Parallel Point-to-Point Shortest Paths and Batch Queries

Xiaojun Dong
UC Riverside
Riverside, CA, USA
xdong038@ucr.edu

Andy Li
UC Riverside
Riverside, CA, USA
ali164@ucr.edu

Yan Gu
UC Riverside
Riverside, CA, USA
ygu@cs.ucr.edu

Yihan Sun
UC Riverside
Riverside, CA, USA
yihans@cs.ucr.edu

Abstract

We propose Orionet, efficient parallel implementations of Point-to-Point Shortest Paths (PPSP) queries using bidirectional search (BiDS) and other heuristics, with an additional focus on batch PPSP queries. We present a framework for parallel PPSP built on existing single-source shortest paths (SSSP) frameworks by incorporating novel pruning conditions. As a result, we develop efficient parallel PPSP algorithms based on early termination, bidirectional search, A* search, and bidirectional A*—all with simple and efficient implementations.

We extend our idea to batch PPSP queries, which are widely used in real-world scenarios. We first design a simple and flexible abstraction to represent the batch so PPSP can leverage the shared information of the batch. Orionet formalizes the batch as a *query graph* represented by edges between queried sources and targets. In this way, we directly extended our PPSP framework to batched queries in a simple and efficient way.

We evaluate Orionet on both single and batch PPSP queries using various graph types and distance percentiles of queried pairs, and compare it against two baselines, GraphIt and MBQ. Both of them support parallel single PPSP and A* using unidirectional search. On 14 graphs we tested, on average, our bidirectional search is 2.9× faster than GraphIt, and 6.8× faster than MBQ. Our bidirectional A* is 4.4× and 6.2× faster than the A* in GraphIt and MBQ, respectively. For batched PPSP queries, we also provide in-depth experimental evaluation, and show that Orionet provides strong performance compared to the plain solutions.

CCS Concepts

• **Theory of computation** → **Shortest paths**; **Parallel algorithms**; **Shared memory algorithms**; **Graph algorithms analysis**.

Keywords

Point-to-Point Shortest Paths, Bidirectional Search, A*, Parallel Algorithms, Graph Algorithms

ACM Reference Format:
Xiaojun Dong, Andy Li, Yan Gu, and Yihan Sun. 2025. Parallel Point-to-Point Shortest Paths and Batch Queries. In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28-August 1, 2025, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3694906.3743311

1 Introduction

In this paper, we propose Orionet, an efficient parallel library for Point-to-Point Shortest Paths (PPSP) queries using bidirectional search and other heuristics, with the additional support of *batch PPSP queries*. PPSP is an important graph processing problem. Given a graph $G = (V, E)$ and two vertices $s, t \in V$, a PPSP query finds the shortest paths from s to t. For simplicity in description, we assume the graphs are undirected in this paper, but all techniques in this paper also apply to directed graphs. It is one of the most widely used primitives on graphs with applications such as navigation, network design, and artificial intelligence [1, 37, 41, 42, 47].

Although in theory, PPSP has the same asymptotic cost as the single-source shortest paths (SSSP) problem (i.e., finding the shortest paths from s to all $v \in V$), in practice, many existing techniques can accelerate PPSP. One of the most widely used techniques is the ***bidirectional search*** (BiDS). At a high level, a BiDS runs SSSP (e.g., Dijkstra's algorithm) from both s and t, with a stop (pruning) condition, such that a PPSP query does not need to traverse the entire graph. Hence, the cost of BiDS can be substantially lower than a complete SSSP query from s (see Fig. 1 as an illustration). On graphs with geometric information, other heuristics, such as A*, can also be used, either independently or together with BiDS.

Sequentially, BiDS has been widely studied and shown to be effective in accelerating PPSP queries. However, despite related work on parallel PPSP [4, 17, 28, 63, 67], we are unaware of any work *parallelizing the BiDS*. One major reason for this gap is the inherent distinction in the approach of parallel SSSP compared to its sequential counterpart. Sequentially, the efficiency of BiDS comes from the fact that SSSP queries from both ends visit vertices in *increasing distance order (i.e., Dijkstra's ordering)*. The standard stop condition for BiDS is when the first vertex v is settled (finalized) from both directions. Hence, BiDS saves much work by skipping vertices, such as those that are further than v from s (and similarly for the backward search). However, to achieve high parallelism, practical parallel SSSP algorithms usually do not visit vertices in ascending order of distance (consider parallel Bellman-Ford). Therefore, it remains unclear *whether and how the bidirectional search, shown effective for sequential PPSP queries, can be parallelized* ***correctly*** *and* ***efficiently***.

Designing parallel A* is relatively straightforward. A* incorporates a heuristic function from the current vertex to the target t, adding it to the tentative distance from s to prune the search in the "wrong" direction. Many existing parallel SSSP algorithms can be adapted for A* [19, 66, 67]. Interestingly, BiDS and A* can be combined as bidirectional A* (BiD-A*). However, all existing sequential BiD-A* algorithms [30, 34] rely on strictly increasing distance orders, similar to BiDS. Thus, efficiently parallelizing BiD-A* also remains an open challenge.

The first contribution of Orionet is ***a framework of algorithms and implementations for efficient parallel PPSP, using techniques including bidirectional search, A* search, and bidirectional A* search.*** To achieve high performance and low coding effort, we adopt a parallel SSSP framework called *stepping algorithms* [19]. We carefully redesigned the framework for PPSP queries, which abstract three user-defined functions for different approaches: 1) the setup of the initial frontier (vertices to be processed in the first round), 2) the prune condition to skip certain vertices, and 3) the process to update the current best answer. Interestingly, this abstraction also extends to batch PPSP queries, which we describe in detail below. Our key algorithmic insight mainly lies in the new prune conditions, which correctly and effectively reduce the searching space, and are also simple for implementation in parallel. We provide the technical details in Sec. 3. Experimentally, we show that our new approaches in Orionet based on BiDS, A*, and BiD-A* greatly outperform existing baselines.

In addition to single PPSP queries, Orionet also provides a simple and flexible interface to support ***batch PPSP queries***. In many real-world scenarios, multiple PPSP queries, either from independent users or as components in the same complex query (see examples below), are required to be evaluated in a short time. Batch PPSP has recently gained much attention and has been studied in many recent papers [27, 28, 36, 43, 44, 57, 61, 62]. Most of them are system work and study techniques such as caching or relabeling to take advantage of multiple SSSP searches. However, we observe a gap in the study of batch PPSP queries: this concept is used to refer to many different but closely related query types. Here, we list several "batch PPSP" query types with possible real-world applications as motivating examples below.

- ***Single-source Many-target (SSMT)*** and symmetrically Many-Source Single-Target: find the shortest paths from one source s to a set of targets $T \subseteq V$. Example: Finding the shortest path from the current location to any nearby Walmart.
- ***Pairwise***: find shortest paths from all sources in $S \subseteq V$ to all targets in $T \subseteq V$. Example: Finding the shortest paths from all Walmart and all their warehouses in a region.
- ***Multi-Stop***: find shortest paths from $s = t_0$ visiting a list of targets $t_1, t_2, \ldots, t_k$ in turn, i.e., a batch of $\{(t_0, t_1), (t_1, t_2), \ldots (t_{k-1}, t_k)\}$. Example: planning a trip with multiple stops on the way.
- ***Subset APSP*** (All-Pairs Shortest Paths): find pairwise shortest paths among a subset of vertices $V' \subset V$. Many algorithms use subset APSP as a building block, for instance, the ideas of hopsets [13, 24] and landmarks [53, 58] for computing approximate shortest paths.
- ***Arbitrary Batch***: find shortest paths for a list of s-t pairs, which can be disjoint or overlapping.

Most existing work focuses on one or a subset of the special cases and lacks a general interface to leverage the shared information from arbitrary query batches. Among the work we know of, some focus on pairwise queries [27, 28, 36], and some others focus on SSMT queries [62].

One may also notice that due to the specificity of each query, the best strategy for them also remains elusive. For instance, the SSMT query with a small number of targets may still benefit from running BiDS from all vertices, but when the target set T becomes larger, one SSSP query from the source may give the best performance. A more interesting example is the multi-stop queries, where the best SSSP-based strategy may be running SSSP from *every other vertex* t_{2i+1} (when the graph is undirected) and combining the results of all $(t_{2i+1}, t_{2i}), (t_{2i+1}, t_{2i+2})$ pairs.

The second contribution of Orionet is a simple abstraction of arbitrary batch PPSP queries, along with several efficient implementations. Orionet formalizes all PPSP queries as a ***query graph*** $G_q = (V_q, E_q)$, where V_q contains all vertices in the query batch, and each edge $(q_i, q_j) \in E_q$ indicates a PPSP query between q_i and q_j. Orionet then analyzes the query graph and attempts to reuse the shortest path information of all the queried vertices. Using the query graph, we develop algorithms for batch PPSP queries based on both BiDS and SSSP. For the BiDS-based algorithm, we extend our PPSP framework to support batch queries by carefully redefining the three functions mentioned above. Specifically, this requires carefully setting the pruning conditions based on all queries in the batch. As mentioned, when a vertex in G_q is involved in many s-t pairs, using an SSSP from this vertex can likely give better performance. A naïve approach is to run SSSP from all sources s_i and save some work in the case that there are many targets associated with it. By abstracting the query graph, a more interesting observation is that the best SSSP-based strategy is to find the ***vertex cover*** of G_q and run SSSP from the cover. For example, for the multi-stop query, the query graph forms a chain, and the vertex cover consists of every other vertex on it. This SSSP-based approach is integrated into Orionet, and for certain special queries, it can outperform BiDS.

For all our algorithms, we highlight their ***simplicity, generality***, as well as ***efficiency***. By modeling related queries into the framework in Alg. 2, each approach can be concisely described by the instantiation with the three user-defined functions. The simplicity also enables high performance with relatively low coding effort, as most optimizations for parallel SSSP can be directly used.

We evaluated Orionet on both single and batch PPSP queries using 14 graphs with different types. For single PPSP queries, we tested Orionet using various distance percentile (1%, 50%, and 99% closest) query pairs. and compared it against two baselines, GraphIt (CGO'20) [67] and MBQ (SPAA'24) [66]. Both of them only support parallel single PPSP and A* using unidirectional search. In Sec. 6, we show our algorithm with bidirectional search achieves significant speedup over the baselines. On average, across all tests, our BiDS is 2.9× faster than GraphIt, and 6.8× faster than MBQ (even excluding MBQ's timeout cases). Our bidirectional A* is 4.4× and 6.2× faster than the A* in GraphIt and MBQ, respectively. These results demonstrate the effectiveness of Orionet, as well as the proposed algorithmic insights in this paper. For batched queries, we provide in-depth experimental evaluation to compare the two solutions in Orionet (BiDS-based and SSSP-based) on various query graph patterns. Our code is publicly available [20]. We present more results and analyses in the full version of this paper [21]. We believe this is the first systematic study of BiDS, A*, and BiD-A* for PPSP, and batch PPSP queries in the parallel setting.

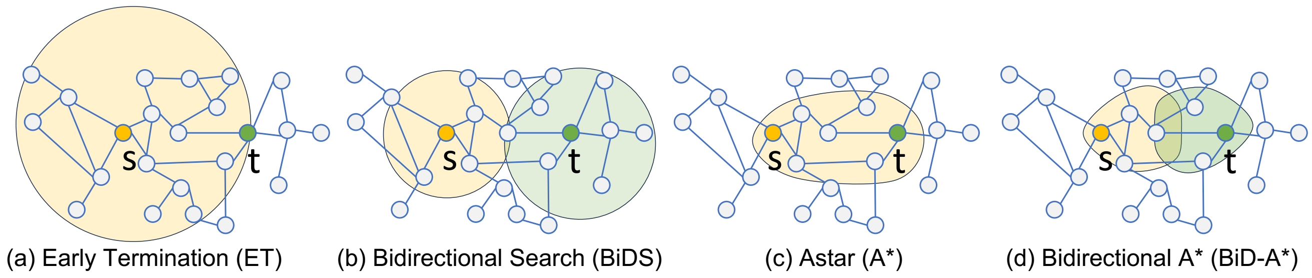

s
t
(a) Early Termination (ET)
(b) Bidirectional Search (BiDS)
(c) Astar (A*)
(d) Bidirectional A* (BiD-A*)

Figure 1: Illustration for PPSP algorithms with early termination, bidirectional search, A*, and bidirectional A*. Details can be found in Sec. 3. (a) Early Termination (ET) runs SSSP from s until t is settled. All vertices with distances smaller than t are processed. (b) Bidirectional Search (BiDS) runs SSSP from both s and t, until both sides settle a common vertex. Each search only touches a small range of vertices around s and t. (c) Astar (A*) runs SSSP from s with heuristic information. The search is roughly "guided" towards t. (d) Bidirectional A* (BiD-A*) combines the advantages of both BiDS and A*. It runs SSSP from both s and t and avoids searching vertices from s that are away from t, and similarly for t.

