Accelerating Exact Constrained Shortest Paths on GPUs (complete version)

Shengliang Lu
National University of Singapore
lusl@comp.nus.edu.sg

Bingsheng He
National University of Singapore
hebs@comp.nus.edu.sg

Yuchen Li
Singapore Management University
yuchenli@smu.edu.sg

Hao Fu
Tianjin University, China
haofu@tju.edu.cn

ABSTRACT

The recently emerging applications such as software-defined networks and autonomous vehicles require efficient and exact solutions for constrained shortest paths (CSP), which finds the shortest path in a graph while satisfying some user-defined constraints. Compared with the common shortest path problems without constraints, CSP queries have a significantly larger number of subproblems. The most widely used labeling algorithm becomes prohibitively slow and impractical. Other existing approaches tend to find approximate solutions and build costly indices on graphs for fast query processing, which are not suitable for emerging applications with the requirement of exact solutions. A natural question is whether and how we can efficiently find the exact solution for CSP.

In this paper, we propose *Vine*, a framework that parallelizes the labeling algorithm to efficiently find the exact CSP solution using GPUs. The major challenge addressed in Vine is how to deal with a large number of subproblems that are mostly unpromising but require a significant amount of memory and computational resources. Our solution is twofold. First, we develop a two-level pruning approach to eliminate the subproblems by making good use of the GPU's hierarchical memory. Second, we propose an adaptive parallelism control model based on the observations that the degree of parallelism (DOP) is the key to performance optimization with the given amount of computational resources. Extensive experiments show that Vine achieves 18× speedup on average over the widely adopted CPU-based solution running on 40 CPU threads. Vine also has over 5× speedup compared with a GPU approach that statically controls the DOP. Compared to the state-of-the-art approximate solution with preprocessed indices, Vine provides exact results with competitive or even better performance.

PVLDB Reference Format:
Shengliang Lu, Bingsheng He, Yuchen Li, and Hao Fu. Accelerating Exact Constrained Shortest Paths on GPUs (complete version). PVLDB, 14(4): XXX-XXX, 2021.
doi:10.14778/3436905.3436914

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/xtra-computing/vine.

Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436914

1 INTRODUCTION

The constrained shortest path (CSP) problems are widely used to formulate many applications. Each edge in the graph of such applications has two properties: a length and a cost. One example is the vehicle routing problem [41], where edge properties are the length and cost of the corresponding road segment. The cost could be road user charges or time costs depending on the application. Moreover, we have witnessed the recent rise of emerging applications such as software-defined networks and autonomous vehicles, which require efficient and exact solutions for CSP. The exact solution requirements come from rigid requirements on safety and conflicts like routing in autonomous vehicles and battery-powered drones [6, 8, 17], or minimizing the risk of detection of an aircraft/-submarine [43, 44]. Those applications cannot tolerate large errors in CSP solutions because errors can lead to inefficiencies, riskiness, and safety issues [31]. In this paper, we study whether and how we can efficiently solve the exact CSP.

The commonly used solution for exact CSP problems is the labeling algorithm based on dynamic programming [13]. It has been well studied and extended to solve a line of CSP problems [7, 18, 41]. In the labeling algorithm, an available path from the source src to vertex v is presented as a label ℓ_v. Similar to Dijkstra's algorithm [5], the labeling algorithm maintains a min-heap of labels and pops the top label ℓ_v at each iteration. New labels are generated by appending edges to ℓ_v. The unpromising labels are pruned, and others are pushed back to the min-heap for further expansion. The objective is to find the optimal labels at destination vertex dst. In this paper, we focus on the efficiency of CSP problems with a single constraint, because they are more commonly used in practice, and efficiently solving such single-constraint CSP problems is already very challenging [41]. The single-constraint CSP and the labeling algorithm are formally presented in Section 2.2.

The labeling algorithm is prohibitively slow in real-world cases due to the significant computation and memory consumption incurred by the large number of labels [14]. It thus significantly limits the applicability of the labeling algorithm to get an exact solution. Therefore, many approximate approaches trade off the solution accuracy for speed. They reduce the number of labels by finding approximate solutions [38], building indices [37], or both [41]. Wang et al. [41] present the state-of-the-art approximate CSP solution, named COLA, which improves the approximate labeling algorithm by partitioning and building indices on the networks. However, the approximated solution is not suitable for many emerging applications with the requirement of exact solutions. Additionally, the expensive preprocessing of building indexes can be inefficient or even impractical for frequently updated graphs.

GPUs are promising platforms for solving large-scale graph problems due to their massive number of cores and high memory bandwidth. Existing efforts have shown great success in a plethora of graph applications on GPUs, such as breadth-first search [26], shortest path [2], PageRank [11] and subgraph enumeration [10]. Nevertheless, the efficient parallel solutions for CSP on GPUs have been largely overlooked. Accordingly, we propose a GPU framework Vine to parallelize the labeling algorithm for the exact CSP solution efficiently. While the GPU's massive parallelism is very suitable for the subproblem parallelism within CSP, the major challenge is how to deal with the exponential growth of subproblems. Furthermore, as these subproblems are mostly unpromising in practice, we need an effective and efficient pruning mechanism to reduce memory and computational resource usage. Specifically, we propose the following two techniques to address those challenges.

First, we propose a two-level pruning solution, which makes use of the hierarchy of memories on the GPU. The first level offers an efficient pruning in the GPU's shared memory that reduces GPU memory contention. The second level prunes labels thoroughly in GPU's global memory.

Second, we propose an adaptive parallelism control model that adjusts the degree of parallelism (DOP) on the GPU at runtime. In our experiments, we observe that a high DOP explores subproblems quickly but brings more unpromising ones that require more computation and memory resources, which turns out to be slow in execution time. Thus, the proposed model is based on the total number of subproblems and the prediction of its growth at runtime, which are affected by both the graphs and queries.

Extensive experiments using 11 real-world graphs that comprise three different applications show that Vine achieves 18× speedups on average over the widely adopted CPU-based solution on 40 CPU threads, and up to 5× speedups compared to a fine-tuned GPU approach that statically controls the DOP. Compared to the state-of-the-art approximate approach, COLA [41], Vine shows competitive or even better performance while providing the exact solution without costly preprocessing. The contributions of this paper are summarized as follows.

- We propose Vine, an acceleration framework for solving exact CSP queries on GPUs. There are two core techniques proposed: 1) an efficient *two-level pruning* technique to prune the subproblems effectively, and 2) a light-weight *adaptive parallelism control model* to adjust the DOP for different graphs and queries at runtime.
- We have conducted extensive experiments to demonstrate that Vine achieves significant speedups over existing solutions.
- Vine achieves competitive or even better performance to approximate approaches, which are commonly used in the previous studies of getting an approximate solution for CSP. This paper shows that, with our careful optimizations on GPUs, Vine is a viable solution for solving the exact CSP.

The rest of this paper is organized as follows. In Section 2, we introduce the definition of CSP, review the related work, and present the GPU background. In Section 3, we present a baseline approach for parallelizing the commonly used labeling algorithm. We give the system overview in Section 4, followed by implementation details in Sections 5 and 6. Section 7 shows the evaluation of Vine. Section 8 concludes this work.

2 BACKGROUND AND RELATED WORK

In this section, we provide the GPU preliminaries, define the CSP problem and labeling algorithm, and review the related work.

2.1 GPU Preliminaries

A single GPU has tens of streaming multiprocessors (SMs). Each SM contains many processing cores. A program executed by the GPU is called a *Kernel*. A kernel executes on multiple thread blocks, each of which consists of many threads. Thread blocks can further be divided into *warps* (each of 32 threads). Each thread block is assigned to a single SM, and the cores inside each SM execute the threads in a SIMT fashion, at the granularity of a single warp. During any given execution cycle, an entire warp of threads only execute a single instruction.

The global memory of the GPU has a very high bandwidth. It can be accessed by all threads executing on a GPU, but the latency is high. In contrast, *shared memory* is faster for data accesses, but has limited capacities (usually around 64KB per SM) and can only be accessed by the threads in the same thread block. There are two conventional optimization methods for the memory hierarchy on GPUs. The first is to maximize coalesced memory accesses for improving the utilization of global memory bandwidth. It requires that the threads in one warp access one contiguous memory region at one time instead of scattered accesses. The second is to maximize the computation within the low-latency shared memory. In this paper, we develop a novel pruning solution to the efficient use of both memory optimization methods.

2.2 CSP and Labeling Algorithm

Definition 1 (CSP problem). A CSP problem is defined on a directed graph $G = (V, E)$, where V is the vertex set and E is the edge set. Each edge $e \in E$ is associated with a *length* $e.l \geq 0$ and a *cost* $e.c \geq 0$. For a path $p = \langle e_1, e_2, \cdots, e_{|p|} \rangle$ in G, the length and cost of p are defined as $p.l = e_1.l \oplus e_2.l \oplus \cdots \oplus e_p.l$ and $p.c = e_1.c \odot e_2.c \odot \cdots \odot e_p.c$, respectively. Following previous works [13, 41], operators $\oplus$ and $\odot$ can be defined as addition, production, min, and max operators depending on the application. Both operators satisfy the monotonicity properties.

Problem statement. This paper focuses on CSP problems with a single constraint. Given a source vertex $src \in V$, a destination vertex $dst \in V$, and a constraint ω, we let $\mathbb{P}$ denote the set of paths joining src to dst, where $\forall p \in \mathbb{P}$, $p.c$ satisfies the constraint ω. A CSP query asks for a shortest path $p^* = \text{argmin}_{p \in \mathbb{P}}\, p.l$.

The constraint satisfaction function depends on the application. We detail the constraint used in the vehicle routing problem shortly in this section, and the constraints of the other two evaluated applications of this work in Section 7.

