Anytime Hybrid Best-First Search with Tree Decomposition for Weighted CSP

David Allouche, Simon de Givry, George Katsirelos,
Thomas Schiex, Matthias Zytnicki

MIAT, UR-875, INRA, F-31320 Castanet Tolosan, France
`firstname.lastname@toulouse.inra.fr`

Abstract. We propose Hybrid Best First Search (HBFS), a search strategy for optimization problems that combines Best First Search (BFS) and Depth First Search (DFS). Like BFS, HBFS provides an anytime global lower bound on the optimum, while also providing anytime upper bounds, like DFS. Hence, it provides feedback on the progress of search and solution quality in the form of an optimality gap. In addition, it exhibits highly dynamic behavior that allows it to perform on par with methods like limited discrepancy search and frequent restarting in terms of quickly finding good solutions.

We also use the lower bounds reported by HBFS in problems with small treewidth, by integrating it into Backtracking with Tree Decomposition (BTD). BTD-HBFS exploits the lower bounds reported by HBFS in individual clusters to improve the anytime behavior and global pruning lower bound of BTD.

In an extensive empirical evaluation on optimization problems from a variety of application domains, we show that both HBFS and BTD-HBFS improve both anytime and overall performance compared to their counterparts.

Keywords: combinatorial optimization, anytime algorithm, weighted constraint satisfaction problem, cost function networks, best-first search, tree decomposition

1 Introduction

Branch and Bound search is a fundamental tool in exact combinatorial optimization. For minimization, in order to prune the search tree, all variants of Branch and Bound rely on a local lower bound on the cost of the best solution below a given node.

Depth First Search (DFS) always develops a deepest unexplored node. When the gap between the local lower bound and a global upper bound on the cost of an optimal solution – usually provided by the best known solution – becomes empty, backtrack occurs. DFS is often used in Constraint Programming because it offers polyspace complexity, it takes advantage of the incrementality of local consistencies and it has a reasonably good anytime behavior that can be further enhanced by branching heuristics. This anytime behavior is however largely destroyed in DFS variants targeted at solving problems with a reasonable treewidth such as BTD [7] or AND/OR search [6].

Best-First Search (BFS) instead always develops the node with the lowest lower bound first. It offers a running global lower bound and has been proved to never develop more nodes than DFS for the same lower bound [22]. But it has a worst-case exponential space complexity and the optimal solution is always the only solution produced.

An ideal Branch and Bound algorithm would combine the best of all approaches. It would have a bearable space complexity, benefit from the incrementality of local consistencies and offer both updated global upper and lower bounds as the problem is solved. It would also not loose all its anytime qualities when used in the context of treewidth sensitive algorithms such as BTD.

With updated global lower and upper bounds, it becomes possible to compute a current global optimality gap. This gap can serve as a meaningful indicator of search progress, providing a direct feedback in terms of the criteria being optimized. This gap also becomes of prime importance in the context of tree-decomposition based Branch and Bound algorithms such as BTD [7] as global bounds for each cluster can typically be used to enhance pruning in other clusters.

In this paper, we introduce HBFS, an hybrid, easy to implement, anyspace Branch and Bound algorithm combining the qualities of DFS and BFS. The only limitation of HBFS is that it may require to compromise the anytime updating of the global lower bound for space. This can be achieved dynamically during search. HBFS can also be combined with a tree-decomposition to define the more complex BTD-HBFS, a BTD variant offering anytime solutions and updated global optimality gap.

On a set of more than 3,000 benchmark problems from various sources (MaxCSP, WCSP, Markov Random Fields, Partial Weighted MaxSAT) including resource allocation, bioinformatics, image processing and uncertain reasoning problems, we observe that HBFS improves DFS in term of efficiency, while being able to quickly provide good solutions – on par with LDS and Luby restarts – and a global running optimality gap. Similarly, HBFS is able to improve the efficiency and anytime capacities of BTD.

2   Background

Our presentation is restricted to binary problems for simplicity. Our implementation does not have such restriction. A binary Cost Function Network (CFN) is a triplet (X, D, W). $X = \{1, \ldots, n\}$ is a set of n variables. Each variable $i \in X$ has a finite domain $D_i \in D$ of values than can be assigned to it. The maximum domain size is d. W is a set of cost functions. A binary cost function $w_{ij} \in W$ is a function $w_{ij} : D_i \times D_j \mapsto [0, k]$ where k is a given maximum integer cost corresponding to a completely forbidden assignment (expressing hard constraints). If they do not exist, we add to W one unary cost function for every variable such that $w_i : D_i \mapsto [0, k]$ and a zero arity constraint $w_\varnothing$ (a constant cost payed by any assignment, defining a lower bound on the optimum). All these additional cost functions will have initial value 0, leaving the semantics of the problem unchanged.

The Weighted Constraint Satisfaction Problem (WCSP) is to find a minimum cost complete assignment: $\min_{(a_1, \ldots, a_n) \in \prod_i D_i} \{w_\varnothing + \sum_{i=1}^n w_i(a_i) + \sum_{w_{ij} \in W} w_{ij}(a_i, a_j)\}$, an optimization problem with an associated NP-complete decision problem.

The WCSP can be solved exactly using Branch and Bound maintaining some lower bound: at each node ν of a tree, we use the local non naive lower bound $\nu.lb = w_\varnothing$ provided by a given soft arc consistency [5]. Each node corresponds to a sequence of decisions $\nu.\delta$. The root node has an empty decision sequence. When a node is explored, an unassigned variable is chosen and a branching decision to either assign the variable

to a chosen value (left branch, positive decision) or remove the value from the domain (right branch, negative decision) is taken. The number of decisions taken to reach a given node ν is the depth of the node, $\nu.depth$. A node of the search tree that corresponds to a complete assignment is called a leaf. At this point, $\nu.lb$ is assumed to be equal to the node cost (which is guaranteed by all soft arc consistencies).

