Complete Anytime Decision Diagram Search with GPU-Accelerated State Expansion

Fabio Tardivo[1][0000-0003-3328-2174], Laurent Michel[1][0000-0001-7230-7130], and Willem-Jan van Hoeve[2][0000-0002-0023-753X]

University of Connecticut, School of Computing, Storrs, CT, USA
fabio.tardivo@uconn.edu, laurent.michel@uconn.edu
Carnegie Mellon University, Tepper School, Pittsburgh, PA, USA
vanhoeve@andrew.cmu.edu

Abstract. We present a GPU-accelerated method for compiling multi-valued decision diagrams (MDDs) for dynamic programming models. MDD construction expands states in layers defined by their distance from the root, a structure that enables independent parallel expansion of all states in a layer. We exploit this by partitioning each layer into limited-size fragments and executing the full expansion of each fragment on the GPU, including successor generation, dominance filtering, duplicate elimination, and pruning of suboptimal states. By systematically traversing fragments in a depth-first manner, we obtain a parallel Complete Anytime Decision Diagram Search that preserves exactness and anytime behavior. Applied to the Traveling Salesman Problem with Time Windows, our implementation yields speedups of up to two orders of magnitude, and outperforms other state-based methods as well as a state-of-the-art dedicated optimization method for this application.

Keywords: Decision Diagrams · Dynamic Programming · GPU Computing · TSPTW.

1 Introduction

Dynamic programming (DP) models provide a general and expressive formalism for representing combinatorial optimization problems through a state space and transitions between states. Two generic solver frameworks have recently been developed around this paradigm. The DIDP system [11] follows the heuristic-search tradition from AI and implements methods such as A* search and Complete Anytime Beam Search (CABS). Decision diagram-based solvers [9,15,20] instead construct relaxed and restricted diagrams that provide dual and primal bounds in a branch-and-bound search [2].

The effectiveness of decision-diagram methods depends on the strength of the relaxed or restricted diagrams. Relaxed diagrams yield dual bounds through state merging, but effective merging rules can be problem-specific and are often weak for applications such as the Traveling Salesman Problem with Time Windows (TSPTW). In these cases, restricted diagrams, that are constructed simply

by retaining the best states per layer and that require no merging rule, become the primary practical mechanism for search and anytime performance.

A key structural property of decision diagrams is that they are compiled layer by layer, while the expansion of a layer is inherently parallel: each state can be processed independently to generate successors, evaluate feasibility, and apply dominance checks. This structure aligns well with modern GPU architectures, which are designed to apply uniform operations to large, contiguous sets of data. Prior work on parallelizing decision-diagram methods, first proposed in [3] and implemented in Ddo, distributes branch-and-bound subproblems across CPU workers, but does not parallelize the construction of a single diagram. Consequently, the core MDD compilation loop remains sequential in existing DD-based solvers, despite its highly parallelizable structure. DIDP offers a parallel version of CABS as well, which is executed on a CPU [12]. Prior work combining GPUs and MDDs has been conducted in the context of Large Neighborhood Search [21].

In this paper, we investigate GPU acceleration of restricted MDD construction for solving DP models. Our method is implemented within the CODD architecture but does not construct relaxed diagrams, and therefore does not require a merging operator. Instead, we solve pure DP models using restricted diagrams only. This brings the approach closer in spirit to heuristic-search methods such as CABS that exhibit strong anytime performance. At the same time, the layer-structured compilation of MDDs offers the opportunity for massively parallel GPU execution. To this end, we partition each layer into bounded fragments and offload their expansion –including successor generation, dominance filtering, and pruning– to the GPU. A depth-first traversal over fragments preserves the anytime and completeness properties of the search. This improves upon sequential Beam-Stack-Search [25], which requires re-expanding nodes during backtracking and lacks equality and dominance filtering.

We evaluate our approach on the Traveling Salesman Problem with Time Windows since it is a challenging benchmark with hard instances even for modern DD-based approaches [19]. Nonetheless, our approach is generic and applicable to any problem that can be modeled in DIDP [11]. The main purpose of the experiments is to demonstrate the value of using GPUs –generically– for solving optimization problems with CADDS.

We experimentally show that the GPU acceleration achieves an order of magnitude speedup compared to the sequential CPU version. We also obtain an order of magnitude speedup compared to the parallel CABS implementation of DIDP, and up to two orders of magnitude to its sequential implementation in Rust as well as the state-of-the-art optimization solver RouteOpt. These results indicate that the structured, layer-by-layer construction of MDDs is particularly amenable to GPU parallelism, and that accelerating the state-expansion step can significantly enhance the scalability of DP-based solution methods.

2 Decision Diagram-Based Optimization

We first introduce the dynamic programming modeling formalism and the associated decision diagram compilation methodology that we will use in this work. We follow the terminology and notation from CODD [15]. We assume optimization problems for which the feasible set of solutions is a finite set P of decision sequences. The sequences are assumed to be bounded but need not be of the same length. Each sequence $x \in P$ has an associated cost $f(x)$ and the objective is to find a feasible sequence with minimum cost.

2.1 Dynamic Programming Model

Let $\mathcal{U}$ be the set of decision labels for the problem. The dynamic program represents solutions as sequences of labels from $\mathcal{U}$, defined by a labeled transition system with an associated cost function. Let $\mathcal{S}$ denote the set of states of the DP model, with initial state $r \in \mathcal{S}$ and goal state $g \in \mathcal{S}$. For each state $s \in \mathcal{S}$, let $\lambda(s) \subseteq \mathcal{U}$ denote the set of applicable labels. The transition function $\tau(s, \ell)$ maps state $s \in \mathcal{S}$ and label $\ell \in \lambda(s)$ to a successor state. Each transition has an associated cost function $c(s, \ell) \in \mathbb{R}$ for state $s \in \mathcal{S}$ and label $\ell \in \lambda(s)$. Each state $s \in \mathcal{S}$ has an associate value function $v(s)$ that is recursively defined as

$$v(s_{i+1}) = \min_{\substack{\ell \in \lambda(s_i) \\ \tau(s_i,\ell)=s_{i+1}}} v(s_i) + c(s_i, \ell) \quad \forall s_i \in \mathcal{S} \setminus \{g\}$$

where we assume the objective is to minimize the value function $v(g)$ of the goal state. We can optionally initialize $v(r)$ with a non-zero constant. The model is correct when the set of label sequences of all transitions from r to g equals P and the sum of their associated costs equals f. The optimal solution can be obtained by backwards induction using the arguments that minimize the value function.