Labeling algorithm. The commonly used solution for the exact CSP is the labeling algorithm based on dynamic programming [13,

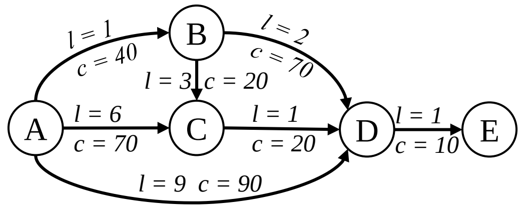


Figure 1: Example of the graph of a vehicle routing problem.

38, 41]. In the labeling algorithm, a path p from source vertex *src* to vertex v is represented as a *label* attached to vertex v, denoted as $\ell_v = (p.l, p.c)$. The result of a CSP query is the label ℓ^*_{dst}, which represents path p^*.

The algorithm maintains $|V|$ *non-dominated label lists* that store non-dominated labels separately for every vertex and a shared min-heap of labels. In each iteration, the algorithm pops the top label ℓ_u from the min-heap and generate a new one ℓ_v from it using edge $e_{u,v}$ via the *Expansion* operation. The *Feasibility* operation checks whether ℓ_v satisfies the constraint ω. The *Dominance* operation checks whether ℓ_v is dominated by a label of vertex v. If ℓ_v is feasible and not dominated by any labels, it will be added to both the min-heap and the non-dominated label list of v, otherwise pruned. The algorithm stops until the optimal label ℓ^*_{dst} is found or the min-heap is empty. We present more details about the three operations (Expansion, Feasibility, and Dominance), using the vehicle routing problem as an example.

Vehicle routing problem. Following previous work [41], we define both $\oplus$ and $\odot$ to be addition + in the vehicle routing problem. The constraint ω is defined as a cost budget so that $\forall p \in \mathbb{P}, p.c \leq \omega$.

Expansion. This operation generates a new label ℓ_v using ℓ_u and edge $e_{u,v}$. It is the transition of a path p presented by ℓ_u to a new path p' presented by ℓ_v. In the vehicle routing problem, $\ell_v = (p'.l, p'.c) = (p.l + e_{u,v}.l, p.c + e_{u,v}.c)$.

Feasibility. This operation checks if the label satisfies the defined constraint. In the vehicle routing problem, the labeling algorithm checks if the cost of a path p is no more than the predefined budget (i.e., $p.c \leq \omega$). Paths that do not satisfy the constraints are infeasible and discarded to reduce the redundancy.

Dominance. This operation checks if a label needs to be pruned. Specifically, let ℓ_v and ℓ'_v present two paths p and p' from *src* to the same vertex v, respectively. In the vehicle routing problem, we define dominance as ℓ_v dominates ℓ'_v *iff* $p.l \leq p'.l$ and $p.c \leq p'.c$. The intuition is that if a longer path p' takes more cost than p, p' will never lead to the optimal results. Therefore, the algorithm prunes the corresponding label ℓ'_v when it is dominated by any label attached to the same vertex.

An example of a road network is shown in Figure 1 and we want to find the CSP from A to E. If there is no constraint, the shortest path is $\langle A \rightarrow B \rightarrow D \rightarrow E \rangle$, with total length 4 and cost usage 120. If we set the cost budget as 100, the shortest path is no longer feasible. The optimal path becomes $\langle A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rangle$, where the length of the path is 6, and the resource usage decreases to 90.

2.3 Related Work

As a classical problem, CSP has been studied for decades [19]. Several families of exact algorithms have been proposed, including Lagrangian relaxation [21], column generation [4], path ranking [39], and integer-linear programming (ILP) [46]. Lagrangian relaxation iteratively calculates the shortest path with link weights being weighted sums of the original length and cost on edges in each iteration [21]. Path ranking works by solving the k-th shortest paths problem and terminates when the first path satisfying all constraints is found [39]. The ILP approach re-formulates CSP problems and then invokes external ILP solvers [46]. However, these approaches are inherently sequential and demonstrate poor practical efficiency. The most widely adopted solution for CSP problems remains to be the labeling algorithm [13] based on dynamic programming, as described in Section 2.2.

Recent approaches focus on providing approximate solutions. Tsaggouris et al. develop a fully polynomial-time approximation scheme [38] to improve both asymptotic complexity and practical performance. Wang et al. [41] present the state-of-the-art work, COLA, which partitions the network, indexes paths between vertices, and finds approximated CSP. However, either the approximated result or high overhead of preprocessing is not always satisfying in practice. We experimentally evaluate the weak points of the approximate approach and compare our system's performance with COLA in Section 7.5.

The massive parallelism and high memory bandwidth of modern GPUs are the main reasons for the adoption of GPUs in graph processing studies. Existing efforts have shown great successes in parallelizing a plethora of graph acceleration works on GPUs [12, 16, 26, 34, 35]. Many frameworks and primitives have also been presented for high-performance graph algorithms on GPUs [22, 42, 45]. However, to the best of our knowledge, there is no acceleration of exact CSP solver and the solutions above cannot efficiently handle the dynamic growth of subproblems. Forcing implementing the labeling algorithm for CSP using frameworks like Gunrock would eventually result in rewriting the data storage and processing model, which are mostly the majorities of the components in such frameworks. We refer the reader to two recent surveys for more details of GPU-based graph processing [9, 36].

3 A BASELINE APPROACH

In this section, we present a baseline approach of the parallel labeling algorithm on the GPU and show our observations to motivate the designs and optimizations in Vine.

3.1 Sequential vs. Parallel Labeling Algorithms

The sequential algorithm stores labels in the min-heap in order and the baseline approach of the parallel algorithm maintains the unexpanded labels in the frontier. The key difference is that the sequential algorithm always selects the top label from the heap to expand and prune others in the heap, whereas the baseline approach expands a fixed number of the frontiers simultaneously in each iteration. We define it as the degree of parallelism (DOP), denoted as a tuning parameter K.

The DOP of baseline parallel approach is a trade-off between the parallelism in GPU processing and the amount of workload (the number of expanding labels). Generally, as illustrated in Figure 2, there are total N labels in the frontiers, K of N frontier labels are expanded at the same time. The expanded labels are used to update the non-dominated label lists. We store the non-dominated label

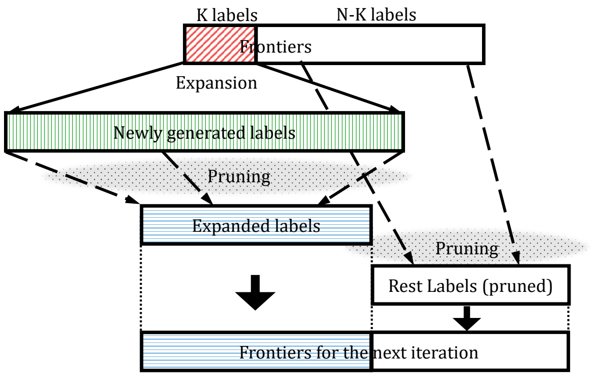


Figure 2: The baseline parallel approach expands K labels from the frontiers, and leverages the outcome to prune dominated labels the rest $N - K$ frontiers in parallel.

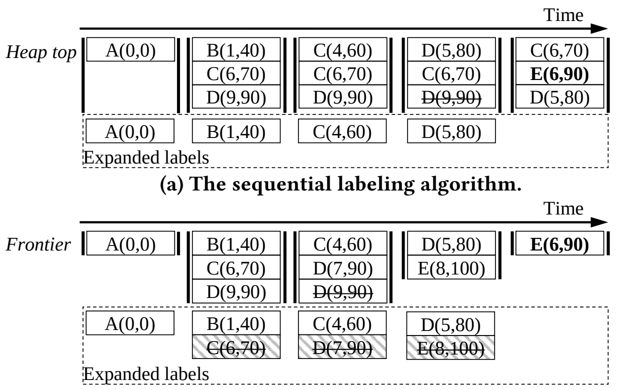


Figure 3: Processing example of Figure 1 using both sequential and parallel labeling algorithms. The cost budget is set to be 100. Dominated labels are stroke out. Unpromising labels are shadowed.

lists in the GPU memory and check each newly expanded label against those lists for pruning. The updated label lists can be used to prune the unexpanded $(N - K)$ frontiers further.

In the parallel algorithm, there may be unpromising yet unpruned labels among the K labels expanded, which results in producing more labels than the sequential algorithm. We use Figure 3 to compare the sequential labeling algorithm and the baseline parallel approach on solving the CSP example in Figure 1. The intermediate results are termed in the label format $v(p.l, p.c)$. Both algorithms start with a label $A(0, 0)$, representing the initial state. In the sequential algorithm, it only expands four labels before termination. In comparison, with the DOP set as two ($K = 2$), the parallel approach expands seven labels in total. There are some unpromising labels among the expanded labels. For example, $C(6, 70)$ is dominated by $C(4, 60)$. However, since the parallel algorithm expands $C(6, 70)$ before we find $C(4, 60)$, we waste the computation on processing $C(6, 70)$ and its following labels $D(7, 90)$ and $E(8, 100)$, which the sequential algorithm would not expand. As a result, the parallel labeling algorithm expands more labels than the sequential one.

3.2 Motivation with Baseline Parallel Algorithm

We experimentally study the impact of different K values in tuning DOP, and analyze the observations from the baseline approach. In the evaluation, we use three query sets Q1, Q2 and Q3 on different applications on 11 real graphs. The experimental setup can be found in Section 7.1.

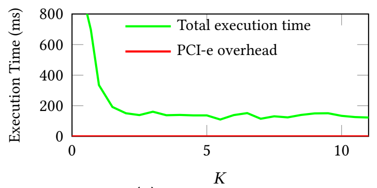
Observation 1: *Label operations, including expansion and pruning, are the core and the most time-consuming operations in CSP.* Parallelizing the labeling algorithm brings a more significant number of labels than sequential execution. It is because those unpromising yet unpruned labels generate even more unpromising ones. Label pruning is very important as it eliminates unpromising labels. However, parallelizing existing pruning operations causes GPU memory contention because a label needs to be checked against labels stored in the non-dominated list and update the list if not being pruned. The read and write in global memory can contribute over 90% of label management's total execution time, thus bottleneck the overall performance. Although the shared memory has low latency, its limited size makes it a challenge to use such a large amount of subproblems efficiently.

