A Multi-Valued Decision Diagram-Based Approach to Constrained Optimal Path Problems over Directed Acyclic Graphs

Mingwei Zhang[1] , **Liangda Fang**[1,2*] , **Zhenhao Gu**[1] , **Quanlong Guan**[1*] and **Yong Lai**[2]
[1]College of Information Science and Technology, Jinan University, Guangzhou 510632, China,
[2]Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, Changchun 130012, China
mingweizhang@stu2022.jnu.edu.cn, {fangld, gql}@jnu.edu.cn

Abstract

Numerous combinatorial optimization problems can be reduced to the optimal path problem over directed acyclic graphs (DAGs). The constrained version of the optimal path problem requires the solution to satisfy a given logical constraint. *BDD-constrained search (BCS)* is an efficient algorithm for the constrained optimal path problem over DAGs. This algorithm considers edges as variables and constraints as Boolean functions and maintains constraints via binary decision diagrams (BDDs), a compact form of Boolean functions. However, BCS involves redundant operations during the search process. To reduce these redundant operations, we use vertices instead of edges as variables and hence represent constraints as multi-valued functions. Due to the multi-valued representation of constraints, we propose a novel algorithm, namely *MDD-constrained search (MCS)*, by using multi-valued decision diagrams (MDDs) instead of BDDs, an efficient representation of multi-valued functions. In addition, we improve MCS via domain reduction in multi-valued functions. Experimental results prove that our proposed algorithm outperforms BCS.

1 Introduction

Numerous combinatorial optimization problems, particularly those that can be solved using dynamic programming paradigm, can be reduced to optimal path problems over directed acyclic graphs (DAGs). Given a weighted DAG, the optimal path problem aims to find a path from a source vertex to a target vertex on the given DAG such that its length is minimal or maximal. These problems include the 0-1 knapsack problem [Kellerer *et al.*, 2004], the Viterbi path problem [Forney, 1973] and the edit distance problem [Wagner and Fischer, 1974]. In many complex real-world scenarios, we do not only find the optimal path on the DAG but also require the optimal path to satisfy a logical constraint. Unfortunately, these constrained optimal path problems become more challenging than optimal path problems without constraints.

Some efficient algorithms targeted at specific problems and constraints have been proposed. For example, Yamada *et al.* [2002] proposed a heuristic method to solve the disjunctively constrained knapsack problem. Oommen [1986] developed an algorithm for the constrained string editing problem. A unified method to these problems is to formalize each problem as an integer linear programming (ILP) instance, which can be solved by state-of-the-art ILP solvers, for example, CPLEX[1] and Gurobi[2]. But the computational time of ILP solvers lacks a theoretical upper bound, making it difficult to estimate the solution time.

BDD-constrained search (BCS) [Nishino *et al.*, 2015] is a unified and efficient approach for the constrained optimization problem, which treats logical constraints as Boolean functions. To maintain logical constraints, this algorithm makes use of binary decision diagram (BDD) [Bryant, 1986], a compact form of Boolean functions. It performs a search on the DAG in a dynamic programming paradigm and employs BDDs to check for equivalent paths to reduce the search space. More importantly, BCS has theoretical upper bounds for computational time and space. BCS solves the constrained optimal path problem over DAGs more efficient than CPLEX. However, since BCS considers the edges as variables and the logical constraint as a Boolean function, it obtains incorrect local constraint w.r.t. a given path in some cases. This leads to extra redundant operations during the search, impacting the efficiency.

To mitigate the above deficiencies of BCS, in this paper, we propose a novel method to the constrained optimization problem, namely *MDD-constrained search (MCS)*. The main idea is to treat logical constraints as multi-valued functions. Each variable of multi-valued functions corresponds to a vertex in a DAG and its domain is the outgoing edges of its corresponding vertex along a special element. We utilize multi-valued decision diagrams (MDDs) [Kam *et al.*, 1998] as the representation of logical constraints. Furthermore, we analyze the time and space complexity of MCS and compare them with the time and space complexity of BCS. Finally, We compare two algorithms BCS and MCS on four datasets with six logical constraints. The experimental results demonstrate MCS shows a substantial improvement over BCS in terms of

*Both are corresponding authors.

[1]https://www.ibm.com/products/ilog-cplex-optimization-studio
[2]https://www.gurobi.com/

search time and memory requirements.

2 Preliminaries

2.1 Directed Acyclic Graphs

Let $G = (\mathcal{V}, \mathcal{E})$ represent a directed acyclic graph (DAG), where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of edges. We use $|\mathcal{V}|$ and $|\mathcal{E}|$ for the number of vertices and edges, respectively. The notation $[v_1, \cdots, v_{|\mathcal{V}|}]$ denotes a sequence of vertices which follow a topological sorted order σ_v. We use $\sigma_v(v_i)$ for the position of v_i in the topological order σ_v. The notation $v_i \prec_{\sigma_v} v_j$ means that vertex v_i precedes v_j in σ_v. Given an edge e, we use $V_s(e)$ and $V_t(e)$ for the source and target vertex of e, respectively. Given a topological sorted order σ_v of vertices, we can define a topological order $\sigma_e : [e_1, \cdots, e_{|\mathcal{E}|}]$ of edges w.r.t. σ_v where $V_s(e_i) \preceq_{\sigma_v} V_s(e_j)$ for $i < j$. A path p is a sequence $[e_{p(1)}, \dots, e_{p(|p|)}]$ of edges where $V_t(e_{p(i)}) = V_s(e_{p(i+1)})$ for $1 \leq i \leq |p| - 1$. A path $[e_{p(1)}, \dots, e_{p(i)}]$ is a *prefix* of p if $i \leq |p|$ and $[e_{p(i)}, \dots, e_{p(|p|)}]$ is a *suffix* of p if $i \geq 1$. We assume each edge e_i has a real-valued weight w_i. The length of a path is the sum of weights of each edge along it. Given two vertices s and t, we use $\mathcal{P}_{st}$ for the set of all paths from s to t, and $\mathcal{E}_{st}$ for the set of edges where each edge e satisfies that $s \preceq_{\sigma_v} V_s(e)$ and $V_s(e) \prec_{\sigma_v} t$.

2.2 Decision Diagrams

Let $f(x_1, \cdots, x_n)$ be a *multi-valued function* where each variable x_i has a finite domain D_i and f returns $\{\top, \bot\}$. We use $|D_i|$ for the size of D_i. *Multi-valued decision diagrams (MDDs)* [Kam *et al.*, 1998] are a directed acyclic graph that compactly represents a multi-valued function. To avoid confusion, we use the terminologies *node* and *arc* for decision diagrams and *vertex* and *edge* for general DAG. An MDD consists of two types of nodes: *terminal* nodes and *internal* nodes. A terminal node is labeled by $\top$ or $\bot$, signifying the Boolean value of a multi-valued function. An internal node n is labeled by a variable $var(n)$ and has $|D_i|$ outgoing arcs where $var(n) = x_i$. We use $child_v(n)$ for the outing arc of n where $var(n) = x_i$ and $v \in D_i$. It means that the variable x_i is assigned to the element v. We also use $child_v(n)$ to denote the successor node of n to which the arc $child_v(n)$ points. We say nodes labeled by the same variable are at the same *level*. For simplicity, we sometimes use a node to denote the subgraph rooted by it and to represent a multi-valued function.

An MDD is *ordered* if the labels of internal nodes in each path from the root node to a terminal node follow the same order. An MDD is *reduced* if it contains neither isomorphic sub-graphs nor nodes where all outgoing arcs points to the same node. An ordered and reduced MDD was proven to be a canonical form of multi-valued functions, that is, any multi-valued function has a unique ordered and reduced MDD w.r.t. a given variable order. In the rest of this paper, we assume that every MDD is ordered and reduced.

