histogram.

Experimental Results

To evaluate the performance of indecision search we compare it to DFS, ILDS, DDS and the following state-of-the-art tree search algorithms:

Weighted Discrepancy Search: (WDS, Bedrax-Weiss 1999) is similar to indecision search, however it weights the value of each heuristic discrepancy with a score between 0 (less preferred) and 1 (most preferred). Discrepancies where the score of the best child and the next-best child are similar will be closer to 1 and discrepancies where the scores are very dissimilar will have values closer to zero. Beginning with a threshold of 1, WDS performs a depth-first search of all nodes for which the

product of the weights along the path from the root to the given node are within the threshold. When all nodes in an iteration are exhausted, the threshold is decreased to allow more nodes to be explored and the search is repeated. Bedrax-Weiss (1999) used offline analysis to find optimal bound schedules and presented results (without CPU times) showing that WDS was surpassed by DFS or ILDS on 3 of the 4 versions of number partitioning that she considered. We propose a novel implementation using the technique of IDA$_{CR}$ (Sarkar et al. 1991) to estimate an appropriate bound on-line, which we call on-line WDS (OWDS). We also tried the estimation approach of IDA$_{CR}$ with BLFS, however it performed worse than the technique described previously.

YIELDS: (YIELDS, Karoui et al. 2007) was introduced for solving CSPs. The algorithm extends LDS by adding a variable choice learning scheme and an early stopping condition. YIELDS weights the variables of a CSP, incrementing a variable's weight whenever the discrepancy limit prevents the search from exploring all domain values for that variable. This weight is used to break ties when the normal variable choice heuristic is used. A higher weight is preferred so that variables that cannot be completely explored despite their domain being non-empty will be visited higher in the tree on subsequent iterations. For problems with non-binary branching, visiting a child of rank k is counted as k discrepancies.

The variable learning scheme of YIELDS can be incorporated into any cost-bounded search for CSPs. We added the YIELDS variable learning method to indecision search, OWDS, DDS, and ILDS for our CSP results. Indecision search, OWDS and DDS benefited from this change on all CSP problem classes. ILDS benefited on all but the latin square domain. In the results presented below, we only show the version of the algorithm which performed the best for the specific domain. We also give results for the plain YIELDS algorithm as it was originally presented.

Indecision RBFS: Recursive best-first search (RBFS, Korf 1993) is an alternative formulation of linear-memory best-first search. An RBFS-based variant of indecision search has the advantage that it is not iterative and therefore does not need to perform bound estimation. RBFS may re-expand nodes many times, however, as it rebuilds previously explored paths. We implemented this RBFS variant, however, we do not present results for it because it was not competitive with the iterative-deepening version.

ILDS variants: In a given iteration of either ILDS or BLFS, the discrepancies may be taken starting at the bottom of the tree first or the top of the tree first. In the following experiments the variant that gave the best performance was chosen. For CSPs, ILDS performed better with discrepancies taken from the top first. On the TSP, the variant of ILDS that prefers discrepancies at the bottom was used. For BLFS, bottom-first always gave better performance. Also for ILDS, all non-preferred child nodes were counted as a single discrepancy in problems with non-binary branching.

Constraint Satisfaction Problems

We ran on three sets of CSP benchmarks. For all problems, the constraints were encoded as binary nogoods and repre-

sented in extension. The variable choice heuristic used was the popular `dom/wdeg` (Boussemart et al. 2004) for the random binary and geometric problems and `min_domain` (Haralick and Elliott 1980) for the latin squares[2]. The value choice heuristic was the `promise` heuristic (Geelen 1992). Forward checking (Haralick and Elliott 1980) was used for constraint propagation after each variable assignment. It is a popular method because propagation is quick for problems with large numbers of constraints and it is simple to implement efficiently. All experiments used a time limit of 5 minutes per instance. For CSPs, we only consider the indecision cost model for use with BLFS. Since the quadratic cost model requires a cost for each leaf to learn on, we only consider it for the COP results. Likewise, since YIELDS is designed for solving CSPs, we only include it in the CSP results. We include plots for both CPU time and nodes visited.

