The AAAI-17 Workshop on
Symbolic Inference and Optimization
WS-17-14

BDD-Constrained A* Search: A Fast Method for Solving Constrained DAG Shortest-Path Problems

Fumito Takeuchi,[1] Masaaki Nishino,[2] Norihito Yasuda,[2] Takuya Akiba,[3]
Shin-ichi Minato,[1] Masaaki Nagata[2]
[1]Graduate School of Information Science and Technology, Hokkaido University
[2]NTT Communication Science Laboratories, NTT Corporation
[3]Preferred Networks, inc.
fumito@ist.hokudai.ac.jp

Abstract

This paper deals with the constrained DAG shortest path problem (CDSP), which finds the shortest path on a given directed acyclic graph (DAG) under any logical constraints posed on taken edges. There exists a previous work that uses binary decision diagrams (BDDs) to represent the logical constraints, and traverses the input DAG and the BDD simultaneously. The time complexity of this BDD-based method is derived from BDD size, and tends to be fast only when BDDs are small. However, since it does not prioritize the search order, there is considerable room for improvement, particularly for large BDDs. We combine the well-known A* search with the BDD-based method synergistically, and implement several novel heuristic functions. The key insight here is that the 'shortest path' in the BDD is a solution of a relaxed problem, just as the shortest path in the DAG is. Experiments, particularly practical machine learning applications, show that the proposed method deceases search time by up to 2 orders of magnitude, with the specific result that it is 2,000 times faster than a commercial solver.

Introduction

The constrained DAG shortest path problem (CDSP) consists of finding a path from a start node to an end node that minimizes the sum of edge weights, subject to not violating any given logical constraints that may be posed on edges. Many combinatorial optimization problems, which are usually solved by dynamic programming (DP) algorithms, can be reduced to DAG shortest path problem equivalents. In the same way, CDSP can solve constrained versions of these combinatorial optimization problems. These include problems of practical importance such as the 0-1 knapsack problem with disjunctive constraints (Yamada, Kataoka, and Watanabe 2002), sequence alignment with user-defined anchor points (Morgenstern et al. 2006), and incorporating knowledge based constraints into statistical machine learning models (Chang, Ratinov, and Roth 2012).

Although a DAG shortest path can be solved in time linear to the number of graph edges, introducing logical constraints makes the problem difficult to solve. Nishino et al. has proposed an algorithm that uses the binary decision diagram (BDD) (Akers 1978; Bryant 1986) to represent the logical constraints and thus solve CDSP (Nishino et al. 2015). Their method, called BDD-constrained search (BCS), simultaneously traverses the input DAG and the BDD. During the traverse, the method checks the logical equivalence of the partial paths to reduce the number of search states. We note that since BCS can be slow when BDDs are large and it does not prioritize the search order, there is considerable room to improve efficiency, particularly for large BDDs.

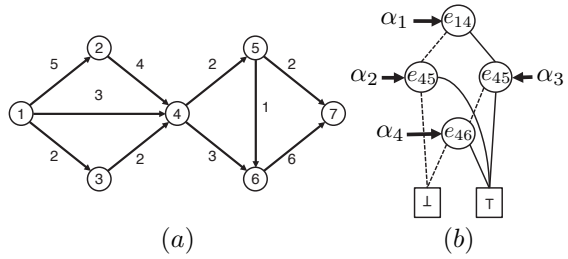

Figure 1: (a) Example of an edge-weighted DAG and (b) BDD corresponding to the Boolean function $F = (e_{14} \wedge e_{46}) \vee e_{45}$ that represents logical constraints.

To address this problem, this paper adds to BCS the well-known A* search. As is often the case, the efficiency of A* search depends on the estimation accuracy of a heuristic function. Since the shortest path in a DAG is a solution of a relaxed version of the original problem, we use this as an estimate of the original solution. Moreover, we focus on the fact that a BDD is also a DAG. We also define a kind of shortest path on the BDD by properly weighting its edges. Accordingly, we employ two types of estimations; one is DAG based and the other is BDD based. Since these estimations are independent of each other, we hope to synergistically combine them to form one heuristic function. We call A* search with this heuristic function BCA*.