The graph $G = (X, E)$ of a CFN has one vertex for each variable and one edge (i, j) for every binary cost function $w_{ij} \in W$. A tree decomposition of this graph is defined by a tree (C, T). The set of nodes of the tree is $C = \{C_1, \ldots, C_m\}$ where C_e is a set of variables ($C_e \subseteq X$) called a cluster. T is a set of edges connecting clusters and forming a tree (a connected acyclic graph). The set of clusters C must cover all the variables ($\bigcup_{C_e \in C} C_e = X$) and all the cost functions ($\forall \{i, j\} \in E, \exists C_e \in C \ s.t. \ i, j \in C_e$). Furthermore, if a variable i appears in two clusters C_e and C_g, i must also appear in all the clusters C_f on the unique path from C_e to C_g in (C, T). If the cardinality of the largest cluster in a tree decomposition is $\omega + 1$ then the *width* of the decomposition is ω. The *treewidth* of a graph is the minimum width among all its decompositions [24].
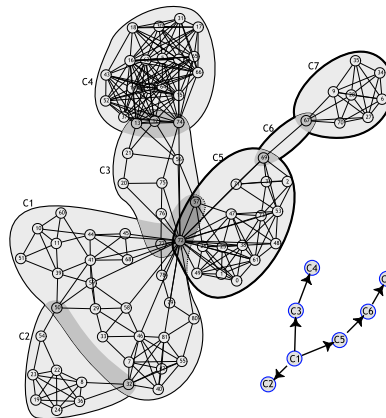
Fig. 1. A tree-decomposition of the CELAR06 radio frequency assignment problem, rooted in C_1 with subproblem P_5 highlighted.

3   Hybrid Best first search

Classical BFS explores the search tree by keeping a list *open* of open nodes representing unexplored subproblems. Initially, this list is reduced to the root node at depth 0. Iteratively, a best node is explored: the node is removed and replaced by its two left and right children with updated decisions, lower bound and depth. In this paper we always choose as best node a node with the smallest $\nu.lb$, breaking ties by selecting a node with maximum $\nu.depth$. The first leaf of the tree explored is then guaranteed to be an optimal solution [14, 22]. The list *open* may reach a size in $O(d^n)$ and, if incrementality in the lower bound computation is sought, each node should hold the minimum data-structures required for soft arc consistency enforcing (in $O(ed)$ per node).

The pseudocode for Hybrid BFS is described as Algorithm 1. HBFS starts with the empty root node in the list of *open* nodes. It then iteratively picks a best node ν from the open list as above, replays all the decisions in $\nu.\delta$ leading to an assignment A_ν, while maintaining consistency. It then performs a depth first search probe starting from that node for a limited number Z of backtracks. The DFS algorithm is a standard DFS algorithm except for the fact that, when the bound on the number of backtracks is reached, it places all the nodes corresponding to open right branches of its current search state in the *open* list (see Figure 2).
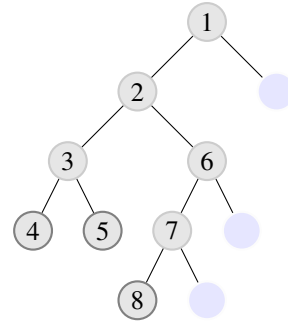
At the price of increased memory usage, this hybrid maintains the advantages of depth-first search. Since it spends a significant amount of its time in a DFS subroutine, it can exploit the incrementality of arc consistency filtering during DFS search without any extra space cost: nodes in the *open* list will just contain decisions δ and lower bound lb, avoiding the extra $O(ed)$ space required for incrementality during BFS. However, each time a node is picked up in *open*, the set of $\nu.depth$ decisions must be "replayed" and local consistency reinforced from the root node state, leading to redundant propagation. This cost can be mitigated to some degree by merging all decisions into one. Hence, a single fixpoint has to be computed rather than $\nu.depth$. Additionally, the cost can be further reduced using other techniques employed by copying solvers [26]. Regardless of these mitigation techniques, some redundancy is unavoidable, hence the number of backtracks performed at each DFS probe should be large enough to avoid excessive redundancy.

Fig. 2. A tree that is partially explored by DFS with backtrack limit = 3. Nodes with a bold border are leaves, nodes with no border are placed in the open list after the backtrack bound is exceeded. Nodes are numbered in the order they are visited.

Second, as it is allowed to perform Z backtracks in a depth-first manner before it picks a new node, it may find new and better incumbent solutions, thus it is anytime. The number of backtracks of each DFS probe should be sufficiently small to offer quick diversification: by exploring a new best node, we are offered the opportunity to reconsider early choices, similarly to what LDS [8] and Luby randomized restarts [18] may offer. Additionally, with early upper bounds, we can also prune the open node list and remove all nodes such that $\nu.lb \geq ub$.

To balance the conflicting objectives of reducing repeated propagation and diversification, we dynamically adjust the amount of backtracks Z that can be performed during one DFS probe by trying to keep the observed rate of redundantly propagated decisions between reasonable bounds (α and β). In all the algorithms here, we assume that the number of nodes ($Nodes$) and backtracks ($Backtracks$) are implicitly maintained during search.

This hybrid does not preserve the polyspace complexity of DFS. However, it can easily be made *anyspace*. If memory is exhausted (or a memory upper bound is reached, with the same effect), the algorithm can switch from bounded DFS to complete DFS. This means that for every node it picks from the open list, it explores the entire subtree under that node. Hence, it will not generate any new open nodes. It can continue in this mode of operation until memory pressure is relieved.

Finally, this method computes stronger global lower bounds than DFS, as the cost of a best node in the open list defines a global lower bound, as in BFS. DFS instead cannot improve on the global lower bound computed at the root until it finally visits the first right branch. In the context of a single instance this is only important in the sense that it provides a better estimation of the optimality gap. However, we will see that this can improve performance in decomposition-based methods.

Function HBFS(clb,cub) : pair(integer,integer)

 $open := \nu(\delta = \varnothing, lb = clb)$;
 while ($open \neq \varnothing$ **and** $clb < cub$) **do**
  $\nu :=$pop($open$) /* *Choose a node with minimum lower bound and maximum depth* */;
  Restore state $\nu.\delta$, leading to assignment A_ν, maintaining local consistency ;
  $NodesRecompute := NodesRecompute + \nu.depth$;
  $cub :=$DFS(A_ν,cub,Z)/* *puts all right open branches in open* */ ;
  $clb := \max(clb, lb(open))$;
  if ($NodesRecompute > 0$) **then**
   if ($NodesRecompute/Nodes > \beta$ **and** $Z \leq N$) **then** $Z := 2 \times Z$;
   else if ($NodesRecompute/Nodes < \alpha$ **and** $Z \geq 2$) **then** $Z := Z/2$;
 return (clb, cub);

  Algorithm 1: Hybrid Best First Search. Initial call: HBFS($w_\varnothing$,k) with $Z = 1$.

