Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

BDD-Constrained Search: A Unified Approach to Constrained Shortest Path Problems

Masaaki Nishino[1] and **Norihito Yasuda**[2] and **Shin-ichi Minato**[2,3] and **Masaaki Nagata**[1]
[1]NTT Communication Science Laboratories, NTT Corporation
[2]ERATO Minato Discrete Structure Manipulation System Project, Japan Science and Technology Agency
[3]Graduate School of Information Science and Technology, Hokkaido University
nishino.masaaki@lab.ntt.co.jp

Abstract

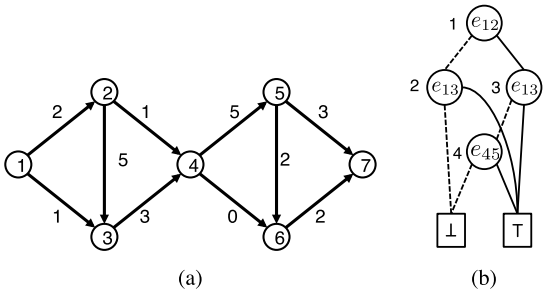
Dynamic programming (DP) is a fundamental tool used to obtain exact, optimal solutions for many combinatorial optimization problems. Among these problems, important ones including the knapsack problems and the computation of edit distances between string pairs can be solved with a kind of DP that corresponds to solving the shortest path problem on a directed acyclic graph (DAG). These problems can be solved efficiently with DP, however, in practical situations, we want to solve the customized problems made by adding logical constraints to the original problems. Developing an algorithm specifically for each combination of a problem and a constraint set is unrealistic. The proposed method, BDD-Constrained Search (BCS), exploits a Binary Decision Diagram (BDD) that represents the logical constraints in combination with the DAG that represents the problem. The BCS runs DP on the DAG while using the BDD to check the equivalence and the validity of intermediate solutions to efficiently solve the problem. The important feature of BCS is that it can be applied to problems with various types of logical constraints in a unified way once we represent the constraints as a BDD. We give a theoretical analysis on the time complexity of BCS and also conduct experiments to compare its performance to that of a state-of-the-art integer linear programming solver.

Introduction

Dynamic programming (DP) is a fundamental tool used for solving various kinds of optimization problems to obtain exact, optimal solutions. Many important problems such as the knapsack problem, the computation of edit distances between string pairs, and the shortest path problem on a graph can be efficiently solved by using DP. To cover practical situations, a natural demand is to extend optimization problems by adding some constraints. For example, suppose that we want to solve a knapsack problem and we know that some combinations of items must not be contained in the solution. Then we represent this property as constraints and solve the problem with them to obtain a solution that does not contain the prohibited combinations of items. However, an optimization problem that can be solved by DP becomes unsolvable with the same method when we add constraints to the problem. Efficient algorithms have been proposed for solving such constraint-added variants (e.g., (Oommen 1986)) of DP-solvable problems, however, they are designed for specific combinations of a problem and a class of constraints, and there may not exist any efficient algorithm for practical problems we want to solve. To develop a made-to-order algorithm for each new combination of a problem and set of constraints is unrealistic. In such cases, we usually resort to using integer linear programming (ILP) solvers, however, they lack theoretical guarantees on the computation time and it is difficult to estimate the solution time.

We propose a unified method for solving a class of optimization problems that are made by adding constraints to existing DP-solvable problems. Here we say that our method is unified since it can solve optimization problems in the same way regardless of the kinds of constraints added. The class of problems our method can solve is the optimization problems that can be solved using DP of finding the shortest (or longest) path on a directed acyclic graph (DAG). Many important optimization problems that are DP-solvable belong to this class. Typical examples in this class includes the 0-1 knapsack problem (Kellerer, Pferschy, and Pisinger 2004), computation of the edit distance (Wagner and Fischer 1974) and the problem of finding the Viterbi path on a Hidden Markov Model (HMM) (Rabiner 1989).

The proposed method, named *BDD-constrained search (BCS)*, uses a binary decision diagram (BDD) (Akers 1978; Bryant 1986) to represent additional logical constraints imposed on an optimization problem. Logical constraints are constraints written as Boolean functions whose variables correspond to edges of the DAG. In the above knapsack problem example, the constraints on pairs of items can be represented by Boolean functions. If there are no additional constraints, a shortest path problem on a DAG can be solved in time proportional to the number of edges due to the optimal substructure property. If additional logical constraints exist, however, the problem loses this property and we have to enumerate all possible paths to solve the problem. Our key idea is to use a BDD as a "guide" to simultaneously judge whether a partial path can be pruned, and find groups of partial paths with which the optimal substructure property holds, i.e., if the shortest path of the DAG contains any partial path contained in the group, then the contained partial



(a) (b)

Figure 1: (a) An example weighted DAG and (b) an example BDD that represents a Boolean function $f = (e_{12} \wedge e_{45}) \vee e_{13}$. These examples are used throughout the paper.

path is always the shortest one in the group. These pruning and grouping operations can drastically reduce the amount of computation and contribute to achieving efficient search under constraints, hence BCS can efficiently solve a broad class of optimization problems.