Latin Square Completion   The latin square completion problem is a popular puzzle that can be encoded easily into a CSP. It has been suggested as a realistic benchmark because instances can be generated randomly but the problem still contains structure. The goal is to fill in a square n by n grid with numbers ranging from 1 to n such that a number appears exactly once in each row and in each column. These problems are easily solved when starting from a blank grid, but become more difficult depending on the percentage of cells pre-filled in (Gomes and Shmoys 2002). Figure 3 shows results on 100 latin squares of order 30 with 42% of the cells already assigned. As the plot shows, indecision search performs the best with OWDS also performing well. ILDS is a distant third. Depth-first search performed extremely poor, solving fewer than 10% of the problems in the time allowed.

Figure 4 shows results on the same problem instances but in terms of logical nodes visited by each algorithm. The same overall ordering of the algorithms is observed in the nodes plot as in the CPU time plot. We define a logical node as a either a branching point in the tree where the search algorithm must make a choice, or a leaf. Note that in CSPs, a logical node could represent several actual variable assignments due to unit propagation. If constraint propagation reduces the domain of a variable to a single value, it will automatically be assigned that value and more constraint propagation and heuristic computation will be triggered. In the plot, it seems that DFS is extremely slow at expanding nodes, which is counter-intuitive since DFS has virtually no overhead. The reason is that unit propagation is happening with higher frequency than the other algorithms. This is likely because DFS is spending more time deep in the tree where domains have been pruned excessively due to constraint propagation. This increases the chances that pruning a value from a variable's domain will cause unit propagation to occur. So while it is true that some algorithms are expanding fewer logical nodes per second, they are all performing approximately equal assignments per second.

---

[2]We found `min_domain` performed better with respect to CPU time because `dom/wdeg` took much longer to compute due to the large numbers of constraints present.
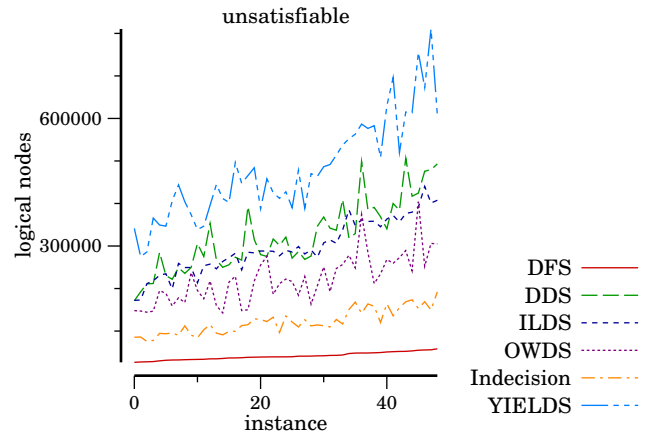

Figure 5: Logical nodes visited per instance to prove random binary CSPs unsatisfiable. Instances ordered such that nodes is increasing for DFS.

Random Binary CSPs   In this type of problem, all constraints are between two randomly chosen variables. A particular class of random binary problems can be described by the tuple $\langle n, m, p_1, p_2 \rangle$ where there are n variables, each having a uniform domain of size m. Exactly $p_1 \cdot n(n-1)/2$ of the possible variable pairs are constrained and exactly $p_2 \cdot m^2$ of the possible value combinations are disallowed. The center plot in Figure 3 shows results on 100 instances with the parameters $\langle 35, 25, 0.3, 0.5 \rangle$. In these problems, ILDS using the YIELDS variable learning performs the best. Indecision search is competitive with the other algorithms, ultimately solving the second largest number of instances in the allotted time. DFS, while performing poorly on latin squares, is more competitive on these random problems. The center plot of Figure 4 shows results for nodes visited by each algorithm for these problems. The ordering is approximately the same except that DFS, while not solving as many problems, visits fewer logical nodes than the other algorithms for the problems that it does solve.

Geometric Quasi-Random CSPs   The geometric quasi-random problems are binary CSPs similar to the ones described by Wallace (1996). These instances are generated by randomly assigning (x, y) coordinates inside a unit square to each variable. Constraints are added between all variables that are within a specified radius of each other. These problems are slightly more structured than pure random binary CSPs because the constraints are clustered according to the coordinates. The problems we tested had 50 variables, each with a domain size of 20. As seen in Figure 3, OWDS and indecision search performed better than all other algorithms. OWDS appears to perform slightly better than indecision search, however they both are able to solve the same number of instances. Figure 4 shows results in terms of nodes in the rightmost plot. This plot closely mirrors the CPU time plot as the algorithms have the same relative performance.