3.1   Related work

Alternate search space exploration schemes have been proposed in the field of heuristic search, as variations of A* search. These schemes can be applied to discrete optimization, yielding other variants of best-first search. However, depth first search is not effective or even feasible in domains where A* search is used: for example, it is possible in planning to have exponentially long sequences of actions when short plans exist. Hence, methods like BRFSL(k) [27] can only do *bounded-depth* DFS probes. Also, in contrast to HBFS, they do not insert the open nodes of the DFS probes into the open list of BFS. Other methods like Weighted best first search [23], ARA* [16] and ANA* [2] weigh future assignments more heavily in order to bias the search towards solutions. We do not need to modify the branching heuristic in any way in HBFS.

 Stratification [1], which solves a weighted MaxSAT instance by iteratively considering larger subsets of its clauses, starting with those that have the highest weight, provides similar benefits to HBFS, as provides solutions quickly and produces lower bounds. This techniques, however, can be viewed as a wrapper over an optimization method and is therefore orthogonal to HBFS.

 Alternate heuristics for choosing the next node to explore may yield different algorithms. When we can identify a preferred value to assign at each choice point, the *discrepancy* of a node ν is the number of right branches in the path from the root to ν. If we always open the node with the smallest discrepancy, set $Z = 1$ and disable the adaptive heuristic, HBFS is identical to Limited Discrepancy Search (LDS)[1] [8].

 In ILP, a closely related approach is so-called BFS with *diving* heuristics [3]. Such heuristics perform a single depth first probe trying to find a feasible solution. Although the idea is quite close to that of HBFS, it is typically restricted to a single branch, the open nodes it leaves are not added to the open node file and is treated separately from the rest of the search process. This is in part motivated by the fact that DFS is considered impractical in ILP [17] and by the fact that the lower bounding method (LP) used is not as lightweight as those used in WCSP.

---

[1] In WCSP optimization, we always have a non naive value heuristic that selects a value (i, a) with minimum unary marginal cost $w_i(a)$ or better, the EAC support [13].

4 Hybrid Best First Search and Tree Decompositions

When the graph of a CFN has bounded treewidth, the $O(d^n)$ worst-case complexity of DFS can be improved using a tree decomposition of the CFN graph. We can trivially observe that the tree decomposition can be rooted by selecting a root cluster denoted C_1. The separator of a non root cluster C_e is $C_e \cap pa(C_e)$, where $pa(C_e)$ is the parent of C_e in T. Local consistency can be enforced on the problem and provide a cluster-localized lower-bound $w_\varnothing^e$ for each cluster C_e. The sum of these cluster-wise lower bounds is a lower bound for the complete problem. Beyond this trivial observation, Terrioux and Jégou [28] and de Givry et al. [7] have extended BTD [9] (which we call BTD-DFS here for clarity) from pure satisfaction problems to the case of optimization (WCSP), in a way similar to AND/OR search [19]. Next, we briefly describe BTD-DFS, as given by de Givry et al, as we base our own algorithm on this.

In BTD-DFS, by always assigning the variables of a cluster before the variables of its descendant clusters, it is possible to exploit the fact that assigning a cluster C_e separates all its child clusters $children(C_e)$. Each child cluster C_f is the root of a subproblem P_f defined by the subtree rooted in C_f which becomes independent of others. So, each subproblem P_f conditioned by the current assignment A_f of its separator, can be independently and recursively solved to optimality. If we memoize the optimum cost of every solved conditioned subproblem $P_e|A_e$ in a cache, then $P_e|A_e$ will never be solved again and an overall $O(nd^{\omega+1})$ time complexity can be guaranteed.

Although this simple strategy offers an attractive worst case theoretical bound, it may behave poorly in practice. Indeed, each conditioned subproblem $P_e|A_e$ is always solved from scratch to optimality. This ignores additional information that can be extracted from already solved clusters. Imagine C_e has been assigned and that we have an upper bound ub (a solution) for the problem $P_e|A_e$. Assume that C_e has two children C_f and $C_{f'}$ and that we have solved the first subproblem $P_f|A_f$ to optimality. By subtracting the lower bound $w_\varnothing^e$ and the optimum of $P_f|A_f$ from ub, we obtain the maximum cost that a solution of $P_{f'}|A_{f'}$ may have in order to be able to improve over ub. Instead of solving it from scratch, we can solve $P_{f'}|A_{f'}$ with this initial upper bound and either find an optimal solution – which can be cached – or fail. If we fail, we have proved a global lower bound on the cost of an optimal solution of $P_{f'}|A_{f'}$. This lower bound can be cached and prevent repeated search if $P_{f'}|A_{f'}$ is revisited with the same or a lower initial upper bound. Otherwise, the problem will be solved again and again either solved to optimality or fail and provide an improved global lower bound. This has been shown to improve search in practice while offering a theoretical bound on time complexity in $O(kn.d^{\omega+1})$ (each time a subproblem $P_f|A_f$ is solved again, the global lower bound increases at least by 1).

In practice, we therefore cache two values, $LB_{P_e|A_e}$ and $UB_{P_e|A_e}$, for every visited assignment A_e of the separator of every cluster C_e. We always assume caching is done implicitly: $LB_{P_e|A_e}$ is updated every time a stronger lower bound is proved for $P_e|A_e$ and $UB_{P_e|A_e}$ when an updated upper bound is found. When an optimal solution is found and proved to be optimal, we will therefore have $LB_{P_e|A_e} = UB_{P_e|A_e}$. Thanks to these cached bounds and to the cluster-wise local lower bounds $w_\varnothing^e$, an improved local lower bound $lb(P_e|A_e)$ for the subproblem $P_e|A_e$ can be computed by recursively summing the maximum of the cached and local bound (see [7]).