Example 1. Let $G = (V, E)$ be a complete directed graph with distances d_{ij} for all $(i, j) \in E$, representing time. We identify $i_0 \in V$ as the depot. Let each location $j \in V$ have a time window $[a_j, b_j]$. The time window $[a_{i_0}, b_{i_0}]$ applies to the final return to the depot. The traveling salesman problem with time windows (TSPTW) is to find a closed tour that starts at the depot, then visits all locations in V exactly once and within their time window, and ends at the depot, with minimum total distance. We allow waiting at location j to meet the lower time window a_j. To model this problem as a dynamic program, we let each state be a tuple (U, i, t) where $U \subseteq V$ is the set of unvisited locations, $i \in V$ is the current location, and $t \geq 0$ is the accumulated time. The available labels for state $s = (U, i, t)$ are $\lambda(s) = \{j \in U : t + d_{ij} \leq b_j\}$. For $j \in \lambda(s)$, the transition function is $\tau\big((U, i, t), j\big) = \big(U \setminus \{j\}, j, \max\{t + d_{ij},\, a_j\}\big)$. The associated transition cost function is $c\big((U, i, t), j\big) = d_{ij}$. The initial state is $(V, i_0, 0)$ and the target state is $(\varnothing, i_0, -)$. A complete tour is obtained by applying a feasible sequence of transitions from the initial state to the goal state. □

2.2 Decision Diagram Representation

In the context of optimizing DP models, a decision diagram is a representation of the state space $\mathcal{S}$ as a weighted layered acyclic directed graph. The first layer is defined as $\mathcal{L}_0 = \{r\}$ and contains the initial state. Each subsequent layer is the set of states that are the successors of the previous layer, i.e., $\mathcal{L}_i = \cup_{s \in \mathcal{L}_{i-1}} \cup_{\ell \in \lambda(s)} \tau(s, \ell)$ for $1 \leq i \leq n$, assuming that each solution sequence has maximum length n. This definition recursively defines the expansion of feasible successor states (via the label function λ) and merges equivalent successor states in each layer. Thus, layer $\mathcal{L}_i$ represents all states that are of distance i from the root state. The directed acyclic graph $D = (N, A)$ is now defined by the state set partitioned as $N = (\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_n)$ and arc set $A = \{(u, v) : \exists \ell \in \lambda(u) \text{ s.t. } v = \tau(u, \ell)\}$. Each arc $(u, v) \in A$ represents the transition $\tau(u, \ell) = v$ for some $\ell \in \lambda(u)$ and has associated weight $c(u, \ell)$. Observe that in the above definition, each layer may contain a goal state g. In standard decision diagram literature, these are usually merged together as a single state in the last layer of the diagram [2], but this is not required in an actual implementation. Each r-g path represents a solution, and the optimal solution is obtained by finding a shortest path from the root to the goal state in D.

A decision diagram is *exact* if all reachable states appear in their corresponding layers. Because the state space can grow exponentially large, [2] proposed using restricted and relaxed diagrams of polynomial size. Let $\text{Sol}(D)$ denote the set of solutions represented by decision diagram D. For each arc-specified r-g path p in D, let $w(p)$ denote its total weight, and let x^p denote its sequence of labels. Recall that P is the set of feasible sequences of the problem and $f(x)$ is the cost of sequence $x \in P$.

Definition 1. *A decision diagram D is* restricted *if* $\text{Sol}(D) \subseteq P$ *and* $w(p) \geq f(x^p)$ *for all r-g paths p in D. A decision diagram D is* relaxed *if* $\text{Sol}(D) \supseteq P$ *and* $w(p) \leq f(x^p)$ *for all r-g paths p in D such that $p \in P$.*

The general approach of decision diagram-based optimization is to compile restricted and relaxed diagrams with a given maximum *width*, the number of states in any layer. Building restricted decision diagrams can be done relatively easily by a top-down compilation method. Starting at the root state, we iteratively expand each layer with its successor state, keeping only the best states (according to some scoring function) within the maximum allotted width. Computationally, this requires sorting the states by their score. Building relaxed decision diagrams is done by merging non-equivalent states whenever the size of layer is more than the maximum width. This requires a merging rule that maps two states to a new merged state without excluding any solution as a result. The merging rule is iteratively applied until the size of the diagram meets the width. Using relaxed diagrams is more involved than using restricted diagrams, because the DP model needs to be extended with a merging rule, and applying the rule is computationally more expensive than a simple sorting of states.

Restricted and relaxed decision diagrams respectively provide a primal and dual bound for the problem. An exact branch-and-bound method can be defined

by identifying a cutset of exact states in the relaxed diagram (for example, the last exact layer), and recursively applying the process to each state in the cutset [2]. This is particularly effective when the relaxed decision diagrams provide strong bounds, justifying the additional computational work of building them. An alternative approach is to iteratively expand the restricted diagram by systematically expanding unexplored states in each layer, until no more states are unexplored [16]. This approach does not necessarily require relaxed decision diagrams, and makes it more suitable for problems for which strong relaxations are difficult to build. In this work, we will adopt the latter approach.

2.3 State Filtering Rules

State-based search methods for DP models can take advantage of several rules to discard states that are provably sub-optimal or infeasible.

Feasibility Rules. The feasibility rules are encoded in the function $\lambda(s)$ that provides the set of feasible labels for state s. By default, this function has a myopic view, i.e., a one-step look-ahead, but it can be enhanced by more involved constraint reasoning to discard labels that lead to infeasibility.

Dominance Rules. Let s_1 and s_2 be two states. A *dominance rule* $\text{dom}(s_1, s_2)$, specifying that s_1 dominates s_2, is a sufficient condition under which the best possible completion from s_1 is at least as good as the best possible completion from s_2. As a result, s_2 can be discarded without compromising exactness of the search.

Local Bound Rules. A state-based local bound function $\text{cost}(s)$ for a state s provides a lower bound on the objective value of any feasible solution that extends s. By default, this is the shortest path value from the root to s, but it can be enhanced by computing the cost-to-go estimate from s to the goal, also known as an admissible heuristic. If $\text{cost}(s)$ exceeds a given primal bound, then s can be discarded without compromising exactness of the search.