General Notations:

G	: input graph $G = (V, E)$	$\delta[v]$	: tentative distance of v
$N(v)$	: neighbor set of v	$d(u, v)$	: true distance from u to v
u-v	: path from u to v	$h(v)$	: heuristic of v

For single PPSP:		For batch PPSP:	
s	: source	G_q	: query graph = $G_q(V_q, E_q)$
t	: target	$N_q(v)$	: neighbor set of v in G_q
$v^{\langle+\rangle}$	: v from the source	q_i	: the i-th source in V_q
$v^{\langle-\rangle}$	: v from the target	$v^{\langle i\rangle}$	: v from q_i
μ	: tentative distance from s to t	$\mu[\langle q_i, q_j\rangle]$	: tentative distance from q_i to q_j
		$\mu_{\max}[i]$	: search radius from source q_i

Table 1: Notations.

Algorithm 1: The Stepping Algorithms Framework

```
Input: A graph $G = (V, E, w)$ and a source node $s$.
Output: The shortest path distances $\delta[\cdot]$ from $s$.
1 $\delta[\cdot] \leftarrow +\infty$
2 $\delta[s] \leftarrow 0$, $F$.Add($s$)        // Initialize and add $s$ to the frontier $F$
3 while $|F| > 0$ do
4     $\theta \leftarrow$ GetDist($F$)   // Get the distance threshold based on the frontier
      // process all $u$ with tentative distances no larger than $\theta$
5     ParallelForEach $u \in F$.Extract($\theta$) do
6         ParallelForEach $v \in N(u)$ do
7             if write_min($\delta[v], \delta[u] + w(u, v)$) then
8                 $F$.Add($v$)   // Add $v$ to the frontier $F$ if $v \notin F$ previously
9 return $\delta[\cdot]$
```

2 Preliminaries

Computational Models. We use the standard binary fork-join model [9, 14]. We assume a set of threads that share a common memory. A process can `fork` two child software threads to work in parallel. When both children complete, the parent process continues. We use the atomic operation `write_min`(p, v), which reads the memory location pointed to by p and writes value v to it if v is smaller than the current value. It returns *true* if the update is successful and *false* otherwise.

Notations. We consider a weighted graph $G = (V, E, w)$ with $n = |V|$, $m = |E|$, and an edge weight function $w : E \to \mathbb{R}^+$. For $v \in V$, we define $N(v) = \{u \mid (v, u) \in E\}$ as the ***neighbor set*** of v. For PPSP queries, we use s as the source and t as the target. In our algorithms, we always maintain $\delta[v]$ as the ***tentative distance*** from the source to vertex v, which will be updated in the algorithm by the relaxations on v. We also use $\delta[v^{\langle\oplus\rangle}]$ to denote the distance for the same vertex v from different sources. For example, in bidirectional search, we use $\delta[v^{\langle+\rangle}]$ to denote the tentative distance from the source and $\delta[v^{\langle-\rangle}]$ from the target. We also denote $d(s, v)$ as the ***true distance*** from s to v. With clear context, we simplify it to $d(v)$. We say a vertex is ***settled*** if its true shortest distance has been computed in $\delta[v]$. A list of notations is summarized in Tab. 1.

Stepping Algorithm Framework. The stepping algorithm framework proposed in [19], is a high-level abstraction of many existing parallel SSSP algorithms, including Dijkstra [12, 18], Bellman-Ford [7, 26], Δ-Stepping [46], ρ-Stepping [19], and more [10, 55]. The framework (see Alg. 1) runs in steps and maintains a ***frontier*** F as the set of vertices to be processed. In each step, it extracts vertices (line 5) that have tentative distances within a certain *threshold* θ (line 4) from the frontier and relaxes their neighbors in parallel (line 6). If a vertex is successfully relaxed (line 7), it will be added to the next frontier (line 8). This framework applies to various algorithms by plugging in different functions for GetDist, which decides the threshold θ of the current step. The algorithmic ideas in this paper can be combined with many SSSP algorithms in the stepping framework. In Orionet, we use Δ^*-stepping [19, 46] as it is one of the most commonly used parallel SSSP algorithms. In Δ^*-stepping, the i-th call of GetDist() returns $i \cdot \Delta$, where Δ is a user-defined parameter.

Other Shortest-Path Algorithms. As one of the most studied problems, there have been many insightful algorithms designed for computing shortest paths. We will review them in Sec. 7.

3 Algorithms for Point-to-Point Shortest Paths

In this section, we present our point-to-point shortest path (PPSP) framework and implementations of early termination (ET), A*, BiDS, and BiD-A* based on this framework. All these techniques have been shown to be effective for sequential PPSP. Although early termination and A* have also been studied in parallel settings, we are aware of little work on using their bidirectional counterparts to achieve high-performance PPSP. A key challenge we observed is that the efficiency of sequential BiDS relies on strong ordering in vertex visits (Dijkstra's order), which is not preserved in parallel

Algorithm 2: Point-to-Point Shortest Path Algorithm Framework

```
Input: A graph $G = (V, E, w)$.
Output: Shortest path between a given pair of vertices or multiple such
        query pairs.
1  $\mu \leftarrow +\infty, \delta[\cdot] \leftarrow +\infty$
2  Init()      // Initialize the frontier $F$ and the distance $\delta[\cdot]$ for the sources
3  while $|F| > 0$ do
4      $\theta \leftarrow$ GetDist($F$)
       // $\oplus$ denotes the source of $u$
5      ParallelForEach $u^{\langle\oplus\rangle} \in F$.Extract($\theta$) do
           // Prune the search at $u^{\langle\oplus\rangle}$ if it cannot contribute to $\mu$
6          if $\neg$Prune($u^{\langle\oplus\rangle}$) then
7              ParallelForEach $v \in N(u)$ do
8                  if write_min($\delta[v^{\langle\oplus\rangle}], \delta[u^{\langle\oplus\rangle}] + w(u, v)$) then
                       // Update $\mu$ using the path through $v$
9                      $\mu$ = UpdateDistance($v^{\langle\oplus\rangle}$)
10                     if $\neg$Prune($v^{\langle\oplus\rangle}$) then F.Add($v^{\langle\oplus\rangle}$)
11 return $\mu$
```

SSSP algorithms. Parallel SSSP algorithms usually update vertices in batches or asynchronously, where strong ordering is not enforced. Hence, in parallel, non-trivial adaptations are required to apply the ideas of BiDS, A*, and BiD-A* effectively to improve performance.

To overcome these challenges, we first design a PPSP framework adapted from the stepping algorithms framework in [19]. Our framework eases the coding effort by abstracting the pruning conditions and, more importantly, facilitates our algorithm on batch PPSP queries. To incorporate bidirectional search, we design a pruning criterion independent of Dijkstra's order and present its adaptation for BiD-A*. Both our BiDS and BiD-A* fit seamlessly into our framework and significantly improve the performance.

Note that both bidirectional search and A* are considered heuristics, which do not improve the worst-case bounds. Our goal in this paper is to show that all these techniques can be the same effective in the parallel setting as in the classic sequential case. The work and span bounds for Orionet are upper-bounded by the underlying parallel SSSP algorithm [19].

In the rest of this section, we first present our framework (Sec. 3.1), then discuss early termination (Sec. 3.2) and A* (Sec. 3.3) as examples to show the simplicity of our framework. We then present our new algorithms on BiDS (Sec. 3.4) and BiD-A* (Sec. 3.5) and also prove the correctness.

3.1 Our Framework

We present our framework in Alg. 2. Under this framework, variants of our PPSP algorithms differ only in how they 1) find the first frontier of a stepping algorithm, specified by the function Init; 2) determine whether further search on a vertex can be pruned, specified by the function Prune; and 3) update the s-t path distance accordingly, specified by the function UpdateDistance.

The algorithm begins by initializing the frontier and setting the source distance to zero using the Init function (line 2) and then proceeds in steps, similar to a regular stepping algorithm. The main difference is that since searches can be from multiple sources (e.g., a bidirectional search or batch queries), each vertex may appear in the frontier multiple times (each from a different source). We use $u^{\langle\oplus\rangle}$ to denote an element in the frontier, where $u \in V$ is a vertex and $\oplus$ denotes the source of the search. For example, in bidirectional searches, a vertex u can appear in both forward and backward searches. We use $u^{\langle+\rangle}$ and $u^{\langle-\rangle}$ to denote these two cases, respectively. As a result, the extracted vertices are those with the closest distances across multiple sources. If a vertex $u^{\langle\oplus\rangle}$ is extracted, we first determine whether it should be pruned using the function Prune (line 6). If not, we process its neighbors regularly. Note that whenever Prune(v) is called, v satisfies $\delta[v] \neq \infty$ since v is either the source or has been updated by other vertices. During the algorithm, we maintain a global variable μ, which is the current shortest distance from s to t. If a vertex $v \in N(u)$ can be relaxed, we use UpdateDistance to update the current shortest distance μ with the path s-v-t, ensuring that μ eventually converges to the exact shortest distance from s to t. If the neighbor v does not satisfy the pruning condition, it is added to the frontier (line 10).

One may note that updating μ requires priority updates, but the contention here is light—existing analysis [56] shows that the content can generally be considered as logarithmic.

In the following, we will present our four PPSP algorithms by showing how they instantiate the functions Init, Prune, and UpdateDistance. We summarize the conditions in Tab. 2.

3.2 Early Termination (ET)

For an algorithm in the stepping algorithm framework, the tentative distances $\delta[v]$ for any vertex v only monotonically decreases. Existing work [63, 67] takes advantage of this monotonicity to prune the exploration of vertices v with $\delta[v] \geq \mu$ as they do not contribute to the final distance. This technique is often referred to as *early termination (ET)* and has been used in existing algorithms, including PnP [63], GraphIt [67], and MBQ [66].

Integrating ET to Orionet is straightforward. The Init function simply initializes the distance for s as zero and adds it to the frontier. Since searches all have the same source s, we omit the $\oplus$ parameter in Tab. 2. Recall that the algorithm maintains μ as the current best distance from s to t. The Prune function skips a vertex v with $\delta[v] \geq \mu$. Accordingly, the UpdateDistance function updates μ when a relaxation results in a smaller distance for t.

3.3 Astar (A*)

A* is widely used in pathfinding, informational search, and natural language processing [5, 32, 35, 49, 65]. When used in graph PPSP queries, A* employs a heuristic function $h(\cdot)$, where $h(v)$ provides an estimate (usually a lower bound) of the distance from v to the target t. The estimate can be the geometric distance (e.g., Euclidean distance) from v to t. As exemplified by Dijkstra's algorithm, when selecting a vertex to search, it chooses the vertex v with the smallest value of $\delta[v] + h(v)$, until t is settled. This process is also referred to as best-first search.

The heuristic function usually satisfies two properties: (1) ***Admissibility***: $\forall v \in V, h(v) \leq d(v, t)$. (2) ***Consistency***: $\forall (u, v) \in E$, $h(u) \leq w(u, v) + h(v)$. Note that when $h(t) = 0$ (true for almost all commonly used heuristics), consistency implies admissibility [51]. In this paper, we assume the heuristic function satisfies consistency. If $h(\cdot)$ is consistent, running any shortest path

	Init()	Prune(v)	UpdateDistance(v)
Single-Directional Searches for PPSP: Early Termination (ET) and A* For source $s \in V$ and target $t \in V$, μ maintains the current shortest distance between s and t.			
ET	$\delta[s] \leftarrow 0$, F.Add(s)	$\delta[v] \geq \mu$	If $v = t$: `write_min`$(\mu, \delta[v])$
A*	$\delta[s] \leftarrow 0$, F.Add(s)	$\delta[v] + h(v) \geq \mu$	If $v = t$: `write_min`$(\mu, \delta[v])$
Bi-Directional Searches for PPSP: Bi-Directional Searches (BiDS) and Bi-Directional A* (BiD-A*) Each vertex $v \in V$ has two copies: $v^{\langle + \rangle}$ denotes the search from the source s, and $v^{\langle - \rangle}$ denotes the search from the target t For source $s \in V$ and target $t \in V$, μ maintains the current shortest distance between s and t.			
BiDS	$\delta[s^{\langle + \rangle}] \leftarrow 0$, F.Add($s^{\langle + \rangle}$) $\delta[t^{\langle - \rangle}] \leftarrow 0$, F.Add($t^{\langle - \rangle}$)	$\delta[v^{\langle \cdot \rangle}] \geq \mu/2$	`write_min`$(\mu, \delta[v^{\langle + \rangle}] + \delta[v^{\langle - \rangle}])$
BiD-A*	$\delta[s^{\langle + \rangle}] \leftarrow 0$, F.Add($s^{\langle + \rangle}$) $\delta[t^{\langle - \rangle}] \leftarrow 0$, F.Add($t^{\langle - \rangle}$)	For Prune($v^{\langle + \rangle}$): $\delta[v^{\langle + \rangle}] + h_F(v) \geq \mu/2$ For Prune($v^{\langle - \rangle}$): $\delta[v^{\langle - \rangle}] + h_B(v) \geq \mu/2$	`write_min`$(\mu, \delta[v^{\langle + \rangle}] + \delta[v^{\langle - \rangle}])$
Multi-Directional Searches for *Batch* PPSP (Multi-PPSP) Let $G_q = (V_q, E_q)$ be the query graph. Each vertex $v \in V$ has $\lvert V_q \rvert$ copies: $v^{\langle i \rangle}$ denotes the search to vertex v from the i-th vertex $q_i \in V_q$. For $\langle q_i, q_j \rangle \in E_q$ (i.e., the distance between q_i and q_j is queried), $\mu[\langle q_i, q_j \rangle]$ maintains the current shortest distance between q_i and q_j.			
Multi-PPSP	For $q_i \in V_q$: $\delta[q_i^{\langle i \rangle}] \leftarrow 0$, F.Add($q_i^{\langle i \rangle}$)	For Prune($v^{\langle i \rangle}$): $\delta[v^{\langle i \rangle}] \geq \mu_{\max}[i]/2$	For Update($v^{\langle i \rangle}$), let q_i be the i-th vertex in V_q: For each $q_j \in N_q(q_i)$: `write_min`$(\mu[\langle q_i, q_j \rangle], \delta[v^{\langle i \rangle}] + \delta[v^{\langle j \rangle}])$ `write_min`$(\mu_{\max}[i], \mu[\langle q_i, q_j \rangle])$