We show pseudocode for the resulting algorithm combining BTD and DFS in Algorithm 2. Grayed lines in this code are not needed for the DFS variant and should be ignored. The algorithm is called on root cluster C_1, with an assignment $A_1 = \varnothing$, a set of unassigned variables $V = C_1$ and initial lower and upper bound clb and cub set respectively to $lb(P_1|\varnothing)$ and k (the maximum cost). The last argument, $RecCall$ is a functional argument that denotes which function will be used to recurse inside BTD-DFS. Here, $RecCall$ will be initially equal to BTD-DFS itself. The algorithm always returns two identical values equal to the current upper bound.[2] Caches are initially empty and return naive values $LB_{P_e/A} = 0$ and $UB_{P_e/A} = k$ for all clusters and separator assignments.

Function BTD-DFS(A,C_e,V,clb,cub,$RecCall$) : pair(integer,integer)

  if ($V \neq \varnothing$) **then**
    $i :=$pop(V) /* *Choose an unassigned variable in C_e* */ ;
    $a :=$pop(D_i) /* *Choose a value* */ ;
    Assign a to i, maintaining local consistency on subproblem $lb(P_e|A \cup \{(i = a)\})$;
    $clb' := \max(clb, lb(P_e|A \cup \{(i = a)\}))$;
    if ($clb' < cub$) **then**
      $(cub, cub) :=$ BTD-DFS($A \cup \{(i = a)\}, C_e, V - \{i\}, clb', cub, RecCall$);
    $C_e.backtracks := C_e.backtracks + 1$;
    if ($\max(clb, lb(P_e|A)) < cub$) **then**
      Remove a from i, maintaining local consistency on subproblem $lb(P_e|A \cup \{(i \neq a)\})$;
      $clb' := \max(clb, lb(P_e|A \cup \{(i \neq a)\}))$;
      if ($clb' < cub$) **then**
        if ($C_e.backtracks < C_e.limit$ **and** $Backtracks < P_e.limit$) **then**
          $(cub, cub) :=$ BTD-DFS($A \cup \{(i \neq a)\}, C_e, V, clb', cub, RecCall$);
        else /* *Stop depth-first search* */
          Push current search node in open list of $P_e|A$ at position clb' ;
  else
    $S := Children(C_e)$;
    /* *Solve all clusters with non-zero optimality gap and unchanged lb or ub* */ ;
    while ($S \neq \varnothing$ **and** $lb(P_e|A) < cub$) **do**
      $C_f :=$pop(S) /* *Choose a child cluster* */ ;
      if ($LB_{P_f|A} < UB_{P_f|A}$) **then**
        $cub' := \min(UB_{P_f|A}, cub - [lb(P_e|A) - lb(P_f|A_f)])$;
        $(clb'', cub'') := RecCall (A, C_f, C_f, lb(P_f|A_f), cub', RecCall)$;
        Update $LB_{P_f|A}$ and $UB_{P_f|A}$ using clb'' and cub'';
    $cub := \min(cub, w_\varnothing^e + \sum_{C_f \in Children(C_e)} UB_{P_f|A})$;
    if $\max(clb, lb(P_e|A)) < cub$ **then**
      Push current search node in open list of $P_e|A$ at position $\max(clb, lb(P_e|A))$;
      $C_e.limit := C_e.backtracks$ /* *Stop depth-first search* */ ;
  return (cub, cub)

Algorithm 2: BTD using depth-first search

---

[2] This is clearly redundant for BTD-DFS, but allows a more uniform presentation with BTD-HBFS

4.1   Using HBFS in BTD

BTD-DFS has two main disadvantages: first, it has very poor anytime behavior, as it cannot produce a solution in a decomposition with k leaves until $k-1$ leaf clusters have been completely solved. This affects the strength of pruning, as values are only pruned if the current lower bound added to the marginal cost of the value exceeds the upper bound. Second, because child clusters are examined in order, only the lower bounds of siblings earlier than C_f in that order can contribute to pruning in C_f. For example, consider a cluster C_e with 3 child clusters $C_{f_1}, C_{f_2}, C_{f_3}$. Assume that $ub = 31$ and under an assignment A, $w_{\varnothing}^e$ has known cost 10 while $P_{f_1}|A_{f_1}$, $P_{f_2}|A_{f_2}$ and $P_{f_3}|A_{f_3}$ all have optimal cost 10, and $lb(P_{f_1}|A_{f_1}) = lb(P_{f_2}|A_{f_2}) = lb(P_{f_3}|A_{f_3}) = 0$. Clearly the subproblem under C_e cannot improve on the upper bound, but when we solve C_{f_1} and C_{f_2}, BTD-DFS does not reduce the effective upper bound at all. However, it may be relatively easy to prove a lower bound of 7 for each of the child clusters. If we had this information, we could backtrack.

HBFS has the ability to quickly provide good lower and upper bounds and interrupts itself as soon as the limit number of backtracks is reached. Using HBFS instead of DFS in BTD should allow to quickly probe each subproblem to obtain intermediate upper and lower bounds for each of them. The upper bounds can be used to quickly build a global solution, giving anytime behavior to BTD. The lower bounds of all subproblems can be used to improve pruning in all other clusters.

The pseudocode of BTD-HBFS is described as Algorithm 3. It takes the same arguments as BTD-DFS but ignores the last one (used only to pass information from BTD-HBFS to BTD-DFS). BTD-HBFS relies on BTD-DFS pseudocode, assuming that all grayed lines of BTD-DFS are active. These reactivated lines in Algorithm 2 impose per-cluster and per-subproblem backtrack limits. Every cluster C_e has a counter $C_e.backtracks$ for number of backtracks performed inside the cluster and an associated limit $C_e.limit$. Every subproblem P_e has a limit $P_e.limit$ on the number of backtracks N performed inside the subproblem. Initially, $C_e.limit = P_e.limit = \infty$ for all $C_e \in C$.

Every subproblem $P_e|A_e$ has its own list of open nodes $P_e|A_e.open$ for each upper bound which it is given. The value of the upper bound participates in the definition of the actual search space that needs to be explored. If the same subproblem is revisited later with a lower upper bound, then the search space shrinks and we can just copy the open list associated with the higher bound and prune all nodes ν such that $\nu.lb \geq ub$. But if the upper bound is more relaxed than any previous upper bound then we need to create a new open list starting with the root node.