We say a multi-valued function f is a *Boolean function*, iff the domain of each variable of f contains only two elements $\{\top, \bot\}$. Similarly to MDDs, *binary decision diagrams (BDDs)* [Bryant, 1986] are a graph-based representation of Boolean functions where each internal node n has exactly two outgoing arcs $child_\top(n)$ and $child_\bot(n)$.

Example 1. *Figure 1(a) depicts a DAG G with 7 vertices and 12 edges. A logical constraint requires valid paths on G to include edge e_6 or both edges e_5 and e_{11} simultaneously. We consider each edge e_i as variable and use the Boolean function $f(e_1, \cdots, e_{12}) : e_6 \vee (e_5 \wedge e_{11})$ to represent the above constraint. Figure 1(b) shows a BDD representing f where internal nodes are represented by circles and terminal nodes are represented by rectangles. The variable of nodes is shown on the left. Each arc $child_\top(n)$ (resp. $child_\bot(n)$) of node n is a solid line marked with the assignment $\top$ (resp. $\bot$). The BDD representing f contains only nodes labeled by three variables e_5, e_6 and e_{11}.*

The logical constraint can be defined by a multi-valued function $g(v_1, \cdots, v_7)$. The domain of v_i is the set of the outgoing edges of v_i together with the special element $\varnothing$ indicating that v_i is never traversed. For example, $D_2 = \{\varnothing, e_3, e_4, e_5, e_6\}$ and $D_5 = \{\varnothing, e_{10}, e_{11}\}$. When $v_2 = e_5$ and $v_5 = e_{11}$, or $v_2 = e_6$, the function g returns $\top$. Figure 1(c) illustrates an MDD representing the multi-valued function g where each variable v_i denotes a vertex of the DAG G shown in Figure 1(a). □

2.3 Optimal Path Problem over DAGs

Given a weighted DAG, the optimal path problem aims to find a path from a source vertex to a target vertex on the given DAG such that its length is minimal or maximal. Numerous combinatorial optimization problems, such as the 0-1 knapsack problem, the edit distance problem, and the Viterbi path problem, can be formalized as optimal path problems over DAGs, especially those that can be solved with dynamic programming (DP). In this paper, we focus on *the constrained optimal path problem over DAGs* that is a variant of optimal path problem.

Definition 1. Given a DAG $G = (\mathcal{V}, \mathcal{E})$, a source vertex s, a target vertex t and a global constraint f on the paths from s to t, the constrained optimal path problem over DAGs aims to find the shortest or longest path from s to t that satisfies f.

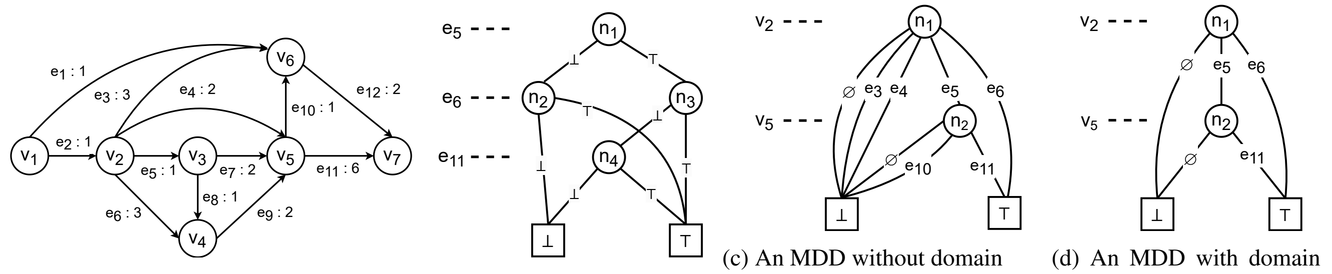
We remark that in the field of constraint programming, the concept of global constraint is used to denote a constraint that can involve an arbitrary set of variables. In this paper, we use the terminology "global constraint" to denote the given logical constraint in the constrained optimal path problem.

3 BDD-Constrained Search

In this section, we first introduce a unified approach for the constrained optimal path problem over DAGs, namely *BDD-constrained search (BCS)*, proposed in [Nishino *et al.*, 2015]. Then, we discuss the deficiencies of BCS.

3.1 The Main Algorithm

The shortest path from the source vertex s to the target vertex t that goes across an intermediate vertex v is the concatenation of the shortest path in $\mathcal{P}_{sv}$ and the shortest path in $\mathcal{P}_{vt}$. This property makes it simple to design an efficient algorithm for the shortest path problem over DAGs with time complexity $O(|\mathcal{E}|)$ [Cormen *et al.*, 2022]. The algorithm first constructs a topological order σ_v of vertices for the given graph G, and then searches the shortest path



(a) A weighted DAG G

(b) A BDD

(c) An MDD without domain reduction

(d) An MDD with domain reduction

Figure 1: A weighted DAG G and three DD representations of the global constraint.

Algorithm 1: BDD-Constrained Search

```
Input: $G = (\mathcal{V}, \mathcal{E})$: a weighted DAG;
       $r$: the root node of the BDD representing the constraint;
       $s$: the source vertex of $G$;
       $t$: the target vertex of $G$.
Output: $p$: the shortest path from $s$ to $t$ under the constraint;
        $l$: the length of $p$.
1  $\sigma_v \leftarrow$ a topological sorted order of vertices
2  $\sigma_e \leftarrow$ a topological sorted order of edges w.r.t. $\sigma_v$
3  foreach $v \in \mathcal{V}$ do
4      $L[v] \leftarrow \emptyset$ and $B[v] \leftarrow \emptyset$
5  $L[s][r] \leftarrow 0$
6  foreach $e_i$ taken in the topological order $\sigma_e$ do
7      $u \leftarrow V_s(e_i)$ and $v \leftarrow V_t(e_i)$
8      foreach $(n, l) \in L[u]$ do
9          $n' \leftarrow$ followBDD$(e_i, n)$
10         if $n' = \perp$ then continue ;
11         if there is no $(n', l') \in L[v]$ or
               $L[v][n'] > l + w_i$ then
12             $L[v][n'] \leftarrow l + w_i$ and $B[v][n'] \leftarrow (e_i, n)$
13 $l \leftarrow L[t][\top]$ and $(e, n) \leftarrow B[t][\top]$
14 $p \leftarrow \emptyset$
15 while $V_s(e) \neq s$ do
16     append the edge $e$ to $p$
17     $(e, n) \leftarrow B[V_s(e)][n]$
18 append the edge $e$ to $p$
```

among vertices arranged in σ_v. For each intermediate vertex v between s and t, we only keep the shortest path p from s to v since p is a prefix of the shortest path from s through v to t.

Unfortunately, the aforementioned property does not hold in the constrained optimal path problem.

Definition 2. Let v be an intermediate vertex between the source vertex s and the target vertex t and f the global constraint on $\mathcal{P}_{st}$. Let p be a prefix path in $\mathcal{P}_{sv}$. A valid suffix path $q \in \mathcal{P}_{vt}$ w.r.t. p is such that the concatenation of p and q satisfies f. We use $\mathcal{R}_p$ for the set of valid suffix path w.r.t. p on $\mathcal{P}_{vt}$.