Table 2: Init, Prune, and Update functions of our PPSP algorithms. The superscript i of a vertex v indicates v is searched from i. "ET": early termination (Sec. 3.2). "A*": astar search (Sec. 3.3). "BiDS": Bidirectional search (Sec. 3.4). "BiD-A*": bidirectional astar search (Sec. 3.5). $h(v)$ is the heuristic of v, e.g., based on the geometric distance from v to s and t. "$h_F(v)$": heuristic used in the forward search. "$h_B(v)$": heuristic used in the backward search. Some examples of the heuristic functions are provided in Sec. 3. For all single PPSP algorithms, we maintain a global variable μ, which is the current best distance from s to t. The UpdateDistance function will update this value, which finally converges to the true distance from s to t. Similarly, in the Multi-PPSP algorithm, for each PPSP query $\langle q_i, q_j \rangle \in E_q$ in the batch, we use $\mu[\langle q_i, q_j \rangle]$ to maintain the current shortest distance between q_i and q_j. We also maintain an array of $\mu_{\max}[i] = \max\{\mu[\langle q_i, q_j \rangle] \mid \langle q_i, q_j \rangle \in E_q\}$, which is the farthest distance that needs to be searched from the i-th source.

algorithm on $G = (V, E, w)$ with $h(\cdot)$ applied is equivalent to running it on an induced graph $G' = (V, E, w')$ with modified weights $w'(u, v) = w(u, v) - h(u) + h(v)$. The induced graph has two useful properties.

Fact 3.1. *Given a graph $G = (V, E, w)$ and a consistent heuristic $h(\cdot)$, its induced graph $G' = (V, E, w')$ with the modified weight $w'(u, v) = w(u, v) - h(u) + h(v)$ has two properties: 1) For all $(u, v) \in E$, $w'(u, v)$ is nonnegative; and 2) Given a source s, $d'(s, t) = d(s, t) - h(s) + h(t)$, where $d(s, t)$ and $d'(s, t)$ are the distances between s and t on G and G', respectively.*

Thus, the induced graph modifies the shortest distance between s and t by a constant shift of $h(t) - h(s)$ while preserving the relative distances. This guarantees the feasibility of running Dijkstra (and other algorithms) on G', with the extra benefit that the search prioritizes expanding vertices closer to the target. Fact 3.1 also implies that we can replace Dijkstra's algorithm with parallel SSSP algorithms in our framework, which yield efficient parallel A* algorithms.

When implementing these algorithms in Orionet, the only difference between ET and A* is in the Prune function: a vertex v can be pruned if it satisfies $\delta[v] + h(v) \geq \mu$. This is because $\delta[v] + h(v)$ provides a lower bound on the shortest path through v. If it is already no less than μ, search from v will not improve the solution. Although A* is simple and can be easily integrated into parallel SSSP algorithms, a previous study [67] reports that A* performs slower than ET. We improve A* with memoization, which enables it to outperform ET, as discussed in Sec. 5.

3.4 Bidirectional Search (BiDS)

BiDS for PPSP has been widely studied sequentially [30, 31, 33, 40, 52] and is shown to have better bounds for certain graph types and better practical performance than ET in general. As shown in Fig. 1 (b), although BiDS performs two searches from both s and t, the total search space can be smaller than a standard PPSP since both searches cover smaller radii. The idea of most sequential BiDS algorithms is to run Dijkstra's algorithm from both sides due to the following theorem.

Theorem 3.2 ([52]). *Given a nonnegative weighted Graph $G = (V, E, w)$, a source s, and a target t, one can use Dijkstra's algorithm to search from both s and t, alternate between the two searches in any way, and terminate when a vertex v has been settled by both searches.*

The algorithm is correct regardless of the alternation strategy used, and existing sequential studies mainly focus on finding better alternation strategies. For example, Nicholson's algorithm [48] selects the vertex with minimum tentative distance among both the forward and backward searches, while Dantzig's algorithm [15, 30] alternates strictly between the forward and backward searches.

Unfortunately, parallel SSSP algorithms achieve parallelism by processing multiple vertices simultaneously and have limited information about which vertices have been settled. Therefore, it has been open on how to efficiently use the bidirectional search to accelerate parallel PPSP. To the best of our knowledge, the only parallel PPSP algorithm that applies BiDS is from PnP [63]. However, unlike sequential BiDS, PnP only uses BiDS in preprocessing to predict which direction (from s or t) results in less computation and then uses the standard unidirectional search with ET.

Our BiDS. We design our BiDS by plugging in the new Prune and UpdateDistance functions in Orionet, which are independent of Dijkstra's order. Note that the key difference here is that, in parallel SSSP, an algorithm only knows the tentative distance on vertex v but cannot determine whether and when it converges to the true distance. In contrast, in Dijkstra's order, every vertex popped from the priority queue is settled with its true distance. Our BiDS is based on an observation that, in BiDS, given any upper bound μ of $d(s,t)$, we can skip a vertex v with tentative distance $\delta[v] \geq \mu/2$. This indicates that any further searches beyond this point are wasteful and thus pruned. Unlike existing parallel PPSP work that does not apply or can only partially apply BiDS, our algorithm can apply BiDS throughout the entire search. We prove the correctness of our algorithm in Thm. 3.3.

We now show how to implement BiDS in our framework in Alg. 2. Similar to the sequential BiDS, our BiDS also starts a forward search from s and a backward search from t, so a vertex can have two copies in the frontier. To distinguish these two copies, we use $\oplus \in \{+,-\}$ to denote the source: $v^{\langle + \rangle}$ represents the copy originating from s, and $v^{\langle - \rangle}$ represents the copy from t. We also use $\delta[v^{\langle + \rangle}] = \delta_s[v]$ and $\delta[v^{\langle - \rangle}] = \delta_t[v]$. Since the searches from both s and t start with themselves, we first add $s^{\langle + \rangle}$ and $t^{\langle - \rangle}$ to the frontier and initialize their distances to zero. Before our algorithm processes a vertex $v^{\langle \cdot \rangle}$ from either direction, the Prune function checks whether $\delta[v^{\langle \cdot \rangle}] \geq \mu/2$ and skips it if so. Finally, when processing a vertex v, the UpdateDistance function always attempts to examine the path s–t–v, and updates μ to $\delta[v^{\langle + \rangle}] + \delta[v^{\langle - \rangle}]$ if it provides a lower value. Since the pruning condition depends only on the tentative distances, it is oblivious to whether a vertex is settled. We formally analyze the correctness of our approach with the theorem below.

Theorem 3.3. *Given a nonnegative weighted graph $G = (V,E,w)$, our BiDS algorithm can correctly compute the shortest s-t distance by running any stepping algorithm on parallel SSSP from both s and t and skipping (not relaxing the neighbors of) any vertex v with a tentative distance $\delta[v] \geq \mu/2$, where μ is the current shortest distance between s and t, updated in Alg. 2.*

Proof. Recall that $\delta[v]$ is the tentative distance and finally converges to the true distance $d(v)$, so we have $\delta[v] \geq d(v)$. Assume the true distance between s and t is $\mu' = d(s,t)$. Note that when μ has reached μ' (i.e., $\mu = \mu'$) in the algorithm, then the shortest distance $d(s,t)$ has been computed.

Hence, we now focus on the case when $\mu' < \mu$. As presented in Fig. 2, assume one of the shortest paths between s and t is $P = \{s, v_1, \cdots, v_{m-1}, v_m, v_{m+1}, \cdots, t\}$. Clearly, for all $v \in P$, $d(s,v) + d(v,t) = \mu'$. We assume vertex v_m divides the path into two halves, $P_f = \{s, v_1, \cdots, v_{m-1}\}$ and $P_b = \{v_{m+1}, \cdots, t\}$, such that vertex $v \in P_f$ satisfies $d(s,v) \leq \mu'/2 < \mu/2$. Similarly, vertex $v \in P_b$ satisfies $d(s,v) \geq \mu'/2$, which implies $d(v,t) = \mu' - d(s,v) \leq \mu'/2 < \mu/2$.

We prove by induction that all true distances on the path will be computed, and the true distance $d(s,t)$ will be correctly updated to the variable μ to be μ' and returned. In the forward search, s is

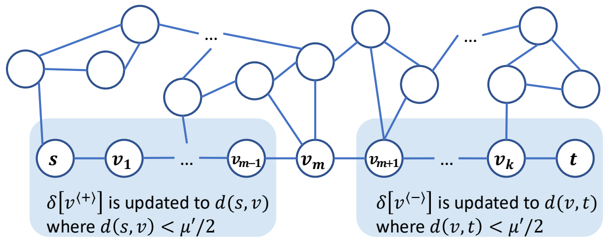

s
v_1
…
v_{m-1}
v_m
v_{m+1}
…
v_k
t
…
…
$\delta[v^{\langle + \rangle}]$ is updated to $d(s,v)$ where $d(s,v) < \mu'/2$
$\delta[v^{\langle - \rangle}]$ is updated to $d(v,t)$ where $d(v,t) < \mu'/2$

Figure 2: Illustration for the proof of Thm. 3.3 for our BiDS algorithm.

the source with $\delta[s^{\langle + \rangle}] = d(s,s) = 0$, which is the base case and not pruned. Hence, s updates v_1 in the first round with $\delta[v_1^{\langle + \rangle}] = d(s,v_1) < \mu/2$ in the forward search. Similarly, all $v_i \in P_f \backslash \{s\}$ are updated by v_{i-1} with $\delta[v_i^{\langle + \rangle}] \leq \mu'/2 \leq \mu/2$ and thus not pruned. Finally, v_{m-1} updates $\delta[v_m^{\langle + \rangle}]$ to the true distance $d(s,v_m)$. Similarly, none of the vertices on P_b are pruned and will finally get their true distance. Finally, v_{m+1} updates $\delta[v_m^{\langle - \rangle}]$ to the true distance $d(v_m,t)$. Since we have the true distance of v_m in both forward and backward searches, we can obtain the true distance between s and t by concatenating the path s–v_m–t. The return value μ is correctly updated by the UpdateDistance function with $\mu = \delta[v^{\langle + \rangle}] + \delta[v^{\langle - \rangle}] = \mu'$, when the later direction is successfully updated. □

Despite its conceptual simplicity, this approach demonstrates very good performance in our experiments and readily fits into our framework. More importantly, we will show that this strategy facilitates our BiD-A* in Sec. 3.5 and our batch PPSP algorithms in Sec. 4. Surprisingly, we did not find existing work that employs such a simple heuristic for BiDS. We attribute such a gap to the inherent difference between sequential and parallel SSSP—sequentially, since only *one* vertex is processed at a time, it is natural (and easy) to find a settled vertex to explore, and thus no "pruning" is needed. We believe the simplicity is particularly interesting as an example of *parallel thinking*—effective parallelization does not always add significant complication to a sequential algorithm, and can be made simple by a different way of thinking.

3.5 Bidirectional Astar (BiD-A*)

With BiDS and A*, it is natural to consider combining them as the bidirectional A^* (BiD-A*). However, BiD-A* is challenging even in the sequential setting. Given two valid heuristic functions $h_s(v)$ estimating the distance to the source and $h_t(v)$ estimating the distance to the target, let $h_F(\cdot)$ be the heuristic used in the forward search (from s) and $h_B(\cdot)$ the heuristic in the backward search (from t). A simple approach is to use $h_F(\cdot) = h_t(\cdot)$ for the forward search and $h_B(\cdot) = h_s(\cdot)$ for the backward search and terminate when the two searches touch. Unfortunately, this solution is incorrect [30, 34]. Intuitively, as in A*, we can consider the induced graph $G' = (V,E,w')$ with modified edge weights $w'(u,v) = w(u,v) - h(u) + h(v)$. In the forward search, an edge (u,v) has weight $w'(u,v) = w(u,v) - h_F(u) + h_F(v)$, while its mirrored edge in the backward search has weight $w'(v,u) = w(v,u) - h_B(v) + h_B(u)$, which are not equal. This difference breaks the consistency of the heuristic and thus leads to incorrect results in this simple solution. To fix this issue, the most commonly used technique [30] is to ensure $h_B(v)+h_F(v) = 0$ for any vertex $v \in V$. Therefore, the modified edge weights w' are identical in both forward and backward searches. We will discuss how our solution achieves a consistent heuristic for BiDS below.