Example 2. Consider the DP model for the TSPTW from Example 1. We can define a dominance rule as follows. Let $s_1 = (U, i, t_1)$ and $s_2 = (U, i, t_2)$ be two states with the same set of unvisited locations U and the same current location i, but possibly different arrival times t_1 and t_2. Then s_1 dominates s_2 if $t_1 \leq t_2$. Indeed, any feasible continuation of s_2 is also feasible from s_1, and the remaining schedule can only benefit from the earlier arrival time t_1. The default local bound for a state s is the value of the shortest r-s path. A simple admissible heuristic can be defined by sorting the distances non-decreasingly in a list L once. For state $s = (U, i, t)$ we can add to $\text{cost}(s)$ the lower bound $\sum_{j=1}^{|U|} L[j]$. A more involved option is to compute a combinatorial bound, for example the value of a minimum spanning tree on the graph induced by U, as is done in [15]. This paper uses the same enhanced heuristic as DIDP. Namely, for each vertex j it precomputes the shortest incoming arc d_j^{in} and outgoing arc d_j^{out}. Then, it establishes $\text{cost}(s)$ by adding $\max\left(\sum_{j \in U \cup i_0} d_j^{in}, \sum_{j \in U \cup i} d_j^{out}\right)$ where i is the last vertex. □

3 Complete Anytime Decision Diagram Search

We next introduce *Complete Anytime Decision Diagram Search* (CADDS), an exact search method that operates on restricted decision diagrams. It formalizes the restricted-diagram expansion strategy introduced in Section 2.2 by systematically exploring the remaining unexplored states in the diagram, using a width-bounded selection scheme analogous to that employed by Complete Anytime Beam Search (CABS) [24] and Beam-Stack-Search [25]. CADDS follows the structure of the decision diagram: rather than maintaining a global beam over the frontier, it maintains and expands bounded sets of states within individual layers and explores these sets in a depth-first manner.

3.1 Motivation and Relation to Complete Beam Search Methods

CABS iteratively constructs restricted search frontiers of bounded size, guided by a scoring function, and restarts the search with increasing beam widths until optimality can be proven. This strategy can find good solutions and provides strong anytime behavior, but it repeatedly regenerates large parts of the search space [24]. Beam-Stack-Search is an upgrade on breadth-first branch-and-bound search in which no layer of the breadth-first search tree is allowed to grow beyond the beam width [25]. Several other variants exist, including Limited Discrepancy Beam Search [7] and Incremental Beam Search [22].

These complete beam search variants are developed for a generic state space without explicitly exploiting the layered structure induced by the DP model. In contrast, CADDS works directly on the layered decision diagram. Each layer groups states by their distance from the root, and the DP transitions define all outgoing arcs from one layer to the next. CADDS preserves the key ideas of beam search, i.e., retaining a subset of states of bounded width, but it does so within each layer of the decision diagram. And like Beam-Stack-Search, CADDS explores the limited-width layers in a depth-first traversal.

3.2 Fragments and Search Structure

The basic search unit in CADDS is a *fragment*. For a given layer $\mathcal{L}_i$ and width parameter w, a fragment is defined as a subset $F \subseteq \mathcal{L}_i$ containing at most w states. In our implementation, F consists of the w best states in $\mathcal{L}_i$ according to a scoring function f_s; if such a function is not provided we select the first w states in F. Intuitively, a fragment represents the portion of the layer that will be expanded next.

CADDS maintains a sequence of layers $(\mathcal{L}_0, \mathcal{L}_1, \dots)$ as in Section 2.2 and a current best solution s^*. At each iteration, the algorithm selects the deepest nonempty layer $\mathcal{L}_i$, extracts a fragment F from $\mathcal{L}_i$, and expands all states in F to generate their successors in $\mathcal{L}_{i+1}$. The remaining states in $\mathcal{L}_i \setminus F$ are left in place and may be expanded later. This induces a depth-first traversal over fragments: the search tends to go deeper whenever possible, but it eventually returns to shallower layers to process the fragments that were left unexplored.

```
   Function: cadds(r: state, w: ℕ, f_s: score function) →s*: best solution
 1 s* ← ⊥                                                  // No solution found yet
 2 ℒ ← ({r}, ∅, ..., ∅)
 3 while ⋃_i ℒ_i ≠ ∅ do
 4 |   i ← max{i : ℒ_i ≠ ∅}                               // Deepest nonempty layer
 5 |   k ← min{w, |ℒ_i|}
 6 |   F ← top-k states of ℒ_i according to f_s
 7 |   S ← calcSuccessors(F, s*)                          // Successors after filtering
 8 |   ℒ_i ← ℒ_i \ F
 9 |   ℒ_{i+1} ← ℒ_{i+1} ∪ S
10 |   forall s ∈ S do
11 |   |   if isSolution(s) ∧ cost(s) < cost(s*) then
12 |   |   |   s* ← s
13 return s*
```

Algorithm 1: Sequential CADDS (high-level specification)

The width parameter w controls both the memory usage and the amount of work that can be exposed to potential parallel execution. For small w, CADDS behaves similarly to a depth-first search, with limited memory but also limited potential for parallelism. For large w, it approaches a breadth-first expansion at each layer, which increases the potential for parallelism but also increases memory usage and delays the discovery of complete solutions.

3.3 Sequential CADDS Algorithm

Algorithm 1 gives a high-level specification of CADDS in its sequential form. The input consists of the initial state r, a width parameter w, and a scoring function f_s on states. The algorithm maintains a layered representation $\mathcal{L} = (\mathcal{L}_0, \mathcal{L}_1, \dots)$ of the current restricted diagram and an incumbent solution s^*. Initially, $\mathcal{L}_0$ contains only the root state and all other layers are empty.

At each iteration, the deepest nonempty layer is selected (line 4), and a fragment F of size at most w is extracted (lines 5–6). The function calcSuccessors expands all states in F and applies the available pruning rules, returning the surviving successors S in the next layer (line 7). The fragment F is then removed from the current layer, and its successors are inserted into $\mathcal{L}_{i+1}$ (lines 8–9). Any complete solutions among the successors update the incumbent solution (lines 10–13).

The algorithm terminates when all layers are empty, which means that every reachable state that was not discarded by feasibility, dominance, or local bound rules has been expanded. Under the assumption that these rules are sound, the final incumbent s^* is therefore optimal.