By selecting different prefix paths from $\mathcal{P}_{sv}$, the sets of

Algorithm 2: `followBDD`

```
Input: $e$: an edge;
       $n$: a BDD node representing the local constraint $f_p$ w.r.t. the path $p$.
Output: $n'$: the BDD node representing the local constraint $f_q$ w.r.t. the path $q$ where $q$ is the concatenation of $p$ and $e$.
1  if $n$ is a terminal node then $n' \leftarrow n$;
2  while $var(n) \prec_{\sigma_e} e$ do        /* $\sigma_e$ is the topological order of edges */
3      $n \leftarrow child_\perp(n)$
4      if $n$ is a terminal node then $n' \leftarrow n$;
5  $n' \leftarrow n$
6  if $var(n) = e$ then $n' \leftarrow child_\top(n)$ ;
```

valid suffix paths on $\mathcal{P}_{vt}$ vary. Hence, the shortest path in $\mathcal{P}_{sv}$ may not be a prefix of the shortest path from s through v to t that satisfies the given logical constraint. We illustrate this with the following example.

Example 2. *Figure 1(a) illustrates a weighted DAG G and 1(b) depicts a BDD representing the global constraint* $f : e_6 \vee (e_5 \wedge e_{11})$. *Suppose that we want to obtain the shortest path from* v_1 *to* v_7 *satisfying the constraint* f. *The set* $\mathcal{P}_{v_1 v_5}$ *of prefix paths contains paths* $p_1 : [e_2, e_5, e_7]$ *and* $p_2 : [e_2, e_6, e_9]$ *while the set* $\mathcal{P}_{v_5 v_7}$ *of suffix paths contains* $p_3 : [e_{10}, e_{12}]$ *and* $p_4 : [e_{11}]$. *If the prefix* p_1 *is chosen, then only the suffix* p_4 *can be chosen so that the concatenation of* p_1 *and* p_4 *is a valid path. Nevertheless, if we choose* p_2*, then both* p_3 *and* p_4 *are able to concatenate with it in that* p_2 *already satisfies the global constraint. The paths* p_1 *and* p_2 *have length* 4 *and* 6*, respectively. The path* $[e_2, e_6, e_9, e_{10}, e_{12}]$ *of length* 9 *with prefix* p_2 *however is shorter than the path* $[e_2, e_5, e_7, e_{11}]$ *of length* 10 *with prefix* p_1. □

A simple extension to the algorithm for the shortest path problem is to save all of the possible paths from s to every intermediate vertex v. This extended algorithm however requires time and space exponential to $|\mathcal{E}|$ and is inefficient.

In order to address the space explosion issue, BCS groups equivalent paths together so as to prune the search space of paths. Two paths p and q are *equivalent*, if they reach the same vertex (that is, $p, q \in \mathcal{P}_{sv}$ for some vertex v) and the two sets of valid suffix paths w.r.t. p and q are identical (that is, $\mathcal{R}_p = \mathcal{R}_q$). If q is longer than p, we only keep the shorter one p for the vertex v and the set of valid suffix paths $\mathcal{R}_p$ since the valid shortest path of $\mathcal{P}_{st}$ does not contain q.

The global constraint f is a logical representation of the set of valid paths of $\mathcal{P}_{st}$. Given a prefix path p on $\mathcal{P}_{sv}$, the *local constraint* f_p w.r.t. p represents the set of valid suffix paths w.r.t. p. If f is represented by a Boolean function, then the subfunction f_p can be obtained by replacing the variable e_i in the constraint f by $\top$ (resp. $\bot$) if $e_i \in p$ (resp. $e_i \notin p$) for every edge $e_i \in \mathcal{E}_{sv}$.

Example 3. *As illustrated in Example 2, path p_1 is $[e_2, e_5, e_7]$. Path p_1 connect vertex v_1 and v_5. The edges in $\mathcal{E}_{v_1 v_5}$ that occur as variables in the global constraint f are e_5 and e_6. Since $e_5 \in p_1$ and $e_6 \notin p_1$, we replace e_5 (resp. e_6) by $\top$ (resp. $\bot$) in f and obtain the local constraint $f_{p_1} = \bot \vee (\top \wedge e_{11}) \equiv e_{11}$, meaning that any suffix path that contains edge e_{11} concatenated with p_1 is valid.* □

The main idea of BCS is to combine equivalent paths from s to v into a group and to save groups of equivalent paths as Boolean functions. BCS keeps only the shortest path for each intermediate vertex v and each logically different local constraint on $\mathcal{P}_{vt}$. Two important components of BCS are (1) deriving the local constraint f_p w.r.t. a prefix path p and (2) combining equivalent local constraints f_q and f_q for every equivalent prefix paths $p, q \in \mathcal{P}_{sv}$. The above two components are accomplished via utilizing BDDs.

The main algorithm of BCS, shown in Algorithm 1, maintains two sets of tables L and B where L stores the length of the shortest path while B stores the backtracking information that is used to obtain the constrained shortest path. Both sets L and B have $|\mathcal{V}|$ tables $L[v]$ and $B[v]$ for $v \in \mathcal{V}$, respectively. The tables $L[v]$ and $B[v]$ are a mapping that takes BDD nodes as keys. Suppose that the current shortest path p from s to v with a local constraint f_p on $\mathcal{P}_{vt}$ is l. The local constraint f_p is represented by a BDD node n. The expression $L[v][n] = l$ (or $(n, l) \in L[v]$) indicates that the path p has length l. The expression $B[v][n] = (e, n')$ means that (1) the last edge of p is e; and (2) the BDD node n' denotes the local constraint w.r.t. the path q, which is the prefix of p without only the edge e.

Algorithm 1 works as follows. It first constructs two topological sorted orders σ_v of vertices and σ_e of edges, initializes every tables $L[v]$ and $B[v]$ as empty table for each vertex v and sets $L[s][r]$ to be 0 where r is the root node of the BDD representing the global constraint (lines 1 - 5). Then, it traverses each edge e_i in the topological sorted order σ_e (lines 6 - 12). Let u and v be the source and target vertex of e_i, respectively. Suppose that the length of the current shortest path p from s to u with the local constraint f_p w.r.t. p represented by n is l (that is, $(n, l) \in L[u]$). We obtain a path q by extending the path p with the edge e_i. The BDD node n' denoting the local constraint f_q w.r.t. q is generated via the subprocedure `followBDD`(e_i, n), which will be elaborated later. When n' is a terminal node $\bot$, denoting that no valid suffix path w.r.t. q exists, Algorithm 1 continues to the next pair. If a shorter path from s to v with the local constraint f_q is found, then the table $L[v]$ and $B[v]$ are simultaneously updated (lines 11 - 12). After traversing all edges, $L[t][\top]$ is the length of the shortest path that satisfies the global constraint, and we can retrieve the edges along the shortest path via the set B of tables (lines 13 - 18).

The subprocedure `followBDD` is illustrated in Algorithm 2. It aims to derive the local constraint f_q w.r.t. the path q from the local constraint f_p w.r.t. p and an edge e where the path q is the concatenation of p and e. It takes the edge e and a BDD node n denoting the local constraint f_p as input, and outputs the BDD node n' denoting the local constraint f_q. If n is a terminal node $\bot$, then there exists no valid suffix path w.r.t. p and hence n' is also $\bot$ (lines 1 and 4). In the case where n is a terminal node $\top$, any suffix path is valid and hence n' is also $\top$ (lines 1 and 4). The variable order of the BDD that represents the global constraint follows the topological order σ_e of edges. The variable $var(n)$ of the BDD node n is an edge e'. Every edge e'' that comes before the edge e and after e' in σ_e is not chosen. Therefore, we continue moving to the $\bot$-child of the BDD node n until we reach a node whose variable either follows or equals to e (line 3). Finally, if the variable of n is e, we move to its $\top$-child since the current path p contains e (line 6). We show the run of Algorithm 1 by the following example.