Observation 2: *The DOP (i.e., the K value) is the key to the performance because of the fact that one needs to carefully examine the trade-off between "exploring" and "exploiting" subproblems.* The trade-off is already illustrated by the example in Figure 3 that $K = 2$ leads to more label expansions than the sequential algorithm. In fact, if we further increase the K value, it will produce even more labels (and even more unpromising labels).

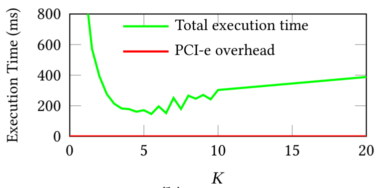
Figure 4 demonstrates the execution time of CSP queries on three graphs with different settings of K. As shown in the figure, the K value has a tremendous impact on performance. On the one hand, setting a large K value for high parallelism would quickly explore subproblems. However, exploration of the subproblems can result in several times slowdown than the optimal DOP setting. The reasons are as follows. First, a higher K leads to an aggressive expansion of labels, resulting in a large number of non-optimal subproblems. In other words, those non-optimal subproblems could have been effectively pruned if we use a smaller K value. Second, we profile the PCI-e overhead when K increases. For the two small graphs (NW and internet), there is no PCI-e overhead in the test. In contrast, for a larger graph like LKS, with a large K, the processing runs out the space on GPU global memory and the overhead of PCI-e data transfer increases dramatically.

On the other hand, setting a small K for low parallelism results in even longer execution time, as shown on the left-end part of the lines. The reasons are the increasing number of iterations, underutilization of the GPU, and degradation of the process. Setting K too large or too small would both result in poor performance.

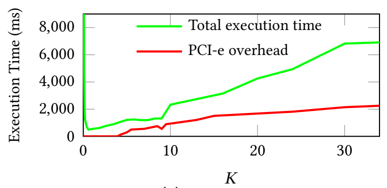
Observation 3: *The best setting of K is graph- and query-dependent.* The optimal K value varies significantly for different graphs with query set Q2, as shown in Figure 5. In another experiment (figures are omitted), we also process three query sets on the same NY graph and find that the best K value varies significantly, changing from 500 to more than 4,900 for different queries. The observation is that the best K is both graph-dependent and query-dependent.



(a) internet



(b) NW



(c) LKS

Figure 4: The execution time of CSP with Q2, with different settings of K on three graphs (internet, NW and LKS).

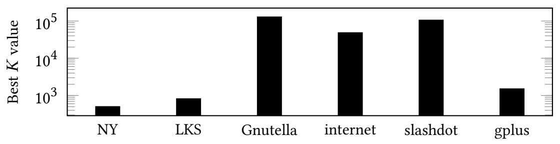


Figure 5: Best K for different graphs using query set Q2.

Table 1: APIs of Label class in Vine.

API Name	Parameter(s)	Description
Expansion	Edge e, Label ℓ_v	Expand and generate label ℓ_v using *this* label and an edge e.
Feasibility	None	Return true if *this* label is feasible, otherwise false.
Dominance	Label ℓ	Return true if *this* label dominates ℓ, otherwise false.

The reason is that the query decides the *src* and *dst*, and affects the total number of subproblems by setting the constraint ω loose or tight; the graph structure affects the number of paths from *src* to *dst*; the lengths and costs of the edges affect the total number of subproblems. A fixed and predefined K value might not capture the growth of the number of subproblems during the traversal, as the travel can access different parts of the graph and different parts of the graph may demonstrate different structures. Instead, adaptively adjusting K at each processing iteration is more promising.

Implications. The above observations motivate the novel design and careful optimizations in Vine. Observation 1 motivates an efficient pruning solution that effectively reduces the number of subproblems while providing high performance by reducing global memory contention. Observations 2 and 3 motivate us to select suitable K values that are adapted to the query and graph structures at runtime. Instead of finding a single predefined value, we use an adaptive model to adjust the K for each iteration.

4 SYSTEM OVERVIEW

We propose Vine, a parallel framework for finding exact CSP solutions. In this section, we first present an overview of the Vine framework, including the API design and the algorithm in Vine. In the following two sections, we present a two-fold solution to deal with the large number of subproblems at runtime. First, we develop an efficient two-level pruning solution (in Section 5), which makes good use of the memory hierarchy of GPU for efficient pruning. Second, we propose a light-weight adaptive parallelism control model to adjust the DOP at runtime by learning and predicting the number of subproblems (in Section 6). In order to differentiate the static DOP control in the baseline approach, we use K_t to represent the DOP of the t-th iteration in the rest of the paper.

Listing 1: An example of Vine for vehicle routing problem.

```
class Edge{
 int len, cost;
};// end of edge class
class Label{
 int vid, l, c; Label *pre = NULL;
 void Expansion (const Edge &e, Label &lv) {
  lv.l = this->l+e.l;
  lv.c = this->c+e.c;
  lv.pre = this;
 }
 bool Feasibility () {
  return this.c <= cost_budget;
 }
 bool Dominance (const Label &l) {
  return this.l <= l.l && this.c <= l.c;
 }
};// end of label class
```

4.1 API Design

Inspired by the success of existing programming frameworks on GPU [15, 42, 45], Vine is designed as a programming framework with three APIs listed in Table 1. The APIs assemble the operations in the labeling algorithm. We show an example of solving the vehicle routing problem using Vine in Listing 1. The code consists of the definition of two classes, *Edge* and *Label*. The Edge class contains length and cost properties. The Label class, with three functions, defines the behavior of a CSP application. The field *pre* is used to backtrack the vertices on the path when we construct the final optimal path.

Users can develop their applications by simply implementing these APIs according to their application logic. The abstraction significantly reduces the burden of programming on the GPU platform. Vine automatically executes these functions in parallel on the GPU.

4.2 Algorithm Overview

Vine is powered by accelerating the labeling algorithm using the GPU. Algorithm 1 shows the labeling algorithm in Vine. There are five major steps in the algorithm, including *initialization*, *parallelism control*, *label expansion*, *label pruning*, and *frontier assembling*.

- *Initialization* (Lines 1 to 3). Vine creates an empty label $src(0, 0)$ and pushes it into the frontier buffer.
- *Parallelism control* (Line 6). In each iteration, a parallelism control model is invoked to determine the DOP for the current iteration t. The model outputs the value K_t by taking the total number of frontiers and the GPU resources as input. Note that we use the per-thread and per-warp coarse-grained workload mapping technique proposed by Gunrock [42].
- *Label expansion* (Line 8). Vine invokes the user-defined *Expansion* function in Label class to generate new labels from the first K_t frontiers. Without being stored in the global memory, the newly

Algorithm 1: Labeling algorithm in Vine.

```
Input   : Graph g, Source label src (0,0)
Output  : Optimal label result
▸Initialization
1  $t \leftarrow 0$;
2  frontiers ← src (0,0);
3  frontierNum ← 1;
4  while frontiers ≠ ∅ do
5      $t \leftarrow t + 1$;
       ▸Parallelism control, executed on the CPU
6      $K_t$ ← AdptParlCtrlModel(frontierNum);
       /* GPU kernel 1                                    */
7      for $\ell \in$ frontiers[1 : $K_t$] do
           ▸Label expansion with the first-level pruning intergrated,
            leveraging Expansion(), Feasibility(), Dominance()
8          hashPrnTbl ← ExpandLabel($\ell$, g);
9          if hashPrnTbl.#EmptyBuckets < threshold then
               ▸Second-level pruning using Dominance()
10             nondmntLst ← 2ndLvlPrn(hashPrnTbl,
                 nondmntLst);
       ▸Clean hash-pruning tables
11     nondmntLst ← 2ndLvlPrn(hashPrnTbl, nondmntLst);
       /* GPU kernel 2                                    */
       ▸Pruning unexpanded frontiers using Dominance()
12     for $\ell \in$ frontiers[$K_t + 1$ : frontierNum] do
13         2ndLvlPrn($\ell$, nondmntLst);
       /* GPU kernel 3                                    */
       ▸Frontier assembling
14     frontiers, frontierNum ← AssembleFrontier();
15 result ← nondmntLst [dst ];
```

generated labels are checked by function *Feasibility()*. Note that Vine picks the top K_t labels to expand because expanding such labels would lead to quicker convergence of the query processing.

• *Label pruning* (Lines 8 to 13). During the expansion, newly generated labels in each thread block of the GPU are pushed to a *hash-pruning table* for the first-level pruning in shared memory (Line 8). The hash-pruning table keeps non-dominated labels that are used to prune the newly generated labels using *Dominance()*.

The hash-pruning table dumps the stored labels to the global memory for thorough pruning in Lines 10 and 11, when the number of empty buckets drops under a threshold and after expanding K_t frontiers, respectively. Non-dominated labels are pushed into the non-dominated label lists so that they can be used to prune others. With the list updated, we prune the labels left in the frontiers to avoid unpromising expansion in Line 13.

• *Frontier assembling* (Line 14). Vine gathers both the non-dominated labels expanded and the rest frontiers after pruning. These labels assemble the frontiers for the next iteration. We adopt the highly efficient prefix-sum-based frontier generation from [26].

The algorithm terminates when the frontier buffer is empty. The non-dominated labels stored in the label list associated with *dst* are the optimal ones (no feasible result if the label list is empty).

5 TWO-LEVEL PRUNING

Design rationales. Due to the large number of unpromising labels generated during searching, an efficient pruning algorithm is needed in order to reduce costly memory accesses to the GPU global memory. We need an approach to achieve a reasonably good pruning power and very low runtime overhead. To this end, we take advantage of the shared memory of the GPU and propose the following two levels of pruning.

The first level offers a very efficient pruning in the shared memory, with reasonable pruning power. However, the shared memory has too limited space (e.g., 64KB per SM) to maintain all non-dominated label lists. Also, pruning in the shared memory only helps prune labels generated within the block in the current iteration but not thoroughly. Therefore, we design a novel data structure to effectively take advantage of shared memory to prune as many labels as possible.