As with BiDS, the ideas in all sequential BiD-A* rely on using Dijkstra's order. To parallelize this idea, we also combine BiD-A* with the high-level idea mentioned in Sec. 3.4 and prove that it can be combined with any consistent heuristic functions.

Our Parallel Bidirectional A*. To enable BiD-A*, our high-level idea is to select the appropriate *consistent* heuristics for both forward and backward searches, ensuring that our algorithms, when running on the original graph with the heuristics, behave as if it were running on a graph (i.e., the induced graph) without the heuristics. This allows us to directly apply our BiDS pruning criterion on the induced graph, as it effectively appears heuristic-free.

To obtain consistency, we adapt the heuristics from previous work [30, 34]. We set $h_F(v) = (h_t(v) - h_s(v))/2$ for the forward search and $h_B(v) = (h_s(v) - h_t(v))/2$ for the backward search, which implies $h_F(v) + h_B(v) = 0$ and thereby preserves heuristic consistency. The intuition here is that, rather than directing the forward search towards the target and the backward search towards the source, both searches are guided towards the "perpendicular bisector" region between the source and target, which is essentially a set of points equidistant from the source and target.

We now present how to implement our BiDS under the framework in Alg. 2. We first add $s^{\langle+\rangle}$ and $t^{\langle-\rangle}$ to the frontier and initialize their distances to zero. These steps are the same as in our BiDS. Note that our algorithm runs in steps, and in each step, it selects a set of vertices within a certain distance threshold (line 4 in Alg. 2). In BiD-A*, we use the sum of the tentative distance and the heuristic function instead of only the tentative distance. In addition, when our algorithm is about to scan a vertex $v^{\langle+\rangle}$ in the forward search, the Prune function checks if $\delta[v^{\langle+\rangle}] + h_F(v) \geq \mu/2$ and skips it if the condition holds. Similarly, for a vertex $v^{\langle-\rangle}$ in the backward search, it checks whether $\delta[v^{\langle-\rangle}] + h_B(v) \geq \mu/2$ instead. Finally, when a vertex v is reached from both directions, an s-t path is found by concatenating s–v and v–t, so the UpdateDistance function updates the current s-t distance μ with $\delta[v^{\langle+\rangle}] + \delta[v^{\langle-\rangle}]$, which will eventually converge to the true distance.

We now show that our algorithm can correctly compute the shortest paths in Thm. 3.4 with *any consistent* heuristic functions. For the page limit, we present the proof in the full paper [21].

Theorem 3.4. *Given a nonnegative weighted graph $G = (V, E, w)$, a source s, a target t, and a consistent heuristic function $h(\cdot)$ (i.e, for all $(u, v) \in E$, $h(u) \leq w(u, v)+h(v)$), our BiD-A* algorithm can correctly compute the shortest s-t distance by running any stepping parallel SSSP algorithms from both s and t, and skipping (i.e., not relaxing the neighbors of) any vertex v having a tentative distance $\delta[v^{\langle+\rangle}]+h(v) \geq \mu/2$ if v originates from the source, or $\delta[v^{\langle-\rangle}] - h(v) \geq \mu/2$ if v originates from the target, where μ is the current shortest s-t distance that is updated in Alg. 2.*

In Sec. 6, we will show that combining BiDS and A* into BiD-A* significantly improves the performance for PPSP.

4 Batch PPSP Queries in Parallel

In practical applications, multiple PPSP queries, either independent or sharing common vertices, can be required in a batch or a complex task. A list of possible query types was reviewed in Sec. 1. Running multiple queries in a batch can be helpful in two aspects. First, it can potentially reduce some redundant searches if the set of queries share common vertices. Second, in the parallel setting, batching multiple queries brings up more work, which can saturate processors to achieve better parallelism.

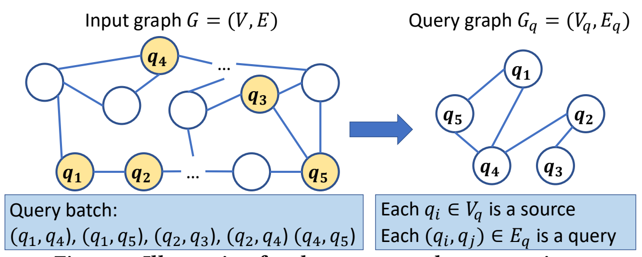
Existing work has considered batching multiple PPSP queries in sequential [27, 28, 36], parallel [61, 62], and distributed settings [43, 44]. Our work differs from them in two aspects. First, each of them usually only focuses on certain query types, such as pairwise queries [27, 28, 36] or single-source multiple-target (SSMT) queries [62]. Second, most of them are system-level improvements for caching or communication cost by techniques such as relabeling, cache line alignment, and some smart vertex marking; the backbone is still some existing PPSP algorithms. In this paper, we focus on algorithmic improvements such as BiDS.

In Orionet, we aim to support *arbitrary batch queries*. We take a sequence of source-target pairs and compute the PPSP among them. A key observation here is that the vertices in the queried batch are often highly relevant: as mentioned in Sec. 1, these queries may come from the same complex query, and the source and target can often overlap. For simple query types such as SSMT, it is easy to reuse the computation since they are all from the same source, but for more complicated cases, it is non-trivial to do so effectively. Our goal is to 1) reuse the computation among the source and/or target vertices as much as possible, 2) keep the implementation reasonably simple, and 3) integrate the useful techniques such as BiDS proposed in Sec. 3. In the following, we first discuss how to preprocess the input queries in Sec. 4.1 and present our BiDS-based algorithms in Sec. 4.2. We also show a baseline algorithm in Sec. 4.3.

4.1 Constructing the Query Graph

One of the advantages of batch PPSP queries is that they can potentially reduce redundant searches. Therefore, the first step of Orionet is to decompose the queries to identify common vertices in the given queries. Given a set of queries $(s, t) \in Q$, we first generate a query graph $G_q = (V_q, E_q)$ such that each query corresponds to an edge in the query graph. As presented in Fig. 7, in the query graph, each vertex $q \in V_q$ denotes a source or target in the query batch, and each edge $(s, t) \in E_q$ denotes a single s-t query in the batch. After constructing the query graph, a batch of general queries boils down to a group of single-source multiple-target (SSMT) queries starting from each $q \in V_q$ and their targets are the neighbors in the query graph (denoted as $N_q(q)$). The structure query graph will help Orionet understand the way to reuse the searches in the batch.

For the special queries mentioned in Sec. 1, each of them has a certain pattern of the query graph. For example, a batch of SSMT queries (or symmetrically many-source single-target) forms a star query graph. The pairwise query batch forms a complete bipartite graph as the query graph. A simple multi-stop query can be represented as a query graph of a chain. When each stop comes with multiple options, there may also be forks branching from a chain. A subset APSP forms a clique query graph on a subset of vertices.


Input graph $G = (V, E)$
Query graph $G_q = (V_q, E_q)$
q_1 q_2 q_3 q_4 q_5
Query batch:
(q_1, q_4), (q_1, q_5), (q_2, q_3), (q_2, q_4) (q_4, q_5)
Each $q_i \in V_q$ is a source
Each $(q_i, q_j) \in E_q$ is a query
Figure 3: Illustration for the query graph construction.

More generally, an arbitrary batch is just a general graph on all vertices (both sources and targets) involved in the batch.

4.2 BiDS-Based Solutions

As discussed in Sec. 3, BiDS can greatly improve performance in a single PPSP query, so algorithmically, it is interesting to see how to integrate BiDS in the batch setting. A straightforward approach is to apply our parallel BiDS algorithm to each query one-by-one serially, which we refer to as the ***Plain*** approach. To further increase parallelism, we can instead run our parallel BiDS algorithm on all queries concurrently, which we refer to as the ***Plain****** approach. However, a batch of queries often share common vertices. After revealing the common vertices, we can reduce the redundant searches originating from the same sources. More importantly, in BiDS, a target can also be considered as a source on undirected graphs. When each vertex can be either source or target and can be involved in multiple PPSP queries, we need to carefully redefine the conditions for pruning and updating the answers. We refer to this approach as the ***Multi-BiDS*** (or ***multi*** for short) approach. Our solution in Orionet is as follows. Since a vertex can be reached from multiple sources, for each vertex u from the original graph, we maintain a copy of it from each of its neighbors in the query graph G_q. We use $u^{\langle i \rangle}$ to denote that vertex u is reached from $q_i \in V_q$, and $\delta[u^{\langle i \rangle}]$ stores the tentative distance from q_i to u (i.e., $\delta[u^{\langle i \rangle}] = \delta_{q_i}[u]$). Our pruning idea is based on pruning the search from a vertex $q_i \in V_q$ when it passes the farthest midpoint of q_i and $v \in N_q(q_i)$.

Although slightly more complicated than a single PPSP query, Orionet also follows the PPSP framework in Alg. 2. To effectively make use of the BiDS, we search from all vertices in the query graph. The Init function starts with adding $q_i \in V$ to the frontier, as all searches start from themselves. Therefore, they add their own copy, which is $q_i^{\langle i \rangle}$ with distance zero, to the frontier.

The problem boils down to setting the Prune and UpdateDistance functions correctly, so we can avoid redundant searches as much as possible. Specifically, Orionet maintains the tentative distances $\mu[\langle q_i, q_j \rangle]$ for each edge $(q_i, q_j) \in E_q$, which are also the outputs of our algorithm. In addition to the best answer for each edge (i.e., query), our algorithm also maintains a “search radius” $\mu_{\max}[i]$ for each $q_i \in V_q$, which indicates that the search starting from the source q_i can be pruned at a vertex u if $\delta[u^{\langle i \rangle}] \geq \mu_{\max}[i]/2$, where $\mu_{\max}[i]$ can be computed by $\mu_{\max}[i] = \max\{\mu[\langle q_i, q_j \rangle] \mid q_j \in N_q(q_i)\}$. If the pruning condition holds, further search on vertex v will not contribute to the final answer to any queries, so we can safely skip $v^{\langle i \rangle}$. When a tentative distance is successfully updated, the UpdateDistance function updates $\mu[\langle \cdot, \cdot \rangle]$ and $\mu_{\max}[\cdot]$ accordingly. For example, if $v^{\langle i \rangle}$ is successfully relaxed, any path distance involving q_i can potentially be updated. Therefore, for $q_j \in N_q(q_i)$, we try to update $\mu[\langle q_i, q_j \rangle]$ by the path q_i–v–q_j, which has a distance $\delta[v^{\langle i \rangle}] + \delta[v^{\langle j \rangle}]$. If $\delta[v^{\langle i \rangle}] + \delta[v^{\langle j \rangle}] < \mu[\langle q_i, q_j \rangle]$, the tentative distance on the edge (q_i, q_j) is updated, thus the search radii $\mu_{\max}[i]$ and $\mu_{\max}[j]$ can potentially be reduced. Then we update the radius $\mu_{\max}[i]$ by $\mu_{\max}[i] = \min\{\mu_{\max}[i], \mu[\langle q_i, q_j \rangle]\}$ (similarly for $\mu_{\max}[j]$).

4.3 SSSP-Based Solutions

Besides the BiDS-based solution mentioned above, one simple way to deal with the batch is to run SSSP from all sources in the batch. We call it a *plain* solution based on SSSP. Although the idea is simple, it remains a reasonably effective solution for cases such as SSMT queries with a moderate number of targets. In our experiments, even with five targets in an SSMT query, running SSSP on the source may outperform other highly optimized solutions.

More generally, we are interested in knowing the minimum number of SSSP queries needed to handle the query graph. Interestingly, for undirected graphs, this reduces to finding a subset of vertices in the query graph to cover all edges, i.e., the ***vertex cover (VC)*** problem. Inspired by this, Orionet includes a solution, which we call the ***VC*** approach: given the query graph, Orionet first finds a vertex cover $V' \subseteq V_q$ of it, runs SSSP from all vertices in V', and combines the results. We note that vertex cover on an arbitrary graph is NP-hard, and finding an optimal solution for a large query graph is infeasible. Orionet finds the VC by enumerating all possibilities for a small query graph and using a greedy algorithm when G_q is large. We experimentally verified the efficiency of the VC-based solution, particularly in cases where the number of edges in the query graph is much larger than the number of vertices in the vertex cover (VC).

4.4 Takeaways and Further Discussions

We present a performance comparison among several approaches for batch PPSP queries in Sec. 6.3, including the two BiDS-based solutions and two SSSP-based solutions. Among multiple graphs and query graph patterns, the Multi-BiDS achieves the best performance in most cases by effectively leveraging shared information among the batch queries.