Finally, the loop in line 1 is interrupted as soon as the optimality gap reduces or the number of backtracks reaches the subproblem limit, making the search more dynamic. If subproblems quickly update their bound, the remaining backtracks can be used in a higher cluster. However, the subproblem under each child cluster is guaranteed to get at least Z backtracks. The result is that we spend most of the time in leaf clusters. When one cluster exhausts its budget, the search quickly returns to the root cluster.

Example 1. Consider the example in figure 3. We have a CFN with the tree decomposition given in the box labeled (C, T) and the search in each cluster is shown in a

box labeled by that cluster's name. Let $N = 2$ and $Z = 1$ in this example. The search visits nodes as they are numbered in the figure. When it reaches node 4, cluster C_1 is completely instantiated and hence it descends into C_2 and after node 7 it descends into C_4. After node 10, we have performed a backtrack in this cluster, and since $Z = 1$ we end this DFS probe and return control to BTD-HBFS. The limit on the number of backtracks in P_4 is still not exceeded, so we choose a new node from the open list, node 11, a conflict. Again we return control to BTD-HBFS and, having exceeded the backtrack limit on P_4, exit this cluster, but with an improved lower bound. Since C_4 exceeded its backtrack limit before improving its lower bound, no more search is allowed in parent clusters. The search is allowed, however, to visit sibling clusters, hence it explores C_5 (nodes 12–14), which it exits with an improved upper bound before exceeding its backtrack limit, and C_3 (nodes 15–18). Once it returns to node 7 after cluster C_5, that node is not closed, because one of the child clusters is not closed. It is instead put back on the open list. Similarly node 4 is put back on the open list of C_1. At that point, best first search picks another node from the open list of C_1, node 19, and continues from there. □

Function BTD-HBFS(A,C_e,V,clb,cub,$\_$) : pair(integer,integer)

> $open :=$ open list of $P_e|A(cub)$;
> **if** $(open = \varnothing)$ **then**
>> **if** exists minimum cub' s.t. $cub' > cub$ **and** $open(P_e|A(cub')) \neq \varnothing$ **then**
>>> $\mid$ $open = \{\nu \in open(P_e|A(cub')) \mid \nu.lb < cub\}$
>> **else**
>>> $\mid$ $open = \{\varnothing\}$ /* *Contains only the root node at position clb* */
> $P_e.limit := Backtracks + N$ /* *Set a global backtrack limit for the subproblem* */ ;
> $clb' := \max(clb, lb(open))$;
> $cub' := cub$;
> **while** $(open \neq \varnothing$ **and** $clb' < cub'$ **and** $(C_e = C_1$ **or** $(clb' = clb$ **and** $cub' = cub$ **and** $Backtracks < P_e.limit)))$ **do**
>> $\nu := \mathsf{pop}(open)$ /* *Choose a node with minimum lower bound and maximum depth* */ ;
>> Restore state $\nu.\delta$, leading to assignment A_ν, maintaining local consistency ;
>> $NodesRecompute := NodesRecompute + \nu.depth$;
>> $C_e.limit := C_e.backtracks + Z$ /* *Set a depth-first search backtrack limit* */ ;
>> $(cub', cub') := \mathsf{BTD\text{-}DFS}(A_\nu,C_e,V_\nu,\max(clb',lb(\nu),lb(P_e|A_\nu)),cub',\mathsf{BTD\text{-}HBFS})$;
>> $clb' := \max(clb', lb(open))$;
>> **if** $(NodesRecompute > 0)$ **then**
>>> **if** $(NodesRecompute/Nodes > \beta$ **and** $Z \leq N)$ **then** $Z := 2 \times Z$;
>>> **else if** $(NodesRecompute/Nodes < \alpha$ **and** $Z \geq 2)$ **then** $Z := Z/2$;
> **return** (clb', cub') /* *invariant clb' ≥ clb and cub' ≤ cub* */ ;

Algorithm 3: Hybrid Best First Search with Tree Decomposition.

BTD-HBFS addresses both the issues of BTD that we identified above. First, it is anytime, because as soon as $UB_{P_e|A_e} < k$ for all subproblems, we can combine the assignments that gave these upper bounds to produce a global solution. Second, it constantly updates lower bounds for all active subproblems, so the search effort in each subproblem immediately exploits all other lower bounds.

Fig. 3. An example of a run of BTD-HBFS. White nodes are open, nodes with a black border are conflicts (square) or solutions (circle). Grey nodes with a white border are explored but put back in the open list. Grey nodes with no border are closed.

Like HBFS, BTD-HBFS can be made *anyspace*, i.e., its memory usage can be limited to any amount beyond what is needed for BTD-DFS, including zero. The cache of bounds can also be limited, at the expense of additional subproblem recomputations, leading to worst-case complexity exponential in the tree decomposition height.

Theorem 1. *Given a CFN P with treewidth ω,* BTD-HBFS *computes the optimum in time $O(knd^{\omega})$ and space $O(knd^{2\omega})$.*

Proof (Sketch). For correctness, observe that BTD-DFS solves independent subproblems separately using DFS, hence using HBFS or any other solution method does not affect correctness. Each leaf node in an internal cluster is closed only when all child clusters are solved, hence all bounds for each subproblem and open node list are correct. Finally, exploration at the root cluster continues until the optimality gap is closed.

Complexity stems from the complexity of BTD and the additional overhead of storing open lists for each separator assignment and upper bound.   □

We implemented a simpler version of this algorithm with better space complexity: each time BTD-HBFS is called on $P_e|A$ with a higher upper bound than previously stored, we wipe the open list and replace it with the root node of C_e. This removes theoretical guarantees on the performance of the algorithm, but does not hurt practical performance, as we will see.

4.2 Related work

AND/OR Branch and Bound search has already been combined with BFS [20]. The resulting AOBF algorithm, has good worst-case time complexity similar to BTD-DFS, but otherwise has the space-intensive non anytime behavior of BFS.