The second-level pruning happens in global memory, which is applied to the outcome of the first-level pruning and has more pruning power. Specifically, new labels are checked against non-dominated labels found in a lazy manner to achieve efficient global memory access.

5.1 Pruning in the Shared Memory

We design a *hash-pruning table* to make use of the limited space of the shared memory. The hash-pruning table indexes a label ℓ_v with a simple hash function by hashing the vertex id v as follows.

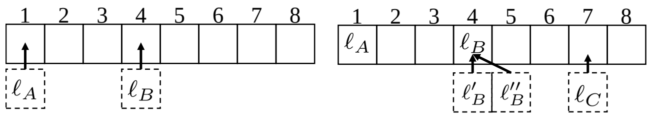
$$idx_{hash} = v \mod (GPU_{shared\_memory\_size}/Label_{size})$$

We employ a linear probing collision resolution since it is cache-efficient [20]. Particularly, we handle the following three conditions when pruning the new label ℓ_v. 1) ℓ_v is inserted if idx_{hash} points to an empty bucket. 2) If the bucket is not empty and one of the two collided labels dominate the other, the dominated one is pruned. 3) If the bucket is not empty and the two labels have different vertex ids or do not dominate with each other, we probe the new label to the next bucket according to linear probing.

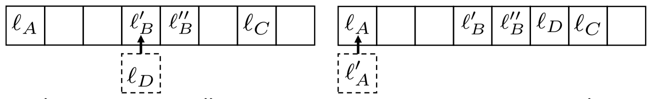
The indexing in a hash-pruning table is fast due to the efficient hashing. Moreover, the hash-pruning table dynamically arranges the limited memory space for efficient occupation. Particularly, when the number of empty buckets drops under a threshold, the hash-pruning table dumps the stored labels to the global memory to make rooms for newly generated labels. Vine dumps the remaining labels in the hash-pruning table after finishing the current iteration to make it empty at the beginning of each iteration since different iterations tend to have very different labels to process.

The hash-pruning table reduces the memory accesses and contentions in the following ways. 1) The hash-pruning operation aggregates the memory accesses of pruning and visits the global memory in bulk. 2) Hash-pruning tables located in the different shared memory of SMs dump contents asynchronously, reducing the contention significantly.

Figure 6 demonstrates an example of the label pruning using the hash-pruning table with eight buckets for brevity. The shared memory of a modern GPU can support more than 8,000 buckets per SM in the vehicle routing problem.



(a) Insert ℓ_A and ℓ_B. **(b) Insert ℓ_B', ℓ_B'', and ℓ_C.**



(c) ℓ_B' dominates ℓ_B; ℓ_B'' probes; ℓ_D is hashed to 4th bucket. **(d) ℓ_D probes; Insert ℓ_A'.**

Figure 6: Collisions and dominance in a hash-pruning table.

- *Hashing:* Suppose label ℓ_A and ℓ_B are hashed and inserted to the 1st and 4th bucket of the empty table, respectively (Figure 6a). Labels ℓ_B' and ℓ_B'' are also hashed to the 4th bucket because they all belong to the same vertex (Figure 6b).
- *Collision and dominance: discard dominated labels.* A dominance comparison is conducted because of the collision between labels of vertex B. In this example, let us assume ℓ_B' dominates ℓ_B. Therefore, we discard ℓ_B and put ℓ_B' at the 4th bucket (Figure 6c).
- *Collision only: probe to the next bucket.* ℓ_B'' is inserted after ℓ_B'. There is also a collision between them. Let us assume they do not dominate each other, and thus ℓ_B'' is probed to the next available buckets (Figure 6c). In another case, a label ℓ_D is also hashed to the 4th bucket taken by ℓ_B'. ℓ_D is then probed to the first available bucket (Figure 6d), which is the 6th bucket. There is no dominance comparison required since ℓ_D and ℓ_B' are from different vertices.

Performance analysis. The number of collisions among labels from the same vertices is essential for the effectiveness of the first-level pruning in the shared memory. Thus, in the following, we develop an analytical model to obtain the expected number of collisions in each iteration.

We assume there are w labels fed into the hash-pruning table in an iteration and they are randomly distributed among vertices for simplicity. We define an indicator function $X(i, j)$ to denote the event of collision between two labels i and j.

$$X(i,j) = \begin{cases} 1, \text{if label } i \text{ and label } j \text{ belong to the same vertex;} \\ 0, \text{otherwise.} \end{cases} \tag{1}$$

Given $X(i, j)$, the expected number of collisions is derived from $\mathbf{E}[X]$. The calculation of $\mathbf{E}[X]$ can be modeled as the birthday problem [40]. The details about this formulation are omitted due to space constraints.

$$\mathbf{E}[X] = \sum_{i=1}^{w} \sum_{j=i+1}^{w} \mathbf{E}[X(i,j)] = \binom{w}{2} \frac{1}{|V|} = \frac{w(w-1)}{2|V|} \tag{2}$$

With the exponential growth in the number of labels, we have $w \gg \sqrt{|V|}$ for most cases. Therefore, we expect a high chance of pruning labels in the hash-pruning table, as also shown in experiments.

5.2 Pruning in the Global Memory

Figure 7 illustrates the design of pruning in the global memory, where labels dumped from the hash-pruning table are checked against the labels stored in the *non-dominated label lists* for every vertex. The blocks with the same color represent labels that are

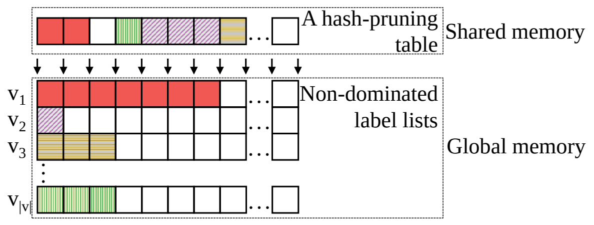


Figure 7: Dump labels to non-dominated label lists.

attached to the same vertex. In Vine, each list is located in a contiguous GPU memory region to enable *coalesced memory accesses.*

Lazy update strategy. If a new label dominates some labels in the non-dominated label list, all of these dominated labels should be removed to avoid a waste of comparisons when pruning other new labels later. However, both removing labels and resizing lists are expensive on GPUs because they require dynamic memory management and locking to avoid conflict among concurrent threads. Accordingly, we propose a practical *lazy update* strategy that only replaces the first dominated label found in a list by the new label. We stop the dominance check immediately and ignore the existence of other dominated labels. If there were some, they remain in the list until being replaced by other inserted labels.

The lazy update sacrifices the opportunities to reduce redundant comparisons but enables the efficiency of the list update. Besides, it keeps the front part of each label list more frequently updated than other parts. Labels that are closer to the optimal results are likely to be located at the front part. Therefore, new labels have a high chance of being pruned by only accessing the front part, reducing the number of memory accesses. The strategy performs very well in practice as the pruning is efficient.

Correctness analysis. Vine will eventually find the optimal path if it exists. Due to the space limitation, we omit the proof here.

6 ADAPTIVE PARALLELISM CONTROL

In this section, we explain how to use the adaptive parallelism control model to adjust the DOP (Line 6 of Algorithm 1). The output of the model is K_t, the DOP of the t-th iteration. Some frequently used notations are summarized in Table 2.

Design rationales. We have observed that the DOP (K_t) is vital for the overall performance in Section 3. On the one hand, a low DOP setting could lead to underutilization of the GPU given that the GPU has tens of thousands of threads. On the other hand, if we keep expanding labels at a high DOP, the number of subproblems would explode and overflow the GPU global memory. Thus, we must resort to costly PCI-e data transfer between the main memory and the GPU.

Despite the tradeoffs in DOP, we find that a higher DOP often achieves higher performance than a significant low DOP.Based on our observation in Section 3, as long as DOP is not set to be widely large, the performance is close to the optimal strategy of varying DOP. Thus, in this paper, we attempt to use higher DOPs in order to utilize the high parallelism of GPUs, with the goal that the memory consumption of all labels can be accommodated by the GPU memory (in order to avoid costly PCI-e traffic during expansion). Another potential benefit of a higher DOP is to provide more labels to prune simultaneously in each iteration, amortizing the overhead of the hash-pruning table and paging between GPU and CPU memory.

Specifically, we consider both the total number of labels and the growth of the number of labels at each iteration. The adaptive model consists of the following design concepts: 1) we dynamically control the total number of labels within the GPU memory by modeling an upper limit for the number of unexpanded labels at each iteration, i.e., the frontier size; 2) given the number of labels in the current iteration satisfying the upper limit, the number of labels in the following iteration should also preserve the limit.

6.1 Control the Total Number of Labels

The total number of labels in a CSP processing is estimated as the sum of two parts, the expanded frontiers and the frontiers in the future. Let M_{total} be the maximum number of labels that can be stored in the GPU global memory and M_t be the number of labels that can be stored in the vacant memory in the t-th iteration. To limit these labels within the GPU memory at t-th iteration, we have:

$$\sum_{i=1}^{t} N_i + \sum_{i=t+1}^{T} \hat{N}_i \le M_{total} \tag{3}$$

The label size is application-dependent. In the vehicle routing problem (Listing 1), a label contains three int values and a pointer, and the label size is 20 bytes. Thus, M_{total} is calculated as the size of GPU global memory divided by the label size. In the formula, T is defined to be the estimated total number of iterations of solving the CSP problem. We assume that the frontier size $\hat{N}$ in the rest $(T-t)$ iterations would be well-controlled by the model and preserve stable under a limit $MaxFtr_{t+1}$, i.e.,

$$\hat{N}_i \le MaxFtr_{t+1}, i \in [t+1, T] \tag{4}$$

$$\therefore \sum_{i=t+1}^{T} \hat{N}_i \le (T-t)MaxFtr_{t+1} \tag{5}$$

We let

$$(T-t)MaxFtr_{t+1} = M_t \tag{6}$$

$$\therefore MaxFtr_{t+1} = \frac{M_t}{(T-t)}, \text{where } M_t = M_{total} - \sum_{i=1}^{t} N_i \tag{7}$$

The $MaxFtr_{t+1}$ value is updated at each iteration with the available GPU space M_t updated.