By abstracting the query graph, we can make use of BiDS and VC and utilize shared information among the query pairs. Such benefit is more significant on undirected graphs than directed graphs, but the idea can simply be extended to directed graphs. Note that on directed graphs, the query points will be separated into sources and targets, which effectively forms a bipartite graph. The same algorithmic idea can just apply. For Multi-BiDS, we start searches from all vertices in the bipartite query graph, using the same idea mentioned in Sec. 4.2. The caveat is that we need to access both the outgoing edges and the incoming edges of a vertex, and some preprocessing is needed when reading the input graph. For the VC-based approach, again assuming we can traverse the graph from both directions, the same idea based on vertex cover also directly applies. Also, the optimal solution can be computed by bipartite graph matching in cubic work or better. Then the searches from the selected vertices, either forward or backward, will cover all queries.

Although bidirectional A* also has good performance in single PPSP queries, defining the heuristic functions becomes unclear when a search involves multiple targets. We leave this as an intriguing future direction to explore.

Space Usage. For BiDS-based approaches, the extra space needed for our Multi-BiDS algorithm is $O(n \cdot |V_q|)$, where $|V_q|$ is the number of *vertices* involved in the batch queries, instead of the number of queries in the batch. For Plain-BiDS, since we run the query one at a time, the extra space is $O(n)$. In comparison, Plain*-BiDS requires $O(n \cdot |E_q|)$ extra space, as we need to maintain the distance array for each query. For SSSP-based approaches, the extra space is upper-bounded by $O(n \cdot |V_q|)$ for both algorithms, since the algorithm needs to instantiate the searches from at most $|V_q|$ sources.

Process Extremely Large Batches. Orionet has a relatively high space usage if the given batches are very large. Therefore, for applications that require a huge number (e.g., $\Omega(n)$) of PPSP queries on the same input graph, some other methodologies with preprocessing may be a better choice. We discuss these ideas in Sec. 7.

5 Implementation Optimizations

Performance Improvement for A* by Memoization. Although applying A* to parallel PPSP is relatively easy, there is still room for improvement empirically. For instance, GraphIt [67] concludes that A* is slower than a simple parallel PPSP with ET in many cases. This is because computing the heuristic function itself could be expensive: it usually requires computing the geometric distance between two multi-dimensional points, which incurs extra cache misses in reading the coordinates. In some cases, the heuristic function is given by spherical distances rather than Euclidean distances (e.g., in real-world road routing), which can be even more expensive. This cost can be particularly expensive since $h(\cdot)$ is computed in the PRUNE function every time we relax a vertex, and a vertex can be relaxed multiple times throughout the algorithm.

Note that the heuristic function $h(\cdot)$ remains unchanged once s and t is selected. Hence, we can use an array to store the heuristics to avoid recomputation. A potential challenge here is that we cannot afford to preprocess this value for *all* vertices: computing the full array takes $O(n)$ cost, but in an effective A* execution, the number of relaxed vertices should be fewer than n. To overcome this challenge, we use memoization: when we need the heuristic function for v for the first time, we compute it on the fly and save the result into the array. For all subsequent queries that require the heuristic function on v, it can be read directly from the array. Our experiments (see Fig. 6) show that memoization greatly improves the performance for A*-based solutions.

Since our implementations are based on the stepping algorithms framework [19], some optimizations are derived from them. We discuss these optimizations in the full paper [21].

6 Experiments

Setup. We run experiments on a 96-core machine (192 hyperthreads) with 4 × 2.1 GHz Intel Xeon Gold 6252 CPUs (each with 36MB L3 cache) and 1.5TB of main memory. Our code is implemented in C++, using ParlayLib [8] to support fork-join scheduler and some parallel primitives. We always use `numactl -i all` to interleave memory for parallel experiments. For each source-target pair, the running time is determined by the average of the five runs after a warmup trial. For each test, we select five source-target pairs in this setting and report the geometric mean for them. Running shortest paths on disconnected pairs can be optimized by a single pass of graph connectivity [16, 22]. Thus, we always select sources and targets from the largest connected component.

		n	m	D	|LCC|%	Heuristics	Notes
Social	OK	3.07M	117M	9	100.0%	-	com-orkut [64]
	LJ	4.85M	42.9M	19	99.9%	-	soc-LiveJournal1 [3]
	TW	41.7M	1.20B	22	100.0%	-	Twitter [38]
	FS	65.6M	1.81B	37	100.0%	-	Friendster [64]
Web	IT	41.3M	1.03B	45	100.0%	-	it-2004 [11]
	SD	89.2M	1.94B	35	98.8%	-	sd_arc [45]
Road	AF	33.5M	44.5M	3948	91.0%	Spherical	Africa [50]
	NA	87.0M	110M	4835	98.9%	Spherical	North-America [50]
	AS	95.7M	122M	8794	94.4%	Spherical	Asia [50]
	EU	131M	166M	4410	98.9%	Spherical	Europe [50]
k-NN	HH5	2.05M	6.50M	1863	81.9%	Euclidean	Household [23, 60]
	CH5	4.21M	14.8M	14479	50.8%	Euclidean	CHEM [25, 60]
	GL5	24.9M	78.7M	21601	49.1%	Euclidean	GeoLife [60, 68]
	COS5	321M	979M	1180	99.8%	Euclidean	Cosmo50 [39, 60]

Table 3: Graphs information. "n" = number of vertices. "m" = number of edges. "D" = diameter. "|LCC|%" = ratio of the largest connected components.

We test 14 graphs in four categories, including social networks, web graphs, road networks, and k-NN graphs. The graph information is given in Tab. 3. For social and web graphs, we download them from the references. Since there are no weights with the original graphs, we generate the weights uniformly at random in the range $[1, 2^{18}]$. For road networks, we parse the longitude, latitude, and length of the road from OpenStreetMap [50]. Since the coordinates are in the form of longitudes and latitudes, we compute the heuristic between two vertices using spherical distance. For k-NN graphs, the original datasets are sets of low-dimensional points. We use GeoGraph [60] to generate the corresponding k-NN graphs, where each point is connected to its k nearest neighbors. We set $k = 5$ for all k-NN graphs. The heuristics for all k-NN graphs are Euclidean distances. For both road networks and k-NN graphs, we use their original weights and coordinates. We symmetrize the directed graphs. The selected graphs cover a wide range of sizes (from million to billion scale) and diameters (from 10 to 10^4). When taking the *average* performance across multiple graphs, we always use the ***geometric mean***.

Additional Experiments. Due to page limits, we provide additional experimental results and analyses in the full version of the paper [21], including comprehensive studies on distance percentiles, scalability, and memoization across all graphs, as well as experiments with larger batch sizes for batch PPSP queries.

6.1 Tested Algorithms

Orionet is implemented on top of the stepping algorithms framework [19], which provides three implementations: ρ-stepping, Δ^*-stepping, and Bellman-Ford. We use Δ^*-stepping (a variant of Δ-stepping [46]) because it has the best performance on large-diameter graphs, which are the primary use cases for A* (road and k-NN

		Social				Web		Road				k-NN				MEAN	
		OK	LJ	TW	FS	IT	SD	AF	NA	AS	EU	HH5	CH5	GL5	COS5	Heur.	ALL
	SSSP [19]	.092	.053	.964	2.35	.463	1.89	.700	1.17	1.56	1.37	.032	.243	.800	1.95	.618	.545
	Ours-ET	.041	.021	.706	.847	.105	.740	.015	.031	.031	.032	.003	.004	.011	.040	.015	.044
	Ours-BiDS	.008	.005	.094	.188	.138	.145	.010	.020	.023	.016	.003	.003	.010	.019	.010	.020
	Ours-A^*	-	-	-	-	-	-	.016	.020	.021	.022	.004	.004	.011	.008	.011	-
1-st	**Ours-BiD-A^***	-	-	-	-	-	-	.009	.017	.019	.011	.004	.004	.010	.004	.008	-
	GI-ET [67]	.027	.016	.417	.167	.062	.401	.029	.057	.062	.052	.116	.003	.070	.058	.041	.058
	GI-A^* [67]	-	-	-	-	-	-	.017	.032	.032	.031	.081	.003	.079	.013	.025	-
	MBQ-ET [66]	.037	.036	1.21	.393	.340	.834	.206	.392	.429	.484	.035	.025	.211	.178	.165	-
	MBQ-A^* [66]	-	-	-	-	-	-	.069	.111	.119	.102	.031	.022	.142	.030	.064	-
	SSSP [19]	.091	.052	.972	2.36	.461	1.88	.701	1.17	1.57	1.36	.031	.245	.795	1.92	.615	.543
	Ours-ET	.077	.049	.878	1.94	.314	1.60	.388	.538	.783	.620	.024	.121	.291	1.08	.313	.341
	Ours-BiDS	.075	.033	.805	1.30	.223	1.08	.304	.315	.498	.348	.012	.089	.352	.515	.206	.239
	Ours-A^*	-	-	-	-	-	-	.342	.241	.479	.303	.025	.140	.416	.132	.197	-
50-th	**Ours-BiD-A^***	-	-	-	-	-	-	.426	.215	.570	.243	.014	.105	.472	.110	.177	-
	GI-ET [67]	.149	.070	2.68	1.64	.609	3.22	1.20	1.06	1.61	1.07	.114	.094	1.59	2.07	.725	.701
	GI-A^* [67]	-	-	-	-	-	-	.993	.479	1.10	.626	.148	.155	1.48	.333	.497	-
	MBQ-ET [66]	.195	.191	2.34	3.85	1.18	3.45	t.o.	t.o.	t.o.	t.o.	.288	.852	t.o.	t.o.	-	-
	MBQ-A^* [66]	-	-	-	-	-	-	1.49	.772	1.65	.727	.148	.684	6.50	.529	.912	-
	SSSP [19]	.090	.052	.968	2.35	.454	1.88	.704	1.17	1.57	1.35	.032	.246	.797	1.94	.618	.543
	Ours-ET	.092	.057	1.01	2.72	.448	1.92	.730	1.18	1.66	1.41	.035	.244	.856	2.14	.648	.570
	Ours-BiDS	.095	.062	1.08	2.48	.416	1.77	.505	.795	.961	.982	.030	.157	.535	1.14	.426	.447
	Ours-A^*	-	-	-	-	-	-	.704	1.04	1.59	1.21	.034	.288	.387	.473	.473	-
99-th	**Ours-BiD-A^***	-	-	-	-	-	-	.858	.976	1.36	1.05	.021	.191	.152	.364	.356	-
	GI-ET [67]	.233	.104	3.37	2.59	.739	4.38	2.17	2.30	3.51	2.46	.121	.189	5.36	3.94	1.46	1.21
	GI-A^* [67]	-	-	-	-	-	-	2.11	2.15	3.73	2.35	.179	.298	1.75	1.34	1.22	-
	MBQ-ET [66]	.311	.245	4.27	6.38	1.43	4.94	t.o.	t.o.	t.o.	t.o.	.589	2.27	t.o.	t.o.	-	-
	MBQ-A^* [66]	-	-	-	-	-	-	3.12	2.66	5.02	2.30	.177	1.61	7.37	1.38	2.02	-

Table 4: Running time (in seconds) on five single-SSSP algorithms. Smaller is better. i-th means the targets are the $i\%$ farthest vertex from the sources. The fastest algorithm on each graph and each percentile is underlined. The full information of the graphs is given in Tab. 3. "SSSP" = Single-source shortest paths [19]. "ET" = Early termination. "BiDS" = Bidirectional search. "A^*" = A^* search. "BiD-A^*" = Bidirectional A^* search. "GI" = GraphIt [67]. "MBQ" = Multi Bucket Queues [66]. "Mean" = Geometric means across graphs with heuristics (road and k-NN graphs) and all graphs. "Heur." = Heuristic-based graphs. "-" = Not applicable. "t.o." = Timeout (exceeding ten seconds). A^* search is not applicable to social and web graphs as they do not have heuristics (coordinates).

graphs). Note that Orionet also supports ρ-stepping and Bellman-Ford, and can easily be integrated with other SSSP algorithms. To find the best parameter Δ for each graph, we use the standard approach by starting with a small initial Δ to run the test and doubling Δ until it converges to the minimum geometric mean running time. For all other baselines using Δ-stepping, we also find and use the best parameter Δ in the same way.

Baselines. We compare Orionet with three SOTA baselines on single PPSP. The first is the original parallel Δ^*-stepping SSSP algorithm from SPAA'21 [19], which is also the base implementation we build upon. The other two are GraphIt from CGO'20 [67] and Multi-Bucket Queue (MBQ) from SPAA'24 [66]. Both of them contain PPSP algorithms with early termination (ET) and A^* in their source code and experiments in their papers. We directly port them (denoted as GI-ET/GI-A^*/MBQ-ET/MBQ-A^*) in our experiments. We select the best Δ for the two baselines as mentioned above, and all other settings and parameters remain unchanged from the given sample scripts. Since MBQ does not support floating-point distances due to bitmasking, we round floating-point values to integers in its experiments, which favors MBQ over other implementations.