The following are key virtues of the BCS.

A unified approach Our algorithm can solve many constrained problems once logical constraints are cast as a BDD. Since BDD can represent any Boolean function, any constraints represented as a Boolean function can be represented as a BDD and hence can be used in BCS.

Upper bounds on the running time We analyze the running time of BCS and show it is bounded by the size of the BDD.

Efficiency We conduct experiments on situations where constrained shortest path search on a DAG is used, and confirm that BCS is faster than CPLEX, a state-of-the-art commercial general purpose ILP solver.

Preliminary

Notations Let $G = (V, E)$ represent a directed acyclic graph, where V is a set of nodes and E is a set of edges. We use $|V|$ and $|E|$ to denote the number of nodes and edges, respectively. Let $\delta_s(e)$ and $\delta_t(e)$ be the source node and the target node of edge e. Each edge e_i $(i = 1, \ldots, |E|)$ has a (possibly negative) real value weight w_i. We assume that both nodes $v_1, \ldots v_{|V|}$ and edges $e_1, \ldots, e_{|E|}$ are in a topological sorted order. We also use the notation e_{ij} to represent the edge that satisfies $\delta_s(e_{ij}) = v_i$ and $\delta_t(e_{ij}) = v_j$. Let p be a path. A path is represented as a set of edges $\{e_{p(1)}, \ldots, e_{p(|p|)}\}$ where $|p|$ is the number of edges contained in p and $p(i)$ is an integer in $1 \leq p(i) \leq |E|$. Path p satisfies $\delta_t(e_{p(i)}) = \delta_s(e_{p(i+1)})$ for $i = 1, \ldots, |p|-1$. Given two nodes $s, t \in V$, let $\mathcal{P}_{s,t}$ be the set of all possible paths that start at s and end at t.

Binary Decision Diagram Binary Decision Diagram (BDD) is a data structure that represents a Boolean function of the form $f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are in $\{0, 1\}$ and f returns $\{0, 1\}$. A BDD represents a Boolean function as a DAG, where every node except leaf nodes has exactly two outgoing edges, namely low-edge and high-edge. Let a non-terminal node be an *intermediate node*. An intermediate node has the label that represents the Boolean variable corresponding to the node. We use $\mathrm{lo}(n)$, $\mathrm{hi}(n)$, and $\mathrm{label}(n)$ to represent the low-edge, the high-edge, and the label of a BDD node n, respectively. We also use $\mathrm{lo}(n)$ and $\mathrm{hi}(n)$ to represent the nodes pointed to by the low-edge and the high-edge of node n, respectively. The intermediate node that comes first in the topological sorted order is the root node of the BDD, and we write it as $root$. A BDD has exactly two leaf nodes $\bot$ and $\top$ and we say these nodes are *terminal nodes*. Figure 1(b) shows an example of a BDD that represents Boolean function $f = (e_{12} \wedge e_{45}) \vee e_{13}$. The numbers to the left of the intermediate nodes are IDs used later, and the BDD has four intermediate nodes $n_1, \ldots, n_4$. An edge with dashed line represents the low-edge of an intermediate node and a normal edge represents the high-edge, and the symbol written in an intermediate node represents the label of the node. A path from the root node to the $\top$-terminal corresponds to an assignment of variables that makes the Boolean function true. In the above example, there are three such paths, and each of them represents variable assignments: $(e_{12} = 1, e_{13} = 1)$, $(e_{12} = 0, e_{13} = 1)$ and $(e_{12} = 1, e_{13} = 0, e_{45} = 1)$. In the following, we assume a BDD is *ordered* and *reduced*. We say that a BDD is ordered if the order of the labels of intermediate nodes is same for all paths from the root to a terminal, and we say that a BDD is reduced if it is an ordered BDD with the minimum number of nodes. It is known that a reduced and ordered BDD is canonical, i.e., a Boolean function has the unique reduced BDD representation for given variable order.

Problems Solved by DAG Path Search

Our BDD-Constrained Search (BCS) method can solve a broad class of optimization problems that can be solved by first constructing a DAG and then finding the shortest (or longest) path on the DAG. We define the class of these problems as DAGOPTPATH. DAGOPTPATH contains many important problems such as the 0-1 knapsack problems, computation of edit distance and its variants, and the problems of finding the Viterbi path of a HMM.