Estimating T is challenging. In our study, we develop a simple way to derive the estimation for T according to graph structures and the application. For the vehicle routing problem, T is capped as the cost budget of the query divided by the minimum cost of the edges in the graph. For the applications on social networks and internet networks, the total number of iterations is usually in hundreds as we observed in our experiments. Therefore, we empirically set T as 1,000 for bottleneck shortest path and optimal trust path selection.

6.2 Control the Growth Rate

We aim to find the suitable K_t to ensure that $N_{t+1} \le MaxFtr_{t+1}$ after expanding K_t labels from N_t. Since N_{t+1} is not known yet in the t-th iteration, we use $\hat{N}_{t+1}$ to denoted the estimated value. As illustrated in Figure 2, $\hat{N}_{t+1}$ could be calculated as:

$$\hat{N}_{t+1} = NewLabel_t + RestLabel_t \le MaxFtr_{t+1} \tag{8}$$

Table 2: Summary of notations.

Input	Description
T	Estimated total number of iterations
N_i	Frontier size of the i-th iteration
$\hat{N}_i$	Estimated frontier size of the i-th iteration
t	Current iteration number
M_{total}	Maximum number of labels that can be stored in the GPU global memory
M_t	Number of labels that can be stored in the vacant GPU global memory in current iteration
K_t	DOP, the number of labels to expand in current iteration
$MaxFtr_{t+1}$	Maximum frontier size in the next iteration
$NewLabel_t$	Number of labels generated from the K_t labels
$RestLabel_t$	Number of unexpanded frontiers after pruning
ω_t	Growth ratio of the frontier size in the current iteration, generated by a forecast model
θ_t	Pruning ratio of the unexpanded frontiers in the current iteration, generated by a forecast model

$NewLabel_t$ represents the new labels expanded from K_t labels, and $RestLabel_t$ represents the unexpanded frontiers after pruning in the t-th iteration. We use ω_t to denote the growth ratio of the frontier size, i.e., $\omega_t = N_{t+1}/N_t$. If we expand K_t labels, based on the growth rate of frontiers, $NewLabel_t$ can be estimated as:

$$NewLabel_t = \omega_t K_t \tag{9}$$

Suppose θ_t is the pruning ratio of the rest $(N_t - K_t)$ unexpanded labels in the frontier. We thus model $RestLabel_t$ as:

$$RestLabel_t = (1-\theta_t)(N_t - K_t) \tag{10}$$

$\hat{N}_{t+1}$ can thus be estimated by substitute Equations 9 and 10 into Equation 8, and we can get K_t that:

$$\begin{aligned} \hat{N}_{t+1} &= \omega_t K_t + (1-\theta_t)(N_t - K_t) \le MaxFtr_{t+1} \\ \therefore K_t &\le (MaxFtr_{t+1} - N_t + \theta_t N_t)/(\omega_t - 1 + \theta_t) \end{aligned} \tag{11}$$

Since $MaxFtr_{t+1}$ and N_t are known at the t-th iteration, to obtain K_t, we need to know the value of ω_t and θ_t. To do so, we adopt a forecast model to predict both ratios ω_t and θ_t.

Autoregression forecast model. Since the prediction methods for ω_t and θ_t are similar, we use the prediction of ω_t as an example. We take the growth ratio of frontier sizes in different iterations as a time series $\{\omega_1, \omega_2, \dots, \omega_{t-1}\}$ where $\omega_i = N_{i+1}/N_i, i \in [1, t-1]$. We employ an autoregression (AR) forecast model (Equation 12) to predict the next item in the series, i.e., ω_t, by learning the trend [1].

$$\omega_t = \sum_{j=1}^{q} \varphi_j \omega_{t-j} + z + \varepsilon_t, \text{ where } q < t. \tag{12}$$

The AR model uses frontier sizes in the previous q iterations. In our implementation, we choose $q = 6$ empirically. $\varphi_1, \dots \varphi_q$ are the derived coefficients by applying least squares [23] on the time series items, z is a constant, and ε_t is the Gaussian white noise.

The AR model can forecast an arbitrary number ($< q$) of iterations by submitting the predicted values back to itself. There are other available approaches for prediction, such as applying neural networks. In this work, we find that AR is light-weight and accurate enough for our purposes.

7 EXPERIMENTAL EVALUATION

This section experimentally evaluates Vine in comparisons with existing exact and approximate approaches.

7.1 Experimental Setup

Hardware. Table 3 summarizes the hardware specification. We conduct experiments on a Linux server with two 10-core Xeon E5-2640v4 CPUs, 256GB memory, and an NVIDIA Tesla P100 GPU, which has 12GB global memory and 56 SMs. We also test the performance of Vine on a desktop GPU, Titan Xp, which has 12GB global memory and 30 SMs. Note, P100 has a larger number of SMs as well as larger shared memory per SM. All GPU programs are compiled with NVIDIA's *nvcc* compiler (version 10.1), and the CPU baseline is compiled with *gcc 8.3* using -O3 flag. In our experiments, the graph data is pre-loaded into the GPU memory. The reported results include the query processing time and the PCI-e transfer time for query input and output. We run all tests five times and present the average time.

Comparisons. We have studied the implementations on the CPU as baselines. Specifically, we note there are some open-source libraries in Boost to solve CSP [28]. Boost only supports sequential algorithms, and thus we parallelize the labeling algorithm [13] based on dynamic programming using OpenMP, running on 40 CPU threads (denoted as **CPU**). Our study finds that our home-grown sequential solution is faster than Boost and our parallel version is even much faster than Boost. Thus, we only report the results for our home-grown parallel solution on the CPU. We discuss more implementation details of the CPU-based approach in the appendix.

To understand the impact of the proposed techniques, we compare Vine with the following implementations. Note that when subproblems cannot fit into the GPU memory, all of the evaluated GPU implementations leverage the UVA (unified virtual addressing), which allows the GPU to access the main memory.

- **Vine (1-level pruning).** This approach is the same as Vine except that the hash-pruning table is disabled.
- **Vine (PC=baseline).** This approach is the same as Vine, except that Vine (PC=baseline) expands all frontier labels in every iteration. PC stands for parallelism control.
- **Vine (PC=static).** Vine (PC=static) is the collection of all the shortest execution for each query with an optimal predefined K. The optimal K is obtained by conducting comprehensive experiments for every single query on each data set with different K settings independently. This is the best case for the static approach. With this strong comparison candidate, we demonstrate the effectiveness of our adaptive control.
- **COLA.** COLA [41] is the state-of-the-art approximate CSP solution based on indexing on road networks. It is a sequential algorithm with indexes. We study the implementation of COLA [41], and we find that it is challenging (possibly for another orthogonal study) to parallelize COLA or optimize it for modern hardware. First, COLA leverages highly customized data structures like maps, heaps, and vectors that are not thread-safe. Furthermore, COLA's pruning procedure is inherently sequential, tightly coupled to the label expansion. Therefore, we use the original code of COLA.

Table 3: Hardware specification.

	2×E5-2640v4	P100 GPU	Titan Xp
# of compute units	20	56	30
# of threads	40	3,584	3,840
Frequency (GHz)	2.4	1.3	1.58
Mem. (GB)	256	12	12
Mem. Bw. (GB/s)	68.3	549	547.7
Cache/Shared Mem. per SM	25MB	64KB	48KB
Mem. type	DDR4	HBM2	GDDR5X

Table 4: Details of data sets.

Type	Data set	|V|	|E|	Size
Road	New York City (NY)	264,346	730,100	14MB
	Florida (FLA)	1,070,376	2,712,798	53MB
	Northwest USA (NW)	1,207,945	2,840,208	58MB
	Northeast USA (NE)	1,524,453	3,897,636	77MB
	Great Lakes (LKS)	2,758,119	6,794,808	141MB
	Synthetic road networks	16,200,000	48,600,000	483MB
Internet	Router Connection (router)	2,113	6,632	1MB
	Internet (internet)	40,164	170,246	2MB
	P2P Gnutella (gnutella)	62,586	147,892	3MB
	Internet Topology (skitter)	1,696,415	11,095,298	277MB
Soc.	Slashdot Network (slashdot)	70,068	358,647	8MB
	Google+ Social Network (gplus)	211,187	1,141,650	36MB

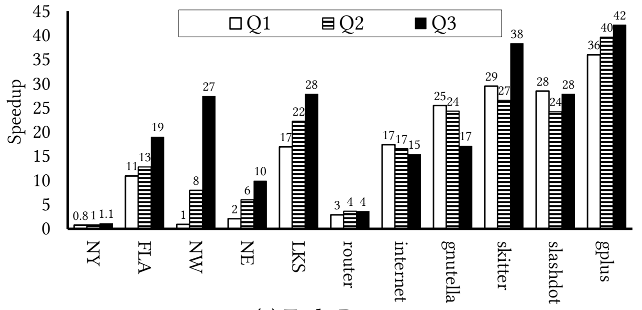
Applications and data sets. We evaluate the performance of Vine with 11 real-world graphs shown in Table 4. We test the following three applications implemented using Vine framework.

(1) *Vehicle routing on road networks.* Following previous works [33, 41], this application is designed to find the shortest path with the total cost under a given cost budget. The road networks are obtained from the 9th DIMACS challenge [3], where edge properties are the road length and the cost.

