Customised Shortest Paths Using a Distributed Reverse Oracle

Arthur Mahéo,[1,*] **Shizhe Zhao,**[1] **Afzaal Hassan,**[2]
Daniel Harabor,[1] **Peter J. Stuckey,**[1] **Mark Wallace**[1]
[1] Monash University, Faculty of IT, Melbourne, Australia
[2] Swinburne University of Technology, Melbourne, Australia
{arthur.maheo,shizhe.zhao,daniel.harabor,peter.stuckey,mark.wallace}@monash.edu
afzaalhassan@swin.edu.au

Abstract

We consider the design and implementation of a centralised oracle that provides commuters with customised and congestion-aware driving directions. Computing directions for a single journey is straightforward, but doing so at city-scale, in real-time, and under changing conditions is extremely challenging. In this work we describe a new type of centralised oracle which combines fast database-driven path planning with a query management system that distributes work across a small commodity cluster of networked machines. Our system allows large-scale changes to the underlying graph metric, from one query to the next, and it supports a variety of query types including optimal, bounded sub-optimal, time-budgeted and k-prefix. Simulated experiments show strong results: we can provide real-time routing for all peak-hour commuter trips in the city of Melbourne, Australia.

Introduction

In commuter road routing a centralised oracle aims to serve driving directions to as many simultaneous users as possible. There exists a large number of simultaneous queries, tens or even hundreds of thousands, and each one must be solved in close to real-time. The planned routes meanwhile must have guarantees: optimal or close to optimal. Further complications include real-time data updates, to reflect changing road conditions, and *customised metrics*, which means the objective function can change from one query to the next.

Ours is an extremely challenging problem which has been considered by all major map providers. Some of these approaches are proprietary and their implementations are unavailable for scientific experimentation. Such is the case for Bing Maps and Google Maps. Other providers (e. g., OpenStreetMap, OpenTraffic) do make available routing implementations and these are based on modern speedup algorithms including Contraction Hierarchies (CHs) (Geisberger et al. 2008, 2012) and other similar techniques (Abraham et al. 2012; Arz, Luxen, and Sanders 2013). Though each of these works can be orders of magnitude faster than reference algorithms, such as Dijkstra and A*, their performance advantages are achieved under the assumption that the input graph remains static. When the graph changes, for example because of a new metric or because new traffic-related information becomes available, these algorithms no longer guarantee returning a shortest or even feasible path. The reason is that such modern algorithms rely on auxiliary data, created offline during a preprocessing step. When the graph changes this data is invalidated and all guarantees are lost.

To handle dynamically changing costs some authors propose to repair auxiliary data (Schultes and Sanders 2007; Geisberger et al. 2012) or else to compute *metric independent* auxiliary data and then *customise* the costs online (Dibbelt, Strasser, and Wagner 2014). After these operations every query is again guaranteed optimal with respect to the new costs and/or metric function. This technology is known to power Bing Maps (Delling et al. 2017) and is believed to be at the heart of Google Maps (Geisberger 2015). The problem is that the cost of each update/repair depends on the size of the changeset. With only a few edges changed these procedures add only a small overhead per query. When the changeset grows large (e. g., the metric changes with each new query) the overhead can dominate runtime, to the point where it becomes faster to find a shortest path using a reference algorithm. See Figures 1 and 2 and for an example. Although improving the customisation time helps (Delling, Kobitzsch, and Werneck 2014), it does not address the issue.

In this work we consider new perspectives and priorities for the design of centralised routing oracles, especially in settings with dynamically changing costs. To wit, instead of optimal routing and repair, we propose that the following characteristics are more desirable:

1. **Anytime search:** centralised oracles should aim for initial solutions fast and better/optimal solutions eventually. This allows actionable plans to be computed and returned sooner, including by a given time budget, as compared to better or best plans that are returned to the user too late.
2. **Prefix paths:** centralised oracles should prioritise and return only the first k steps toward the destination. Prefixes provide concrete directions commuters can execute and they can be faster to compute than entire paths. This increases throughput (e. g., the oracle can handle more simultaneous queries) and reduces replanning time when directions are invalidated (e. g., due to missing a turn).
3. **No repair:** centralised oracles should seldom repair auxiliary data. When edge costs change, precomputed data

*Corresponding author: arthur.maheo@monash.edu
Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