Example 4. *We continue Example 2. The BDD representing the global constraint $f : e_6 \vee (e_5 \wedge e_{11})$ is illustrated in Figure 1(b), and its root node is n_1. Initially, the table $L[v_1]$ contains only one pair $(n_1, 0)$. Then, Algorithm 1 traverses edges in the topological order $[e_1, \cdots, e_{12}]$. Edge e_1 starts at v_1 and ends at v_6. For the pair $(n_1, 0) \in L[v_1]$, the subprocedure* `followBDD`(e_1, n_1) *returns n_1. Since $L[v_6][n_1]$ is not computed yet, we add an element $(n_1, 1)$ (resp. $(n_1, (e_1, n_1))$) to table $L[v_6]$ (resp. $B[v_6]$). Algorithm 1 handles other edges in the same way. Finally, the length of the shortest path satisfying the global constraint is 9, stored in $L[v_7][\top]$. We recover the shortest path $[e_2, e_6, e_9, e_{10}, e_{12}]$ via backtracking the entry $B[v_7][\top] = (e_{12}, \top)$.* □

3.2 Deficiencies of BDD-Constrained Search

BCS reduces many redundant searches on paths via grouping the equivalent paths together and maintaining equivalent local constraints w.r.t. prefix paths in BDDs. Hence BCS performs well in solving the constrained optimal path problem over DAGs. However, we observe that BCS still involves redundant operations and illustrate this in the following.

Let e be an edge from vertex u to vertex v and σ_v a topological order of vertices. We say an edge e' is *e-forbidden* w.r.t. σ_v, if $u \preceq_{\sigma_v} V_s(e')$ and $V_s(e') \prec_{\sigma_v} v$. When edge e is chosen, any e-forbidden edge cannot be chosen from the DAG. However, when deriving the local constraint w.r.t. the path p with the last edge e, BCS still considers these e-forbidden edges and hence involves the following three types of redundant operations: (1) It processes e-forbidden edges that occurs in the BDD; (2) It generates an incorrect local constraint w.r.t. p, that is, a Boolean function represented by a BDD does not conform to the local constraint w.r.t. p; and (3) The two tables $L[v]$ and $B[v]$ include unnecessary information due to the incorrect local constraint.

Example 5. *We continue with Example 2. The source vertex of edge e_1 is v_1 and the target vertex is v_6. Clearly, e_1 is also a path from v_1 to v_6. The topological order*

Algorithm 3: `followMDD`

```
Input: $e$: an edge;
       $n$: an MDD node representing the local
            constraint $g_p$ w.r.t. the path $p$.
Output: $n'$: the MDD node representing the local
             constraint $g_q$ w.r.t. the path $q$ where $q$ is
             the concatenation of $p$ and $e$.
1 if $n$ is a terminal node then $n' \leftarrow n$;
2 if $var(n) = V_s(e)$ then $n \leftarrow child_e(n)$ ;
3 while $var(n) \prec_{\sigma_v} V_t(e)$ do    /* $\sigma_v$ is the
    topological order of vertices */
4     $n \leftarrow child_{\varnothing}(n)$
5     if $n$ is a terminal node then $n' \leftarrow n$;
6 $n' \leftarrow n$
```

σ of vertices is $[v_1, v_2, v_3, v_4, v_5, v_6, v_7]$. It follows that the vertices, which are after v_1 and before v_6 w.r.t. σ, are $\{v_1, v_2, v_3, v_4, v_5\}$. The set of e_1-forbidden edges is $\{e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}\}$. Edge e_2 has the same source vertex v_1 as e_1. Any path contains at most one outgoing edge for each vertex. When e_1 is chosen, the remaining path does not contain edge e_2. Similarly, the remaining path does not go through the vertices v_2, v_3, v_4 and v_5. Hence the outgoing edges of the above vertices cannot be chosen.

We hereafter show the redundant operations of BCS. After initialization, Algorithm 1 first handles edge e_1. In the case where e_1 is chosen, we obtain a path $p : [e_1]$. Any e_1-forbidden edge, including e_5 and e_6, cannot be chosen. In addition, the global constraint f requires a valid path to contain edge e_6 or both edges e_5 and e_{11} simultaneously. Therefore any path with prefix p fails to satisfy f and the local constraint w.r.t. p is $\perp$.

Since e_1 precedes e_5 in the topological order of edges, Algorithm 1 invokes the subprocedure `followBDD`(e_1, n_1) *that yields the BDD node n_1, and then computes the two entries $L[v_6][n_1]$ and $B[v_6][n_1]$. In fact, the Boolean function $e_6 \vee (e_5 \wedge e_{11})$ represented by n_1 is an incorrect local constraint w.r.t. p. In addition, for each path $q \in \mathcal{P}_{v_1 v_6}$, there is no local constraint w.r.t. q represented by n_1. As a result, the two tables $L[v_6]$ and $B[v_6]$ contain unnecessary entries $L[v_6][n_1]$ and $B[v_6][n_1]$ for the incorrect local constraint. The vertex has only one outgoing edge e_{12}. In the subsequent subprocedure* `followBDD`(e_{12}, n_1)*, it is inevitable to assign the Boolean value $\perp$ to n_1 and n_2 (line 3), meaning that e_5 and e_6 are not chosen in the path whose prefix is p.* □

4 MDD-Constrained Search

As pointed out in the previous section, BCS involves three types of redundant operations. In this section, we introduce the proposed algorithm, MDD-constrained search (MCS), reducing the redundant operations of BCS. Then, we propose two improvements: domain reduction and DAG partition to represent constraints in more compact MDDs.

4.1 MDD-Constrained Search

In order to mitigate the deficiencies of BCS, we design a novel search algorithm, namely *MDD-constrained search*, which adopts the same main search framework illustrated in Algorithm 1 as BCS. The main idea is to represent the global constraint in multi-valued function where each variable is a vertex in a DAG and the domain of each variable is the outgoing edges of its corresponding vertex along with the special element $\varnothing$. Furthermore, we make use of MDDs as the representation of global constraints instead of BDDs. Finally, we develop a subprocedure `followMDD` instead of `followBDD` in BCS.

We begin by elaborating how to gain the local constraint in multi-valued function. Suppose that p is a prefix path of $\mathcal{P}_{sv}$ and g is a multi-valued function representing the global constraint on $\mathcal{P}_{st}$. The local constraint g_p w.r.t. p is obtained as follows: for every edge $e_i \in \mathcal{E}_{sv}$, the value of the variable $V_s(e_i)$ in the constraint g is assigned to be e_i if $e_i \in p$ and to be $\varnothing$ if there is no edge $e_j \in \mathcal{E}_{sv}$ s.t. $e_j \in p$ and $V_s(e_i) = V_s(e_j)$.

The subprocedure `followMDD`, illustrated in Algorithm 3, takes the edge e and the MDD node n representing the local constraint g_p as input, and outputs the MDD node n' representing the local constraint g_q. The MDD node n' is also a terminal node if n is (line 1). When n is $\perp$ (resp. $\top$), q concatenated with any suffix path is invalid (resp. valid). Any vertex that comes between u and v in the topological order of vertices is not traversed. Therefore, we go to the arc $child_{\varnothing}(n)$ of the MDD node n until we reach a terminal node or a node whose variable either follows or equals to v (lines 3 - 5).

Suppose that a path p connects vertex u to vertex v, with e as its last edge. Due to the use of multi-valued function representing local constraint, MCS has the following advantages: (1) It does not consider all e-forbidden edges but simply assigns $\varnothing$ to each source vertex of them; (2) It always obtains the correct local constraint w.r.t. the given path p, which confirmed by the Theorem 1; and (3) There is no unnecessary information caused by incorrect local constraints in tables $L[v]$ and $B[v]$. We show how MCS reduces the redundant operations of BCS with the following example.