The 0-1 knapsack problem The 0-1 knapsack problem (Kellerer, Pferschy, and Pisinger 2004) is the problem of finding a subset of the items that maximizes the sum of scores of selected items, while satisfying the constraint that the sum of the costs of the selected items is less than the limit K. If there are N items and the cost of each element is an integer value, it can be solved in $\theta(NK)$ time by using DP. The DP algorithm prepares table $T[i, j]$ with $(N + 1) \times (K + 1)$ entries, and we can find the optimal solution by recursively updating the elements of the table. We can transform the table into a corresponding edge-weighted DAG by regarding each entry in the table, $T[i, j]$, as a node, and setting edges between nodes that are used in the recursive update of entries. For example, let c_i and w_i be the cost and the score of the i-th item. Then we update $T[i, j]$ as

$$T[i, j] = \max(T[i-1, j], T[i-1, j-c_i] + w_i).$$

Then the corresponding DAG has two edges between the pairs of nodes $(T[i-1, j], T[i, j])$ and $(T[i-1, j-c_i], T[i, j])$ whose weights are 0 and w_i, respectively. We

solve the longest path problem on the DAG to obtain the optimal solution.

Computation of the edit distance The edit distance (Wagner and Fischer 1974) is a popular metric used for measuring the similarity between sequences. It is defined as the minimum number of edit operations needed to transform one sequence into another. The edit distance can also efficiently be computed by using a DP algorithm that runs in $O(NM)$ time if the two strings have lengths of N and M. The DP also can be formulated as a shortest path problem on a DAG. Other than the computation of the edit distance, sequence alignment problems and dynamic time warping can be solved with similar DP algorithm.

Finding the Viterbi path The Viterbi path is an assignment of values to hidden states that maximizes the score function, and the Viterbi algorithm is a DP algorithm that can efficiently find the Viterbi path. The Viterbi algorithm is used in many applications such as speech recognition and part-of-speech (POS) tagging using a Hidden Markov Model (HMM) (Rabiner 1989). Given a sequence of length N and each symbol is labeled with one of K different labels, the Viterbi algorithm can find the optimal assignments of labels in $O(NK^2)$. This algorithm can be seen as finding the longest path on a DAG that has $N \times K$ nodes and $(N-1) \times K^2 + 2K$ edges, and each edge has a weight that is defined by the transition probability and emission probability of the HMM.

Shortest Path Search with Logical Constraints

Optimization problems that belong to DAGOPTPATH appear in many practical situations, and it is natural to solve these problems with additional logical constraints that reflect the demands inherent in the problem domain. However, problems become far more difficult if we impose logical constraints in more complex forms like prohibiting pairs of edges from being included in a path. One of the most frequently used tools for solving problems with arbitrary forms of logical constraints is the general purpose ILP solver. Some ILP solvers can solve large-scale optimization problems in practical time, but they lack theoretical guarantees on the computation time and it is difficult to estimate the solution time.

BDD-constrained Search Algorithm

BDD-constrained search is a unified approach for solving constrained shortest (or longest) path problems on a DAG. The input of the method is a weighted DAG and a reduced and ordered BDD. The DAG corresponds to the problem we want to solve. The BDD represents the Boolean function $f(e_1, \ldots, e_{|E|})$ that represents the logical constraints, where $e_1, \ldots, e_{|E|}$ are Boolean variables corresponding to edges in the DAG.

We first introduce an example problem used in the following explanation. We use the DAG shown in Fig.1 (a). We assume the edges of the DAG are topologically sorted and have the order e_{12}, e_{13}, e_{23}, e_{24}, e_{34}, e_{45}, e_{46}, e_{56}, e_{57}, e_{67}. We use this order as the order of the BDD labels. The weight of each edge is written in the figure. We try to find the shortest path from node $s = v_1$ to $t = v_7$ under the logical constraint $f = (e_{12} \wedge e_{45}) \vee e_{13}$. The BDD in Fig. 1 (b) represents f.

We first review why the DAG shortest path search algorithm can work in time proportional to the number of edges. Suppose that we are trying to find the shortest path from s to t that goes through node v. Then the set of all possible paths is made by concatenating all possible combinations of a path in $\mathcal{P}_{s,v}$ and a path in $\mathcal{P}_{v,t}$. This means that when we search for the shortest path, we only have to remember the shortest path in $\mathcal{P}_{s,v}$ since it is always contained in the shortest path from s to t that goes through v. The standard $O(|E|)$ shortest path algorithm on a DAG uses this property and only updates information about the shortest path for each node. This property is, however, not true for constrained shortest path problems. If there exist two paths p_1, p_2 that go from s to v, then the sets of paths from v to t that satisfy the logical constraints may be different depending on the paths taken. We thus have to store the information about both paths, p_1 and p_2, at node v, since the shorter path may not be contained in the shortest path that satisfies the constraints. Since the amount of information we have to store for a node depends on the number of possible paths from s to the node and it may grow exponentially with the number of edges in a DAG, we need exponential computation time and memory for constrained path search.