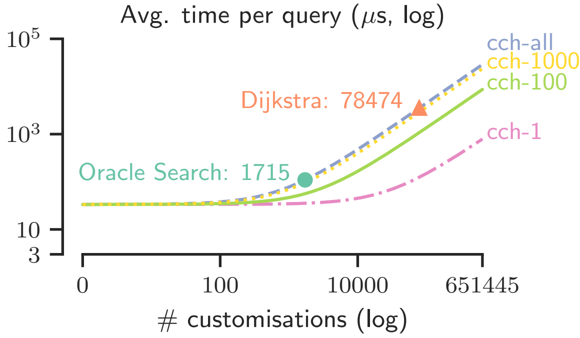

Figure 1: We compare Oracle Search and CCH in a simulated setup of Melbourne (Australia). The metric periodically changes and after every change CCH *customises* (i. e., repairs) its auxiliary data. No matter the size of the changeset (1 edge, 100 edges or even all), CCH performance quickly degrades as the number of customisations grows: from one per queryset (651K total) to one per query. After 1715 customisations, CCH becomes slower than Oracle Search and eventually slower than Dijkstra search. This experiment shows the main advantage of CCH: being able to amortise the cost of customisation over many subsequent queries.

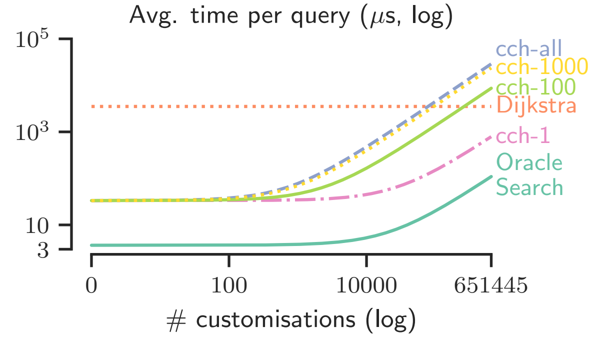

Figure 2: We compare Oracle Search and CCH in a simulated setup of Melbourne (Australia) where some percentage of all queries (651K total) requires a customised metric w_i, while all remaining queries are solved by a default metric w_0, such as network distance or freeflow travel time. CCH performance is identical to Figure 1 but Oracle Search is substantially faster. This experiment shows the main advantages of Oracle Search: fast query performance, which comes from exploiting precomputed w_0 paths, and no additional repair work, which CCH is forced to undertake when the metric changes.

should instead be exploited to guide online search by providing strong upper- and lower-bounds on the optimal cost. This allows frequent graph updates and large changesets without costly online updates.

We describe the design and implementation of such a system and demonstrate its efficacy using a small cluster of six commodity machines. Our approach combines a fast real-time path planning algorithm, called Oracle Search, with a simple workload management system that distributes queries across a cluster. Our system is thus a *realtime*, *centralised* routing engine with a *distributed* oracle. We differentiate real- and anytime as, while the underlying search algorithm is indeed anytime, returning successive paths over the network would incur too much overhead.

Oracle Search generalises CPD Search (Bono et al. 2019), a recent and database-driven path planner designed for dynamic cost settings. A main feature of CPD Search is that the A* heuristic function provides incumbent paths as well as lower bound estimates. In this work we consider a variety of new database types which can substantially improve performance and also dramatically reduce the *resident set size* – i. e., the amount of data loaded into RAM to solve any given query. For evaluation, we consider a simulated traffic scenario of Melbourne, Australia. Results show that we can process hundreds of thousands of queries *per second* in driving conditions similar to Melbourne's morning-hour peak.

Problem Statement

We consider path planning problems in graphs with dynamically changing edge costs. We take as input a graph $G = (V, E)$ where $E \subseteq V \times V$ and where each edge $(i, j) \in E$ has an associated and positive cost $w(i, j)$. The set of all edge costs, w, is called the *graph metric*. In addition to the graph and the metric we are also given pairs of vertices $(s, t) \in V$ (respectively the *start* and *target*). Each pair is called an *instance* and a valid solution to an instance is any path π from s to t, that is a sequence of edges $[(s, n_1), (n_1, n_2), \ldots, (n_m, t)]$. The cost of the path π is the sum of weights of its constituent edges denoted $w(\pi)$. Our **objective** is to find a path from s to t which minimises total edge cost. Let $sp(s, t)$ represent this shortest path. We use $+\!\!+$ to represent path concatenation.