The memory complexity involves δ, i.e., an upper bound on the branching factor for states in F ($\delta \geq \max\{\, |\lambda(s)| : s \in F \,\}$). Since each iteration generates at most $w \cdot \delta$ successors, and the algorithm expands the deepest nonempty layer, the memory complexity is $O(|\mathcal{L}| \cdot w \cdot \delta)$.

Function: calcSuccessors(F: set of states, s^*: best solution)$\rightarrow S$: successors
1 $T \leftarrow \{\tau(s,\ell) : s \in F,\ \ell \in \lambda(s)\}$ `// Successor states`
2 $S \leftarrow \{s : s \in T, \mathrm{cost}(s) < \mathrm{cost}(s^*), \nexists s' \in T : \mathrm{dom}(s', s)\}$ `// Apply filters`
3 **return** S

Algorithm 2: Sequential successor generation and pruning

3.4 Successor Generation and Pruning

Algorithm 2 specifies the calcSuccessors procedure in abstract form. Given a fragment F and incumbent solution s^*, it generates all successors of states in F and applies the pruning rules introduced in Section 2.3.

The set T contains all feasible successors obtained by applying the transition function τ to each state in F and each feasible label in $\lambda(s)$. The local bound function $\mathrm{cost}(\cdot)$ is used to discard states whose bound exceeds the cost of the incumbent solution (line 2). Dominance rules, as defined in Section 2.3, are used to discard states that are dominated by other successors in T. The surviving states form the set S of successors that will be inserted into the next layer of the decision diagram. While described as two separate phases, the abstract specification does not assume any particular implementation of the dominance test or heuristic bound. The next section refines this procedure to obtain a GPU-accelerated implementation that expands all states in a fragment in parallel.

3.5 Anytime Behavior and Exactness

CADDS is an anytime algorithm because it maintains a valid incumbent solution throughout the search. Whenever a complete solution is generated in calcSuccessors, it is compared to the current incumbent, and the better of the two is retained. The incumbent therefore improves monotonically as the search proceeds, and the current best solution can be reported at any time.

Proposition 1. *The complete anytime decision diagram search defined in Algorithm 1 is an exact search.*

Proof. Each iteration removes a fragment F from some layer $\mathcal{L}_i$ and inserts its (filtered) successors into $\mathcal{L}_{i+1}$. States that are not included in F remain in $\mathcal{L}_i$ and must eventually be selected as part of a fragment in a later iteration, unless they are discarded by sound pruning rules. Since we always choose the deepest nonempty layer, every reachable, nondominated, and non-suboptimal state is eventually expanded. Therefore, when all layers are empty, the incumbent solution is optimal. When no solution is returned, the problem is infeasible. □

In summary, CADDS provides an exact search procedure over layered decision diagrams with anytime behavior, and its structure naturally exposes potential for fragment-level parallelism. In the next section, we show how to exploit this parallelism on GPUs by offloading the expansion and pruning of fragments to the device.

4 GPU-Accelerated State Expansion

In this section we describe how to accelerate the state expansion step of CADDS on a GPU. The goal is to offload the most expensive part of the search, namely the generation and filtering of successors for a fragment, while keeping the overall control of the search on the CPU. Although GPUs often accelerate matrix operations, they are capable of much more [10]. A useful abstraction model of a GPU is a collection of multi-core processors (each with its own dedicated cache) that excels at parallelizing independent loop iterations. The key ideas are to represent states in a compact, contiguous layout, to partition fragments into batches that fit on the device, and to run successor generation, local bound filtering, and dominance checks on the GPU.

We note that a parallel beam search method was proposed in [6] while a parallel CABS implementation has been implemented in DIDP [12]. Both methods utilize CPU workers, whereas we focus on massive parallelization on a GPU.

4.1 Design Objectives

The CALCSUCCESSORS procedure in Algorithm 1 is the natural target for GPU acceleration. Given a fragment F of states from some layer $\mathcal{L}_i$ and an incumbent solution s^*, it generates all feasible successors, discards suboptimal and dominated states, and returns the remaining successors S for layer $\mathcal{L}_{i+1}$. For problems such as the TSPTW, fragments can contain tens of thousands of states, and the number of raw successors can be much larger. In this regime, the cost of state expansion dominates the total running time.

The design of the GPU implementation is guided by the following objectives: First, we want to exploit data parallelism by applying the same operations to large sets of states. Second, we want to keep the memory layout simple and contiguous to favor coalesced memory access. Third, we aim to limit device memory usage by processing large fragments in batches of bounded size. Fourth, we want to preserve the pruning behavior of the sequential specification, within the efficient context of each batch. The CPU remains responsible for managing the layers, selecting fragments, updating the incumbent solution, and orchestrating calls to the GPU.

4.2 Fragments, Batches, and Data Layout

Recall that a fragment F is a subset of a layer $\mathcal{L}_i$ of size at most w. To offload a fragment, the CPU packs the states in F into a contiguous array and transfers it to the GPU. Device memory is limited, and the number of successors per state may be large, so it is not always possible to expand the entire fragment at once. We therefore partition F into disjoint *batches* $B_0, \ldots, B_k$ and process them sequentially on the device. The batch size is chosen as large as possible under an upper bound on the memory required for storing the batch, its successors, and temporary working arrays. Let M be the the maximum number of states that can be stored in the GPU memory, δ the upper bound on the branching factor,

Function: calcSuccessorsGPU(F: fragment, s^*: best solution, δ: branching factor)→(S: successors, δ': successor branching factor)