For batched queries, we did not find publicly available implementations for general batch PPSP, so we compare four different approaches in Orionet (all described in Sec. 4): 1) **Multi-BiDS**: a BiDS-based solution using Multi-BiDS, 2) **Plain-BiDS**: a plain BiDS-based solution running parallel BiDS for each query one by one serially, 3) **Plain*-BiDS**: another plain BiDS-based solution running parallel BiDS for all queries simultaneously, 4) **VC-SSSP**: an SSSP-based solution using VC, and 5) **Plain-SSSP**: a plain solution running parallel SSSP from all sources.

6.2 Single PPSP Query

Overall Performance. We evaluate our PPSP algorithms with early termination (ET), bidirectional search (BiDS), A^*, and bidirectional A^* (BiD-A^*). We compare them with running SSSP from the source (SSSP), the PPSP implementations in GraphIt [67], and MBQ (GI-ET / GI-A^* / MBQ-ET / MBQ-A^*), both of which implement unidirectional search with early termination and A^*. The results are available in Tab. 4, where each row represents an algorithm and each column represents a graph. Note that the performance of ET, BiDS, and A^*-based solutions is highly dependent on the distance between the two vertices. Therefore, we evaluate our algorithms in three different distance percentiles (1-st, 50-th, and 99-th). A query at the x-th distance percentile means the target is the $x\%$ farthest vertex from the source. Since we do not have the coordinates on social and web graphs, we only run A^* and BiD-A^* on road and k-NN graphs.

Comparing with an SSSP Implementation. We first compare our implementation with the straightforward baseline using SSSP. On social and web graphs, both ET and BiDS always achieve better performance than SSSP when the given queries are within the 50-th distance percentile. However, when the distance percentile increases to the 99-th, SSSP can achieve better performance than ET and BiDS. This is because, for the 99-th percentile setting, both ET and BiDS are unable to prune many vertices and thus need to search almost all vertices in the graph. Pruning only a few vertices cannot mitigate the extra overheads (e.g., maintaining μ and searching from two directions) compared to SSSP.

On road and k-NN graphs with geometric information, we can also compare A* and BiD-A*. Comparing the average performance across all graphs, BiD-A* achieves the *best average performance on all distance percentiles*. It achieves 76.0×, 3.49×, and 1.73× speedups compared to SSSP on 1-st, 50-th, and 99-th distance percentiles respectively. If we only compare BiDS and A* to ET, we can see that both techniques can effectively reduce the running time in most test cases. On average, across all graphs, their improvement on different percentiles can be 30–60%. BiDS is slightly more efficient than A*. The reason is that A* requires to save the coordinates and compute the heuristics, which can increase both the I/O and computation cost. In contrast, BiDS only needs to initiate two searches, and the other steps are almost the same as in ET. Adding distance heuristics by A* is also very effective. A* and BiD-A* achieves up to 252× and 441× speedups compared to SSSP at the 1-st percentile, and still have up to 4.10× and 5.32× speedups at the 99-th percentile.

On road and k-NN graphs, BiDS and A* still enable significant speedup over SSSP or ET in many cases, even at 99-percentile. One reason might be that the SSSP algorithms on social and web graphs are highly optimized, which leaves little space for further improvement with additional techniques. For road and k-NN graphs, which are sparser and inherently more difficult to achieve good parallelism [19], using BiDS or A* still improves performance even within the 99% closest vertices on many graphs. In addition, A* and BiD-A* can further use the coordinates information to prioritize the search direction and improve the performance. In general, the increasing distance still makes the advantage of A* and BiD-A* smaller, and in certain cases, they can be slower than SSSP (i.e., A* on AS and BiD-A* on AF) on the 99-th percentile. As discussed above, this is mainly due to the inadequacy of prunes and the extra overhead on the heuristics computations.

Comparing with SOTA PPSP Implementations. We now compare Orionet with GraphIt [67] and MBQ [66]. We first compare the standard PPSP query without geometric information (thus A* is not applicable). GraphIt is almost always faster than MBQ, except for two cases, and therefore we focus on comparing with GraphIt below. For ET, on social and web graphs, GI-ET has slightly better performance at the 1-st percentile, while MBQ-ET and Ours-ET are comparable. When the queried vertices are farther, Ours-ET becomes more efficient. On road and k-NN graphs, Ours-ET is almost always faster than GI-ET except for one graph, CH5, where it remains competitive within 20%. On the geometric mean across all graphs, our ET is 1.33×, 2.05×, and 2.12× faster than GraphIt on the three distance percentiles, respectively. Orionet further accelerates the queries by bidirectional search, leading to a final speedup of 2.8× compared to GI-ET on average and 6.8× (or more, considering the timeout cases) faster compared to MBQ-ET.

We now compare the A* performance across Orionet, GraphIt, and MBQ. On the geometric mean across all road and k-NN graphs, Our A* is 2.26×, 2.52×, and 2.59× faster than GraphIt on the three distance percentiles and is 5.75×, 4.63×, and 4.28× faster than MBQ. Part of the performance gain comes from memoization techniques, which reduce redundant heuristic computations. With further integrating the bidirectional search, Orionet finally achieves 4.4× speedup over GraphIt on average, and is 6.2× faster than MBQ.

The Study on Different Distance Percentiles. PPSP usually performs faster than SSSP. However, the overheads of maintaining the pruning conditions and searching bidirectionally become more visible when the queried pairs are farther. Therefore, we provide an in-depth analysis of how the running time scales with fine-grained distance percentiles. We pick a random source s from the largest connected component and select the targets based on the distances from s. The first target is the 10-th closest vertex from s, and we double the distance rank for every subsequent target (20-th, 40-th, etc.) until we reach the farthest vertex. We select a representative graph from each category and present the results in Fig. 4. Results on all graphs are in the full paper [21].

On social and web graphs, ET is not very effective. Both ET and BiDS can be slower than simply running SSSP when the queried vertices are far. This is because these graphs have many high-degree vertices. Once the PPSP search expands on these vertices, they can significantly increase the search space and thus increase the running time. Nevertheless, BiDS still achieves about 10× speedup when the queried vertices are below the 1-st distance percentile.

On road and k-NN graphs, the running time grows almost linearly with increasing distance percentiles. Both BiDS and A* can improve the performance compared to ET. When the queried vertices are close (below 1-st percentile), BiDS has the best performance in general. When the distances are above the 1-st percentile threshold, BiD-A* becomes the best. This is because when the vertices are far enough, BiD-A* can guide the search towards the midpoint more precisely and thus can effectively prune more redundant searches in the wrong direction than BiDS.

Self-Relative Speedups. To measure parallelism, we present the self-relative scalability curves on one representative graph from each graph category in Fig. 5. On the four representative graphs, all of our algorithms, as well as the SSSP implementation from the previous work, have good scalability, achieving 31.0–67.8× speedups. Interestingly, the simpler the algorithms are, the better scalability they have, as plain algorithms have more work due to disoriented search and thus can better saturate the processors with sufficient work. Algorithms with more advanced techniques, such as BiDS, skip many vertices with the heuristics, which leads to less load balance and lower parallelism. The full results on each graph are available in the full paper [21].

Performance Study on Memoization. We test how memoization helps reduce the cost in A*. We perform the same test as in Tab. 4 with and without memoization. In Fig. 6, we show the average performance on road networks and k-NN graphs, respectively. All numbers are normalized to ET. The full results on each graph are presented in the full paper [21].

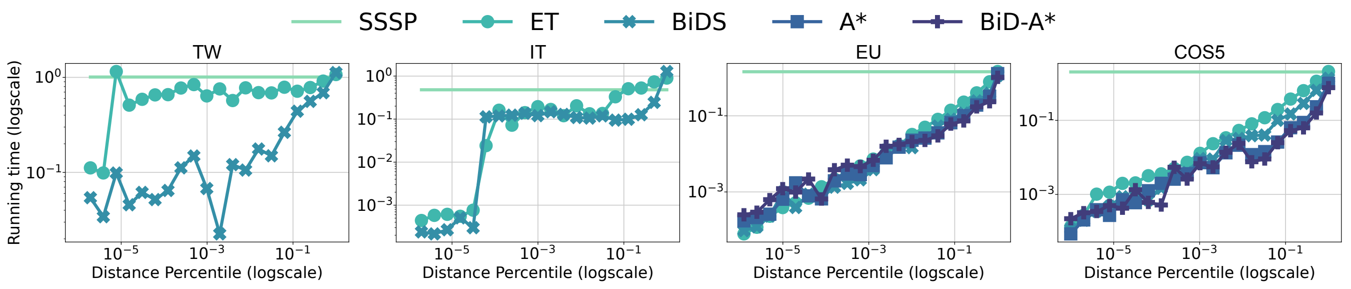


Figure 4: Running time vs. distance percentile for single PPSP algorithms on four representative graphs from each category. Lower is better.

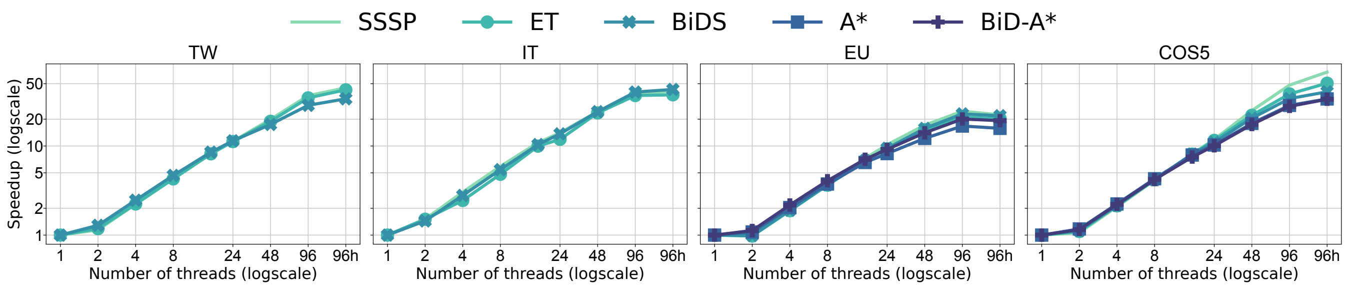


Figure 5: Self-relative speedups of our single PPSP algorithms on four representative graphs from each category. Higher is better.

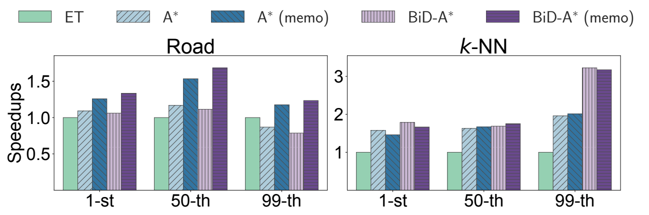


Figure 6: Relative performance over ET with and without memoization on road and k-NN graphs. Numbers are running times normalized to ET. ET is always one. Higher is better. We take the geometric mean on all road and k-NN graphs. "memo" denotes memoization.

Without memoization, A* and BiD-A* can be slower than ET, which is consistent with the findings in previous work. Using memoization, the running time is reduced by 15% on average. Saving and reusing the heuristics greatly mitigate the overhead on the additional computations. Memoization is more useful on road networks than in k-NN graphs, which is due to the more complicated heuristic computations for spherical distances in road graphs than the Euclidean distances in k-NN graphs. Memoization improves A* and BiD-A* by up to 1.74× on road graphs.