We assume such problems can be solved in two phases: online and offline. During the *offline phase* the graph is fixed with given weights w_0 and instances are unknown. We assume that we are free to preprocess the graph for as long as necessary in order to create additional auxiliary data structures. During the *online phase* we are given sets of instances together with revised weights which must be solved together as quickly as possible. In other words we aim to maximise throughput (queries per second) for the set of instances. We assume auxiliary data is available at this point and that we are free to exploit it. Our suggested approach, Oracle Search, has access to an auxiliary database with perfect information w. r. t. w_0 but needs to perform search to find optimal paths w. r. t. new weights.

Cost updates. Edge costs can change from one query to the next. These changes model exogenous events, such as changing traffic conditions, or they can represent penalties, added on a per query basis in order to derive individualised plans – e. g., safest routes, simplest routes, scenic routes and so on. We use $w(i, j)$ to indicate the cost of edge (i, j) with

respect to the current metric and $w_0(i,j)$ to indicate the cost with respect to the initial or preprocessing-time metric. We allow edge costs to increase or decrease with each update but we assume the updated value is always at least as large its corresponding initial cost – i. e., for any edge $(i,j) \in E$ we have that $w_0(i,j) \leq w(i,j)$. We base our oracle on the *free-flow* cost, which is the fastest travel time possible on a road segment – subject to respecting the law.

Note that our problem is distinct from *Personalised Routing* (Funke and Storandt 2015) where the authors assume a fixed set of metrics (i. e., each edge has a vector of static costs) and where each query is *personalised* with respect to a linearised set of user-specified weights (one per metric).

Oracle Search

We describe a family of anytime search algorithms intended for path planning in settings with dynamically changing costs. Collectively known as *Oracle Search*, each member of this family has access to an eponymous *path oracle* $O(s,t)$.

Definition 1. *A* path oracle *is a function $O(s,t)$ that returns a tuple $\langle \pi, lb, ub \rangle$ where π is an s-t path and the values $ub = w(\pi)$ and $lb = w_0(\pi)$ are upper and lower bounds on the cost of the shortest s-t path $w(sp(s,t))$.*

Once constructed, a path oracle can provide strong heuristic estimates in the context of anytime A* search. Two important differences that distinguish our approach, and which provide compelling advantages, are the following:

1. For each expanded node the heuristic returns a concrete path π from s to t. The cost $w_0(\pi)$ bounds the optimal solution from below and drives the search toward the target. The cost $w(\pi)$ meanwhile bounds the optimal solution from above; allowing more nodes to be pruned and helping the search to close the optimality gap faster (recall that A* only bounds from below).
2. Early termination, which means the search stops when a quality threshold is achieved or a time limit is exceeded. In many cases these conditions can be satisfied long before the target comes off the OPEN list. Upon termination Oracle Search returns the best known feasible path. We give a pseudo-code description in Algorithm 1.

There are many possible instantiations of Oracle Search, each one characterised by the choice of path oracle. For example in CPD Search (Bono et al. 2019) the path oracle takes the form of a Compressed Path Database (CPD, Botea 2011; Botea and Harabor 2013). In this work we generalise CPD Search and we consider several alternative path oracles which can improve performance and reduce the size of active memory during pathfinding search.

Oracles. The usual approach to constructing an oracle is to undertake (offline) an all pairs shortest path computation w. r. t. the initial graph metric w_0. For each vertex s and t we record in a *first-move table*, $\mathbf{fm}[s,t]$, the identity of the next edge on the optimal path, from s toward t. Once the table is computed we *extract* any w_0-shortest path using the following recursive procedure: extract the move $\mathbf{fm}[s,t]$ and follow the corresponding arc; repeating as necessary until the target is reached.

Algorithm 1: Oracle-Search(w, s, t, ϵ): Parameters s and t indicate the start and target, w encodes actual edge-costs. O is the shortest path oracle. The fst function returns the first argument of a tuple. This algorithm guarantees solutions are ϵ-optimal.

```
1  closed ← ∅; open ← {s}
2  for n ∈ N do g[n] ← ∞;
3  g[s] ← 0; f[s] ← 0; p[s] ← []
4  u ← ∞; I ← s
5  while open ≠ ∅ do
6      n ← arg min{f[n'] | n' ∈ open}
7      if n = t then return p[n] ;
8      if ϵf[n] ≥ u then return p[I] ++fst O(I, t) ;
9      open ← open − {n}
10     closed ← closed ∪ {n}
11     for (n, m) ∈ E, m ∉ closed do
12         if g[n] + w(n, m) < g[m] then
13             p[m] ← p[n] ++ [(n, m)]
14             g[m] ← g[n] + w(n, m)
15             ⟨π, lb, ub⟩ ← O(m, t)
16             f[m] ← g[m] + lb
17             if u > g[m] + ub then
18                 u ← g[m] + ub; I ← m
19             open ← open ∪ {m}
20 return ⊥                                  ▶ No solution
```