The poor anytime ability of BTD has been addressed by breadth-rotating AND/OR search (BRAO) [21]. BRAO *interleaves* DFS on all components, so it can combine the incumbents of all components to produce a global solution. However, as it performs DFS on each component, it does not produce better lower bounds.

OR-decomposition [12] is an anytime method that exploits lower bounds produced by other clusters by performing DFS in which it interleaves variable choices from all components, and uses caching to achieve the same effect as BTD. However, the global lower it computes depends on the partial assignments of all components. Thus it may revisit the same partial assignment of one component many times. This may also inhibit its anytime behavior, as a high cost partial assignment in one component will prevent other components from reaching good solutions. Moreover, the local lower bound for each component is only updated by visiting the right branch at its root.

Russian Doll Search [25], uses DFS to solve each cluster of a rooted tree decomposition in topological order. This method is not anytime, as it cannot produce a solution until it starts solving the root cluster. Moreover, it computes lower bounds that are independent of the separator assignment, hence can be lower than their true value.

5 Experimental Results

We used benchmark instances including stochastic graphical models from the *UAI evaluation* in 2008 and 2010, the *Probabilistic Inference Challenge 2011*, the *Weighted Partial Max-SAT Evaluation 2013*, the MiniZinc Challenge 2012 and 2013, Computer Vision and Pattern Recognition problems from OpenGM2 [3] and additional instances from the CostFunctionLib [4]. This is a total of more than 3,000 instances that can be encoded as Cost Function Networks, available at `http://genoweb.toulouse.inra.fr/~degivry/evalgm`, with domain sizes that range from $d = 2$ to 503, $n = 2$ to $903,884$ variables, and $e = 3$ to $2,912,880$ cost functions of arity from $r = 2$

---

[3] `http://hci.iwr.uni-heidelberg.de/opengm2/`
[4] `https://mulcyber.toulouse.inra.fr/projects/costfunctionlib`

to 580. We used `toulbar2` version 0.9.8.0-dev[5] on a cluster of 48-core Opteron 6176 nodes at 2.3 GHz with 378 GB RAM, with a limit of 24 simultaneous jobs per node.

In all cases, the local lower bound is provided by maintaining EDAC [13]. The variable ordering includes both weighted-degree [4] and last-conflict [15] heuristics. The value ordering is to select the EAC support value first [13]. All executions used a min-fill variable ordering for DAC preprocessing. For HBFS, we set the node recomputation parameters to $[\alpha, \beta] = [5\%, 10\%]$ and the backtrack limit N to $10,000$.

The methods based on BTD use a different value ordering heuristic: if a solution is known for a cluster, it keeps the same value if possible and if not uses EAC support values as the previous methods. A min-fill ordering is used for building a tree decomposition. Children of a cluster are statically sorted by minimum separator size first and smallest number of subproblem variables next.

Our aim is to determine whether HBFS is able to improve over DFS both in terms of number of problems solved (including the optimality proof) and in its anytime behavior. Similarly, we compare BTD-HBFS to BTD-DFS. We include in our comparison two methods that are known to significantly improve the upper bound anytime behavior of DFS: Limited Discrepancy Search [8] and DFS with Luby restarts [18].

We also include results from BRAO [21] using the `daoopt` solver[6] with static mini-bucket lower bounds of different strength (*i-bound* set to 15 and 35) and without local search nor iterative min-fill preprocessing. `daoopt` is restricted to cost functions expressed by complete tables, hence we exclude most MaxSAT families (except MIPLib and MaxClique) in tests where we use it.

5.1   Proving optimality

We show in figure 4 a cactus plot comparing all the algorithms that do not use tree decompositions, but also include BTD-HBFS^{r_k} for reference (see below). We see that HBFS is the best performing decomposition-unaware algorithm. It outperforms DFS and DFS with Luby restarts significantly, and slightly outperforms LDS.

Although our benchmark set includes very large instances, and our HBFS implementation does not include automatic control of space usage, no instance required more than 32 GB. The median memory usage was 36.8 MB for DFS and 38.2 MB for hybrid BFS. The worst-case largest ratio between HBFS and DFS was $\frac{1,425.1MB}{29.3MB} = 48.7$ on MRF Grid instance `grid20x20.f15` (unsolved in one hour by both methods).

In figure 5, we compare algorithms exploiting a tree decomposition (BTD-like) or a pseudo-tree (BRAO).We see that BTD-HBFS slightly outperforms BTD-DFS, both outperforming BRAO. However, many of these instances have large treewidth and BTD-like methods are not ideal for these. Even for instances with small treewidth, the decomposition is often a deep tree in which each cluster shares all but one variables with its parent. In these cases, following the tree decomposition imposes a static variable ordering on BTD, while HBFS degrades to DFS.Finding good tree decompositions is not straightforward [10, 11]. A simple way to improve one is to merge clusters until no
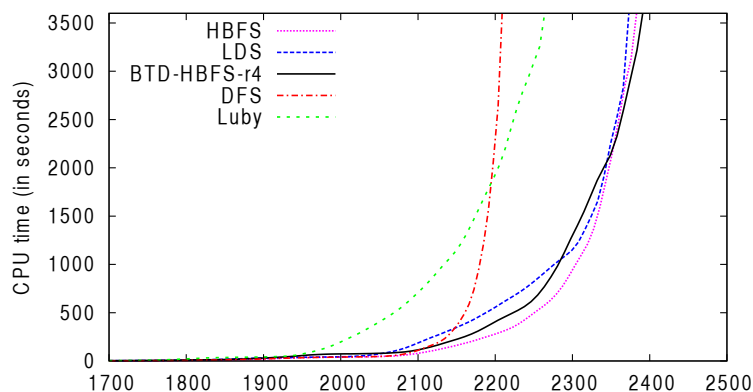

Fig. 4. Number of solved instances within a given time. Methods in the legend are sorted at time=20min.