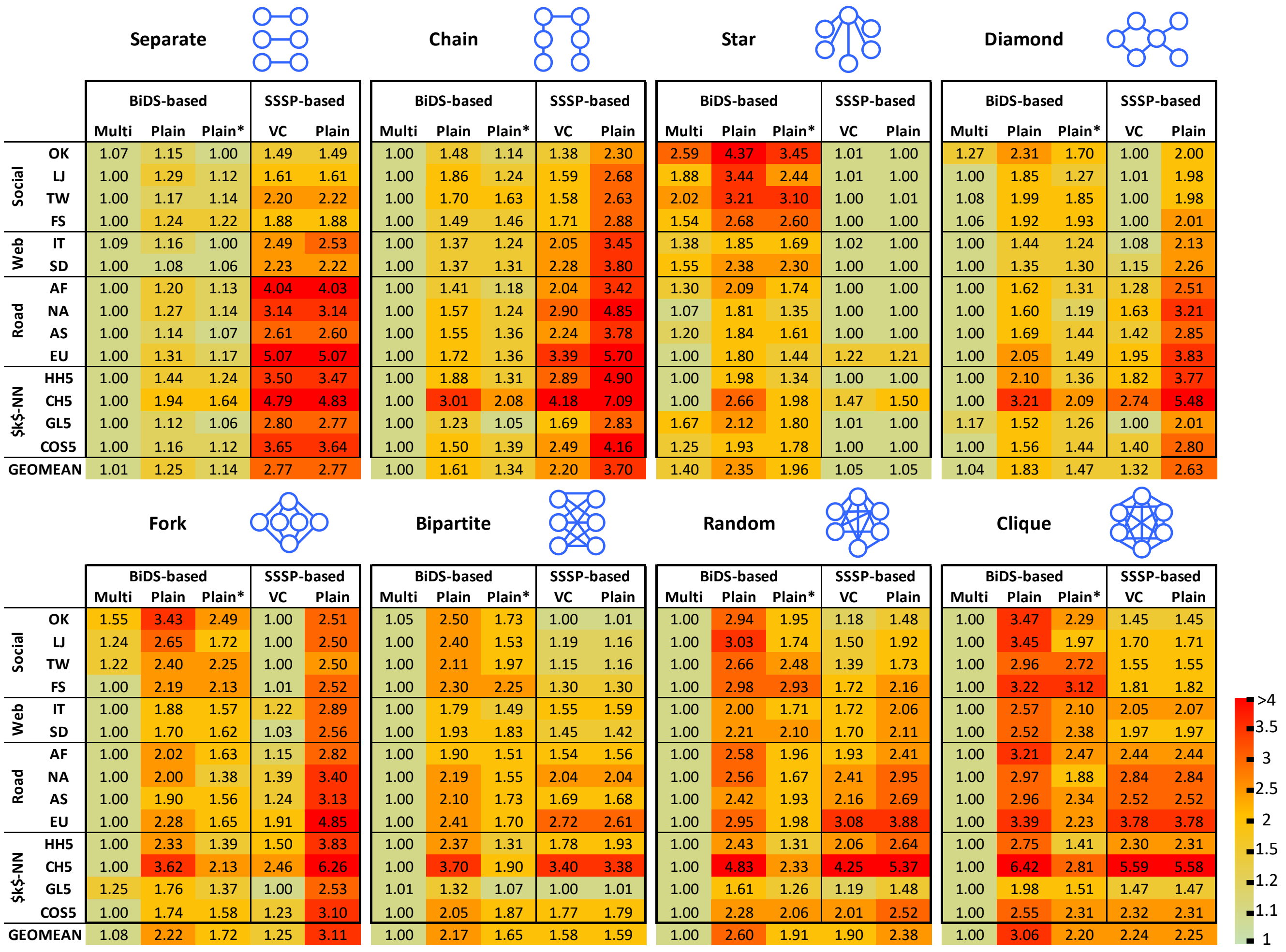
6.3 Batch PPSP Queries

We discuss the tested batch PPSP algorithms in Sec. 6.1 and present our experiments in Fig. 7. We generate eight types of batch queries to simulate different use cases, as discussed in Sec. 1 and 4. Their query graphs are outlined in Fig. 7. The numbers in Fig. 7 represent relative running time normalized to the fastest approach on each graph and each query graph. To compare across queries, we fix the number of sources to six and vary only the queries between them. The tested algorithms are discussed in Sec. 6.1. Note that since the vertices are randomly picked, the expected distance is at about the 50-th percentile.

Overall, our Multi-BiDS *achieves the best performance on about 80% of the tests.* Even in the worst case (star), Multi-BiDS is within 50% slower than the fastest baseline on average. The speedups of Multi-BiDS over other baselines become larger when the query graph becomes denser (i.e., more sources are shared in the queries). For example, on cliques, Multi-BiDS is up to 2.81× faster (on CH5) even compared to the best baseline, and 2.20× on average. Similarly, on other query graph patterns where each vertex may be involved in more queries, such as chain, bipartite, and random, Multi-BiDS have an overwhelming advantage over the others.

Even on separate queries (three disjoint s-t pairs) where no sources are shared among different queries, Multi-BiDS still benefits from running all queries in parallel. Compared to Plain-BiDS, Multi-BiDS achieves competitive performance on social and web graphs and achieves up to 94% speedups on road and k-NN graphs. This indicates another benefit of using Multi-BiDS on the batch: SSSP/PPSP algorithms usually suffer from insufficient parallelism because road and k-NN graphs are sparse and have large diameters. Batching the queries can naturally saturate the processors with sufficient jobs. Compared to Plain*-BiDS, Multi-BiDS achieves a 14% speedup on average, which we believe is due to better cache access patterns from collocating the queries. On average, Multi-BiDS is 25% faster than Plain-BiDS and 14% faster than Plain*-BiDS.

On other query graph patterns where the queries do not share vertex heavily, other approaches can be favorable. For example, on star queries, only one SSSP is needed for SSSP-based algorithms, while Multi-BiDS needs to search from all six sources. On social and web graphs, the advantage of running from only one source is obvious. However, on road and k-NN graphs, Multi-BiDS can still achieve comparable or even better performance (EU and CH5). The reason is that on these graphs, BiDS is already about 2.75-3.91×



Separate

		BiDS-based			SSSP-based	
		Multi	Plain	Plain*	VC	Plain
Social	OK	1.07	1.15	1.00	1.49	1.49
	LJ	1.00	1.29	1.12	1.61	1.61
	TW	1.00	1.17	1.14	2.20	2.22
	FS	1.00	1.24	1.22	1.88	1.88
Web	IT	1.09	1.16	1.00	2.49	2.53
	SD	1.00	1.08	1.06	2.23	2.22
Road	AF	1.00	1.20	1.13	4.04	4.03
	NA	1.00	1.27	1.14	3.14	3.14
	AS	1.00	1.14	1.07	2.61	2.60
	EU	1.00	1.31	1.17	5.07	5.07
k-NN	HH5	1.00	1.44	1.24	3.50	3.47
	CH5	1.00	1.94	1.64	4.79	4.83
	GL5	1.00	1.12	1.06	2.80	2.77
	COS5	1.00	1.16	1.12	3.65	3.64
GEOMEAN		1.01	1.25	1.14	2.77	2.77

Chain

		BiDS-based			SSSP-based	
		Multi	Plain	Plain*	VC	Plain
Social	OK	1.00	1.48	1.14	1.38	2.30
	LJ	1.00	1.86	1.24	1.59	2.68
	TW	1.00	1.70	1.63	1.58	2.63
	FS	1.00	1.49	1.46	1.71	2.88
Web	IT	1.00	1.37	1.24	2.05	3.45
	SD	1.00	1.37	1.31	2.28	3.80
Road	AF	1.00	1.41	1.18	2.04	3.42
	NA	1.00	1.57	1.24	2.90	4.85
	AS	1.00	1.55	1.36	2.24	3.78
	EU	1.00	1.72	1.36	3.39	5.70
k-NN	HH5	1.00	1.88	1.31	2.89	4.90
	CH5	1.00	3.01	2.08	4.18	7.09
	GL5	1.00	1.23	1.05	1.69	2.83
	COS5	1.00	1.50	1.39	2.49	4.16
GEOMEAN		1.00	1.61	1.34	2.20	3.70

Star

		BiDS-based			SSSP-based	
		Multi	Plain	Plain*	VC	Plain
Social	OK	2.59	4.37	3.45	1.01	1.00
	LJ	1.88	3.44	2.44	1.01	1.00
	TW	2.02	3.21	3.10	1.00	1.01
	FS	1.54	2.68	2.60	1.00	1.00
Web	IT	1.38	1.85	1.69	1.02	1.00
	SD	1.55	2.38	2.30	1.00	1.00
Road	AF	1.30	2.09	1.74	1.00	1.00
	NA	1.07	1.81	1.35	1.00	1.00
	AS	1.20	1.84	1.61	1.00	1.00
	EU	1.00	1.80	1.44	1.22	1.21
k-NN	HH5	1.00	1.98	1.34	1.00	1.00
	CH5	1.00	2.66	1.98	1.47	1.50
	GL5	1.67	2.12	1.80	1.01	1.00
	COS5	1.25	1.93	1.78	1.00	1.00
GEOMEAN		1.40	2.35	1.96	1.05	1.05

Diamond

		BiDS-based			SSSP-based	
		Multi	Plain	Plain*	VC	Plain
Social	OK	1.27	2.31	1.70	1.00	2.00
	LJ	1.00	1.85	1.27	1.01	1.98
	TW	1.08	1.99	1.85	1.00	1.98
	FS	1.06	1.92	1.93	1.00	2.01
Web	IT	1.00	1.44	1.24	1.08	2.13
	SD	1.00	1.35	1.30	1.15	2.26
Road	AF	1.00	1.62	1.31	1.28	2.51
	NA	1.00	1.60	1.19	1.63	3.21
	AS	1.00	1.69	1.44	1.42	2.85
	EU	1.00	2.05	1.49	1.95	3.83
k-NN	HH5	1.00	2.10	1.36	1.82	3.77
	CH5	1.00	3.21	2.09	2.74	5.48
	GL5	1.17	1.52	1.26	1.00	2.01
	COS5	1.00	1.56	1.44	1.40	2.80
GEOMEAN		1.04	1.83	1.47	1.32	2.63

Fork

		BiDS-based			SSSP-based	
		Multi	Plain	Plain*	VC	Plain
Social	OK	1.55	3.43	2.49	1.00	2.51
	LJ	1.24	2.65	1.72	1.00	2.50
	TW	1.22	2.40	2.25	1.00	2.50
	FS	1.00	2.19	2.13	1.01	2.52
Web	IT	1.00	1.88	1.57	1.22	2.89
	SD	1.00	1.70	1.62	1.03	2.56
Road	AF	1.00	2.02	1.63	1.15	2.82
	NA	1.00	2.00	1.38	1.39	3.40
	AS	1.00	1.90	1.56	1.24	3.13
	EU	1.00	2.28	1.65	1.91	4.85
k-NN	HH5	1.00	2.33	1.39	1.50	3.83
	CH5	1.00	3.62	2.13	2.46	6.26
	GL5	1.25	1.76	1.37	1.00	2.53
	COS5	1.00	1.74	1.58	1.23	3.10
GEOMEAN		1.08	2.22	1.72	1.25	3.11

Bipartite

		BiDS-based			SSSP-based	
		Multi	Plain	Plain*	VC	Plain
Social	OK	1.05	2.50	1.73	1.00	1.01
	LJ	1.00	2.40	1.53	1.19	1.16
	TW	1.00	2.11	1.97	1.15	1.16
	FS	1.00	2.30	2.25	1.30	1.30
Web	IT	1.00	1.79	1.49	1.55	1.59
	SD	1.00	1.93	1.83	1.45	1.42
Road	AF	1.00	1.90	1.51	1.54	1.56
	NA	1.00	2.19	1.55	2.04	2.04
	AS	1.00	2.10	1.73	1.69	1.68
	EU	1.00	2.41	1.70	2.72	2.61
k-NN	HH5	1.00	2.37	1.31	1.78	1.93
	CH5	1.00	3.70	1.90	3.40	3.38
	GL5	1.01	1.32	1.07	1.00	1.01
	COS5	1.00	2.05	1.87	1.77	1.79
GEOMEAN		1.00	2.17	1.65	1.58	1.59

Random

		BiDS-based			SSSP-based	
		Multi	Plain	Plain*	VC	Plain
Social	OK	1.00	2.94	1.95	1.18	1.48
	LJ	1.00	3.03	1.74	1.50	1.92
	TW	1.00	2.66	2.48	1.39	1.73
	FS	1.00	2.98	2.93	1.72	2.16
Web	IT	1.00	2.00	1.71	1.72	2.06
	SD	1.00	2.21	2.10	1.70	2.11
Road	AF	1.00	2.58	1.96	1.93	2.41
	NA	1.00	2.56	1.67	2.41	2.95
	AS	1.00	2.42	1.93	2.16	2.69
	EU	1.00	2.95	1.98	3.08	3.88
k-NN	HH5	1.00	2.43	1.31	2.06	2.64
	CH5	1.00	4.83	2.33	4.25	5.37
	GL5	1.00	1.61	1.26	1.19	1.48
	COS5	1.00	2.28	2.06	2.01	2.52
GEOMEAN		1.00	2.60	1.91	1.90	2.38

Clique

		BiDS-based			SSSP-based	
		Multi	Plain	Plain*	VC	Plain
Social	OK	1.00	3.47	2.29	1.45	1.45
	LJ	1.00	3.45	1.97	1.70	1.71
	TW	1.00	2.96	2.72	1.55	1.55
	FS	1.00	3.22	3.12	1.81	1.82
Web	IT	1.00	2.57	2.10	2.05	2.07
	SD	1.00	2.52	2.38	1.97	1.97
Road	AF	1.00	3.21	2.47	2.44	2.44
	NA	1.00	2.97	1.88	2.84	2.84
	AS	1.00	2.96	2.34	2.52	2.52
	EU	1.00	3.39	2.23	3.78	3.78
k-NN	HH5	1.00	2.75	1.41	2.30	2.31
	CH5	1.00	6.42	2.81	5.59	5.58
	GL5	1.00	1.98	1.51	1.47	1.47
	COS5	1.00	2.55	2.31	2.32	2.31
GEOMEAN		1.00	3.06	2.20	2.24	2.25

Figure 7: Heatmap on batch PPSP queries. The numbers are the running times normalized to the fastest algorithm on each test instance. Smaller or green is better. "GEOMEAN" = geometric means on all graphs. We run two BiDS-based algorithms and two SSSP-based algorithms as introduced in Sec. 4. For BiDS-based algorithms, Multi is to evaluate all queries in a batch, Plain is to runs each query with our parallel BiDS algorithm one at a time, and Plain* is to run all queries simultaneously in parallel, and each query is a parallel BiDS algorithm. For SSSP-based algorithms, VC is to run the SSSP algorithm from the minimum required sources induced by the vertex cover of the query graph, and Plain is to run the SSSP algorithm from all given sources.

faster than SSSP with a single query (see Tab. 4), and running the queries in a batch also diminishes some redundant searches.