Example 6. *We continue with Example 5. The MDD representing the global constraint $g : (v_2 = e_5 \wedge v_5 = e_{11}) \vee v_2 = e_6$ is illustrated in Figure 1(c). We select the first edge e_1 and obtain a path $p : [e_1]$. The target vertex of e_1 is v_6. The local constraint on $\mathcal{P}_{v_6 v_7}$ w.r.t. p is $\perp$.*

To obtain the local constraint w.r.t. the path p, MCS invokes the subprocedure `followMDD`(e_1, n_1)*. Since v_6 is a vertex that follows v_2 in the topological order of vertices,* `followMDD` *assigns $\varnothing$ to the MDD node n_1 and reaches the $\perp$-node. The two edges e_5 and e_6 are e_1-forbidden, no valid suffix path when e_1 is chosen. Hence, $\perp$ is the correct local constraint w.r.t. p and there is no unnecessary entries stored in $L[v_6]$ and $B[v_6]$. In addition, MCS does not consider both e_5 and e_6, expect for assigning $\varnothing$ to one vertex v_2.* □

Theorem 1. *Let e be an edge from vertex u to vertex v, and p and q two paths where q is the concatenation of p and e. Let n be the MDD node representing the correct local constraint w.r.t. p. The subprocedure* `followMDD(e, n)` *returns the MDD node n' representing the correct local constraint w.r.t. q.*

Proof. By assumption, the MDD node n represents the correct local constraint g_p w.r.t. p, that is, for every edge $e_i \in$

Algorithm 4: `followMDD′`

```
Input: $e$: an edge;
       $n$: an MDD node representing the local constraint $g_p$ w.r.t. the path $p$.
Output: $n'$: the MDD node representing the local constraint $g_q$ w.r.t. the path $q$ where $q$ is the concatenation of $p$ and $e$.
1  if $n$ is a terminal node then $n' \leftarrow n$;
2  if $var(n) = V_s(e)$ then
3      if $e$ in the domain of $var(n)$ then
4          $n \leftarrow child_e(n)$
5      else
6          $n \leftarrow child_\varnothing(n)$
7  while $var(n) \prec_{\sigma_v} V_t(e)$ do    /* $\sigma_v$ is the topological order of vertices */
8      $n \leftarrow child_\varnothing(n)$
9      if $n$ is a terminal node then $n' \leftarrow n$;
10 $n' \leftarrow n$
```

$\mathcal{E}_{su}$, the value of the variable $V_s(e_i)$ in the constraint g is assigned. The subprocedure `followMDD(e, n)` handles every vertex that comes after u and before v (lines 2 - 5 in Algorithm 3). Hence, for each edge $e_j \in \mathcal{E}_{uv}$, the variable $V_s(e_j)$ is assigned. Thus n' denotes the correct local constraint w.r.t. q. □

4.2 Domain Reduction

In the previous subsection, we showed the advantage of using multi-valued functions to represent constraints over Boolean functions. Each vertex in general has a huge number of outgoing edges in a large-scale DAG. Hence, the domain of each vertex in the multi-valued function has many elements and each node in the MDD has a great number of outgoing arcs. It takes much time to create the MDD representing the global constraint.

To solve the above problem, we propose domain reduction for multi-valued function. An edge e_i with source vertex v is *irrelevant* to a global constraint g, if $g(e_1, \cdots, \varnothing, \cdots, e_n) = g(e_1, \cdots, e_i, \cdots, e_n)$ for every outgoing edge e_i of each vertex v_i. To reduce the domain of vertex v, all irrelevant edges of v can be combined into the special element $\varnothing$. The notation $v = \varnothing$ indicates that either the outgoing edge of v is one of the irrelevant edges or v is never traversed. For example, Figure 1(c) represents $g : (v_2 = e_5 \wedge v_5 = e_{11}) \vee v_2 = e_6$ as the MDD without domain reduction while Figure 1(d) depicts the MDD with domain reduction. In the former MDD, the node with label v_2 has 5 outgoing arcs, whereas in the latter, it has only 3. As e_3 and e_4 does not occur in g and hence is irrelevant, the reduced domain of the variable v_2 is $\{\varnothing, e_5, e_6\}$.

We also adjust the subprocedure `followMDD`. The adjusted subprocedure, namely `followMDD′`, illustrated in Algorithm 4, is the same as `followMDD` except the following. If an edge e is in the reduced domain of the vertex $var(n)$, we choose the $child_e(n)$; otherwise, we choose $child_\varnothing(n)$ (lines 3 - 6). The subprocedure `followMDD′` has only one more step than `followMDD`, which is to determine whether an edge is in the domain of a vertex (lines 3 - 6). This step has no influence on the variable assignment process. Therefore, `followMDD′(e, n)` always outputs the correct local constraints as well.

Theorem 2. *Let e be an edge from vertex u to vertex v, and p and q two paths where q is the concatenation of p and e. Let n be the MDD node representing the correct local constraint w.r.t. p. The subprocedure* `followMDD′(e, n)` *returns the MDD node n' representing the correct local constraint w.r.t. q.*

4.3 DAG Partition

In 0-1 knapsack and Viterbi path problems, we can partition the set of vertices into mutually exclusive groups so as to reduce the number of variables in the global constraint. For example, in the DAG representing a Viterbi path problem instance, some vertices have the same meaning that assigns different hidden states to the same specific output o. In this case, these vertices can be combined into a group. Meanwhile, the outgoing edges of a vertex from the group can also be grouped together, since they represent the operation of choosing one specific state for o simultaneously. Thus, the constraint of assigning the state to o can be reduced into a constraint on the edge group in the Viterbi path problem.

Let $G = (\mathcal{V}, \mathcal{E})$ be a DAG, s the source vertex, and t the target vertex. Suppose that we have a topological order σ_v of vertices and a topological order σ_e of edges. Partitioning $\mathcal{V}$ into a collection $\mathcal{V}_\mathcal{G}^1, \cdots, \mathcal{V}_\mathcal{G}^n$ satisfies the following two conditions: (1) for every $1 \leq i \leq n-1$, every two vertices $v \in \mathcal{V}_\mathcal{G}^i$ and $v' \in \mathcal{V}_\mathcal{G}^{i+1}$, we have $v \prec_{\sigma_v} v'$; and (2) any path from s to t goes through at most one vertex of each group $\mathcal{V}_\mathcal{G}^i$ for $1 \leq i \leq n$. Similarly, partitioning $\mathcal{E}$ into a collection $\mathcal{E}_\mathcal{G}^1, \cdots, \mathcal{E}_\mathcal{G}^m$ satisfies : (1) for every $1 \leq i \leq m-1$, every two edges $e \in \mathcal{E}_\mathcal{G}^i$ and $e' \in \mathcal{E}_\mathcal{G}^{i+1}$, we have $e \prec_{\sigma_e} e'$; and (2) any path from s to t contains at most one edge of each group $\mathcal{E}_\mathcal{G}^i$ for $1 \leq i \leq m$.

In general, the domain of each group $\mathcal{V}_\mathcal{G}^i$ consists of each outgoing edge group of each vertex $v \in \mathcal{V}_\mathcal{G}^i$ together with the special element $\varnothing$. In 0-1 knapsack and Viberti path problems, any path that starts at s and ends with t has to go through one vertex from each vertex group. In this case, the domain of the group $\mathcal{V}_\mathcal{G}^i$ does not contain the special element $\varnothing$ if all outgoing edge groups of each vertex $v \in \mathcal{V}_\mathcal{G}^i$ are contained in the constraint.

5 Complexity Analysis

In this section, we analyze the time and space complexity of both search algorithms: BCS and MCS.

We first introduce two concepts for complexity: *width* and *forbidden nodes* and elaborate them with an example.

Definition 3. The width of a MDD is the maximum number of nodes pointed to by the edges in a cut of the MDD at a level of nodes.