1 $S \leftarrow [\,]$ // Initialize successor array
2 $\delta' \leftarrow 0$ // Initialize successor branching factor
3 $b \leftarrow \lfloor M/(1+\delta+\delta t) \rfloor$ // Largest batch size
4 $\{B_0, B_1, \ldots, B_k\} \leftarrow \text{Partition}(F, b)$ // Divide F in batches
5 **forall** $B \in \{B_0, B_1, \ldots, B_k\}$ **do**
6 $\quad T_0 \leftarrow [\,]$ // Feasible successors of B
7 $\quad$ **parall** $s \in B$ **do**
8 $\quad\quad$ **parall** $\ell \in \lambda(s)$ **do**
9 $\quad\quad\quad T_0 \leftarrow T_0 \parallel [\tau(s,\ell)]$ // Array concatenation (thread-safe)
10 $\quad T_1 \leftarrow [\,]$ // Filter sub-optimal states
11 $\quad$ **parall** $s \in T_0$ **do**
12 $\quad\quad$ **if** $\text{cost}(s) < \text{cost}(s^*)$ **then**
13 $\quad\quad\quad T_1 \leftarrow T_1 \parallel [s]$
14 $\quad H \leftarrow \text{parSortBy}(T_1, h \circ \pi)$ // Array of states sorted by hash
15 $\quad R \leftarrow [\bot, \ldots, \bot]$ // State dominance flags
16 $\quad$ **parall** $i \in \{0, \ldots, |H|-1\}$ **do**
17 $\quad\quad$ **forall** $j > i$ *s.t.* $h(\pi(H[i])) = h(\pi(H[j]))$ **do**
18 $\quad\quad\quad$ **if** $H[i] = H[j] \vee \text{dom}(H[i], H[j])$ **then**
19 $\quad\quad\quad\quad R[j] \leftarrow \top$ // State $H[j]$ is dominated
20 $\quad\quad\quad$ **else if** $\text{dom}(H[j], H[i])$ **then**
21 $\quad\quad\quad\quad R[i] \leftarrow \top$ // State $H[i]$ is dominated
22 $\quad T_2 \leftarrow [\,]$ // Filter dominated states
23 $\quad$ **parall** $i \in \{0, \ldots, |H|-1\}$ **do**
24 $\quad\quad$ **if** $R[i] = \bot$ **then**
25 $\quad\quad\quad T_2 \leftarrow T_2 \parallel [H[i]]$
26 $\quad$ **parall** $s \in T_2$ **do**
27 $\quad\quad \delta' \leftarrow \max(\delta', |\lambda(s)|)$ // Update branching factor
28 $\quad S \leftarrow S \parallel T_2$
29 **return** (S, δ')

Algorithm 3: GPU-based successor generation and pruning

and t a constant accounting for temporary buffers, we choose the largest integer b, representing the batch size, such that

$$b + b \cdot \delta + b \cdot \delta \cdot t \leq M. \tag{1}$$

The batch $B_j \subseteq F$ of size at most b form a partition of F. The b states in B_j produce up to $b \cdot \delta$ successor states, each of which require auxiliary space $b \cdot \delta \cdot t$.

4.3 GPU Successor Generation and Pruning

We next refine the CALCSUCCESSORS procedure to its GPU-based counterpart. Algorithm 3 shows the high-level structure of the procedure. It starts by initial-

izing the successor array S and branching factor δ'. It then computes the largest batch size b for fragment F, and partitions F into batches $B_0, B_1, \ldots, B_k$ using the function PARTITION(F, b) (the details are omitted for brevity). It then processes each batch B in sequence, by generating its feasible successors (lines 6-9), discarding suboptimal states based on the local bounding rule (lines 10-13), and discarding dominated states (lines 14-25). The resulting array of remaining successors states is added to S, and the next batch is considered. Each stage uses a temporary array (denoted T_0, T_1, T_2) to store the intermediate set of successor states. Observe the use of PARALL in the algorithm, denoting the parallel execution of a set of instructions on the GPU.

The computationally most demanding part of the algorithm is identifying dominated states, because this requires a pairwise comparison of states. To check dominance efficiently in our setting, we first identify the components of the states that take the same value in the dominance rule. Let $s = (a_1, a_2, \ldots, a_k)$ be the state definition as a list of attributes and let $\mathrm{dom}(s, s')$ be the dominance rule under consideration. Let $J = \{j : a_j = a'_j \wedge \mathrm{dom}(s, s')\}$ be the subset of state attributes a_j that are identical in the application of $\mathrm{dom}(s, s')$. We let $\pi(s) = (a_j : j \in J)$ be the projection of state s under the dominance rule.

Example 3. Consider the TSPTW for which the dominance rule $\mathrm{dom}(s_1, s_2)$ is defined for states $s_1 = (U, i, t_1)$ and $s_2 = (U, i, t_2)$ with the same value for U and for i. The associated projection of state s is $\pi(s) = (U, i)$. □

We use the projection of a state relative to the dominance rule as follows. First, we sort the successor states by their hash function value that is defined on the projection π (function PARSORTBY$(T_1, h \circ \pi)$ in line 14) in a temporary array H. We then execute parallel threads for each state in H to flag dominance (line 16); this can be done quickly for contiguous elements with the same hash value (lines 17-21), because these are the only states to which the dominance rule may apply. We flag all dominated states in a temporary array R, which is subsequently used to discard dominated states in parallel (lines 22-25), storing the remaining states in the temporary array T_2. An important feature of this approach is that dominance is only applied within each batch. This improves the computational performance on the GPU, as it avoids an explicit quadratic number of comparisons and maintains a flat, contiguous memory layout. The cost of sorting T_1 by the hash value is offset by the simplicity of the subsequent dominance checks. The potential downside, however, is that fewer dominated states can be identified across a layer compared to CABS.

Algorithm 3 concludes by appending the outcome T_2 of each batch to the successor array S, and updating the successor branching factor. It then returns the resulting complete set of successors S and the associated branching factor δ'.

Note: Algorithm 3 defines the successor computation at the abstract level of states. In the actual GPU implementation, F is represented as an array of nodes of the form (s, c, p), where s is the state, c is its cost (the current value of the local bound $\mathrm{cost}(s)$), and p is a compact representation of the path from the root to s. This provides access to the information needed for the transition rule and

the pruning rules. We keep the entire path in memory because each fragment is discarded after processing it. This design choice improves the performance, even if it decreases the maximum batch size b for partitioning the fragment.

4.4 Integration with CADDS

The procedure calcSuccessorsGPU integrates into CADDS by replacing the sequential call to calcSuccessors in Algorithm 1 (line 7). The CPU provides the fragment F, the current incumbent solution s^*, and an estimate δ of the branching factor for states in F. The GPU returns the successors S and an updated branching factor δ', which is used to estimate the batch sizes for the next layer. From the point of view of the CADDS algorithm, the semantics of the state expansion step remain unchanged: a fragment is removed from the current layer, its surviving successors are inserted into the next layer, and any complete solutions among the successors are used to update the incumbent. The only difference is that successor generation and pruning are now carried out in parallel on the GPU.