A first-move table stores $|V|^2$ entries and as $|V|$ grows the total space consumption can become prohibitive. This situation has motivated a variety of works that try to reduce the size of the table using lossless compression, with the currently most successful strategy being some variety of run-length encoding (Strasser, Botea, and Harabor 2015). The resulting *Compressed Path Database* (CPD) trades a small amount of online performance[1] for a two or even three orders reduction in space.

Reverse oracles. A reverse oracle is a first-move table indexed on a target node t – i. e., for every node t we store an array that contains the optimal first move for every s towards t. This way of indexing has the same precomputation costs as in the forward case but comes with some compelling advantages. For example, in a forward oracle (e. g., CPD) each first move is extracted from a different row and the set of required rows is *a priori* unknown (i. e., the entire CPD must be loaded into RAM). With a reverse oracle however we query *only a single row* to extract a complete s to t path.

Forward CPDs encode one-to-all information, where many adjacent target nodes share the same first move from a common source. This compresses very efficiently. Reverse CPDs meanwhile encode all-to-one information, and here first moves can differ even for adjacent source nodes (due to topological changes in the graph). A main consequence is that a reverse CPD can be many times larger than an equivalent forward CPD. We refer the interested reader to Zhao et al. (2020) for a more detailed discussion.

[1] Extraction takes log-time w. r. t. the compressed string length.

Type		Size (MB)	Speedup (vs. Dijkstra)	
			Static (w_0)	Dynamic (w)
Dijkstra		0	1	1
CPD	Fwd	118	182.312	42.877
	Rev[2]	49938	n/a	n/a
Table	Fwd	13423	586.015	62.917
	Rev	13423	812.236	73.941
	(*Trim*	*3800*)		

Table 1: Oracle Search performance with different path oracles on the benchmark from Figure 1. *Speedup* is the mean increase in query processing time. Column *Static* indicates free-flow travel times (metric w_0) and *Dynamic* indicates the congestion costs (w). In the static case the path oracle is sufficient to solve each problem optimally; in the dynamic case we perform search.

In this work we omit the compression step entirely and propose a new type of reverse oracle that operates directly on first-move tables. As we will see, this seemingly naïve approach actually has several strong advantages:

1. **Hardware caching:** we need only one row per query and we can store that row in a low level cache on the processor. This can eliminate expensive memory read operations which otherwise occur after every cache miss.
2. **Software caching:** if two queries share the same target and metric their w-optimal paths can overlap. By caching extracted path data in these cases we can further improve oracle performance. This is similar to the "heuristic cache" optimisation developed for forward CPDs in (Bono et al. 2019).
3. **Memory requirements:** each row is independent from all the rest, which means we can load arbitrary subsets of the first-move table into memory, and we can store those subsets across different machines. This allows us to trivially parallelise the queryset computation and it also mitigates the increased storage cost (since there is no compression).

Choosing an Oracle

Choosing a graph oracle is a case of balancing memory requirements with performance. To recap: *forward oracles* allow first-move data to be effectively compressed but this comes at the cost of online performance. Meanwhile *reverse oracles* suffer from ineffective compression but improved performance, as only one row is required to extract a complete path and computing each first move requires only constant time. Table 1 gives a summary, comparing tradeoffs for different instantiations of optimal Oracle Search.

The experiment shows reverse oracles can be several times faster than forward oracles. Notice too reverse tables are more storage efficient than reverse CPDs. This is because compressing reverse data with run-length encoding is inefficient. In our implementation the reverse table stores 4 bits of information per move while our reverse CPD stores 32 bits per run (4 bits for the move and 28 bits for the index). Although there is some evidence of compression (the reverse CPD is not seven times larger; *only* 2.72 times) the savings are overwhelmed by the index size. The last row, Trim, shows the actual/active space requirements for the reverse oracle – i. e., the total storage for all reverse rows accessed during search. This indicates that not all nodes appear as targets in the queryset. If we knew these nodes in advance we could have only computed and stored their corresponding rows. This reinforces one of the the main advantages of reverse oracles: that the data can be trivially partitioned and, potentially, computed only in part.