Definition 4. Let e be an edge and σ_e a topological order of edges. Let $\mathcal{E}_e$ be a set of edges where each edge e' of $\mathcal{E}_e$ is such that (1) $V_t(e') = V_t(e)$ and that (2) e is e'-forbidden.

Let e'' be the minimal element of $\mathcal{E}_e$ w.r.t. σ_e. Let F_B be a BDD representing the global constraint. The forbidden node of level e in F_B is a node that is between the level e'' and e.

Example 7. *Figure 1(a) depicts a DAG G with a topological order σ_e of edges. Figure 1(b) illustrates a BDD F_B representing the global constraint $f = e_6 \vee (e_5 \wedge e_{11})$, where only levels e_5, e_6 and e_{11} contain nodes. The number of nodes pointed to by the edges in the cuts of levels e_5, e_6, and e_{11} is* 2, 3, *and* 2, *respectively. Hence the width of F_B is* 3.

The set $\mathcal{E}_{e_8}$ contains one edge e_6, and the minimal element of $\mathcal{E}_{e_8}$ w.r.t. σ_e is also e_6. Two BDD nodes n_2 and n_3 are between level e_6 and e_8 in F_B. Hence, both n_2 and n_3 are forbidden nodes of level e_8. □

The time and space complexity of BCS are given in the following theorem.

Theorem 3. *Let $G = (\mathcal{V}, \mathcal{E})$ be a DAG and F_B a BDD representing the global constraint. Let w_B be the width of F_B and γ the maximum number of forbidden nodes of a level in F_B. The time complexity of BCS is $O(|\mathcal{E}|^2 \cdot (w_B + \gamma))$, and the space complexity is $O(|\mathcal{V}| \cdot (w_B + \gamma))$.*

Proof. For a vertex v, the sizes of $L[v]$ and $B[v]$ depend on the number of BDD nodes that are used as keys. Let $\mathcal{E}_v$ be a set of edges where $\mathcal{E}_v = \{e' \mid V_t(e') = v\}$. Let σ_e be a topological order of edges and e the maximal element of $\mathcal{E}_v$ w.r.t. σ_e. We first focus on correct local constraints. The two tables $L[v]$ and $B[v]$ have at most as many keys as nodes pointed to by the edges that intercepts a cut of F_B at level e. The number of these nodes is at most w_B. We now consider incorrect local constraints. For the vertex v, BCS obtains at most as many different incorrect local constraints as forbidden nodes of level e. The number of these forbidden nodes is at most γ. As a result, $L[v]$ and $B[v]$ have at most $w_B + \gamma$ keys. Thus the space complexity is $O(|\mathcal{V}| \cdot (w_B + \gamma))$.

The update process in Algorithm 1 (lines 6 - 12) performs at most $|\mathcal{E}| \cdot (w_B + \gamma)$ times. Each update process invokes the subprocedure `followBDD`, which performs the traverse process in Algorithm 2 (lines 2 - 4) at most $|\mathcal{E}|$ times. Therefore, the time complexity is $O(|\mathcal{E}|^2 \cdot (w_B + \gamma))$. □

We remark that the time and space complexity of BCS given in this paper is different from that in [Nishino *et al.*, 2015], where the time and space complexity are $O(|\mathcal{E}| \cdot w_B)$. This is because Nishino *et al.* [2015] did not consider incorrect local constraints and the time complexity of the subprocedure `followBDD`. The space complexity is $O(|\mathcal{V}| \cdot (w_B + \gamma))$ rather than $O(|\mathcal{E}| \cdot w_B)$ since the numbers of $L[v]$ and $B[v]$ are equal to the number of vertices.

Finally, we provide the time and space complexity of MCS.

Theorem 4. *Let $G = (\mathcal{V}, \mathcal{E})$ be a DAG, F_M a MDD representing the global constraint and w_M the width of F_M. The time complexity of MCS is $O(|\mathcal{E}| \cdot |\mathcal{V}| \cdot w_M)$, and the space complexity of MCS is $O(|\mathcal{V}| \cdot w_M)$.*

Proof. Let v be a vertex in the given DAG and u the previous vertex of v in the topological order σ_v. Two tables $L[v]$ and $B[v]$ have at most as many keys as nodes pointed to by the edges that intercepts a cut of F_M at level u, and the number of these nodes is at most w_M. As a result, the space complexity of MCS is $O(|\mathcal{V}| \cdot w_M)$. The update process in Algorithm 1 (lines 6 - 12) for MCS performs at most $|\mathcal{E}| \cdot w_M$ times. Since each update process invokes the subprocedure `followMDD` and the traverse process in `followMDD` (lines 2 - 5 in Algorithm 3) executes at most $|\mathcal{V}|$ times, the time complexity of MCS is $O(|\mathcal{E}| \cdot |\mathcal{V}| \cdot w_M)$.

Compared with `followMDD`, `followMDD′` only has one more step that determines if an edge is in the domain of a vertex (lines 3 - 6 in Algorithm 4). This process executes at most once in each `followMDD′`. As a result, MCS with domain reduction has the same time and space complexity as MCS without domain reduction. □

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{V}_\mathcal{G}\rvert$	$\lvert\mathcal{E}_\mathcal{G}\rvert$
DAG	5,979	53,364	-	-
Knapsack	188,198	367,047	202	401
Edit distance	15,128	43,459	-	-
Viterbi path	11,379	351,788	369	113,48

Table 1: Summary of Datasets.

Since MCS always obtains the correct local constraint w.r.t. a path (Theorems 1 and 2), the time and space complexity of MCS does not contain the parameter γ where γ is the maximum number of forbidden nodes at a level in the BDD representation of the global constraint. BDD treats edges as variables while MDD treats vertices as variables. Hence, the assignment process in `followBDD` performs at most $|\mathcal{E}|$ times, while that in `followMDD` executes at most $|\mathcal{V}|$ times. In most constrained optimal path problems, the number $|\mathcal{E}|$ of edges in DAGs is much larger than the number $|\mathcal{V}|$ of vertices. Although the width w_M of the MDD representation of the global constraint is not guaranteed to be smaller than the width w_B of BDD representation in theory, the experimental results show that w_M is always less than or equal to w_B. In summary, MCS has better time and space complexity than BCS.

6 Experiments

6.1 Settings and Datasets

As Nishino *et al.* [2015] did not provide the source code of BCS, we reproduce this algorithm. Both BCS and MCS are implemented in C++[3]. We use the CUDD package [Somenzi, 2015] to construct BDD and the Meddly library [Babar and Miner, 2010] to MDD. It is worth noting that our implementation of BCS has similar performance as the original BCS proposed in [Nishino *et al.*, 2015] due to simplicity of BCS illustrated in Algorithm 1. All experiments were conducted on a Linux machine with Intel Core i7-10700 2.90GHz CPU and 16 GB RAM.

We compare our proposed algorithm MCS to BCS on the following four categories of datasets: the shortest path problem over DAGs, the 0-1 knapsack problem [Kellerer *et al.*, 2004], the edit distance problem [Wagner and Fischer, 1974] and the Viberti path problem [Forney, 1973]. We remark that

[3]The source code and datasets are publicly available in https://github.com/someMing/MDD-constrained-search.

all datasets were mentioned in [Nishino *et al.*, 2015]. We contacted Nishino *et al.*, but they did not provide the datasets. We therefore download all datasets from the original source and preprocess the datasets according to the method in [Nishino *et al.*, 2015]. So the numbers of vertices and edges are a bit different from those in [Nishino *et al.*, 2015].

The information about the datasets is summarized in Table 1. The columns $|\mathcal{V}|$ and $|\mathcal{E}|$ represent the number of vertices and edges of the transformed DAGs for each problem, respectively. The columns $|\mathcal{V}_\mathcal{G}|$ and $|\mathcal{E}_\mathcal{G}|$ indicate the number of vertex groups and edge groups of DAGs via DAG partition for 0-1 knapsack problem and Viberti path problem, respectively.