Preliminary

Notation

Let $G = (V, E)$ be a DAG, where V is a set of vertices and E is a set of edges. Let $|V|$ and $|E|$ be the numbers of vertices and edges, respectively. We use $v_1, v_2, \ldots, v_{|V|}$

to represent vertices. We use e_{ij} to represent the directed edge whose source and target vertices are v_i and v_j, respectively. We assume that every DAG is edge-weighted, i.e., each edge, $e_{ij} \in E$, has a real value weight, w_{ij}. We also assume that vertices $v_1, \ldots, v_{|V|}$ follow a topological order. Let p be a path on the DAG and represent it as a subset of E. Let the length of path p be the sum of the weights, w_{ij}, of the edges e_{ij} contained in p. Given vertices $u, v \in V$, let $\mathcal{P}_{u,v}$ be the set of all paths that start at u and end at v.

Binary Decision Diagram

The BDD is a data structure for representing the Boolean function $F(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are logical variables and n is the number of arguments. The BDD consist of two types of nodes: terminal nodes ($\bot$ and $\top$) and non-terminal nodes. Every non-terminal node is associated with a logical variable as its label, and it has two outgoing edges called lo-edge and hi-edge. A node with no parent node is called *root*. Every BDD node recursively represents a Boolean function. $\bot$ and $\top$ represent the Boolean functions *false* and *true*, respectively. Suppose that the lo-edge and hi-edge of node α with the label x_i point to nodes that represent the Boolean functions F_0 and F_1, respectively. Then, α represents the Boolean function $F(x_i, \ldots, x_n) = (\bar{x_i} \wedge F_0) \vee (x_i \wedge F_1)$. Figure 1 (b) shows an example of a BDD representing $F = (e_{14} \wedge e_{46}) \vee e_{45}$, where e_{14}, e_{46}, and e_{45} are logical variables. Dashed edges represent lo-edges and a solid edges represent hi-edges. In this example, α_3 represents the subfunction $F_{\alpha_3} = (\bar{e_{45}} \wedge e_{46}) \vee e_{45}$. A path from the root to the $\top$ node corresponds to an assignment of variables that makes the Boolean function true. And we define the *width* of a BDD as the maximum number of nodes pointed to by the edges in a cut of the BDD at a level of nodes.

BDD-Constrained Search

Before explaining BCS, we briefly remind the reader why solving DAG shortest path problems without constraints is fast. The point is that the shortest $s-t$ path that goes through v is obtained by concatenating the shortest path from s to the vertex $v \in V$ with the shortest path from v to t. This is because which path is chosen as $s - v$ does not affect the remaining $v - t$ path. By using this feature, we can solve the problem with a DP algorithm that computes and stores the shortest path from s to every vertex $v \in V$ in topological order, thus the computing time is $O(|E|)$. In contrast, we cannot apply the same DP algorithm to CDSP, because the path obtained by concatenating the shortest $s - v$ path with the shortest $v - t$ path may not satisfy the constraints.

BCS solves CDSP with a DP algorithm by checking the equivalence of constraints posed on the set of paths. Let F be the Boolean function representing the logical constraints whose variable order is taken to be the topological order of the DAG. Suppose that we take partial path $p \in \mathcal{P}_{s,v}$ from start vertex s to intermediate vertex v, then the subfunction F_p, representing the constraints posed over the remaining paths from v to t, is obtained by the partial assignment: $e = 1$ if $e \in p$, $e = 0$ if $e \notin p$ and the source vertex of e precedes v. This subfunction means that the constraints on the remaining $v - t$ path must be satisfied. Suppose that there are two paths $p, q \in \mathcal{P}_{s,v}$ that satisfy $F_p = F_q$. Then, the set of possible paths from v to t is the same regardless of whether p or q is taken. Therefore, if p is longer than q, then the shortest $s - t$ path that goes through v while satisfying constraint F must not contain p as a partial path because any $s - t$ path that satisfies F and contains p as a partial path can be shortened by replacing p by q. We can prune the search starting from p. In this manner, we can use the equivalence of constraints to prune the search on non-optimal partial paths.

Algorithm 1 BDD-constrained search algorithm

Input: Weighted DAG $G = (V, E)$, a BDD representing the logical constraints on path $s, t \in V$
Output: The shortest path from s to t that satisfies the logical constraints

```
1: Cost[s][root] ← 0,
2: for Select every e_ij ∈ E in topological order do
3:     (v, u) ← source and target vertices of e_ij
4:     for all BDD node α which is key to Cost[v][α] do
5:         β ← followBDD(e_ij, α)
6:         if β = ⊥ then continue
7:         if label(β) = e_ij then
8:             β ← hi(β)
9:         if Cost[u][β] > Cost[v][α] + w_ij then
10:            Cost[u][β] ← Cost[v][α] + w_ij
11:            Back[u][β] ← (e_ij, α)
12: (e, α) ← Back[t][⊤]
13: while Source vertex of e is not s do
14:     u ← source vertex of e
15:     Output e;
16:     (e, α) ← Back[u][α]
17: return Cost[t][⊤]
```