We use the DAG in Fig.1 (a) and the BDD in Fig. 1 (b) to show what happens with a constrained shortest path search problem. We try to find the shortest path that satisfies the logical constraint $f = (e_{12} \wedge e_{45}) \vee e_{13}$. We compare the two paths $p_1 = v_1 \to v_2 \to v_4$ and $p_2 = v_1 \to v_3 \to v_4$ that go from v_1 to v_4. There are 3 different paths from v_4 to $t = v_7$. If we take p_1 then taking either $v_4 \to v_5 \to v_7$ or $v_4 \to v_5 \to v_6 \to v_7$ can be a path satisfying the logical constraint. If we take p_2, then all three paths from v_4 to $t = v_7$ satisfy the constraint. This difference requires us to store two different pieces of information at node v_4.

We can prevent this explosion in the amount of information stored at a node by finding equivalent paths in terms of the set of remaining paths. We compare path p_1 with another path $p_3 = v_1 \to v_2 \to v_3 \to v_4$ from s to v_4. The set of possible paths we can take from v_4 to t when we take p_3 is the same as the case as when we take p_1. This result means we have to store the information of the shorter path of either p_1 or p_3, since if p_1 is shorter than p_3, the shortest path that contains p_1 is always shorter than the shortest path that contains p_3. This suggests that the amount of information we have to store for node v is decided by the number of groups of equivalent paths from s to v, and a group of paths is determined by the set of possible paths that connect the node and t.

Definition 1. *Given path p that starts at s and ends at v, let the* remaining paths $\mathcal{R}(p, f)$ *be the set of paths that is a subset of $\mathcal{P}_{v,t}$ and every $p' \in \mathcal{R}(p, f)$ concatenated with p satisfies constraint f.*

BCS uses the BDD to (1) check whether a path satisfies the constraints or not, and (2) manage the information about the groups of equivalent paths. Since to explicitly represent

Algorithm 1 BDD-constrained search algorithm

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

1: $T[s][root] \leftarrow 0$,
2: **for** $i = 1$ to $|E|$ **do**
3: $v_s \leftarrow \delta_s(e_i), \quad v_t \leftarrow \delta_t(e_i)$
4: **for all** pair (n, l) stored in $T[v_s]$ **do**
5: $n' \leftarrow \mathrm{followBDD}(e_i, n)$
6: **if** $n' = \bot$ **then** continue
7: **if** $\mathrm{label}(n') = e_i$ **then**
8: $n' \leftarrow \mathrm{hi}(n')$
9: **if** $T[v_t][n'] > l + w_i$ **then**
10: $T[v_t][n'] \leftarrow l + w_i, \quad B[v_t][n'] \leftarrow (e_i, n)$
11: $(e, n) \leftarrow B[v_{|V|}][\top]$
12: **while** $\delta_s(e) \neq v_1$ **do**
13: Output e; $(e, n) \leftarrow B[\delta_s(e)][n]$
14: **return** $T[v_{|V|}][\top]$

$\mathcal{R}(p, f)$ for each path p is unrealistic, we represent them by the combination of node v in the DAG and the logical constraints imposed on the set of paths $\mathcal{P}_{v,t}$ that are represented by a BDD node. Since the logical constraints determine the remaining paths $\mathcal{R}(p, f)$, we can regard the two paths as having the same set of remaining paths if the paths end at the same node and the corresponding logical constraints are the same. We can obtain the logical constraints that the edges in $\mathcal{P}_{v,t}$ must satisfy by following BDD edges. Since a reduced and ordered BDD has canonical form, we can judge the equivalency of two logical constraints represented by BDDs by checking whether the two constraints correspond to the same BDD node or not. We simultaneously follow both the DAG and the BDD in order that the current position at the BDD represents the logical constraints that must be satisfied by the remaining set of paths. In other words, if two partial paths are given, we follow the paths on the DAG and on the BDD, and if they reach the same nodes both in the DAG and the BDD, it means that the set of remaining paths are equivalent. Similarly, if we follow the BDD edges and if we reach the $\bot$ terminal node, it means the path does not satisfy the logical constraints and we terminate the search.