Unsatisfiable Random Binary CSPs   In this experiment we ran on only unsatisfiable problems. In order to prove a problem unsatisfiable, the search algorithm must visit the entire search tree. This puts all of the discrepancy based al-
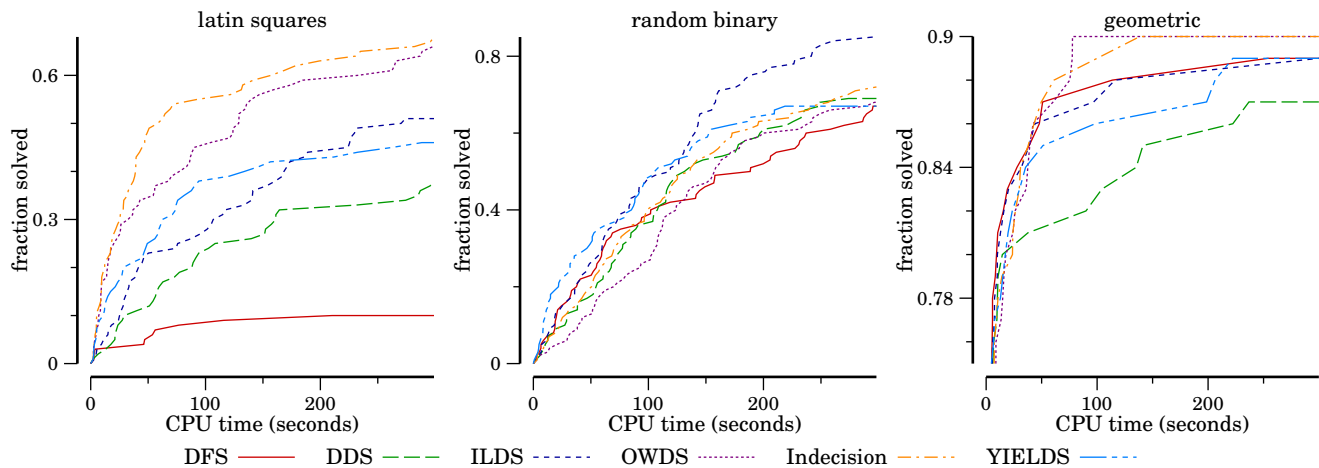
Figure 3: CSPs: The fraction of problems solved as a function of CPU time when using each search strategy.
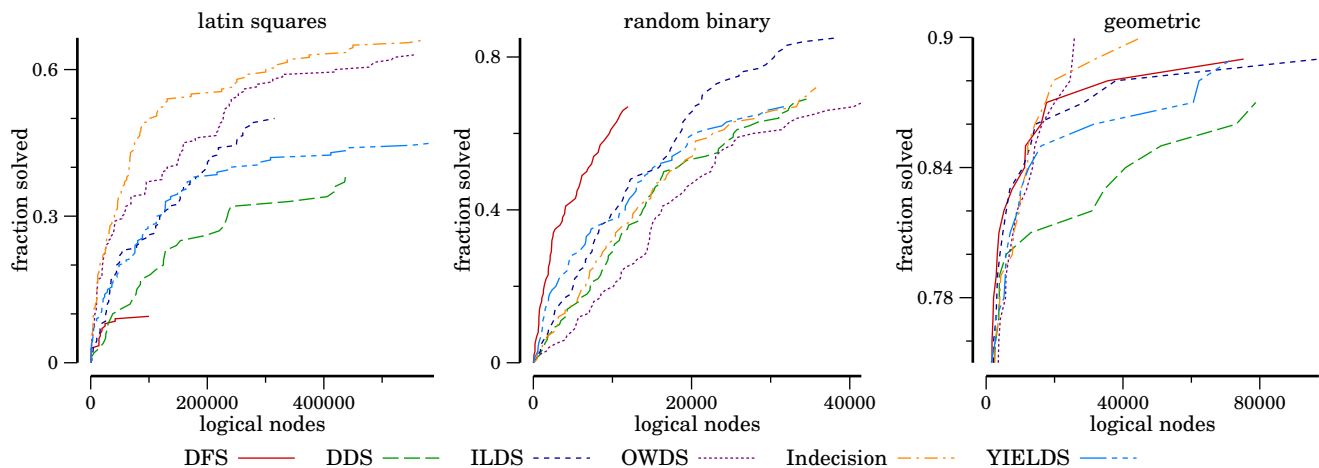