To exploit the equivalence of constraints, we have to (i) update F_p along with path p and (ii) check the equivalence of F_p and F_q for different paths $p, q \in \mathcal{P}_{s,v}$; BDD is used for these operations. For (i), updating F_p along with path p corresponds to assigning values to variables in F. This operation corresponds to following lo-edges or hi-edges of the BDD representing F. Therefore, there always exists a BDD node α that represents F_p for any path p. For (ii), the equivalence of Boolean functions can be checked by using the canonicity of BDDs, i.e., if there are two $s - v$ paths p, q satisfying $F_p = F_q$, then the BDD nodes that correspond to F_p and F_q are always the same. Hence, we can check the equivalence of F_p and F_q in constant time.

We show an example in Figure 1. The Boolean function represented by this BDD is $F = (e_{14} \wedge e_{46}) \vee e_{45}$. If we take path $p = \{e_{12}\}$, then F_p becomes e_{45} by assigning $e_{12} = 1$ and $e_{14}, e_{13} = 0$ to F. F_p corresponds to the BDD node α_2 and can be reached from the root node by following the appropriate paths. Let q, r be the paths $q = \{e_{12}, e_{24}\}$ and $r = \{e_{13}, e_{34}\}$. Then, $F_q = F_r = e_{45}$, and both functions correspond to BDD node α_2. In this case, the lengths of q, r are, respectively, 9 and 4. Thus, we prune the search for paths that contain q because replacing q by r always shortens the path.

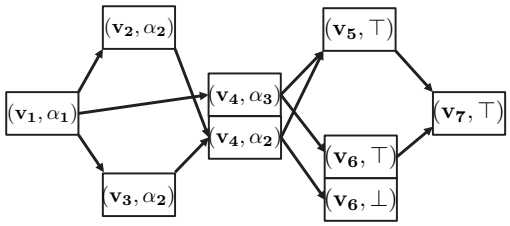

Figure 2: Example of the process of BCS algorithm.

BCS represents the search state by the pair (v, α), where v is a current end of partial path $s - v$ and α is the BDD node that represents the subfunction defined by the path. Thus start DAG vertex s corresponds to state $(s, root)$, and the goal vertex t corresponds to state $(t, \top)$. If edge e_{ij} exists and β is a subfunction of α, we define the direct transition from (v_i, α) to (v_j, β), with transition cost is w_{ij}. With this definition, the CDSP solution corresponds to finding successive transitions from state $(s, root)$ to $(t, \top)$ with minimum cost. Algorithm 1 shows the procedure of BCS. Subroutine followBDD(e, α) visits BDD nodes by following lo-edges from node α until the label of the visited node is not less than e and returns the last visited node β. Subroutines $\mathrm{hi}(\alpha)$ and $\mathrm{lo}(\alpha)$ return the node pointed to by the hi-edge and lo-edge of α, respectively. The time complexity of BCS is $O(|E|W)$, where $|E|$ and W denote the number of edges in the DAG and BDD width, respectively. Consequently, BCS is fast only if BDD width is small. Figure 2 shows an example of BCS solving the problem shown in Figure 1.

BDD-Constrained A* Search

In this section, we incorporate A* search into BCS. For each state (v, α), we calculate value $f(v, \alpha)$, an estimate of the shortest path length that encompasses the state. Let $g(v, \alpha)$ be the shortest path length from $(s, root)$ to (v, α). Let $h(v, \alpha)$ be a heuristic function that is monotonic and returns the estimated length from state (v, α) to goal state $(t, \top)$. Then $f(v, \alpha)$ is defined as

$$f(v, \alpha) = g(v, \alpha) + h(v, \alpha)\,.$$

BCA* works as shown in Algorithm 2. We also use two tables $Cost[v][\alpha]$ and $Back[v][\alpha]$. The list named *open* list is used to store the set of candidate states to be explored. After initializing $Cost[s][root]$ (line 1) and appending $(s, root)$ to the open list (line 2), BCA* selects a state whose value of $f(v, \alpha)$ is minimum in the open list (line 4). Then, for every edge e_{ij} whose source vertex is v, it calculates state (u, β) that can be reached from (v, α) by using e_{ij} (lines 6–10). It then computes the score of (u, β) and updates $Cost[u][\beta]$ and the open list (lines 11–16).