More generally, we can observe that a VC-based SSSP solution is almost always better (up to 2.55× times faster) than a plain SSSP solution since the number of SSSPs running by VC is no more than the plain SSSP solution. As expected, on graphs where a small number of vertices can cover all edges, such as star and fork graphs, the VC-based solution can even outperform the solution based on Multi-BiDS, especially on social and web graphs. As mentioned, this is because SSSP on these graphs is highly optimized, which makes the additional work on maintaining BiDS more significant. Therefore, running one or two SSSPs may be cheaper than running BiDS from all six vertices.

7 Related Work

Existing Parallel Algorithms for BiDS and Batch PPSP. Although BiDS is shown to be effective in accelerating sequential PPSP queries, we are unaware of any fully parallelized BiDS algorithm throughout the query phase. Some existing works [54, 59] parallelize bidirectional Dijkstra with two threads, one for the forward search and one for the backward search, which naturally does not scale to more than two threads. PnP [63] uses bidirectional searches in preprocessing to predict the more efficient direction (from either the source or the target) to perform the search. Once predicted, it continues the search in only one direction, either forward or backward.

There have also been studies on batch PPSP queries [27, 36, 43]. Some of them [27, 36] support only a specific type of query, such as SSMT or subset APSP, and do not generalize to arbitrary batches. Multilyra [43] studies batch PPSP queries in the distributed system setting, with the goal of processing a batch of queries at a time and thus amortizing the communication cost. However, it does not consider the connections between the given queries.

Other Algorithms for Computing Shortest Paths. We note that there is a large body of work on efficiently computing shortest paths (mostly in the sequential setting), and we refer the audience to an excellent survey [5]. Most other techniques require some preprocessing to accelerate query time. Some famous ones include contraction hierarchies (CH) [29], transit nodes routing [6], pruned landmark labeling (PLL) [2], and ALT (to accelerate A*) [30]. Most of these ideas are orthogonal to the techniques studied in this paper: BiDS, A*, and batch PPSP, which do not require preprocessing. We should note that preprocessing in shortest-path algorithms is double-edged—while queries can be significantly accelerated, the preprocessing can also take much time, and sometimes much more space as well (e.g., PLL). The preprocessing-free methodology studied in this paper is potentially preferred in certain scenarios, such as when fewer total queries are performed, graphs are larger (so less auxiliary space is available), and/or graphs change frequently.

8 Conclusions

In this paper, we systematically study parallel point-to-point shortest paths (PPSP). We propose Orionet, a parallel library for PPSP with multiple algorithms for both single and batch PPSP queries. Algorithmically, we propose a list of techniques to integrate bidirectional search (BiDS) and A* into existing parallel SSSP algorithms to achieve simple implementation with high parallelism. Orionet supports BiDS, A*, and their combination BiD-A*. We unify the methodologies for all of them in a PPSP framework in Alg. 2 by plugging in three user-defined functions. Using the framework, all these algorithms are simple, easy to program, and have high performance.

For batch queries, we give a novel abstraction based on the query graph. This abstraction allows the algorithmic ideas to be studied independently of the query types. We show how to use vertex cover and our BiDS technique to support arbitrary query batches, which can be especially efficient when the queries share vertices. Our Multi-BiDS for batch PPSP can also be implemented by our PPSP framework in a simple way.

We conducted a thorough experimental study and showed that with the careful algorithmic design, both BiDS and A* indeed lead to significant improvements in the parallel setting. Namely, BiDS and A* can consistently improve the PPSP performance, and BiD-A*, which combines their advantages, usually gives the best overall performance, which aligns with the observations in the sequential setting. Our BiDS-based solutions outperform state-of-the-art unidirectional PPSP solutions such as GraphIt [67] and MBQ [66]. In the batch PPSP setting, our proposed techniques effectively reuse shared information in the batch and consistently achieve better performance than the plain implementations.

Acknowledgments

This work is supported by NSF grants CCF-2103483, IIS-2227669, and TI-2346223, NSF CAREER Awards CCF-2238358 and CCF-2339310, the UCR Regents Faculty Development Award, and the Google Research Scholar Program. We thank the anonymous reviewers for their useful comments.

References

[1] 2024. GraphHopper. https://www.graphhopper.com/.
[2] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 349–360.
[3] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 44–54.
[4] Michael Barley, Patricia Riddle, Carlos Linares López, Sean Dobson, and Ira Pohl. 2018. GBFHS: A generalized breadth-first heuristic search algorithm. In *Proceedings of the International Symposium on Combinatorial Search*, Vol. 9. 28–36.
[5] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Mü ller Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. 2016. Route planning in transportation networks. *Algorithm engineering: Selected results and surveys* (2016), 19–80.
[6] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. 2007. Fast routing in road networks with transit nodes. *Science* 316, 5824 (2007), 566–566.
[7] Richard Bellman. 1958. On a routing problem. *Quarterly of applied mathematics* 16, 1 (1958), 87–90.
[8] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.
[9] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.
[10] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. 2016. Parallel Shortest Paths Using Radius Stepping. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 443–454.
[11] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*. 595–602.
[12] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D Zaroliagis. 1998. A parallel priority queue with constant time operations. *J. Parallel and Distrib. Comput.* 49, 1 (1998), 4–21.
[13] Nairen Cao, Jeremy T. Fineman, and Katina Russell. 2020. Efficient construction of directed hopsets and parallel approximate shortest paths. In *ACM Symposium on Theory of Computing (STOC)*. 336–349.
[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition)*. MIT Press.
[15] George Dantzig. 1963. *Linear programming and extensions*. Princeton university press.
[16] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: a framework for static and incremental parallel graph connectivity algorithms. *Proceedings of the VLDB Endowment (PVLDB)* 14, 4 (2020), 653–667.
[17] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. 2016. Customizable contraction hierarchies. *J. Experimental Algorithmics* 21 (2016), 1–49.
[18] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959).
[19] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 184–197.
[20] Xiaojun Dong, Andy Li, Yan Gu, and Yihan Sun. 2025. Orionet: Parallel Point-to-Point Shortest Paths and Batch Queries. https://github.com/ucrparlay/Orionet.
[21] Xiaojun Dong, Andy Li, Yan Gu, and Yihan Sun. 2025. Parallel Point-to-Point Shortest Paths and Batch Queries. *arXiv preprint:2506.16488* (2025).
[22] Xiaojun Dong, Letong Wang, Yan Gu, and Yihan Sun. 2023. Provably Fast and Space-Efficient Parallel Biconnectivity. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 52–65.
[23] Dheeru Dua and Casey Graf. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/.
[24] Michael Elkin and Ofer Neiman. 2019. Hopsets with constant hopbound, and applications to approximate shortest paths. *SIAM J. Comput.* 48, 4 (2019), 1436–1480.
[25] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.
[26] Lester R Ford Jr. 1956. *Network flow theory*. Technical Report. Rand Corp Santa Monica Ca.
[27] Robert Geisberger and Peter Sanders. 2010. Engineering time-dependent many-to-many shortest paths computation. In *10th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'10)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[28] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental Algorithms (WEA)*. Springer, 319–333.

[29] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact routing in large road networks using contraction hierarchies. *Transportation Science* 46, 3 (2012), 388–404.

[30] Andrew V Goldberg and Chris Harrelson. 2005. Computing the shortest path: A search meets graph theory. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Vol. 5. 156–165.

[31] Andrew V Goldberg, Haim Kaplan, and Renato F Werneck. 2006. Reach for A*: Efficient point-to-point shortest path algorithms. In *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 129–143.

[32] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[33] Robert Holte, Ariel Felner, Guni Sharon, and Nathan Sturtevant. 2016. Bidirectional search that is guaranteed to meet in the middle. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30.

[34] Takahiro Ikeda, Min-Yao Hsu, Hiroshi Imai, Shigeki Nishimura, Hiroshi Shimoura, Takeo Hashimoto, Kenji Tenmoku, and Kunihiko Mitoh. 1994. A fast algorithm for finding better routes by AI search techniques. In *Proceedings of VNIS'94-1994 Vehicle Navigation and Information Systems Conference*. IEEE, 291–296.

[35] Dan Klein and Christopher D Manning. 2003. A* parsing: Fast exact Viterbi parse selection. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*. 119–126.

[36] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. 2007. Computing many-to-many shortest paths using highway hierarchies. In *Algorithm Engineering and Experiments (ALENEX)*. SIAM, 36–45.

[37] Sunita Kumawat, Chanchal Dudeja, and Pawan Kumar. 2021. An extensive review of shortest path problem solving algorithms. In *2021 5th International Conference on Intelligent Computing and Control Systems (ICICCS)*. IEEE, 176–184.

[38] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *International World Wide Web Conference (WWW)*. 591–600.

[39] YongChul Kwon, Dylan Nunley, Jeffrey P Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. 2010. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *International Conference on Scientific and Statistical Database Management*. Springer, 132–150.

[40] Michael Luby and Prabhakar Ragde. 1989. A bidirectional shortest-path algorithm with good average-case behavior. *Algorithmica* 4 (1989), 551–567.

[41] Dennis Luxen and Christian Vetter. 2011. Real-time routing with OpenStreetMap data. In *SIGSPATIAL international conference on advances in geographic information systems*. 513–516.

[42] Thomas L Magnanti and Prakash Mirchandani. 1993. Shortest paths, single origin-destination network design, and associated polyhedra. *Networks* 23, 2 (1993), 103–121.

[43] Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. 2019. Multilyra: Scalable distributed evaluation of batches of iterative graph queries. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 349–358.

[44] Abbas Mazloumi, Chengshuo Xu, Zhijia Zhao, and Rajiv Gupta. 2020. BEAD: Batched Evaluation of Iterative Graph Queries with Evolving Analytics Demands. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 461–468.

[45] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. 2014. Web Data Commons — Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph.

[46] Ulrich Meyer and Peter Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms* 49, 1 (2003), 114–152.

[47] Joseph SB Mitchell et al. 2000. Geometric Shortest Paths and Network Optimization. *Handbook of computational geometry* 334 (2000), 633–702.

[48] T Alastair J Nicholson. 1966. Finding the shortest route between two points in a network. *The computer journal* 9, 3 (1966), 275–280.

[49] Nils J Nilsson. 2009. *The quest for artificial intelligence*. Cambridge University Press.

[50] OpenStreetMap contributors. 2010. OpenStreetMap. https://www.openstreetmap.org/.

[51] Judea Pearl. 1984. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc.

[52] Ira Pohl. 1969. *Bi-directional and heuristic search in path problems*. Technical Report. Stanford Linear Accelerator Center, Calif.

[53] Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu. 2012. Approximate shortest distance computing: A query-dependent local landmark scheme. *IEEE Transactions on Knowledge and Data Engineering* 26, 1 (2012), 55–68.

[54] Luis Henrique Oliveira Rios and Luiz Chaimowicz. 2011. Pnba*: A parallel bidirectional heuristic search algorithm. In *ENIA VIII Encontro Nacional de Inteligê ncia Artificial*.

[55] Hanmao Shi and Thomas H. Spencer. 1999. Time-Work Tradeoffs of the Single-Source Shortest Paths Problem. *Journal of Algorithms* 30, 1 (1999), 19–32.

[56] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. 2013. Reducing Contention Through Priority Updates. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 152–163.

[57] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. 2014. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment* 8, 4 (2014), 449–460.

[58] Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Ba ñuelos, Jaak Vilo, and Marlon Dumas. 2011. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *ACM International Conference on Information and Knowledge Management*. 1785–1794.

[59] Gintaras Vaira and Olga Kurasova. 2011. Parallel bidirectional dijkstra's shortest path algorithm. In *Databases and Information Systems VI*. IOS Press, 422–435.

[60] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2021. GeoGraph: A Framework for Graph Processing on Geometric Data. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 38–46.

[61] Chengshuo Xu, Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. 2020. SimGQ: Simultaneously evaluating iterative graph queries. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 1–10.

[62] Chengshuo Xu, Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. 2022. SimGQ+: Simultaneously evaluating iterative point-to-all and point-to-point graph queries. *Journal of parallel and distributed computing* 164 (2022), 12–27.

[63] Chengshuo Xu, Keval Vora, and Rajiv Gupta. 2019. Pnp: Pruning and prediction for point-to-point iterative graph analytics. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 587–600.

[64] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[65] Wei Zeng and Richard L Church. 2009. Finding shortest paths on real road networks: the case for A. *International journal of geographical information science* 23, 4 (2009), 531–543.

[66] Guozheng Zhang, Gilead Posluns, and Mark C Jeffrey. 2024. Multi Bucket Queues: Efficient Concurrent Priority Scheduling. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 113–124.

[67] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing ordered graph algorithms with GraphIt. In *International Symposium on Code Generation and Optimization (CGO)*. 158–170.

[68] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning transportation mode from raw gps data for geographic applications on the web. In *International World Wide Web Conference (WWW)*. 247–256.