6.2 Logical Constraints

We consider two types of global constraints: *Checkpoint* and *Disjunction* that are used in [Nishino *et al.*, 2015].

Checkpoint Constraints. Suppose that we have several disjoint subsets $\mathcal{E}_1, \cdots, \mathcal{E}_n$ of edges. The checkpoint constraint requires any valid path to include at least one edge from each set $\mathcal{E}_i$ for $1 \leq i \leq n$. The checkpoint constraints are generated via the pair of two parameters (n, ρ). The pair (n, ρ) means that we first randomly select edges with probability ρ and then randomly divide them into n sets of equal size. We generate three constraints using the parameters $(3, 0.01)$, $(5, 0.05)$, and $(8, 0.2)$ for each problem instance, and call them *Check1*, *Check2*, and *Check3*.

Disjunctive Constraints. Given a collection of pairs of edges, the disjunctive constraint requires any valid path cannot contain both edges from the same pair. The knapsack problem with disjunctive constraint is the disjunctively constrained knapsack problem (DCKP) [Yamada *et al.*, 2002]. The disjunctive constraints are generated via the pair of two parameters (k, ρ). The pair (k, ρ) means that we first select k-best edges with smaller (resp. greater) weight in the shortest (resp. longest) path problem, and generate all the possible pair from selected edges. Then we randomly select disjunctive pairs from them with probability ρ. We generate three constraints using the parameters $(50, 0.01)$, $(60, 0.7)$, and $(100, 0.8)$ for each problem instance, and mention them *Dis1*, *Dis2*, and *Dis3*.

6.3 Results

The experimental results are shown in Table 2. The columns "Nodes", "Arcs", and "Width" denote the number of nodes, the number of arcs, and the width of compiled decision diagrams, respectively. The columns "Entries" and "Total time" indicate the number of entries of sets L and B and the total running time (milliseconds) in both search algorithms BCS and MCS, respectively. As the numbers of nodes and entries in the MCS with and without domain reduction are identical, we display them together. The total time contains the time of compiling constraints into decision diagrams and of search the optimal path. Due to space limitation, we only present the total time here. In most instances, the compilation time accounts for a small proportion of the total time (1% - 26%), except the shortest path problem over DAGs and the edit distance problem with disjunctive constraints. On the one hand, the decision diagrams representing disjunctive constraints have a huge size, requiring more compilation time. On the other hand, the number of k-best edges defined by disjunctive constraints is much smaller compared to the number of edges, resulting in a large number of equivalent paths and less search time. The best data for columns "Nodes", "Arcs", "Width" , "Entries" and "Total time" is the smallest data among the three algorithms BCS and MCS with and without domain reduction. We highlight the best data in bold.

MDDs vs BDDs. We begin by comparing the size of BDDs and MDDs with domain reduction. The former has at least as many nodes as the latter. Except the shortest path problem over DAGs with Check2 and Check3 constraints and the edit distance problem with Check3 constraint, MDDs with domain reduction have fewer or an equal number of arcs compared to BDDs. The width of MDDs is at least as less as that of BDDs. This indicates that MDDs with domain reduction is a more compact form of representing logical constraints than BDDs.

The nodes and width in MDDs with and without domain reduction are identical. However, the number of arcs in MDDs with domain reduction is significantly less than that in MDDs without domain reduction. Especially for the shortest path problem over DAGs and the Viterbi path problem, domain reduction achieves at least 3.57 times compression rate. This indicates that domain reduction can effectively reduce the number of arcs in MDDs. As an exception, the number of arcs of both are identical on the Knapsack problem. This is because each vertex group in the DAGs has only two outgoing edge groups, hence there is no improvement via domain reduction.

MCS vs BCS. MCS is consistently faster than BCS, expect in the instance Knapsack with Dis2 constraint. This is because in this instance, BCS does not obtain many incorrect local constraints, resulting in close search times for both algorithms. However, MCS uses 439ms for compilation while BCS just needs 181ms. In most instances, MCS substantially outperforms BCS, for example, the running time of BCS is nearly 11.60 times that of MCS in the Viterbi path problem with Check3 constraint. In addition, MCS stores less entries of sets L and B than BCS. This is because MCS always derives the correct constraint w.r.t. a given path, resulting in a substantial reduction of redundant operations. These experimental results demonstrate that MCS can reduce the search space compared to BCS, leading to less memory requirements and running time.

We can observe that domain reduction is able to accelerate the MCS algorithm in 15 out of 24 instances. In these instances, most variables have domains with large sizes. Domain reduction significantly reduce the number of arcs in the MDDs, thereby decreasing the compilation time. For other 9 instances, the sizes of domains of variables are small. The compact improvement on sizes due to domain reduction can be neglected. Moreover, MCS with domain reduction requires additional operations to determine if the input edges is in the domain of the vertices, increasing the search time.

We do not make a comparison between MCS and CPLEX. It was demonstrated in [Nishino *et al.*, 2015] that BCS performs better than CPLEX on constrained optimal path

Dataset	Constraint	BCS					MCS						
		Nodes	Arcs	Width	Entries	Total time	Nodes	Width	Entries	w.o. domain reduction		with domain reduction	
										Arcs	Total time	Arcs	Total time
DAG	Check1	2,028	4,052	**8**	214,050	608.3	**1,878**	**8**	**33,243**	41,666	133.9	**4,018**	**121.6**
	Check2	42,148	**84,292**	**32**	1,018,807	11,168.7	**32,708**	**32**	**135,450**	627,011	5,797.6	86,545	**5,381.7**
	Check3	1,356,392	**2,712,780**	**256**	7,537,362	261,368.8	**683,697**	**256**	**1,020,471**	10,423,998	**173,900.3**	2,918,333	174,664.9
	Dis1	992	1,980	**193**	8,069	16.8	**896**	**193**	**5,995**	15,999	8.5	**1,908**	**7.7**
	Dis2	4,489	8,974	**154**	10,630	112.7	**3,735**	**154**	**6,033**	88,462	108.9	**8,245**	**40.3**
	Dis3	12,160	24,316	**230**	16,427	748.7	**8,907**	**230**	**6,252**	247,248	488.6	**21,078**	**136.8**
Edit distance	Check1	1,696	3,388	**8**	173,702	103.5	**1,690**	**8**	**109,488**	6,658	**50.9**	**3,386**	55.3
	Check2	34,464	68,924	**32**	836,424	760.8	**33,624**	**32**	**443,447**	132,006	368.2	**69,404**	**361.4**
	Check3	1,137,920	**2,275,836**	**256**	6,200,251	6,523.8	**1,013,827**	**256**	**3,250,573**	3,976,887	**6,453.8**	2,339,889	7,319.8
	Dis1	**268**	**532**	**65**	16,236	14.7	**268**	**65**	**15,658**	1,060	**8.5**	**532**	8.7
	Dis2	**5,894**	**11,784**	**203**	33,030	159.3	**5,894**	**203**	**23,705**	23,120	80.1	**11,784**	**53.1**
	Dis3	**13,083**	**26,162**	**252**	83,003	1,007.9	**13,083**	**252**	**47,991**	51,982	258.9	**26,162**	**237.5**
Knapsack	Check1	**10**	**16**	**4**	446,217	134.5	**10**	**4**	**444,369**	**16**	**128.9**	**16**	149.7
	Check2	141	278	32	2,038,107	477.0	**66**	**16**	**1,039,981**	**128**	**284.8**	**128**	392.5
	Check3	8,645	17,286	256	8,831,122	2,072.5	**412**	**32**	**2,686,862**	**820**	844.6	**820**	**659.3**
	Dis1	**432**	**860**	**128**	8,263,524	1,849.8	**432**	**128**	**8,121,206**	**860**	**1,785.7**	**860**	1,990.0
	Dis2	**5,399**	**10,794**	**204**	17,360,640	**4,092.6**	**5,399**	**204**	**16,941,349**	**10,794**	4,110.3	**10,794**	4,784.4
	Dis3	**12,533**	**25,062**	**261**	24,167,685	6,285.5	**12,533**	**261**	**23,565,084**	**25,062**	**5,891.9**	**25,062**	6,807.6
Viterbi path	Check1	410	816	**8**	96,939	101.0	**377**	**8**	**85,531**	11,625	75.7	**815**	**74.1**
	Check2	8,630	17,256	32	592,629	665.2	**2,571**	**16**	**176,702**	79,639	206.8	**7,861**	**178.8**
	Check3	304,600	609,196	256	10,579,620	23,825.2	**41,360**	**128**	**1,424,669**	1,282,098	2,432.1	**319,703**	**2,054.5**
	Dis1	**320**	**636**	**73**	193,597	232.8	**320**	**73**	**184,359**	9,858	187.0	**636**	**184.6**
	Dis2	**5,313**	**10,622**	**191**	1,069,285	1,287.4	**5,313**	**191**	**907,930**	164,641	1,062.8	**10,622**	**961.7**
	Dis3	**12,818**	**25,632**	**247**	1,841,867	2,867.2	**12,818**	**247**	**1,448,725**	397,296	2,440.0	**25,632**	**1,934.0**