The Heuristic Function

In this subsection we describe the heuristic function used for BCA*. Since the shortest path in the DAG is a solution of a relaxed problem of the original problem, the value is always shorter than or equal to the solution of the original problem. Thus we can use this as the estimate needed by the heuristic

Algorithm 2 BDD-Constrained A* Search

Input: Weighted DAG $G = (V, E)$, $s, t \in V$, BDD, monotonic heuristic function h
Output: shortest path that satisfies constraints represented by BDD from s to t on DAG

```
 1: Cost[s][root] ← 0
 2: add (s, root) to open
 3: while open is not empty do
 4:     (v, α) ← the state with the minimum f(v, α) in open
 5:     if Cost[v][α] + h(v, α) < f(v, α) then continue
 6:     if (v, α) = (t, ⊤) then break
 7:     for all edges e_ij ∈ E such that source vertex is v do
 8:         u ← target vertex of e_ij,  β ← followBDD(e_ij, α)
 9:         if β = ⊥ then continue
10:         if label(β) = e_ij then
11:             β ← hi(β)
12:         if (u, β) is in open then
13:             if f(u, β) ≤ Cost[v][α] + w_{i,j} + h(u, β) then
14:                 continue
15:         Cost[u][β] ← Cost[v][α] + w_{i,j}
16:         Back[u][β] ← (e_{i,j}, α)
17:         f(u, β) ← Cost[u][β] + h(u, β)
18:         add (u, β) to open
19: (e, α) ← Back[t][⊤]
20: while Source vertex of e is not s do
21:     u ← source vertex of e
22:     Output e
23:     (e, α) ← Back[u][α]
24: return g(t, ⊤)
```

function. Similarly, because BDD is also a DAG, we can also use the definition of the 'shortest path' of the BDD as an estimate. More specifically, CDSP can be interpreted as finding $X \subseteq E$ that minimizes the sum of the weights of the edges contained in X and satisfies the following conditions: (i) X corresponds to a path from s to t in the problem DAG, and (ii) X makes the Boolean function $F(X) = true$. By relaxing these two conditions independently, we can introduce two types of cost estimations.

By relaxing condition (ii), the problem becomes a DAG shortest path problem. Let $h_D(v)$ denote the shortest path length from v to t in the DAG. The estimated shortest path length for state $h(v, \alpha)$ is defined as:

$$h(v, \alpha) = h_D(v).$$

Next, we relax condition (i). The relaxed problem is to find assignments of 0, 1 to all $e_{ij} \in E$ that minimize $\sum e_{ij} w_{ij}$ and satisfy $F = 1$. We can calculate the values for all BDD nodes using Knuth's Algorithm B (Knuth 2009), which is a DP algorithm that runs in time proportional to the number of BDD nodes.

For example, in Figure 1, we can get the value of $h_B(\alpha)$; $h(\top) = 0, h(\alpha_4) = h(\top) + 3 = 3, h(\alpha_3) = \min\{h(\alpha_4) + 0, h(\top) + 2\} = 2$, and so on.

If $h_B(\alpha)$ denotes the shortest path length from α to $\top$ in the BDD, the estimated shortest path length for state $h(v, \alpha)$ is defined as:

$$h(v, \alpha) = \sum_{e_{ij} \in E_{v,\alpha}} \min(w_{ij}, 0) + h_B(\alpha),$$

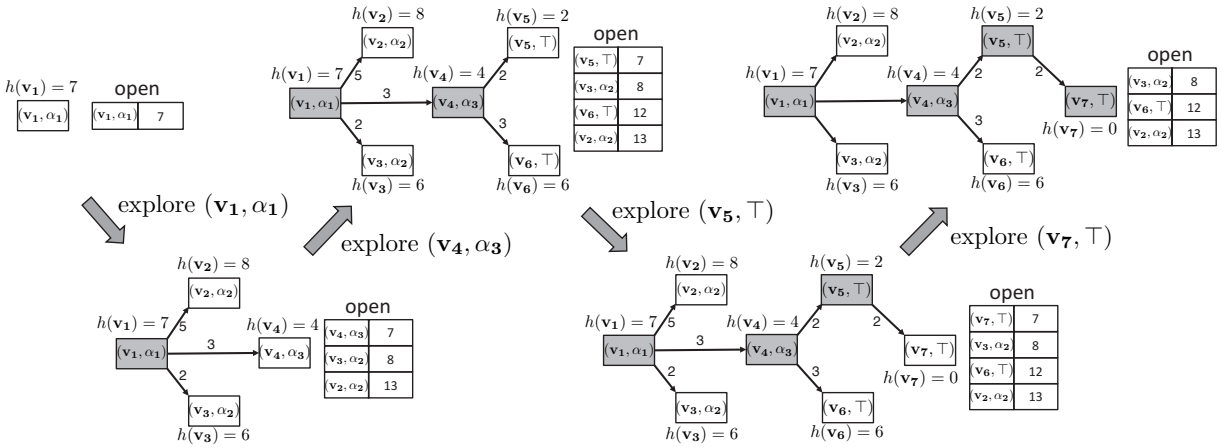

Figure 3: Example of the process of BCA* algorithm.