separator has size greater than k, even if this increases width. We call the algorithms that apply this BTD-DFS^{r_k} and BTD-HBFS^{r_k}. Figure 5 includes results for BTD-DFS^{r_4} and BTD-HBFS^{r_4}. BTD-DFS^{r_4} is significantly better than BTD-DFS and BTD-HBFS^{r_4} outperforms BTD-DFS^{r_4} by an even greater margin. BTD-HBFS^{r_4} is also the overall best performer as shown in figure 4.
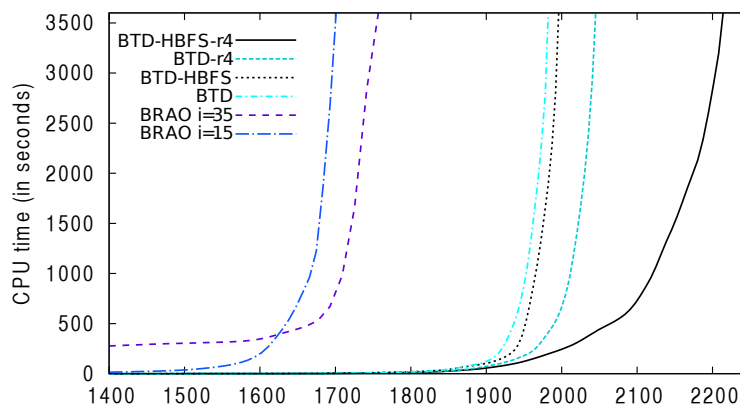

Fig. 5. Number of solved instances as time passes on a restricted benchmark set (without MaxSAT). Methods in the legend are sorted at time=20min.

5.2   Anytime behavior

To analyze the algorithms' anytime behavior, we first show in figure 6 the evolution of the lower and upper bounds for two instances: the SPOT5 404 (left) and the RLFAP CELAR06 instances (right). We solve both instances using DFS, LDS, DFS with Luby restarts, HBFS, BTD-DFS and BTD-HBFS. In both instances, we see that HBFS and BTD-HBFS improve significantly on the upper bound anytime ability of DFS and BTD-DFS, respectively. Moreover, the lower bound that they report increases quickly in the

beginning and keeps increasing with time. For all other algorithms, the lower bound increases by small amounts and infrequently, when the left branch of the root node is closed. The HBFS variants are as fast as the base algorithms in proving optimality.
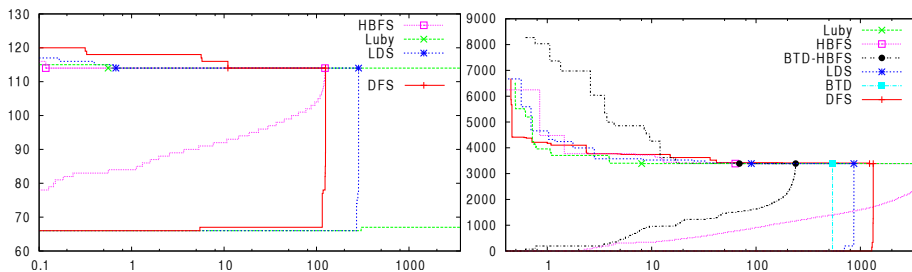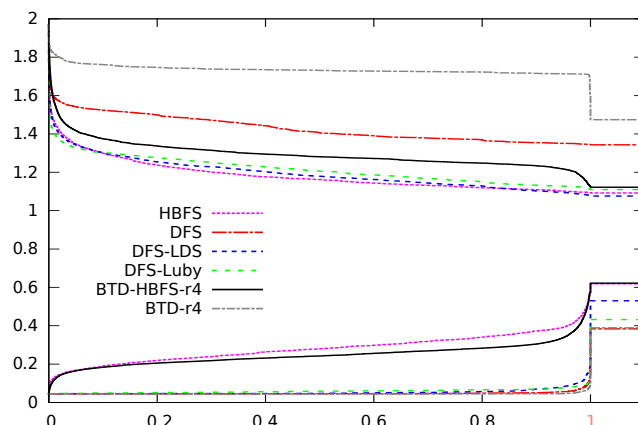

Fig. 6. Evolution of the lower and upper bounds (Y axis, in cost units) as time (X axis, in seconds) passes for HBFS, Luby restart, LDS, and DFS on SPOT5 404 instance (left) and for the RLFAP CELAR06 instance (right). Methods are sorted in increasing time to find the optimum. For each curve, the first point represents the time where the optimum is found and the second point the time (if any) of proof of optimality.

In figure 7, we summarize the evolution of lower and upper bounds for each algorithm over all instances that required more than 5 sec to be solved by DFS. Specifically, for each instance I we normalize all costs as follows: the initial lower bound produced by EDAC (which is common to all algorithms) is 0; the best – but potentially suboptimal – solution found by any algorithm is 1; the worst solution is 2. This normalization is invariant to translation and scaling. Additionally, we normalize time from 0 to 1 for each pair of algorithm A and instance I, so that preprocessing ends at time 0 and each run finishes at time 1. This time normalization is different for different instances and for different algorithms on the same instance. A point $\langle x, y \rangle$ on the lower bound line for algorithm A in figure 7 means that after normalized runtime x, algorithm A has proved on average over all instances a normalized lower bound of y and similarly for the upper bound. We show both the upper and lower bound curves for all algorithms evaluated here. In order for the last point of each curve to be visible, we extend all curves horizontally after 1.0.

This figure mostly ignores absolute performance in order to illustrate the evolution of upper and lower bounds with each algorithm, hence cannot be interpreted without the additional information provided by the cactus plots in figures 4 and 5. It confirms that HBFS improves on DFS in terms of both upper and lower bound anytime behavior and similarly for BTD-HBFS^{r4} over BTD-DFS^{r4}, with the latter being especially dramatic. The two HBFS variants are, as expected, significantly better than all other algorithms in terms of the lower bounds they produce. HBFS produces solutions of the same quality as the most dynamic variants, LDS and DFS-Luby, while BTD-HBFS^{r4} is slightly worse than this group but significantly better than BTD-DFS^{r4}.

Despite the fact that time to solve an instance is normalized away in figure 7, it does give some information that is absent from the cactus plots and that is the average normalized lower and upper bounds at time 1. Figure 7 tells us that LDS finds the

Fig. 7. Average evolution of normalized upper and lower bounds for each algorithm.

best solution most often, as its upper bound curve is the lowest at time 1. It is followed closely by the HBFS variants and DFS-Luby, while DFS and BTD-DFS^{r_4} are significantly worse. On the other hand, while LDS looks only slightly worse than the HBFS variants in the cactus plot, HBFS and BTD-HBFS^{r_4} give better lower bounds in those instances that they failed to solve, so their lower bound curves are higher at point 1.