Figure 4: CSPs: The fraction of problems solved as a function of logical nodes visited when using each search strategy.

gorithms, indecision search included, at a disadvantage because they re-expand nodes across search iterations. DFS is optimal in this respect because it only visits each node once. To test the amount of overhead incurred from revisiting nodes, we compare the performance of all algorithms to that of DFS in terms of nodes visited. Figure 5 shows the number of nodes visited by each algorithm on an instance by instance basis. The problems ran on were 47 unsatisfiable random binary CSPs generated with the parameters $\langle 40, 11, 0.96, 0.1 \rangle$. The instance numbers in the plot are assigned such that the number of nodes visited by DFS is increasing. As expected, DFS visits the fewest nodes to prove each instance unsatisfiable. Indecision search clearly visits the second fewest nodes, and OWDS the third fewest nodes. As mentioned in the section describing the bound estimation technique, as long as the number of nodes that indecision search visits on each iteration is double the number visited on the previous iteration, the amount of overhead due to node re-expansions is bounded. This experiment shows that the bound estimation procedure of BLFS works well in practice. Coupled with the previous experiments, indecision

search has been shown to provide robust performance across different types of CSPs as well as on satisfiable and unsatisfiable instances.

Traveling Salesman Problem

The *traveling salesman problem* (TSP) is a combinatorial optimization problem that asks for the shortest tour of a set of cities, given the distance between each pair of cities. We use a straightforward tree search where each node represents a partial tour of the cities. To break symmetries, we select an initial city as the start city and another city that must appear in the second half of the tour (Pearl and Kim 1982). Our value ordering heuristic sorts successors on the cost of the partial tour represented by the node plus the minimal spanning tree of the remaining cities. This value gives a lower bound on cheapest tour that may extend from the partial tour.

We used three different instance sets for this domain. The first two sets, "usquare" with 50 and 100 cities, were generated by placing the respective number of cities in random locations on the unit square and using Euclidean distance. The final set, "pkhard," is a set of instances with random
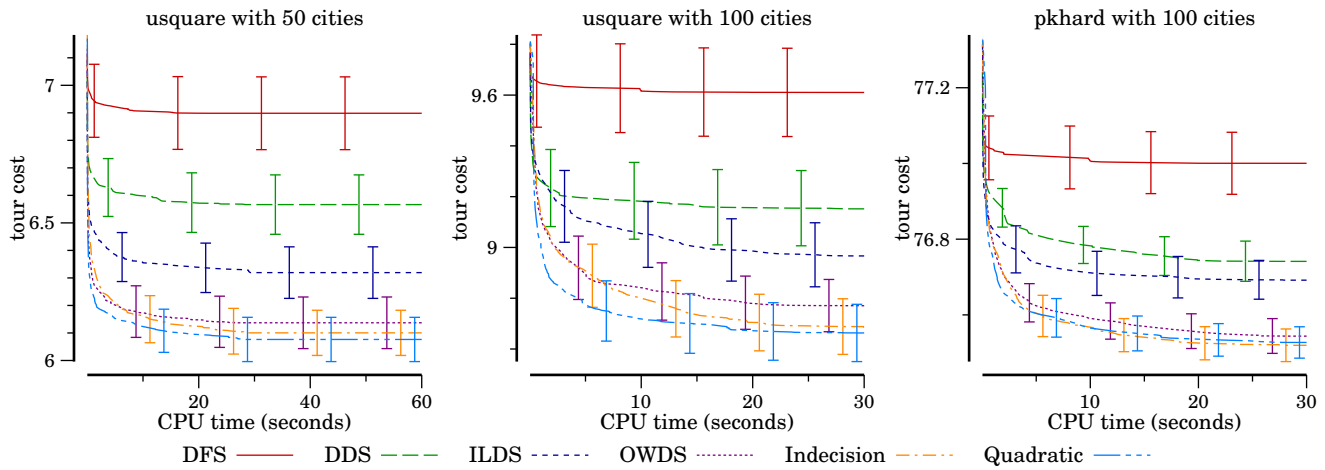
Figure 6: Traveling salesman problem: tour cost as a function of CPU time when using each search strategy.
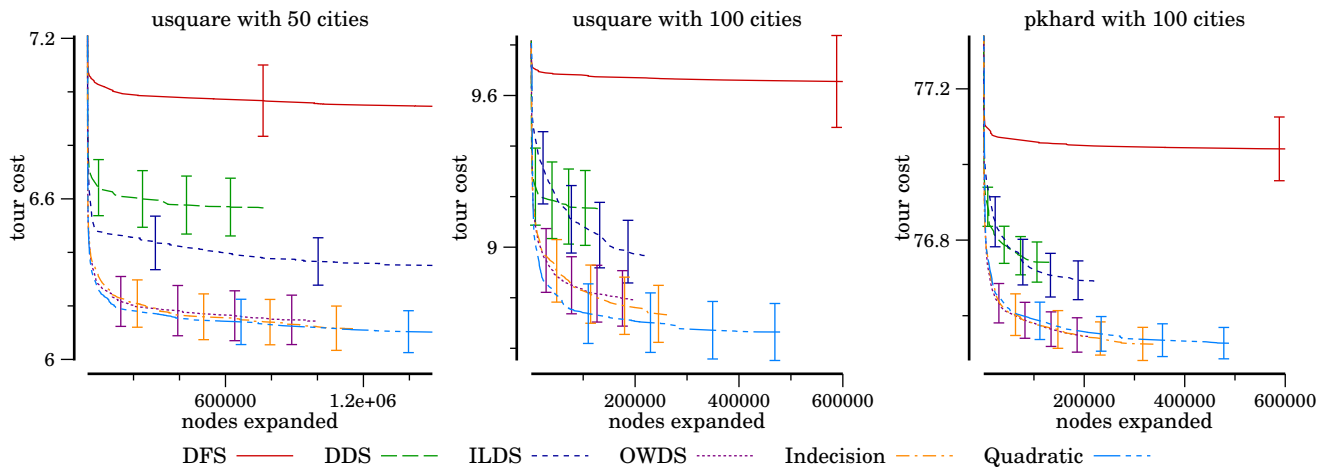