where $E_{v,\alpha}$ is the set of edges that come after the first e_{ij} whose source vertex is v and that come before the edge that corresponds to $\mathrm{label}(\alpha)$ in topological order. We need the first term because the Boolean function that represents the logical constraints on remaining paths at vertex v is defined over the set of all edges that come after the first e_{ij} whose source is v in topological order. On the other hand, the Boolean function represented by the BDD node is defined on the set of all edges that come after $\mathrm{label}(\alpha)$. Let $|BDD|$ be the number of BDD nodes. The running time to finish the above estimation for all BDD nodes is $O(|E| + |BDD|)$, because computing the cumulative sum for all edges takes $O(|E|)$ time and the computation of $h(\alpha)$ takes $O(|BDD|)$ time.

Here we combine these two together into our heuristic function. Because $h_D(v, \alpha)$ and $h_B(v, \alpha)$ are independent of each other, we can use these functions in parallel. Therefore, the proposed heuristic function is defined as follows:

$$h_{D\&B}(v, \alpha) = \max(h_D(v, \alpha), h_B(v, \alpha)).$$

Since $h_D \leq h_{D\&B}$ and $h_B \leq h_{D\&B}$, this heuristic function can, by definition, give a closer estimate than h_D or h_B. Since both h_D and h_b are monotonic, obviously $h_{D\&B}$ is also monotonic.

Example of BCA*

Figure 3 shows an example of BCA*. The input DAG and BDD are shown in Figure 1(a) and (b), respectively. The BDD corresponds to the Boolean function $F = (e_{14}, \wedge e_{46}) \vee e_{45}$. Due to space limitation, the heuristic function illustrated considers only the estimation from the DAG side. For states $v_1, \ldots, v_7$, the values of the DAG heuristic function $h(v_i, \alpha)$ are, $7, 8, 6, 4, 2, 6$, and 0, respectively. Search states are represented as rectangles, and the arcs between them represent transitions between states. Gray-colored rectangles represent explored states. We initialize the search by appending the initial search state (v_1, α_1) to the open list. Then, we remove the state from the list and explore it and add three states (v_2, α_2), (v_3, α_2), (v_4, α_3) can be reached from (v_1, α_1) to the open list. Because the lengths of the current shortest paths from the start state to each state are $Cost[v_2][\alpha_2] = 5$, $Cost[v_3][\alpha_2] = 2$, and $Cost[v_4][\alpha_3] = 3$, the estimated scores are $f(v_2, \alpha_2) = Cost[v_2][\alpha_2] + h(v_2, \alpha_2) = 13$, $f(v_3, \alpha_2) = Cost[v_3][\alpha_2] + h(v_3, \alpha_2) = 8$, and $f(v_4, \alpha_3) = Cost[v_4][\alpha_3] + h(v_4, \alpha_3) = 7$, respectively. Because $f(v_4, \alpha_3)$ is the smallest among the states in the open list, it is used in the next step. By repeating this procedure, we explore states in the order $(v_1, \alpha_1) \rightarrow (v_4, \alpha_3) \rightarrow (v_5, \top) \rightarrow (v_7, \top)$ to finally reach the goal state $(v_7, \top)$ and finish the search. We eventually get the shortest path $\{e_{14}, e_{45}, e_{57}\}$ and its length, 7. After the search ends, we explore 4 states, while BCS will explore 9 states as shown in Figure 2.

Complexity

Theorem 1. *When the open list is implemented as a priority queue with a binary heap, the worst-case time complexity of BCA* is $O(|E|W \log(|E|W))$. The worst-case space complexity is $O(|V|W)$.*

Proof. Because the number of states expanded in BCS is $O(|E|W)$, this is also true for BCA*. Thus, the number of states stored in the binary heap is also $O(|E|W)$. Because storing each state in the binary heap requires $O(\log(|E|W))$, the worst-case time complexity of BCA* is $O(|E|W \log(|E|W))$.