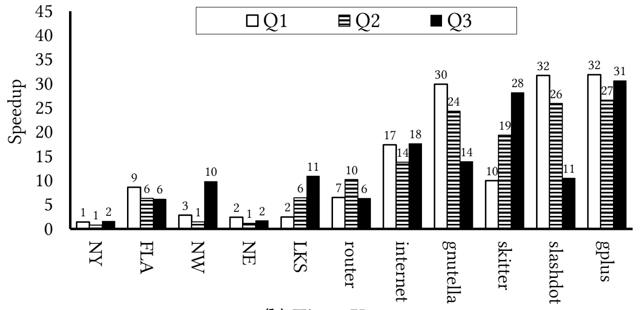
(2) *Bottleneck shortest path with bandwidth constraint on Internet networks.* This problem is to find the fastest path between two vertices with a minimum bandwidth requirement, where the vertices represent access points in the network. Graphs are obtained from [24, 32], where edge attributes are the link latency and bandwidth. Note that the latency is the additive objective and the bandwidth is the min-max QoS measure, which is quantified in the range of $[0, 10]$ to represent different levels of bandwidth.

(3) *Optimal trust path selection on social networks.* The objective of this application is to find the path with the highest trust to a person [25]. A vertex in these social networks represent one person, and edges between vertices are assigned with a trust value ($\mathcal{T} \in [0, 1]$) and a social intimacy value ($\gamma \in [0, 1]$). The optimal path satisfies the constraint that the attenuated aggregation of social intimacy (i.e., the multiplication of γ on the edges) is greater than a predefined threshold. Different from the other two applications, optimal trust path is based on a non-additive constraint. We take the negation of $\mathcal{T}$'s logarithmic transformations as each edge's length to formulate the problem as finding the shortest path with social intimacy as the cost. We adopt the method in [25] to generate $\mathcal{T}$ and γ randomly.

Query sets. We adopt the rule proposed in [41] to produce road network queries. When generating queries, the *src* and *dst* vertices are generated randomly. All these queries are divided into different sets in terms of the lengths of their shortest-length paths. We pick



(a) Tesla P100



(b) Titan Xp

Figure 8: The speedups of CSP queries on different graphs (Vine over CPU) using two different GPUs.

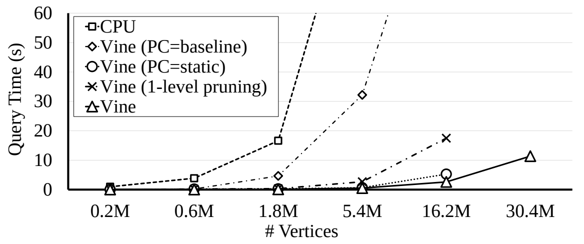


Figure 9: The query execution time of different solutions on synthetic road networks using query set Q2.

three query sets Q1, Q2, and Q3, that have the path lengths within the range of $[d_{min}/8, d_{min}/4)$, $[d_{min}/4, d_{min}/2)$, and $[d_{min}/2, +\infty)$, respectively, where d_{min} is the diameter of the graph. Constraints are also generated randomly between the minimum cost of all paths and the cost of the shortest-length path from *src* to *dst*. Q1 is designed to be relatively small in search space, Q2 is moderate, and Q3 is large. Each query set consists of 20 queries. We follow the same concept to generate three query sets for each internet graphs. The aggregation of social intimacy in queries for social network graphs are set as $Q1 : \gamma \geq 0.07$, $Q2 : \gamma \geq 0.04$, and $Q3 : \gamma \geq 0.01$.

Experiment outline. Section 7.2 presents the overall performance comparison with the CPU-based approaches. Sections 7.3 and 7.4 evaluate the impact of two-level pruning and adaptive parallelism control, respectively. We compare Vine with COLA in Section 7.5.

7.2 Overall Performance Comparison

Figure 8 shows speedup results of Vine over CPU using P100 and Titan Xp, respectively. The speedup is calculated as the execution time of the CPU divided by that of Vine. Vine gets a significant speedup using a single GPU over using 40 CPU threads (i.e., 18× speedups on average). The maximum speedup on P100 GPU is 42× and the minimum is 0.8× on the NY graph. The maximum speedup on Titan Xp is 32× and the minimum is also 1× on the NY graph. Since NY is a relatively small network, the GPU related overhead, including PCI-e transfer and kernel invocation, takes over 70% of the end-to-end execution time, which is not negligible, unlike other cases. In order to handle the large search space of the CSP problem, massive thread parallelism and memory bandwidth of the GPU are both desirable features (as shown in Table 3). These lead to significant performance improvement over CPUs.

We also compare the performance of Vine on P100 and Titan Xp. We observe that Vine performs better on P100 than that on Titan Xp with 1.8× speedups on average. The main reason for this performance difference is that P100 has more SMs, as well as larger shared memory per SM than Titan Xp. Therefore, Vine takes advantage of larger shared memory and achieves more efficient pruning on P100. As a result, we study the profiling results, and observe that Vine on P100 has 3.3× fewer global memory transactions than that on Titan Xp.

Performance-price ratio. Vine shows a high performance-price ratio. The price of the P100 GPU is 3× than the two-socket CPU and 5× than Titan Xp. We define the performance-price ratio as: $1/time/price$. With up to 42× speedups, the performance-price ratio of Vine is much higher than its CPU counterpart's for most of the cases, showing up to 13× better cost-efficiency. Titan Xp has even higher performance-price ratio as it achieves more 50% of P100's performance on average but only costs 1/5 of the price of P100, delivering even higher (up to 22×) cost efficiency on average over the CPU-based counterpart.

Performance-power ratio. Vine also shows a high performance-power ratio compared to CPU. The thermal design powers (TDP) of the P100 and Titan Xp are both 250W, which is 1.4× of the two-socket CPUs. With up to 42× speedups on GPUs, Vine delivers much better energy efficiency than the CPU-based counterparts.

Scalability. Vine gains higher speedups on larger graphs and scales better than CPU. We further evaluate the scalability of Vine using synthetic road networks. As shown in Figure 9, Vine (on P100) shows very good scalability. We terminate the query if it runs more than 1 minute. Note that 30.4M is the maximum graph size that the current machine can support in this test. The design of Vine does not have limits on graph size, and the graph size is limited in the sense that the graph data and intermediate results need to be in the main memory and GPU memory.

Through the comparison among GPU baselines with the increased graph size, we have the following observations on the proposed techniques. First, compared to Vine (PC=baseline), all the GPU implementations with DOP tuning scale well. As the graph size increases, adjusting the DOP shows even more significant impact on the performance. Vine's adaptive parallelism control effectively shows minimum influence caused by the increasing graph size. Second, the two-level pruning mitigates the costly label management especially when the graph size is large. Particularly, Vine (1-level pruning) shows similar performance as Vine and Vine (PC=static) on small graphs but performs much worse than Vine on the graph with 16.2M vertices.

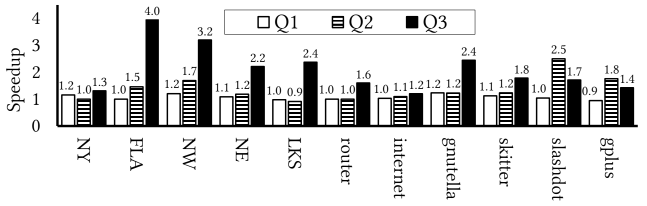
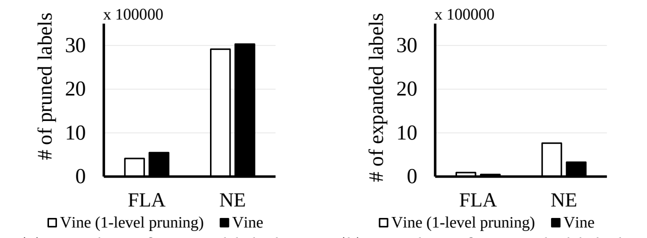
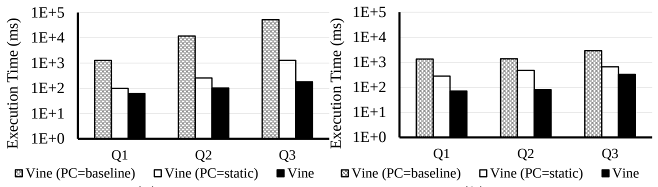


Figure 10: The speedup of CSP queries on different graphs (Vine over Vine (1-level pruning)).



(a) Number of pruned labels. **(b) Number of expanded labels.**

Figure 11: The number of expanded and pruned labels of Vine (1-level pruning) and Vine using query set Q2.



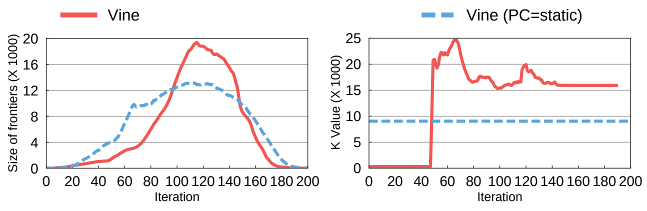
(a) NW **(b) NE**

Figure 12: Query efficiency of Vine using different parallelism control strategies on different graphs.

7.3 Impact of Two-level Pruning

To understand the impact of the two-level pruning optimization with the hash-pruning table, we further conduct experiments with Vine and Vine (1-level pruning) in this part. We present the result collected by running Vine on P100. As shown in Figure 10, we can draw an overall conclusion that Vine outperforms Vine (1-level pruning) on most of the graphs with up to 4× speedups. Vine shows little speedup on solving Q1 for most of the cases. Because of the tight constraint of Q1, most of the generated labels are not feasible. These unfeasible labels are discarded immediately without comparison with others. Thus, the two-level pruning does not help to prune much but even brings slight overhead. Nevertheless, with the pruning optimization, Vine still outperforms Vine (1-level pruning) for 31 out of 33 test cases and is never more than 10% slower than Vine (1-level pruning).

Analysis on the efficiency of label pruning. Efficient pruning reduces the total number of subproblems, which makes it easier to find the optimal results. As shown in Figure 11a, Vine prunes more labels than Vine (1-level pruning). It turns out that Vine finishes query faster with around 50% number of labels fewer than Vine (1-level pruning), as shown in Figure 11b. Besides reducing the total amount of labels, the hash-pruning table also significantly reduces the global memory contention by buffering updates. Vine shows similar pruning outcomes on other graphs and queries at different scales. For brevity, we show only the above cases, where Vine achieves 1.2× and 1.5× speedups, respectively.