We show the BDD-constrained search algorithm in Alg. 1. The input of the algorithm is a pair of a weighted DAG and the BDD that represents the logical constraints. We first initialize two sets of tables, T and B, where T is used for storing the lengths of the possible paths, and B stores information used for backtracking. Both sets consist of $|V|$ tables $T[v]$ and $B[v]$ for $v \in V$. Each $T[v]$ is related to DAG node v and is a table that takes BDD node n as a key and returns the shortest path length l from node s to v among the paths whose corresponding logical constraints imposed on the remaining paths are represented by BDD node n. Each $B[v]$ is a table that also takes BDD node as a key and returns the pair of a DAG edge e and BDD node n, which is used in backtracking. We first initialize every $T[v]$ and $B[v]$ $(v \in V)$ as empty tables, and then set $T[s][root] \leftarrow 0$ (line 1). Next, we consult every edge $e \in E$ in topological sorted order and update $T[\delta_t(e)]$ using $T[\delta_s(e)]$, where we use v_s and v_t to represent $T[\delta_s(e)]$ and $T[\delta_t(e)]$, respectively (lines 2 to 10). For each pair (n, l) of BDD node n and path length l stored in $T[v_s]$, we update the BDD node so as that it corresponds to the logical constraints after taking current edge e, and store the results in $T[v_t]$. Function $\mathrm{followBDD}(e, n)$ takes edge e and BDD node n and returns the first BDD node, n', reached by following only the low-edges from n and satisfies either the label of n', which corresponds to an edge of the DAG, is equal or larger than e, or n' is a terminal node. If we reach BDD node $\bot$, it means the corresponding path does not satisfy the given constraints and we terminate the search (line 6). If $\mathrm{label}(n') = e_i$, then we follow the high-edge of node n' to update n' (lines 7 to 8). We then use obtained n' to update $T[v_t]$ and $B[v_t]$ (lines 9 to 10). After traversing all edges, $T[v_{|V|}]$ will contain at most one entry whose key is $\top$ and whose value $l^* = T[v_{|V|}][\top]$ corresponds to the length of the shortest path that satisfies the given constraints. We then perform backtracking to recover the optimal solution (lines 12 to 13).

Example 2. We use the DAG and the BDD shown in Fig.1. First we set $T[v_1][n_1] \leftarrow 0$ where $n_1 = root$. Then update nodes by picking up edges in the order $e_{12}, e_{13}, \ldots, e_{67}$. When we pick up the first edge e_{12}, we lookup table $T[v_1]$ and find key-value pair $(n_1, 0)$. We calculate the new score and the new BDD node. The new score is $0 + w_{12} = 2$, and since the label of n_1 is e_{12}, $\mathrm{followBDD}(e_{12}, root)$ returns n_1, and we obtain the new BDD node $n_3 \leftarrow \mathrm{hi}(n_1)$. We store $T[v_2][n_3] \leftarrow 2$ and $B[v_2][n_3] \leftarrow (e_{12}, n_1)$. After finishing the loop for edges e_{34}, we have two key-value pairs $(\top, 4)$ and $(n_4, 3)$ in $T[v_4]$. There are three paths from s to v_4 but two paths $v_1 \to v_2 \to v_4$ and $v_1 \to v_2 \to v_3 \to v_4$ result in the same BDD node n_4, only one entry is stored. When we pick up e_{46} and update $T[v_6]$ from $(\top, 4)$ and $(n_4, 3)$, since $\mathrm{followBDD}(e_{46}, n_4) = \bot$, we do not update from the pair $(n_4, 3)$. Finally we get $T[v_7] = \{(\top, 6)\}$ and the length of the shortest path is 6 and the path $v_1 \to v_3 \to v_4 \to v_6 \to v_7$ is recovered from entries in B.

Theoretical Analysis

We give the upper bound on the time and space complexity of the BCS algorithm. We can estimate them from the number of edges of the DAG and the form of the BDD representing the constraints.

Definition 3. *Let the* width *of a BDD be the maximum number of nodes pointed to by the edges in a cut of the BDD at a level of nodes.*

Theorem 4. *The time and space complexity of Algorithm 1 are $O(|E|W)$, where W is the width of the BDD.*

Proof. The maximum numbers of entries in $T[v]$ and $B[v]$ depend on the number of different BDD nodes used as keys. Since $\mathrm{followBDD}(e, n)$ always returns a BDD node that is the child of an edge that intercepts a cut of the BDD at a level of nodes. It means $\mathrm{followBDD}(e, n)$ returns at most W different nodes for each v, and the update process in Alg.1 (lines 2 to 10) is performed at most $|E|W$ times. The function $\mathrm{followBDD}(e, n)$ takes, in total, number of steps that is

Table 1: Summary of the datasets.

Dataset	$\|V\|$	$\|E\|$
DAG	4,177	25,208
Knapsack	188,952	368,067
Edit distance	10,609	31,416
POS tagging	12,468	571,411

proportional to the number of BDD nodes. Hence the time complexity of the algorithm is $O(|E|W)$. □

This result suggests that our method runs efficiently if it is applied to problems with logical constraints represented as the BDD with small numbers of nodes.