The time complexities of the DAG heuristic function and BDD heuristic function are $O(|E|)$ and $O(|E| + |BDD|)$, respectively. Because $|E| \leq |E|W$ and $|BDD| \leq |E|W$, they are less complex than BCA*. Therefore, the worst-case time complexity of the whole algorithm is also $O(|E|W \log(|E|W))$. With regard to space complexity, because $O(W)$ states exist at most for each vertex in the DAG, the worst-case space complexity is $O(|V|W)$. □

Discussion

BCS can be extended to solve constrained shortest path problems when there are many-to-one correspondences be-

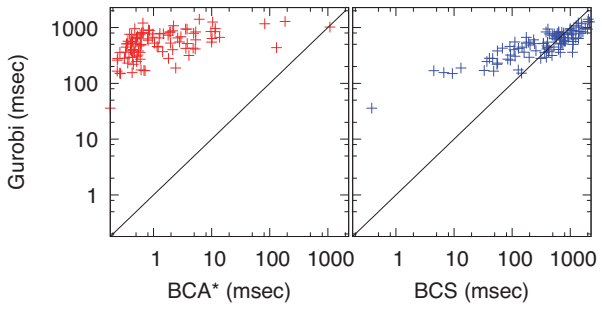

Figure 4: Time comparison between Gurobi and BCA* (left) / BCS (right)

tween DAG edges and Boolean variables. In this setting, the value of the variable is true if and only if a path includes one of the edges. This representation reduces the size of the BDD. This extension allows for more natural representations of some kinds of constraints. For example, the action of taking an item corresponds to deleting multiple edges in the DAG that represents a 0-1 knapsack problem. Thus, logical constraints imposed on taken items are naturally represented as constraints on groups of edges of the DAG. BCA* can also handle this many-to-one correspondence, and the DAG heuristic function can be used without any modification. The BDD heuristic function requires a small modification in that the weight of every variable is set to the minimum weight of the edge that corresponds to the variable. In the next section, we use this constraint-on-group setting in experiments on 0-1 knapsack problems that involve searching for a Viterbi path.

With small modifications, BCA* can also be used to solve the constrained DAG longest path problem.

Experiments

Experimental Settings

We conducted two experiments using two types of practical problems: the Viterbi path finding problem of hidden Markov models (HMMs), and the 0-1 Knapsack problem.

The first experiment assumes sequential labeling, a kind of machine learning task. We followed the settings in (Chang, Ratinov, and Roth 2012), where the HMM is assumed to be used for identifying the role, such as 'author' and 'title', of each word in a given citation text snippet. The sequential labeling task correspond to finding the Viterbi path of the HMM. The Viterbi path is the sequence of hidden states that maximizes the likelihood scores. If we see the searching space as a DAG, the Viterbi path correspond to the longest path in the DAG. We trained the HMM with 10 training samples and used 99 samples as test data. Both sets were obtained from (Chang, Ratinov, and Roth 2012). There are 12 types of constraints. Some of them are non-local, complex constraints. One example, called AppearOnce, demands that the same type of role must appear consecutively with no separation. We compared the proposed algorithm with the commercial solver Gurobi 6.5.1.

pair ratio	time (msec)		
γ	BCA*	Yamada 02	BCS
0.001	1.89	2.43	79.95
0.005	2.16	3.54	19140.33
0.01	47.31	37.20	N/A
0.5	4.33	1509.41	294.42
0.9	1.70	60.93	16.28
0.95	1.70	39.83	16.40

Table 1: Results from solving DCKP.

As the second experiment, we considered a kind of 0-1 knapsack problem called disjunctively constrained knapsack problem (DCKP)(Yamada, Kataoka, and Watanabe 2002; Hifi and Michrafy 2006). DCKP is a kind of 0-1 knapsack problem with disjunctive constraints, i.e. constraints on pairs of items that cannot be selected together. We can find an optimal solution of a 0-1 knapsack problem by using the pseudo polynomial time DP algorithm. This DP algorithm is equivalent to the problem of finding an optimal path on a DAG. Thus DCKP can be seen as a DAG shortest path problem with disjunctive constraints posed on DAG edges. We made instances of a knapsack problem with 100 items and budget size of 1000. We randomly assigned an integer in the range [1, 100] to the weight and value of each item. As for constraints, disjunctive pairs were randomly selected according to the probability parameter $\gamma = \frac{(\#disj.pairs)}{(\#items)(\#items-1)/2}$. We varied γ as follows: $0.001, 0.005, 0.01$, and 0.5. Since the size of the BDD depends on items selected in each pair as well as the number of disjunctive pairs, it is hard to predict the complexity of finding the shortest path from just the number of disjunctive pairs. Therefore, we prepared 10 instances for each setting. We compared the proposed method with Yamada02, a dedicated algorithm for DCKP that uses Lagrangian relaxation and the branch and bound method.