The combination of fragment-based search and GPU-accelerated state expansion yields two complementary benefits. First, it exposes large amounts of parallel work whenever layers contain many states, which is precisely the regime where sequential solvers can have scalability issues. Second, it preserves the anytime and exactness properties of CADDS, since the pruning rules applied on the GPU are the same as in the sequential specification and dominance is applied in a sound, batch-local manner.

5 Experimental Evaluation

We next present an experimental evaluation using the TSPTW as a case study, using five TSPTW benchmark sets from the literature [1,8,19,17,18] for a total of 637 instances. We compare the following methods:

- *CADDS-GPU:* GPU implementation of CADDS, without scoring function.
- *CADDS-GPU-Greedy:* GPU implementation with cost$(\cdot)$ as scoring function.
- *CADDS-Sequential:* Sequential version of CADDS, without scoring function.
- *DIDP-Parallel:* Multi-threaded implementation of CABS in DIDP [12].
- *RPID:* Sequential Rust implementation of CABS in DIDP [13].
- *RouteOpt:* State-of-the-art exact solver for VRPTW that implements a branch-price-and-cut algorithm [23].

We do not include an extension of the relaxed-DD-based approach from CODD [15] to the TSPTW, as preliminary experiments showed it to be non-competitive for this problem. All CADDS[1], DIDP, and RPID models use the same dynamic programming model, dominance rules, and cost heuristics following the literature for TSPTW [5,4,14].

[1]Code available at https://github.com/ldmbouge/CODD/tree/CPAIOR26

The hardware used is an AMD Ryzen Threadripper PRO 7995WX[2], 768 GB of RAM, and a NVIDIA RTX 6000 Ada Generation[3]. The system operates on Ubuntu Linux 24.04 LTS and uses CUDA 12.5, GCC 13.3, Rust 1.90, and Gurobi 13.0. We bound CADDS-Sequential to 48 GB of working memory to process fragments and mimic the GPU memory size. The parallel CABS in DIDP uses 96 threads, 1 per CPU core with no other restrictions. We apply a 10-minute timeout per instance.

The main results are presented in Table 1 over a set of 506 instances that were solved within the timeout by the CADDS and DIDP solvers. The reason is that all these solvers *use the same DP model specification* which makes it more meaningful to compare their performance in terms of time and states explored. For each benchmark and each solution method, the table reports the total solving time over all solved instances, the maximum solving time, the geometric mean of the solving time, the total number of search states, and the maximum memory. Unsolved instances never exhausted memory. All CADDS methods adjust $|F|$ at each iteration to yield a single batch of maximal size according to eq. (1). We note that RouteOpt is not designed for asymmetric or infeasible instances, which are present in four benchmarks, and we indicate this with a dash (−).

GPU Acceleration. We first inspect the impact of the GPU acceleration, for which we compare the total time of CADDS-GPU and CADDS-GPU-Greedy with the baseline CADDS-Sequential. The GPU acceleration achieves a speedup of 10-20x compared to the sequential version, while having a lower memory footprint. This confirms our expectation that state expansion can indeed be offloaded very efficiently on a GPU to be handled in parallel. Observe that the scoring function can have a (marginal) computational benefit, but not always.

Solver Comparison. We next compare the performance of CADDS with CABS in DIDP. The sequential versions CADDS-Sequential and RPID are competitive, although CADDS-Sequential is on average about three times faster while using more memory. Naturally the multi-threaded implementation of DIDP is more competitive. Yet, the GPU still derives speed-ups from 2.6x (Solomon-Pesant) to 16.7x (Solnon Infeasible) compared to DIDP-Parallel. The total number of nodes explored gives a more nuanced story. Indeed, for some benchmarks, CADDS requires significantly fewer nodes than DIDP variants, while for others (e.g., Solomon-Pesant) the opposite holds. The latter happens exactly when the speedups are the weakest (2.6x). An analysis of the behavior indicates that it should be attributed to the weakening of the dominance check caused by the reliance on fragments and batches (recall that dominance is local to fragments and batches in CADDS). That also manifests itself in the increased memory usage (fewer dominated nodes are discarded).

The performance of RouteOpt (on the symmetric and feasible benchmark sets) is weaker than the DP solvers, as it generally solves fewer instances within the time limit. Because it uses strong relaxations during search, it explores fewer search nodes than the CADDS and DIDP variants. The total number of nodes

[2]96 Cores at 5.1 GHz, 384 MB of L3 Cache

[3]18176 CUDA Cores at 2.5 GHz, 96 MB of L2 Cache, 48 GB of VRAM.

Benchmark	Solver	Solved	T_{sum}[s]	T_{max}[s]	T_{GM}[s]	N_{sum}[$\times 10^6$]	M_{max}[GB]
AFG	CADDS-GPU	45	9.95	1.6	0.06	66.7	1.8
	CADDS-GPU-Greedy	45	10.78	1.9	0.06	66.7	1.8
	CADDS-Sequential	45	120.55	30.6	0.00	66.7	7.8
	DIDP-Parallel	45	109.52	22.6	0.37	202.3	2.1
	RPID	45	642.28	179.7	0.09	201.6	1.7
	RouteOpt	–	–	–	–	–	–
Gendreau Dumas Extended	CADDS-GPU	84	98.83	21.6	0.19	740.5	7.0
	CADDS-GPU-Greedy	84	112.12	31.5	0.19	823.9	7.3
	CADDS-Sequential	84	1278.87	301.2	0.00	726.5	24.4
	DIDP-Parallel	84	824.35	104.9	1.53	1453.8	2.8
	RPID	84	3720.75	469.8	1.67	1449.0	1.9
	RouteOpt	60	2442.90	368.3	7.17	0.0	2.9
Solnon (Feasible)	CADDS-GPU	235	60.53	7.2	0.05	683.7	10.2
	CADDS-GPU-Greedy	235	62.13	7.1	0.05	673.3	10.0
	CADDS-Sequential	235	643.14	102.6	0.00	690.7	28.4
	DIDP-Parallel	235	505.86	41.9	0.28	1452.7	4.0
	RPID	235	2513.05	306.3	0.05	1441.7	2.9
	RouteOpt	219	3064.80	341.8	3.46	0.0	5.3
Solnon (Infeasible)	CADDS-GPU	95	18.89	3.8	0.00	254.6	4.2
	CADDS-GPU-Greedy	95	20.11	4.5	0.00	260.3	5.2
	CADDS-Sequential	95	221.45	52.7	0.00	256.1	14.3
	DIDP-Parallel	95	317.96	72.2	0.13	772.3	7.1
	RPID	95	1508.56	447.9	0.01	771.5	5.1
	RouteOpt	–	–	–	–	–	–
Solomon Pesant	CADDS-GPU	21	29.17	16.3	0.07	404.8	24.3
	CADDS-GPU-Greedy	21	29.17	16.3	0.07	404.8	24.3
	CADDS-Sequential	21	607.36	326.1	0.00	698.4	47.7
	DIDP-Parallel	21	75.33	35.4	0.34	220.0	3.8
	RPID	21	435.48	213.7	0.10	219.9	2.6
	RouteOpt	–	–	–	–	–	–
Solomon Potvin Bengio	CADDS-GPU	26	46.63	26.8	0.07	764.1	18.7
	CADDS-GPU-Greedy	26	36.36	14.0	0.08	425.3	19.8
	CADDS-Sequential	26	731.85	365.1	0.00	696.5	44.6
	DIDP-Parallel	26	167.05	69.2	0.39	489.4	4.5
	RPID	26	1128.60	523.8	0.07	485.0	3.0
	RouteOpt	–	–	–	–	–	–