Discussion

Constraints on the groups of edges The BCS algorithm imposes constraints on combinations of edges. We can extend it to impose logical constraints on combinations on groups of edges if some conditions are satisfied. This setting is more natural when a group of edges have the same meaning in a DAG. For example, in a DAG corresponding to solving a 0-1 knapsack problem, a edge means to choose an item or not and more than one edge corresponds to the same action. We use one BDD node to represent a group of DAG edges that correspond to the same action. This representation can reduce the size of the BDD that represents constraints.

We can use the constraints on groups of edges if (i) the groups are ordered, i.e., let $G_1, G_2 \subseteq E$ be groups, then $e < e'$ is satisfied for all pairs of $e \in G_1$ and $e' \in G_2$, and (ii) a path must not have more than two edges that are contained in the same group. If we use the groups of edges for the 0-1 knapsack problem, the above two conditions will be satisfied and the resulting BDD representation is compact. In experiments, we use constraints on group representation for two problems that corresponds to DP algorithms for the 0-1 knapsack Problem and the Viterbi algorithm.

Zero-suppressed binary Decision Diagram (ZDD) We can use a Zero-suppressed binary Decision Diagram (ZDD) (Minato 1993) instead of a BDD in the BCS algorithm. A ZDD is a variant of a BDD that can represent a Boolean function that returns 0 for almost all inputs in a much smaller DAG than a BDD; we can speed up the BGS algorithm by choosing the smaller of either the BDD or ZDD representation.

Experiments

We conducted experiments to evaluate our BCS algorithm in situations where the problems in the DAGOPTPATH are solved with additional constraints.

Datasets We use four datasets, *DAG*, *Knapsack*, *Edit distance*, and *Part-of-speech (POS) tagging*, each of them are typical datasets for four different tasks. *DAG* is the task of finding a shortest path in the DAG that represents the citation network of high energy physics phenomenology papers on arXiv[1]. Since the original graph contains cycles, we use the timestamp of the nodes to remove some edges to make it a DAG, and then extract the largest connected components of the DAG. We randomly assign an integer weight in the range $[1, 100]$ to each edge. *Knapsack* is the DAG that derives from the DP algorithm for a 0-1 knapsack problem. We design a 0-1 knapsack problem that consists of 200 items, and the cost and the weight of an item are randomly selected from the range $[1, 100]$. We set the size of the knapsack to 1000. *Edit distance* is the DAG used in the computation of the edit distance between two strings. The two strings are selected from the DBLP-dataset[2] which is frequently used in evaluating edit distance computation methods (e.g., (Wang, Li, and Feng 2012; Zhang et al. 2010)). We selected the 2 strings that have median lengths in the dataset, each consists of 102 characters. *POS tagging* is the DAG created when applying the HMM-based POS tagging algorithm to a word sequence. We use the HMM-based POS tagger contained in the Natural Language Toolkit Python library[3]. The length of the word sequence we used consists of 271 words and we tagged the sequence with 46 different labels. The weights of the DAG are determined by the parameters of the POS tagger trained using the Penn Treebank. The summaries of the DAGs are shown in Tab.1.

Constraints We set two different constraints *Checkpoint* and *Disjunction* on each dataset. *Checkpoint* first chooses small groups of edges $G_1, \ldots, G_N \subseteq E$ and then sets a constraint that path p must have at least one edge that is common with each group, i.e., path $p \subseteq E$ must satisfy $p \cap G_i \neq \emptyset$ for all $i = 1, \ldots, N$. *Checkpoint* would be appear in situations where we know a part of the desirable solutions in advance. *Disjunction* is defined by a set of pairs of edges that must not be contained in the solution path. *Disjunction* assumes the situation where we know the combinations of edges that must not be contained in a feasible solution. The knapsack problem under disjunctive constraints is known as the disjunctively constrained knapsack problem (DCKP) (Yamada, Kataoka, and Watanabe 2002; Hifi and Michrafy 2006).

For *Checkpoint* constraints, we first randomly select edges with probability ρ, and then partition the set of selected edges into L groups. We used the pairs of parameters $(L, \rho) = (3, 0.01)$ and $(5, 0.05)$ in experiments. We mention these two settings as CHK1 and CHK2, respectively. For *Disjunction* constraints, we first select the k-best edges that have higher scores, and then randomly generate disjunctive pairs of the selected edges with pair probability γ. We used the pairs of parameters $(k, \gamma) = (50, 0.01)$ and $(100, 0.8)$ in the experiments. We mention these two constraints as DIS1 and DIS2, respectively.

Settings Our BCS algorithm was implemented in C++, and we used the CUDD library[4] to construct the BDDs representing logical constraints. We compared our algorithm with CPLEX 12.5.1.0, a state-of-the-art commercial general purpose ILP solver. All experiments were run on a Linux server with a Xeon X5680 3.33 GHz CPU and 48 GB RAM.