6 Conclusions

Hybrid BFS is an easily implemented variant of the Branch and Bound algorithm combining advantages of BFS and DFS. While being a generic strategy, applicable to essentially any combinatorial optimization framework, we used it to improve Depth First Branch and Bound maintaining soft arc consistency and tested it on a large benchmark set of problems from various formalisms, including Cost Function Networks, Markov Random Field, Partial Weighted MaxSAT and CP instances representing a variety of application domains in bioinformatics, planning, resource allocation, image processing and more. We showed that HBFS improves on DFS or DFS equipped with LDS or restarts in terms of number of problems solved within a deadline but also in terms of anytime quality and optimality gap information.

HBFS is also able to improve Tree Decomposition aware variants of DFS such as BTD, being able to solve more problems than the previous DFS based BTD on the same set of benchmarks. BTD is targeted at problems with relatively low treewidth and has been instrumental in solving difficult radio-link frequency assignment problems. On such problems, BTD-HBFS provides to BTD the same improvements as to DFS.

Its ability to provide feedback on the remaining search effort, to describe the current remaining search space in a list of open nodes and to decompose search in self-interrupted DFS probes makes it a very dynamic search method, very attractive for implementing multi-core search.

References

1. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving sat-based weighted maxsat solvers. In: Milano, M. (ed.) Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7514, pp. 86–101. Springer (2012), http://dx.doi.org/10.1007/978-3-642-33558-7_9
2. van den Berg, J., Shah, R., Huang, A., Goldberg, K.: ANA*: Anytime nonparametric A*. In: Proceedings of Twenty-fifth AAAI Conference on Artificial Intelligence (AAAI-11) (2011)
3. Berthold, T.: Primal heuristics for mixed integer programs. Master's thesis, Technischen Universität Berlin (2006), urn:nbn:de:0297-zib-10293
4. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI. vol. 16, p. 146 (2004)
5. Cooper, M., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M., Werner, T.: Soft arc consistency revisited. Artificial Intelligence 174(7), 449–478 (2010)
6. Dechter, R., Mateescu, R.: And/or search spaces for graphical models. Artificial intelligence 171(2), 73–106 (2007)
7. de Givry, S., Schiex, T., Verfaillie, G.: Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP. In: Proc. of the National Conference on Artificial Intelligence, AAAI-2006. pp. 22–27 (2006)
8. Harvey, W.D., Ginsberg, M.L.: Limited discrepency search. In: Proc. of the 14th IJCAI. Montréal, Canada (1995)
9. Jégou, P., Terrioux, C.: Hybrid backtracking bounded by tree-decomposition of constraint networks. Artif. Intell. 146(1), 43–75 (2003), http://dx.doi.org/10.1016/S0004-3702(02)00400-9
10. Jégou, P., Terrioux, C.: Combining restarts, nogoods and decompositions for solving csps. In: Schaub, T., Friedrich, G., O'Sullivan, B. (eds.) ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014). Frontiers in Artificial Intelligence and Applications, vol. 263, pp. 465–470. IOS Press (2014), http://dx.doi.org/10.3233/978-1-61499-419-0-465
11. Jégou, P., Terrioux, C.: Tree-decompositions with connected clusters for solving constraint networks. In: O'Sullivan, B. (ed.) Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8656, pp. 407–423. Springer (2014), http://dx.doi.org/10.1007/978-3-319-10428-7_31
12. Kitching, M., Bacchus, F.: Exploiting decomposition in constraint optimization problems. In: Principles and Practice of Constraint Programming. pp. 478–492. Springer (2008)
13. Larrosa, J., de Givry, S., Heras, F., Zytnicki, M.: Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In: Proc. of the 19th IJCAI. pp. 84–89. Edinburgh, Scotland (Aug 2005)
14. Lawler, E., Wood, D.: Branch-and-bound methods: A survey. Operations Research 14(4), 699–719 (1966)
15. Lecoutre, C., Saïs, L., Tabary, S., Vidal, V.: Reasoning from last conflict(s) in constraint programming. Artificial Intelligence 173, 1592,1614 (2009)
16. Likhachev, M., Gordon, G.J., Thrun, S.: ARA*: Anytime A* with provable bounds on suboptimality. In: Advances in Neural Information Processing Systems. p. None (2003)
17. Linderoth, J.T., Savelsbergh, M.W.: A computational study of search strategies for mixed integer programming. INFORMS Journal on Computing 11(2), 173–187 (1999)

18. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. In: Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the. pp. 128–133. IEEE (1993)
19. Marinescu, R., Dechter, R.: AND/OR branch-and-bound for graphical models. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005. pp. 224–229. Professional Book Center (2005), `http://www.ijcai.org/papers/1556.pdf`
20. Marinescu, R., Dechter, R.: Best-first AND/OR search for graphical models. In: Proceedings of the National Conference on Artificial Intelligence. pp. 1171–1176. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999 (2007)
21. Otten, L., Dechter, R.: Anytime and/or depth-first search for combinatorial optimization. AI Communications 25(3), 211–227 (2012)
22. Pearl, J.: Heuristics — Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley Publishing Comp. (1985)
23. Pohl, I.: Heuristic search viewed as path finding in a graph. Artificial Intelligence 1(3), 193–204 (1970)
24. Robertson, N., Seymour, P.D.: Graph minors. II. Algorithmic aspects of tree-width. Journal of algorithms 7(3), 309–322 (1986)
25. Sanchez, M., Allouche, D., de Givry, S., Schiex, T.: Russian doll search with tree decomposition. In: IJCAI. pp. 603–608 (2009)
26. Schulte, C.: Comparing trailing and copying for constraint programming. In: Schreye, D.D. (ed.) Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999. pp. 275–289. MIT Press (1999)
27. Stern, R., Kulberis, T., Felner, A., Holte, R.: Using lookaheads with optimal best-first search. In: AAAI (2010)
28. Terrioux, C., Jégou, P.: Bounded backtracking for the valued constraint satisfaction problems. In: CP. pp. 709–723 (2003)