Table 1. Solving performance on instances solved by all DP solvers.

appears as zero in the table as it is too small for the scale.. However, CADDS-GPU outperforms RouteOpt by about two orders of magnitude.

Performance Plots. The plots in Figure 1 show the percentage of solved instances as a function of time for each method, grouped by benchmark. For this analysis, all instances were included. On the AFG and Gendreau-Dumas-Extented, DIDP-Parallel closes one more instance than CADDS-GPU. Once again, it appears that this is a consequence of the dominance weakening. On the feasible Solnon instances, CADDS-GPU solves six more instances than any other solver. Yet, for all the plots, CADDS-GPU dominates in terms of anytime performance, climbing to a high success rates early on, and retaining that advantage throughout in almost all classes.

6 Conclusions

We introduced a GPU implementation of the state expansion process for a complete anytime decision diagram search for dynamic programming models. We rely on the structure of restricted decision diagrams of limited width to create fragments of each node layer, which are processed by the GPU using a massively

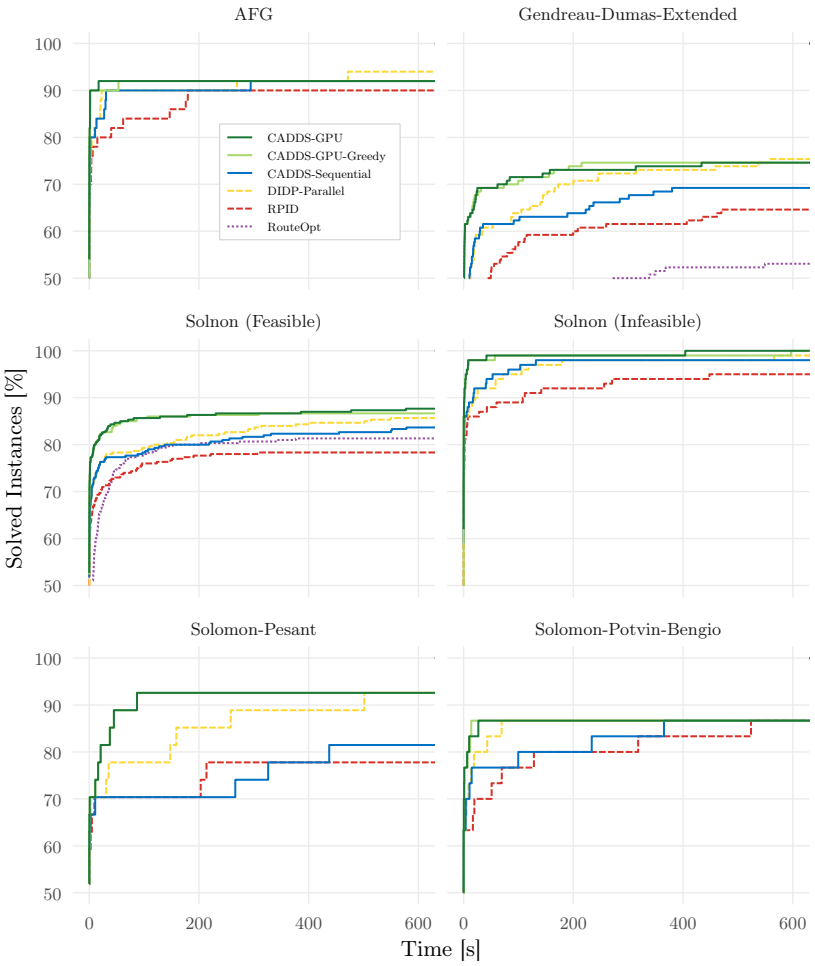

Fig. 1. Empirical cumulative distributions of performance across all instances (truncated at 50% for readability).

parallel implementation. By traversing the fragments of the restricted decision diagram in a depth-first manner, we obtain an exact search method with anytime behavior. We empirically demonstrate on the Traveling Salesman Problem with Time Windows that GPU acceleration obtains an order of magnitude speedup compared to the sequential version, and up to two orders of magnitude speedup compared to the CABS solver in DIDP and the dedicated solver RouteOpt. We conclude that GPU acceleration of state-based search methods provides substantial computational improvements, at no cost to the modeler who states their problem as a dynamic programming model.

Acknowledgments. The authors would like to thank Prof. John D. Owens for his thoughtful insights on GPU-based hash tables and duplicate detection.