All experiments were performed on a Linux machine with Xeon X5680 3.33 GHz CPU and 48 GB RAM. We implemented BCS and BCA* with C++ and constructed BDDs using the CUDD library.[1]

Results

For the sequential labeling task, Figure 4 compares the results of BCA* and Gurobi (left) along with the results of BCS and Gurobi (right). Both axis have log scale. As we can see, BCA* outperforms Gurobi in almost all cases; one of the 99 cases was the exception. BCA* is at least 100 times faster than Gurobi, and in the best case the ratio is about 2,000, note that the worst case ratio is around 0.28. In contrast, the right figure for BCS shows a neutral result. Actually it outperformed Gurobi in only 54 cases.

Table presents the measured results of DCKP. Each value is the average of 10 tests. In this table, N/A indicates that BCS couldn't obtain the result due to memory outage. Since BCA* expands fewer states than BCS, it requires less memory than BCS, thus BCA* could obtain shortest paths in all cases. From this table, we can see that BCA* dramatically

[1] http://vlsi.colorado.edu/~fabio/CUDD/

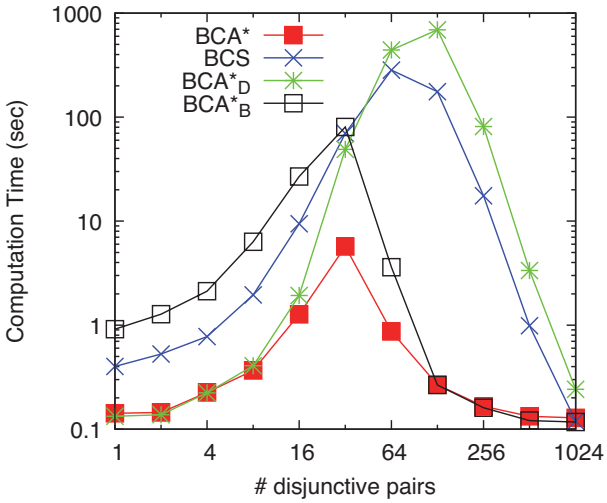

Figure 5: Computation time for various numbers of disjunctive pairs. BCA^*_D, BCA^*_B indicates the BCA* whose heuristic function considers only DAG shortest path and BDD shortest path, respectively.

improves on the speed of BCS. It is even faster than an algorithm designed for this problem in many cases.

In order to investigate the behavior of the two components of the heuristic function, we exhaustively varied the intensity of the constraint using smaller problems. We tested DCKP again with 50 items (i.e. 1225 pairs). The range of the weights and values are [1, 50] and the capacity is 500. We varied the number of randomly selected disjunctive pairs from 2^0 to 2^{10}. Again, we repeated 10 tests for each instance. We compared four methods: BCA* with combined heuristic function (BCA*), BCS, BCA* whose heuristic function only considers DAG shortest path (BCA^*_D), and BCA* whose heuristic function only considers BDD shortest path (BCA^*_B). The result is shown in Figure 5. We can see that BCA* is always faster than BCS and the ratio is at most 700 times. In cases with few disjunctive pairs, since the solution would be close to the solution of the DAG shortest path problem, estimations from the DAG side work fine and BCA^*_D is the fastest. On the other hand, if there are many disjunctive pairs, disjunctive constraints become the dominant factor. As a result, estimations from the BDD side are effective and BCA^*_B is fastest. However, both BCA^*_D and BCA^*_B run slower than BCS when their estimation is not close to the optimal value. This is often the case with general A*, since we have to pay the cost of using a priority queue to maintain the open list. We can see that the combined heuristic function is always to superior to its components, this means that BDD-side and DAG-side two estimations work synergistically.

Related Work

CDSP can be seen as a special case of the constrained shortest path problem (CSP), which generally does not limit the input graph to a DAG. However, most studies of CSP handle a special type of constraint, which limits the total consumption of edge-associate non-negative values, called resources in (Handler and Zang 1980; Santos, Coutinho-Rodrigues, and Current 2007). This type of CSP is specifically called the resource-constrained shortest path problem (RCSP) (Zhu and Wilhelm 2012; Pugliese and Guerriero 2013). Since we assumed arbitrary logical constraints, CDSP can also be seen as a generalization of RCSP. As another view of generalization, the multiple resource constraint shortest path problem (MRCSP) accepts multiple resource constraints.