[2]We omit a full Reverse CPD comparison due to size; tests on smaller instances indicate performance on par with Fwd Table.

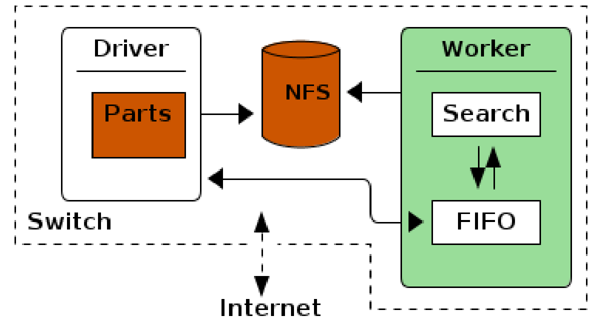

Figure 3: Schematic view of our cluster

Distributed Oracle Search

We deploy Oracle Search on a six-machine cluster managed with a simple script that distributes queries using standard GNU utilities. We use the following nomenclature:

Driver means the **head node** of the cluster. We read and distribute queries from this machine.

Worker means a machine in the cluster that is not the driver. On each **worker node**, we will load a *resident process* (the solver) with which we will communicate using a **FIFO**: a kernel-level file descriptor which allows for inter-process communication.

Partition means a division of a larger set of queries, all of which must be solved. The driver defines the partitions and subsequently allocates each one to a worker.

Our entire distributed approach works as follows. During the *offline phase*, we compute the reverse oracle. This involves a complete Dijkstra search for every node in the graph which records the first move in the (reverse) path towards the target. The rows are distributed across the different machines. (Notice that this limits each worker to solving only queries for its associated target nodes). We also load the underlying graph on each worker. During the *online phase*, we distribute path planning queries to the workers, including edge-cost changes, and we solve them in parallel, with one instance of Oracle Search for each available core. Finally, we collect a summary of results from each distributed computation.

Currently our system is a prototype which we use for feasibility testing. That means we use batched sets of pre-

generated queries instead of reading them in real-time from a stream. We also do not simulate real-time edge-cost changes but assume that cost updates from the changeset have already been applied to the graph, before receiving an associated query. We follow the same strategy in Figures 1 and 2 when comparing with CCH – i. e., we suppose the graph labels have already been updated and we measure only the time needed to customise auxiliary data.

Systems such as ours often come with a set of trade-offs, mainly due to the overhead incurred from distributing jobs across the (local) network (McSherry, Isard, and Murray 2015). In the experimental section we consider ways to reduce such overheads when most of the computational load comes from external functions (in our case, external calls to Oracle Search). Our main experiment draws on simulated congestion data for the city of Melbourne, Australia. We use congestion as an example of a "changing metric." When evaluating the efficacy of the system we consider throughput (as measured in queries per second) and solution quality (as measured by path cost w. r. t. the current graph metric w).

Simulating Melbourne

We use a road graph of Melbourne, Australia provided by OpenStreetMaps (OSM, OpenStreetMap contributors 2017). The graph has 167,758 nodes and 459,793 edges. We compute free-flow travel time for every edge by taking the edge length and dividing by max speed.

Our demand is computed from 785 unique origin-destination pairs provided by the 2012–2016 Victorian Integrated Survey of Travel and Activity (VISTA, Department of Transport, Victoria 2016). Every VISTA origin or destination represents a unique statistical area for census data collection purposes. We use the data as follows:

1. **Compute scenarios.** For each VISTA origin-destination pair, the survey reports an associated number of trips occurring during different periods of the day. The total number of commuter trips in the VISTA data is 651,445. We take these trips as the total set of queries to be solved. The VISTA data aggregates queries by centroids, which map to statistical areas in Melbourne. From the centroids, we generate concrete start and target locations within those areas and use these as the queryset.
2. **Model congestion.** We create a 8–9am model of Melbourne, the busiest time during a typical day. VISTA reports 77K sampled trips in that time period but the 2016 Australian Census (Australian Bureau of Statistics 2016) reports 1.2M *commuter trips* per day by car. Assuming approximately half of all trips occur in the morning, we scale the 8–9am VISTA data by a factor of 8 to obtain 616K peak-hour trips. We then simulate these trips on the graph of Melbourne using the software package Aimsun (Aimsun 2019). The result is a perturbed graph with increased edge costs which we treat as a congestion model.