Figure 7: Traveling salesman problem: tour cost as a function of nodes visited when using each search strategy.

distances as recommended by Pearl and Kim (1982). Each set contained 40 instances.

Figure 6 gives the results of these experiments. The x axis in each plot is the CPU time in seconds and the y axis is the cost of the best tour found so far. The error bars indicate the 95% confidence intervals. The figure shows that DFS saw little improvement in solution quality, most likely because it was stuck in the left-most portion of the search tree. While DDS and ILDS tended to find better and better solutions as time passed, the quality of the solutions were not very competitive. OWDS appears to have given the next best performance. Finally, BLFS with the quadratic and indecision cost models gave the best performance on all three sets of instances. The quadratic model tended to perform better on the unit square instance set, showing more advantage over indecision as the problem difficulty increased.

Figure 7 shows the results of the same problems but in terms of nodes visited. The algorithms appear in the same ordering in terms nodes as they do in terms of CPU time. Note that DFS actually expands tens of millions of nodes in every problem but the x-axis has been clamped to bet-

ter show the performance of the other algorithms. There is no noticeable change in the solution cost that DFS achieves for node values greater than what the plots are clamped at. BLFS using quadratic or indecision cost models and OWDS consistently found better solutions per nodes expanded, with the quadratic model performing the best across all three problem sets. The additional information that these algorithms use in order to guide search seems to give them an advantage over the other search algorithms evaluated.

Discussion

While improving search order is one way to avoid the pitfalls of DFS, there are several other methods that have been developed to achieve similar results. For CSPs, Hulubei and OSullivan (2006) show that the `dom/wdeg` heuristic coupled with maintaining arc consistency (MAC) for constraint propagation is able to reduce the heavy tails experienced by DFS when solving certain problems BLFS can be seen as a complementary technique that avoids heavy-tail behavior by intelligent backtracking. Grimes and Wallace (2006) also achieve good performance by adding probing and random-

ized restarting to DFS to avoid becoming stuck in one portion of the search tree for too long and to improve the effectiveness of variable choice heuristics which learn from experience (e.g. `wdeg`).