Table 2: Experimental Results of BCS and MCS.

problems over DAGs. So is MCS.

7 Related Work

Some algorithms were proposed for some optimal path problems with specific constraints. Samphaiboon and Yamada [2000] developed a dynamic programming algorithm for the precedence-constrained knapsack problem where the constraint requires that some items can only be selected if their corresponding preceding items have been selected. Multiple sequence alignment with user-defined anchor points takes user-defined homology information as constraints, and a method to solve this problem was introduced in [Morgenstern *et al.*, 2006]. Oommen [1986] proposed an algorithm to solve the edit distance problem with the constraint involving the number and type of edit operations, and Petrović and Golić [1993] focused on the edit distance problem with more complex constraints. These above algorithms only solve optimal path problems with specific constraints. In contrast, our MCS approach is a general method to the constrained optimal path problem with arbitrary global constraint that is represented by an MDD. In real applications, we sometimes require solutions to satisfy multiple constraints. MDDs support polytime conjunction operation so as to combine different constraints efficiently.

MDDs provide a compact and efficient representation for multi-valued functions and large sets. Therefore, MDDs are widely used in the fields of constraint programming and combinatorial optimization. Domain propagation is a crucial step for constraint programming. The results of domain propagation are maintained in a constraint store. Andersen *et al.* [2007] adopted MDDs for constraint store and showed that propagation over MDDs solves all-different problems more efficient than typical domain propagation. A prize-collecting scheduling problem requires selecting a subset of elements from a ground set and ordering the selected elements. For this problem, Horn *et al.* [2021] utilized MDDs to encode the solutions and proposed a novel construction scheme for MDDs. An extension to MDDs, namely AND/OR MDDs, introducing the AND/OR search space, was proposed in [Mateescu and Dechter, 2006]. Later, Mateescu *et al.* [2007] used AND/OR MDDs to represent the optimal set of solutions for constraint optimization problems.

8 Conclusions

In this paper, we have proposed a new algorithm, namely MDD-constrained search (MCS), to solve the constrained optimal path problem over DAGs. It considers the logical constraints as multi-valued functions, and uses multi-valued decision diagrams to handle the constraints. Compared with BCS, MCS can always obtain the correct local constraint w.r.t. a given path, and hence reduce numerous redundant operations. From both the time and space perspective, our proposed method is significantly superior to BCS.

Acknowledgments

This paper was supported by National Natural Science Foundation of China (62276114 and 62377028), Guangdong Basic and Applied Basic Research Foundation (2023B1515120064), Science and Technology Planning Project of Guangzhou (202206030007, Nansha District: 2023ZD001), Jilin Province Natural Science Foundation (20240101378JC), Key Laboratory of Smart Education of Guangdong Higher Education Institutes, Jinan University (2022LSYS003), the Fundamental Research Funds for the Central Universities, JNU (21623202) and the Fundamental Research Funds for the Central Universities, JLU (93K172022K03).

References

[Andersen *et al.*, 2007] Henrik Reif Andersen, Tarik Hadzic, John N Hooker, and Peter Tiedemann. A Constraint Store Based on Multivalued Decision Diagrams. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP-2007)*, volume 4741 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 2007.

[Babar and Miner, 2010] Junaid Babar and Andrew Miner. Meddly: Multi-terminal and Edge-Valued Decision Diagram Library. In *Proceedings of the Seventh International Conference on the Quantitative Evaluation of Systems (QEST-2010)*, pages 195–196, 2010.

[Bryant, 1986] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computer*, 100(8):677–691, 1986.

[Cormen *et al.*, 2022] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter Single-source shortest paths in directed acyclic graphs, pages 616–619. MIT press, 4th edition, 2022.

[Forney, 1973] G. David Forney. The Viterbi Algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[Horn *et al.*, 2021] Matthias Horn, Johannes Maschler, Günther R. Raidl, and Elina Rönnberg. A*-based construction of decision diagrams for a prize-collecting scheduling problem. *Computers & Operations Research*, 126:105125, 2021.

[Kam *et al.*, 1998] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1-2):9–62, 1998.

[Kellerer *et al.*, 2004] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004.

[Mateescu and Dechter, 2006] Robert Mateescu and Rina Dechter. Compiling Constraint Networks into AND/OR Multi-valued Decision Diagrams (AOMDDs). In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP-2006)*, volume 4204 of *Lecture Notes in Computer Science*, pages 329–343. Springer, 2006.

[Mateescu *et al.*, 2007] Robert Mateescu, Radu Marinescu, and Rina Dechter. AND/OR Multi-valued Decision Diagrams for Constraint Optimization. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP-2007)*, volume 4741 of *Lecture Notes in Computer Science*, pages 498–513. Springer, 2007.

[Morgenstern *et al.*, 2006] Burkhard Morgenstern, Sonja J. Prohaska, Dirk Pöhler, and Peter F. Stadler. Multiple sequence alignment with user-defined anchor points. *Algorithms for Molecular Biology*, 1(1):1–12, 2006.

[Nishino *et al.*, 2015] Masaaki Nishino, Norihito Yasuda, Shin-ichi Minato, and Masaaki Nagata. BDD-Constrained Search: A Unified Approach to Constrained Shortest Path Problems. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI-2015)*, pages 1219–1225, 2015.

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

[Petrović and Golić, 1993] Slobodan V. Petrović and Jovan D.J. Golić. String Editing Under a Combination of Constraints. *Information Sciences*, 74(1):151–163, 1993.

[Samphaiboon and Yamada, 2000] N. Samphaiboon and T. Yamada. Heuristic and Exact Algorithms for the Precedence-Constrained Knapsack Problem. *Journal of Optimization Theory and Applications*, 105:659–676, 2000.

[Somenzi, 2015] Fabio Somenzi. CUDD: CU Decision Diagram Package Release 3.0.0. http://vlsi.colorado.edu/~fabio/CUDD/, 2015. Accessed: 2015-12-31.

[Wagner and Fischer, 1974] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168–173, 1974.

[Yamada *et al.*, 2002] Takeo Yamada, Seija Kataoka, and Kohtaro Watanabe. Heuristic and Exact Algorithms for the Disjunctively Constrained Knapsack Problem. *Information Processing Society of Japan Journal*, 43(9), 2002.