Since CDSP accepts arbitrary logical constraints, MRCSP can be converted into CDSP. Conversely, some classes of the logical constraints can be represented as MRCSP instances. For example, DCKP can be converted into an MRCSP instance whose number of constraints equals the number of disjunctive pairs. However, since existing studies treat very few constraints, we suspect that this approach would impractical with dozens or more disjunctive pairs. The A$*$ approach has also been studied for MRCSP(Liu and Ramakrishnan 2001). It has some common points with our method in the sense that unconstrained DAG shortest paths are used by a heuristic function.

CDSP can also be solved using only BDDs and the AND operation. More precisely, we prepare a BDD representing the paths of the DAG and a BDD representing the constraints and then build the BDD of these two BDDs. Finding the shortest path in this BDD will give the optimal solution. However, this approach has major drawbacks compared to BCS or BCA*. To allow determination of the intersection, the two BDDs must have same variable sets. On the other hand, BCS just requires the BDD to represent the constraints posed on DAG edges. This difference becomes clearer when some edges in a DAG have the same meaning. Such situations frequently appear in DAGs derived from standard dynamic programming methods. For example, in the knapsack-problem, taking an item will impact a group of multiple DAG edges. In this case, BCS/BCA* has less complexity than the pure BDD-based approach.

The combination of BDD and A* is considered in (Edelkamp and Reffel 1998; Torralba and Alcázar 2013), where BDD is used for improving the efficiency of the search process. BDD here is used for managing the search state of A*. On the other hand, in BCA*, BDD is rather a part of the problem and used represent the constraint. Therefore, the BDD roles differ.

Conclusion

We proposed BDD-constrained A* search (BCA*), which efficiently solves constrained DAG shortest path problems. BCA* is a A* search variant based on an algorithm that uses BDD. We introduced a novel heuristic function for BCA*. Using the fact that both the DAG shortest path and the BDD shortest path are solutions to the same relaxed problem, the heuristic function consists of combining these two factors. Experiment showed it offers dramatically improved performance; the proposed method runs up to 2,000 times faster than a commercial solver.

References

Akers, S. B. 1978. Binary decision diagrams. *Computers, IEEE Trans. on* 100(6):509–516.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Trans. on* 100(8):677–691.

Chang, M.-W.; Ratinov, L.; and Roth, D. 2012. Structured learning with constrained conditional models. *Mach. Learn.* 88(3):399–431.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In *Proc. of KI*, 81–92.

Handler, G. Y., and Zang, I. 1980. A dual algorithm for the constrained shortest path problem. *Networks* 10(4):293–309.

Hifi, M., and Michrafy, M. 2006. A reactive local search-based algorithm for the disjunctively constrained knapsack problem. *Journal of the Operational Research Society* 57(6):718–726.

Knuth, D. E. 2009. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional.

Liu, G., and Ramakrishnan, K. 2001. A* prune: an algorithm for finding k shortest paths subject to multiple constraints. In *In Proc. of INFOCOM*, volume 2, 743–749.

Morgenstern, B.; Prohaska, S. J.; Pöhler, D.; and Stadler, P. F. 2006. Multiple sequence alignment with user-defined anchor points. *Algorithms for Molecular Biology* 1:6.

Nishino, M.; Yasuda, N.; Minato, S.; and Nagata, M. 2015. BDD-constrained search: A unified approach to constrained shortest path problems. In *Proc. of AAAI*, 1219–1225.

Pugliese, L. D. P., and Guerriero, F. 2013. A survey of resource constrained shortest path problems: Exact solution approaches. *Networks* 62(3):183–200.

Santos, L.; Coutinho-Rodrigues, J.; and Current, J. R. 2007. An improved solution algorithm for the constrained shortest path problem. *Transportation Research Part B: Methodological* 41(7):756–771.

Torralba, Á., and Alcázar, V. 2013. Constrained symbolic search: On mutexes, BDD minimization and more. In *Proc. of SOCS*, 175–183.

Yamada, T.; Kataoka, S.; and Watanabe, K. 2002. Heuristic and exact algorithms for the disjunctively constrained knapsack problem. *Information Processing Society of Japan Journal* 43(9):2864–2870.

Zhu, X., and Wilhelm, W. E. 2012. A three-stage approach for the resource-constrained shortest path as a sub-problem in column generation. *Computers & Operations Research* 39(2):164–178.