For many tree search problems, CSPs especially, different techniques such as the variable choice heuristic and the level of constraint propagation used can greatly influence the structure of the resulting search tree. BLFS is purely a search algorithm and is independent of how the search tree is generated. Irregardless of those techniques, the tree that is generated needs be searched for a solution and BLFS aims to do this in the most intelligent way possible. To be effective however, BLFS requires useful heuristic features for its cost model. Some of the modern CSP solving techniques rendered useless the `promise` heuristic that we employ. To be able to take advantage of the most cutting edge techniques for solving CSPs, we modified a state-of-the-art CSP solver[3] to use indecision search. After evaluating a wide variety of benchmarks, we found that the `promise` value ordering heuristic provided no more useful guidance than just ordering the values lexicographically (and was much slower to compute). With no useful heuristic, indecision search performed poorly. Perhaps better results could be achieved if a better value choice heuristic were used.

BLFS is motivated by best-first search techniques used for shortest path problems. We have shown how to solve CSPs and COPs by adapting best-first search to work on bounded-depth trees and have provided two possible cost models to use during search: the indecision model and the quadratic model. While it was shown how these cost models could be use to improve search order in CSPs and COPs, it remains a topic of future work to see if they can also be used to improve shortest path algorithms.

Conclusion

BLFS is a complete tree search method that visits nodes in approximately best-first order while still retaining the linear memory complexity of depth-first search. We presented two types of cost models to use with this framework: indecision and the quadratic model. Indecision search takes advantage of the quantitative heuristic information available at a node to visit leaves in an approximately best-first ordering. Priority is given to places in the search tree where the heuristic is less decisive in its ranking. In the quadratic model, node costs are represented as quadratic functions learned from leaf costs encountered during search using on-line regression. We also presented a technique that uses an online model of the search tree to provide bound estimations to guide cost-bounded search.

An empirical evaluation showed that indecision search can outperform other algorithms in both CSPs and COPs. We also found that even when indecision search wasn't the best, it was more consistent across domains than other algorithms, leading to more robust performance. In COPs, BLFS using the quadratic model was consistently one of best performing algorithms. The ideas introduced in BLFS bring together shortest path heuristic search from artificial intelligence and heuristic tree search from constraint programming and operations research, helping to unify the field of combinatorial search.

References

Bedrax-Weiss, T. 1999. *Optimal Search Protocols*. Ph.D. Dissertation, University of Oregon.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *Proceedings of ECAI-04*, 146–150.

Geelen, P. A. 1992. Dual viewpoint heuristics for binary constraint satisfaction problems. In Neumann, B., ed., *Proceedings of ECAI-92*, 31–35.

Gomes, C. P., and Shmoys, D. 2002. Completing quasigroups or latin squares: A structured graph coloring problem. In *Computational Symposium on Graph Coloring and Generalizations*.

Grimes, D., and Wallace, R. 2006. Learning from failure in constraint satisfaction search.

Haralick, R. M., and Elliott, G. L. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* 14(3):263 – 313.

Harvey, W. D., and Ginsberg, M. L. 1995. Limited discrepancy search. In *Proceedings of IJCAI-95*, 607–613. Morgan Kaufmann.

Hulubei, T., and OSullivan, B. 2006. The impact of search heuristics on heavy-tailed behaviour. *Constraints* 11:159–178.

Karoui, W.; Huguet, M.-J.; Lopez, P.; and Naanaa, W. 2007. YIELDS: A yet improved limited discrepancy search for CSPs. In *Proceedings of CPAIOR-07*, 99–111.

Korf, R. E. 1985. Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of IJCAI-85*, 1034–1036.

Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

Korf, R. E. 1996. Improved limited discrepancy search. In *Proceedings of AAAI-96*, 286–291. MIT Press.

Mitchell, T. M. 1997. *Machine Learning*. New York: McGraw-Hill.

Pearl, J., and Kim, J. H. 1982. Studies in semi-admissible heuristics. *IEEE Trans PAMI* 4(4):391–399.

Ruml, W. 2002. Heuristic search in bounded-depth trees: Best-leaf-first search. In *Working Notes of the AAAI-02 Workshop on Probabilistic Approaches in Search*.

Sarkar, U.; Chakrabarti, P.; Ghose, S.; and Sarkar, S. D. 1991. Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence* 50:207–221.

Wallace, R. 1996. Analysis of heuristic methods for partial constraint satisfaction problems. In Freuder, E., ed., *Principles and Practice of Constraint Programming CP96*, volume 1118 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 482–496.

Walsh, T. 1997. Depth-bounded discrepancy search. In *Proceedings of IJCAI-97*.

---

[3]We used Mistral, winner of the 2009 CSP competition. It is written by Emmanuel Hebrard and available from `http://4c.ucc.ie/~ehebrard/Software.html`