(a) Frontier sizes. **(b) DOP control with K values.**

Figure 13: Frontier sizes and DOP (i.e., K_t and K) in different iterations on the NE graph.

7.4 Impact of Parallelism Control

Figure 12 presents the execution time of Vine (PC=baseline), Vine (PC=static), and Vine on NW and NE graph using P100. We observe similar behaviors on other data sets using the other GPU, Titan Xp, and thus we omit the results for brevity. We have the following observations. First, both static and adaptive parallelism control help improve the performance over the baseline, because the total amount of work is reduced since there is more "exploitation". The controlled DOP helps reduce global memory accesses and thus also reduces the execution time.

Second, Vine performs better than Vine (PC=static) since the static parallelism control overlooks the distinctions of iterations. As the growth of the number of labels changes depending on the graph structure and query at runtime, a static value can hardly perform well at all iterations. Vine, instead, proficiently tailors K_t for each iteration according to both the estimated total number of labels and the growth of the number of labels during the process.

Impact to frontier size. Taking one of the query processing on the NE graph as an example, we show the trend of the frontier size in Figure 13a. As the first 100 iterations shown in the figure, Vine maintains a smaller frontier size than Vine (PC=static) and expands more labels in the middle iterations. Vine turns to converge faster and achieves 1.3× speedups over Vine (PC=static) in this case.

One of the benefits Vine obtains from the adaptive parallelism control model is changing K_t for a different stage of processing. We show the adaptive value K_t and the static value of K in Figure 13b. The trend of frontier expansion is steady at the beginning, and thus adaptive DOP control is not activated yet. After the frontier size begins to flatten, the model encourages increasing DOP. Vine gets a steeper trend of frontier sizes, which leads to a fast exploration in the middle stages, which Vine (PC=static) cannot achieve. At this point, the adaptive control model guides Vine to slow the exploration and decrease DOP. At around iteration 120, Vine has increased DOP to accelerate the convergence.

In summary, the adaptive parallelism control can help achieve a reliable performance improvement.

7.5 Comparison with Approximate Approaches

Since COLA [41] is the state-of-the-art approximate solution for CSP, we performed a study with the original source code from COLA, and have the following findings.

First, the preprocessing is time-consuming, which makes COLA mostly for static graphs in practice. We present the total preprocessing time of COLA in Figure 14. COLA has a tuning parameter (approximation ratio α) to control the solution quality, with value no smaller than one. A larger α leads to smaller preprocessing time but a more significant relative error. For example, it can take hours to more than a half-day to index a single graph. This overhead of preprocessing (i.e., index construction) becomes impractical in the real-world as the changes in road conditions and charges call for real-time processing. We discuss more details on the extensions of dynamic graphs in the appendix.

Second, the solution error of COLA highly depends on the tuning parameter α and the graph structure. Setting $\alpha = 1.4$ leads to a shorter preprocessing time, but the relative error could be as high as 14%. The applications with requirements of exact solution may not be able to tolerate the relative error of the result. On the other hand, although setting $\alpha = 1.005$ leads to a very small error on some small graph like NY, it leads to a larger error on other graphs (meanwhile leads to very high preprocessing time).

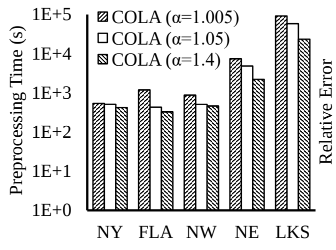
Finally, we compare the query response time of Vine and COLA. Figure 15 shows the query time of Vine and COLA with different approximation ratios (α). We observe that COLA has better performance with a higher approximation ratio (also leading to larger errors). Compared with COLA's performance with the smallest error ($\alpha = 1.005$), Vine's performance is competitive and even better on larger graphs. We emphasize a few points that are not shown in this comparison. First, COLA builds indexes, but Vine does not. Second, Vine leverages the GPU acceleration, but COLA is sequential on the CPU. Third, Vine processes exact CSP results, but COLA targets approximated results. Our comparison with COLA demonstrates that: by efficiently leveraging the GPU acceleration, we can deliver the exact CSP results whose performance is comparable and even faster than the state-of-the-art solution.

7.6 Discussions

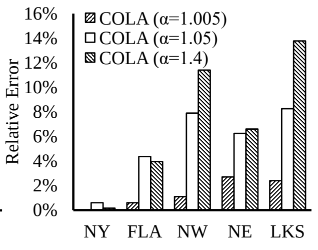
Summary of experimental findings. The experimental findings are summarized as follows. First, compared to the labeling algorithm on 40 CPU threads, Vine achieves 18× speedups on average. Second, Vine outperforms Vine (1-level pruning) up to 4× due to the efficient two-level pruning. Third, both static and adaptive parallelism control help improve the query performance over the baseline, and our adaptive control provides a better performance improvement for different queries on different graphs. Last but not least, compared to the state-of-the-art approximate solution, Vine produces exact solutions at a competitive or even better performance without time-consuming preprocessing.

The significant improvement on accelerating exact CSP on GPUs enables some potential applications of exact CSP that were not used before because of its low performance. For example, in network cases with response time requirements of 10 ms, traditional approaches take more than one second, which cannot fulfill the rigid user requirement. In safety-critical cases, highly accurate or exact solutions are required to route automated vehicles to avoid collisions [8, 29]. Vine has the potential of enabling exact CSPs in those applications, because 1) Vine is much faster in finding exact results than other existing exact solutions, and 2) given a time budget of execution, Vine provides comparable or even better results than the state-of-the-art approximate solution.

Limitations of Vine. First, Vine leverages the support of UVA between CPUs and GPUs to handle out-of-core processing of labels.



(a) Preprocessing time.



(b) Relative error.

Figure 14: Preprocessing time and relative error of COLA with different configuration (i.e., α) on road networks.

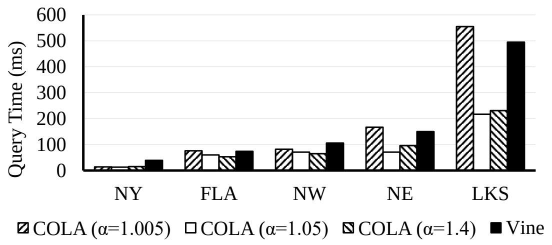


Figure 15: Execution time of Vine and COLA with different α on different road networks.

The total main memory space in the system limits the graph size that Vine can handle. It will be interesting future work to study 1) how to efficiently support even large graphs on the disk, and 2) how to scale efficiently in a multi-node GPU cluster.

Second, the CSP problem is NP-hard [14] and Vine only provides empirical speedups by proposing efficient parallel approaches. We acknowledge that Vine is not a new algorithm or an approach that is theoretically better than existing ones. Nevertheless, the labeling algorithm solves the CSP problem in pseudo-polynomial time [13]. Moreover, as mentioned in [27, 30], many real-world applications in practice do not exhibit worst-case properties. In this paper, we demonstrate that Vine efficiently parallelizes the existing exact algorithm, leveraging GPU for a high performance in practice.

8 CONCLUSIONS

Motivated by the recent rise of emerging applications that require efficient and exact solutions for CSP, we present Vine, a novel and practical framework for accelerating exact CSPs on the GPU. There are two techniques proposed to tackle the extremely large number of subproblems. First, we develop an efficient two-level pruning to eliminate the unpromising subproblems by making good use of the hierarchical memory on the GPU. Second, we propose a lightweight adaptive model to control the degree of parallelism that adapts to the number of subproblems and their growth at runtime. Vine achieves more than an order of magnitude speedup over the CPU counterpart. Compared to the state-of-the-art approximate solution, Vine produces exact solutions at a competitive or even lower latency without costly preprocessing. With the novel design and optimizations on GPUs, Vine can be a viable and cost-effective solution for exact CSPs in many emerging applications.

ACKNOWLEDGMENTS

This work is in part supported by a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore.

Yuchen Li's work is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award No.: MOE2019-T2-2-065). We also thank Liyun Dai, Guangyi Qiao, and Shiheng Chen for the initial discussions of the work.

REFERENCES

[1] Chris Chatfield and Haipeng Xing. 2019. *The analysis of time series: an introduction with R.*

[2] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. 2014. Work-efficient parallel GPU methods for single-source shortest paths. In *IPDPS (2014).* 349–359.

[3] Camil Demetrescu, Andrew V Goldberg, and David S Johnson. 2009. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge.* Vol. 74.

[4] Martin Desrochers, Jacques Desrosiers, and Marius Solomon. 1992. A new optimization algorithm for the vehicle routing problem with time windows. *Operations research* 40, 2 (1992), 342–354.

[5] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[6] Kevin Dorling, Jordan Heinrichs, Geoffrey G Messier, and Sebastian Magierowski. 2016. Vehicle routing problems for drone delivery. *Transactions on Systems, Man, and Cybernetics: Systems* 47, 1 (2016), 70–85.

[7] Dominique Feillet, Pierre Dejax, Michel Gendreau, and Cyrille Gueguen. 2004. An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. *Networks: An International Journal* 44, 3 (2004), 216–229.

[8] Ewgenij Gawrilow, Ekkehard Köhler, Rolf H Möhring, and Björn Stenzel. 2008. Dynamic routing of automated guided vehicles in real-time. In *Mathematics–Key Technology for the Future.* 165–177.

[9] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. 2019. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology* 34, 2 (2019), 339–371.

[10] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *SIGMOD (2020).* 1067–1082.

[11] Wentian Guo, Yuchen Li, Mo Sha, and Kian-Lee Tan. 2017. Parallel personalized pagerank on dynamic graphs. *Proceedings of the VLDB Endowment* 11, 1 (2017), 93–106.

[12] Wentian Guo, Yuchen Li, and Kian-Lee Tan. 2020. Exploiting Reuse for GPU Subgraph Enumeration. *Transactions on Knowledge and Data Engineering* (2020).

[13] Pierre Hansen. 1980. Bicriterion path problems. In *Multiple criteria decision making theory and application.* 109–127.