[1] https://snap.stanford.edu/data/cit-HepPh.html

[2] http://www.informatik.uni-trier.de/~ley/db/

[3] http://www.nltk.org/

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

Table 2: Comparison with CPLEX.

Dataset	Constraints	BDD nodes	width	Steps	BDD time (ms)	BCS time (ms)	CPLEX time (ms)
DAG	CHK1	1,010	8	72,862	40.6	8.8	170
	CHK2	20,010	32	310,668	46.3	154	540
	DIS1	98	24	25,763	40.8	2.3	100
	DIS2	13,570	259	60,319	61.2	5.1	130
Knapsack	CHK1	14	8	698,458	2.9	66.9	72,590
	CHK2	188	32	3,714,558	3.1	331	146,880
	DIS1	529	192	11,514,987	3.1	1,060	186,000
	DIS2	13,352	257	45,593,454	40.4	3,961	>3,600 sec
Edit distance	CHK1	1,222	8	230,140	45.6	14.1	2,590
	CHK2	25,040	32	929,231	51.2	75.0	1,050
	DIS1	238	64	121,674	9.0	8.0	1,790
	DIS2	13,410	255	1,410,769	64.9	99.1	2,620
POS tagging	CHK1	42	8	3,061,346	4.0	55.4	6,120
	CHK2	948	32	14,770,508	4.1	304	7,770
	DIS1	134	32	571,412	3.7	14.5	5,190
	DIS2	13,862	268	74,739,926	41.9	138	14,400

Results Experimental results are shown in Tab.2, where width is the width of the BDD, and the steps is the actual number of updating processes executed on the problem. BCS time and BDD time are running time for BCS algorithm and BDD construction, respectively. We can see that for all settings the proposed method can find the optimal path faster than CPLEX.

We can see that the gap between the theoretical upper bound of steps derived from Theorem 4 and the practical number of steps differs between settings. This difference is caused by (i) the number of unused constraints, and (ii) the difference between the width of the BDD and actual number of different nodes of each level of the BDD. For example, if we set logical constraints $f = e_{12} \wedge e_{13} \wedge g$ on the DAG in Fig. 1 (a) , where g is an arbitrary logical constraint, then the search will soon terminate regardless of g since no paths satisfy $e_{12} \wedge e_{13}$. The upper bound, $|E| \times W$, is always bigger than the actual number of steps, and we can directly estimate the solution time from the number of steps. The upper bound is useful for estimating the efficiency of the BCS algorithm.

Comparison with a state-of-the-art algorithm for DCKP

We compared the performance of the proposed algorithm with a problem-specific state-of-the-art algorithm for DCKP. The comparison used the exact solution algorithm proposed in (Yamada, Kataoka, and Watanabe 2002), which solves a DCKP instance by implicitly enumerating possible solutions while applying pruning rules to reduce the number of possible intermediate states to enable efficient search [5]. We used datasets used in (Yamada, Kataoka, and Watanabe 2002) and (Hifi and Michrafy 2006), made by randomly setting the cost and weight of items, and then set disjunctive constraints on the possible pairs of items. We use two schemes for setting item cost and weight: the first scheme sets a randomly selected integer in the range $[1, 100]$ for both cost and weight, while the second scheme first randomly sets the cost of an item in the same way as the first scheme, then set the weight of an item as item cost + 10. The second scheme generates a correlated dataset. We set the number of items to 100, and set disjunctive constraints on pairs of items with probability $\gamma = 0.001, 0.002, 0.005$.

[5]Since the algorithm proposed in (Hifi and Michrafy 2006) is an approximation algorithm, we did not compared it with the proposed algorithm.

Table 3: Experimental results on solving DCKP.

Dataset	γ	BCS time (ms)	Yamada 02 time (ms)
Random	0.001	58.3	10.2
	0.002	224	14.2
	0.005	10,050	18.2
Correlated	0.001	58.1	403
	0.002	228	7,613
	0.005	10,056	54,377

The results are shown in Tab. 3. We can see that the performance of the baseline method become worse when it is applied to correlated instances. This is because the performance of the method depends on how well the pruning rules work, and few candidates can be pruned if item costs and weights are correlated. Since the proposed method does not impose any pruning based on scores, its running time depends only on the sizes of both the BDD and DAG, and shows competitive performance when they are small.