References

1. Ascheuer, N.: Hamiltonian Path Problems in the On-line Optimization of Flexible Manufacturing Systems. Ph.D. thesis, Konrad-Zuse-Zentrum für Informationstechnik, Berlin (1996), https://nbn-resolving.org/urn:nbn:de:0297-zib-5328
2. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Decision diagrams for optimization. Springer (2016). https://doi.org/10.1007/978-3-319-42849-9
3. Bergman, D., Ciré, A.A., Sabharwal, A., Samulowitz, H., Saraswat, V.A., van Hoeve, W.J.: Parallel combinatorial optimization with decision diagrams. In: Simonis, H. (ed.) Proceedings of CPAIOR. Lecture Notes in Computer Science, vol. 8451, pp. 351–367. Springer (2014). https://doi.org/10.1007/978-3-319-07046-9_25
4. Coppé, V., Gillard, X., Schaus, P.: Modeling and exploiting dominance rules for discrete optimization with decision diagrams. In: Dilkina, B. (ed.) Proceedings of CPAIOR. Lecture Notes in Computer Science, vol. 14742, pp. 226–242. Springer (2024). https://doi.org/10.1007/978-3-031-60597-0_15
5. Dumas, Y., Desrosiers, J., Gélinas, É., Solomon, M.M.: An optimal algorithm for the traveling salesman problem with time windows. Oper. Res. **43**(2), 367–371 (1995). https://doi.org/10.1287/OPRE.43.2.367
6. Frohner, N., Gmys, J., Melab, N., Raidl, G.R., Talbi, E.: Parallel Beam Search for Combinatorial Optimization. In: Workshop Processing of ICCP. pp. 21:1–21:8. ACM (2022). https://doi.org/10.1145/3547276.3548633
7. Furcy, D., Koenig, S.: Limited discrepancy beam search. In: Kaelbling, L.P., Saffiotti, A. (eds.) Proceedings of IJCAI. pp. 125–131. Professional Book Center (2005), http://ijcai.org/Proceedings/05/Papers/0596.pdf
8. Gendreau, M., Hertz, A., Laporte, G., Stan, M.: A generalized insertion heuristic for the traveling salesman problem with time windows. Oper. Res. **46**(3), 330–335 (1998). https://doi.org/10.1287/OPRE.46.3.330
9. Gillard, X., Schaus, P., Coppé, V.: Ddo, a generic and efficient framework for mdd-based optimization. In: Bessiere, C. (ed.) Proceedings of IJCAI. pp. 5243–5245. ijcai.org (2020). https://doi.org/10.24963/IJCAI.2020/757
10. Hwu, W.m.W., Kirk, D.B., El Hajj, I.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 4 edn. (2022). https://doi.org/10.1016/C2020-0-02969-5
11. Kuroiwa, R., Beck, J.C.: Domain-independent dynamic programming: Generic state space search for combinatorial optimization. In: Koenig, S., Stern, R., Vallati, M. (eds.) Proceedings of ICAPS. pp. 236–244. AAAI Press (2023). https://doi.org/10.1609/ICAPS.V33I1.27200
12. Kuroiwa, R., Beck, J.C.: Parallel beam search algorithms for domain-independent dynamic programming. In: Wooldridge, M.J., Dy, J.G., Natarajan, S. (eds.) Proceedings of AAAI. pp. 20743–20750. AAAI Press (2024). https://doi.org/10.1609/AAAI.V38I18.30062
13. Kuroiwa, R., Beck, J.C.: RPID: rust programmable interface for domain-independent dynamic programming. In: de la Banda, M.G. (ed.) Proceedings of CP. LIPIcs, vol. 340, pp. 23:1–23:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2025). https://doi.org/10.1609/AAAI.V38I18.30062
14. Libralesso, L., Bouhassoun, A., Cambazard, H., Jost, V.: Tree search for the sequential ordering problem. In: Giacomo, G.D., Catalá, A., Dilkina, B., Milano, M., Barro, S., Bugarín, A., Lang, J. (eds.) Proceedings of ECAI. Frontiers in Artificial Intelligence and Applications, vol. 325, pp. 459–465. IOS Press (2020). https://doi.org/10.3233/FAIA200126

15. Michel, L., van Hoeve, W.: CODD: A decision diagram-based solver for combinatorial optimization. In: Proceedings of ECAI. vol. 392, pp. 4240–4247. IOS Press (2024). https://doi.org/10.3233/FAIA240997
16. O'Neil, R.J., Hoffman, K.: Decision diagrams for solving traveling salesman problems with pickup and delivery in real time. Operations Research Letters **47**(3), 197–201 (2019). https://doi.org/10.1016/J.ORL.2019.03.008
17. Pesant, G., Gendreau, M., Potvin, J., Rousseau, J.: An exact constraint logic programming algorithm for the traveling salesman problem with time windows. Transp. Sci. **32**(1), 12–29 (1998). https://doi.org/10.1287/TRSC.32.1.12
18. Potvin, J., Bengio, S.: The vehicle routing problem with time windows part II: genetic search. INFORMS J. Comput. **8**(2), 165–172 (1996). https://doi.org/10.1287/IJOC.8.2.165
19. Rifki, O., Solnon, C.: On the phase transition of the euclidean travelling salesman problem with time windows. J. Artif. Intell. Res. **82**, 2167–2188 (2025). https://doi.org/10.1613/JAIR.1.18334
20. Rudich, I., Cappart, Q., Rousseau, L.: Improved peel-and-bound: Methods for generating dual bounds with multivalued decision diagrams. J. Artif. Intell. Res. **77**, 1489–1538 (2023). https://doi.org/10.1613/JAIR.1.14607
21. Tardivo, F., Pontelli, E.: Parallel declarative solutions of sequencing problems using multi-valued decision diagrams and gpus. In: Cheney, J., Perri, S. (eds.) Proceedings of PADL. Lecture Notes in Computer Science, vol. 13165, pp. 191–207. Springer (2022). https://doi.org/10.1007/978-3-030-94479-7_13
22. Vadlamudi, S.G., Aine, S., Chakrabarti, P.P.: Incremental beam search. Information Processing Letters **113**(22-24), 888–893 (2013). https://doi.org/10.1016/J.IPL.2013.08.010
23. You, Z., Yang, Y.: RouteOpt: An Open-Source Modular Exact Solver for Vehicle Routing Problems. SSRN (2025). https://doi.org/10.2139/ssrn.5314242
24. Zhang, W.: Complete anytime beam search. In: Mostow, J., Rich, C. (eds.) Proceedings AAAI. pp. 425–430. AAAI Press / The MIT Press (1998), http://www.aaai.org/Library/AAAI/1998/aaai98-060.php
25. Zhou, R., Hansen, E.A.: Beam-stack search: Integrating backtracking with beam search. In: Biundo, S., Myers, K.L., Rajan, K. (eds.) Proceedings of ICAPS. pp. 90–98. AAAI (2005), http://www.aaai.org/Library/ICAPS/2005/icaps05-010.php