[14] Refael Hassin. 1992. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations research* 17, 1 (1992), 36–42.

[15] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K Govindaraju, and Tuyong Wang. 2008. Mars: a MapReduce framework on graphics processors. In *PACT (2008).* 260–269.

[16] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P Sadayappan. 2017. Multigraph: Efficient graph processing on gpus. In *PACT (2017).* 27–40.

[17] Saghar Hosseini, Ran Dai, and Mehran Mesbahi. 2013. Optimal path planning and power allocation for a long endurance solar-powered UAV. In *ACC (2013).* 2588–2593.

[18] Stefan Irnich and Guy Desaulniers. 2005. Shortest path problems with resource constraints. In *Column generation.* 33–65.

[19] Hans C Joksch. 1966. The shortest route problem with constraints. *Journal of Mathematical analysis and applications* 14, 2 (1966), 191–197.

[20] Daniel Jünger, Christian Hundt, and Bertil Schmidt. 2018. WarpDrive: Massively parallel hashing on multi-GPU nodes. In *IPDPS (2018).* 441–450.

[21] Alpar Juttner, Balazs Szviatovski, Ildikó Mécs, and Zsolt Rajkó. 2001. Lagrange relaxation based method for the QoS routing problem. In *INFOCOM (2001)*, Vol. 2. 859–868.

[22] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *HPDC (2014).* 239–252.

[23] Steven J Leon, Ion Bica, and Tiina Hohn. 1998. *Linear algebra with applications.* Vol. 6.

[24] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In *SIGKDD (2005).* 177–187.

[25] Guanfeng Liu, Yan Wang, and Mehmet A Orgun. 2010. Optimal social trust path selection in complex social networks. In *AAAI (2010).* 1391–1398.

[26] Hang Liu and H Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. In *SC (2015).* 1–12.

[27] Lawrence Mandow and JL Pérez De La Cruz. 2009. A memory-efficient search strategy for multiobjective shortest path problems. In *AAAI (2009).* 25–32.

[28] Drexl Michael. 2006. Boost Graph Library: Resource-Constrained Shortest Paths. https://www.boost.org/doc/libs/1_73_0/libs/graph/doc/r_c_shortest_paths.html.

[29] Rolf H Möhring, Ekkehard Köhler, Ewgenij Gawrilow, and Björn Stenzel. 2005. Conflict-free real-time AGV routing. In *Operations Research.* 18–24.

[30] Matthias Müller-Hannemann and Karsten Weihe. 2001. Pareto shortest paths is often feasible in practice. In *WAE (2001).* 185–197.

[31] Luigi Di Puglia Pugliese and Francesca Guerriero. 2013. A survey of resource constrained shortest path problems: Exact solution approaches. *Networks* 62, 3 (2013), 183–200.

[32] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *AAAI (2015).*

[33] Antonio Sedeño-Noda and Sergio Alonso-Rodríguez. 2015. An enhanced K-SP algorithm with pruning strategies to solve the constrained shortest path problem. *Appl. Math. Comput.* 265 (2015), 602–618.

[34] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating dynamic graph analytics on GPUs. *Proceedings of the VLDB Endowment* 11, 1 (2017), 107–120.

[35] Mo Sha, Yuchen Li, and Kian-Lee Tan. 2019. Gpu-based graph traversal on compressed graphs. In *SIGMOD (2019).* 775–792.

[36] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: A survey. *Computing Surveys* 50, 6 (2018), 1–35.

[37] Sabine Storandt. 2012. Route planning for bicycles—exact constrained shortest paths made practical via contraction hierarchy. In *ICAPS (2012).*

[38] George Tsaggouris and Christos Zaroliagis. 2009. Multiobjective optimization: Improved FPTAS for shortest paths and non-linear objectives with applications. *TOCS (2009)* 45, 1 (2009), 162–186.

[39] Piet Van Mieghem, Hans De Neve, and Fernando Kuipers. 2001. Hop-by-hop quality of service routing. *Computer Networks* 37, 3-4 (2001), 407–423.

[40] David Wagner. 2002. A generalized birthday problem. In *CRYPTO (2002).* 288–304.

[41] Sibo Wang, Xiaokui Xiao, Yin Yang, and Wenqing Lin. 2016. Effective Indexing for Approximate Constrained Shortest Path Queries on Large Road Networks. *Proceedings of the VLDB Endowment* 10, 2 (2016).

[42] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *PPoPP (2016).* 265–266.

[43] Alan R Washburn. 1990. Continuous Autorouters, with an Application to Submarines. (1990).

[44] Michael Zabarankin, Stan Uryasev, and Robert Murphey. 2006. Aircraft routing under the risk of detection. *Naval Research Logistics* 53, 8 (2006), 728–747.

[45] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *Transactions on Parallel and Distributed Systems* 25, 6 (June 2014), 1543–1552.

[46] Mark Ziegelmann. 2007. *Constrained Shortest Paths and Related Problems - Constrained Network Optimization.*

A ADDITIONAL IMPLEMENTATION DETAILS

CPU baseline. We have developed an efficient parallel implementation on the CPU. Specifically, the CPU-based implementation takes advantage of the fine-grained dynamic scheduling provided by OpenMP. It leverages efficient hashing and dynamic memory management provided by the STL libraries. However, our proposed techniques like adaptive DOP are more specific to GPU, which are not applied to the CPU implementations. The reasons are listed as follows.

First, the two-level pruning is not designed for the CPU. The two-level pruning only applies to the GPU as it is designed to overcome the costly dynamic memory management on GPUs, especially when we perform pruning in parallel. On the CPU, the pruning can be simpler and more effective because CPU threads can easily manipulate dynamic-sized label lists.

Second, the DOP of the CPU implementation is fixed at 40 in our experimental evaluation, as the adaptive parallelism has a small impact on the CPU. One the one hand, the adaptive model is specifically designed to limit the number of subproblems for architectures with low memory-to-core ratio (i.e., available memory divided by the number of cores). Particularly, the GPU we used only has 12GB memory and more than three thousand cores, while the CPU has 256GB memory with 20 cores (40 threads with hyper-threading

Listing 3: An example of Vine for SPPTW.

```
class Edge{
 int len, time;
 int openT, closeT;
};// end of edge class
class Label{
 int vid, l, c; Label *pre = NULL;
 void Expansion (const Edge &e, Label &lv) {
  lv.l = this->l+e.l;
  if (this->c < e.openT) //edge not available
   lv.c = e.openT + e.time;
  else
   lv.c = this->c + e.time;
  if (lv.c > e.closeT)
   lv.c = -1;
  lv.pre = this;
 }
 bool Feasibility () {
  return this.c > 0;
 }
 bool Dominance (const Label &l) {
  return this.l <= l.l && this.c <= l.c;
 }
};// end of label class
```

Listing 2: An example of Vine for 2-CSP.

```
class Edge{
 int len, cost;
};// end of edge class
class Label{
 int vid, l, r, h; Label *pre = NULL;
 void Expansion (Edge e, Label &lv) {
  lv.l = this->l+e.l;
  lv.c = this->c+e.c;
  lv.h = this->h + 1;
  lv.pre = this;
 }
 bool Feasibility () {
  return this.c <= cost_budget
      && this.h <= hop_limit;
 }
 bool Dominance (const Label &l) {
  return this.l <= l.l && this.c <= l.c
      && this.h <= l.h;
 }
};// end of label class
```

enabled). On the other hand, compared to tens of thousands of threads on the GPU, the smaller number of CPU threads tend to generate a much smaller number of non-optimal problems. Thus, the CPU version is less likely to overwhelm the main memory by unpromising subproblems. We also experimentally verify that applying the adaptive parallelism control has little impact on the CPU-based implementations (showing less 1% differences to the static control of DOP).

B EXTENSIONS OF VINE

Multi-constraint shortest path problem. Vine is general to be extended for finding multi-constraint shortest paths. Here, we present two examples, as shown in code Listings 2 and 3. The first application is a variant of vehicle routing problem with an additional constraint that limits the number of vertices visited. Since there are two constraints, we denoted it as 2-CSP. The second application is the shortest path problem with time window constraints (SPPTW) [4], where each edge will only be available between the open time and close time of its time window.

Dynamic graphs support. We have two points for extending Vine to support dynamic graphs. First, in comparison with existing approaches with indexes, Vine is more friendly to dynamic graphs in the sense that Vine can process CSP queries without time-consuming preprocessing or index maintenance. Second, to further extend Vine for supporting dynamic graphs, we can seamlessly extend Vine's graph storage using GPMA [34], a GPU-based dynamic graph storage scheme. Vine works on a snapshot of the graph provided by GPMA. It could be more interesting to study how to incrementally compute CSP on dynamic graphs.

C DISCUSSION

Vine on dense graphs. Due to the NP-hardness of CSP, the performance of Vine can worsen, for example, when the graph is very dense. Fig. 16 shows Vine's performance on dense graphs solving CSP queries with the same setting as the vehicle routing problem, where we apply a set of synthetic graphs with $|V|$=0.01M and varying degrees. We observe that the execution time increases dramatically when the average degree is less than 1,800. The reason is that a denser graph provides more routing options, which makes the optimal one harder to find. As a result, it takes Vine more than a minute to solve CSP. Still, Vine shows up to two orders of magnitude speedups over the parallel CPU implementation.

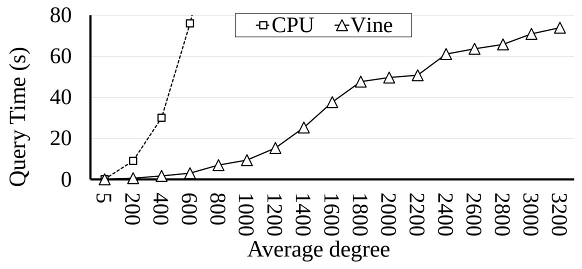


Figure 16: The query execution time of Vine on synthetic graphs with 0.01M vertices and varying degrees, solving Q2.