Related Work

For each specific problem in DAGOPTPATH, algorithms for solving it under some constraints have been proposed. The 0-1 knapsack problem has many variants with additional constraints. For example, the tree knapsack problem (Lukes 1974; Hirao et al. 2013), the preceding knapsack problem (Ibarra and Kim 1978), and the disjunctively constrained knapsack problem (Yamada, Kataoka, and Watanabe 2002; Hifi and Michrafy 2006) are problems made by adding some constraints to the original 0-1 knapsack problem. Constrained Edit distance (Oommen 1986) is a variant of the edit distance that sets constraints on the number of edit opera-

tions. The Viterbi algorithm is used in many speech recognition and natural language processing applications, and recently constraints are used for compensating lack of sufficient training data to improve the performance in POS tagging (Zhao and Marcus 2012). Previous algorithms for solving constrained problems are specific in the sense that they assume the constraint forms are specific. Our BCS framework is general since it can handle arbitrary logical constraints represented by a BDD. We can use efficient operations for combining BDDs to easily combine different constraints to make another constraint.

The binary decision diagram (BDD) (Akers 1978; Bryant 1986) was originally used in the fields in digital-system design and related applications (Bryant 1992). In AI fields, it is known as a kind of compiled knowledge representation (Darwiche and Marquis 2002), and is used in many applications such as weighted model counting (Fierens et al. 2011). BDD is known to be used for efficiently solving linear Boolean programming problems (Knuth 2011; Lai, Pedram, and Vrudhula 1994), which is a combinatorial optimization problem of finding x that maximizes $w_1x_1 + \cdots + w_nx_n$ subject to $f(x_1, \ldots, x_n) = 1$, where f is a Boolean function represented by a BDD.

Recently, new algorithms for solving combinatorial optimization problems that use BDD and its variants have been proposed (Bergman, van Hoeve, and Hooker 2011; Bergman et al. 2014). Since the BDD that has all possible solutions may become huge, Bergman et al. construct small BDDs named relaxed or restricted BDDs to compute lower bound or approximate solutions. Our method is different from theirs in that we use BDD to represent the additional constraints and do not construct a BDD that represents all possible solutions. This separation of the problem and the additional constraints can make BDD size relatively small, the manipulation of which is much easier.

Conclusion

We have proposed the BDD-Constrained Search (BCS) algorithm. It can solve an important class of optimization problems that can be efficiently solved with DP algorithm, the problems solved as a shortest path search problem on a DAG, with some additional constraints. The proposed method is general since it can use arbitrary constraints represented as a BDD. Its time complexity can be estimated from the size of the problem DAG and the constraint BDD, and it shows better performance in experiments that assume concrete dynamic programming problems than a commercial ILP solver.

References

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

Bergman, D.; Cire, A. A.; van Hoeve, W.-J.; and Yunes, T. 2014. BDD-based heuristics for binary optimization. *Journal of Heuristics* 20(2):211–234.

Bergman, D.; van Hoeve, W.-J.; and Hooker, J. N. 2011. Manipulating MDD relaxations for combinatorial optimization. In *Proc. of CPAIOR*, 20–35.

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

Bryant, R. E. 1992. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys* 24(3):293–318.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17(1):229–264.

Fierens, D.; den Broeck, G. V.; Thon, I.; Gutmann, B.; and Raedt, L. D. 2011. Inference in probabilistic logic programs using weighted CNF's. In *Proc. of UAI*, 211–220.

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.

Hirao, T.; Yoshida, Y.; Nishino, M.; Yasuda, N.; and Nagata, M. 2013. Single-document summarization as a tree knapsack problem. In *Proc. of EMNLP*, 1515–1520.

Ibarra, O. H., and Kim, C. E. 1978. Approximation algorithms for certain scheduling problems. *Mathematics of Operations Research* 3(3):197–204.

Kellerer, H.; Pferschy, U.; and Pisinger, D. 2004. *Knapsack problems*. Springer.

Knuth, D. E. 2011. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley.

Lai, Y.-T.; Pedram, M.; and Vrudhula, S. B. 1994. EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on* 13(8):959–975.

Lukes, J. A. 1974. Efficient algorithm for the partitioning of trees. *IBM Journal of Research and Development* 18(3):217–224.

Minato, S. 1993. Zero-suppressed BDDs for set manipulation in combinatiorial problems. In *Proc. of DAC*, 272–277.

Oommen, B. J. 1986. Constrained string editing. *Information Sciences* 40(3):267–284.

Rabiner, L. 1989. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE* 77(2):257–286.

Wagner, R. A., and Fischer, M. J. 1974. The string-to-string correction problem. *Journal of the ACM (JACM)* 21(1):168–173.

Wang, J.; Li, G.; and Feng, J. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proc. of SIGMOD*, 85–96.

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).

Zhang, Z.; Hadjieleftheriou, M.; Ooi, B. C.; and Srivastava, D. 2010. B^{ed}-tree: an all-purpose index structure for string similarity search based on edit distance. In *Proc. of SIGMOD*, 915–926.

Zhao, Q., and Marcus, M. 2012. Exploring deterministic constraints: from a constrained english POS tagger to an efficient ILP solution to chinese word segmentation. In *Proc. of ACL*, 1054–1062.