To compute the congestion model we undertake a mesoscopic simulation of Melbourne. In a mesoscopic simulation every vehicle is considered a separate entity, but their behaviour is simplified – e. g., vehicles are either stopped or travelling at speed; there is no acceleration, but turn costs are taken into account. To model route choice (i. e., how vehicles react to traffic) we used a dynamic user equilibrium with the method of successive averages (Florian, Mahut, and Tremblay 2008), as implemented in Aimsun. Vehicles were released in a uniform fashion for four routing cycles and reaction time was set at $1.2\,\mathrm{s}$, mimicking human drivers.

The simulation was warmed up using 5 minutes of the same demand and then ran for a further 60 minutes. This produces modified travel times on about 12% of all edges – i. e., a likely optimistic but still plausible model for 8–9am congestion in Melbourne. Indeed, this relatively small perturbation actually impacts 89% of queries (580K).[3]

Experimental Setup

We implement Oracle Search[4]in C++ using G++ 7.4 with `-O3` on Ubuntu 18.04. Our test environment is a dedicated hardware cluster comprising six Intel NUCs (model `NUC817BEH`). These are commodity system-on-a-chip hardware, each featuring a four-core i7-8559U CPU running at 2.7GHz. We install 2 × 16GB RAM per machine and an NVMe solid state drive for storage. The machines have access to most of these resources but 2GB RAM is reserved for the operating system. The cluster is interconnected with an off-the-shelf gigabit router.

On each worker, we spin a thread which loads into memory the path oracle data, the graph and (for simplicity) any perturbed costs. The worker then opens a FIFO and *waits* for configuration data and for the start signal which begins the solving phase.

On the driver, we load the queryset and divide the instances into partitions – one per worker for simplicity. Each partition is copied to a shared network drive (NFS) and then the driver sends a signal to the workers along with configuration data.

The configuration data sets the parameters for the search (e. g., termination criteria) and the location of the queryset. Upon receiving the start signal each worker reads the queryset, performs the search and returns summary statistics per instance – e. g., solution cost, time elapsed, nodes expanded, etc.

Small querysets. The performance of a distributed system can be strongly impacted by communication overheads. For example, the time spent copying query data to the workers must be carefully weighed against the expected time savings that can be derived from parallelisation. To mitigate such issues we apply a simple rule: for querysets with $< 10\mathrm{K}$ instances[5] we perform the search on the driver. This means we also run a resident process on the driver. This configuration still involves copying data but only locally and the overheads are *much smaller* than over the network.

[3]Measured as #queries which require some amount of search.

[4]We use *warthog* as pathfinding library (https://bitbucket.org/dharabor/pathfinding, tag: `socs21`) and have uploaded the supplementary code and data for the experiments at: https://doi.org/10.5281/zenodo.4785122

[5]This value was found experimentally.

Results

To measure the performance of our distributed system we consider queries per second (i. e., throughput), which we measure in two different ways.

On the workers: we measure the time taken to read the queries, the time spent doing search, and the time to output the aggregated statistics. We will report one boxplot per configuration where each datapoint is the throughput on *a single worker.* An ideal system with no overheads would thus have a throughput five times higher on our cluster.

On the driver: we measure the time elapsed between sending out queries to workers and collecting all results; this includes the communication overheads.

In distinct experiments we then consider a variety of different settings including unbounded suboptimal search, bounded suboptimal search and time-budgeted search. In a fourth experiment we examine the scalability of our system and the impact of communication overheads on throughput.

Experiment #1: Any Route At All

In this experiment every query is solved by returning the shortest free-flow path. Solving such queries requires no search, only recursive move extractions from the oracle. Each path is w_0-optimal but its cost w. r. t. to w is unbounded suboptimal. We consider full path extraction and k-prefix queries where we only extract and return the first k moves of the w_0-optimal path. Results in Figure 4 are *per worker*.

We see that our system scales extremely well for any value of k including up to 720K full path queries per second on average – i. e., enough to provide simultaneous directions to every commuter in our demand model at every second (communication overheads notwithstanding).

In Figure 5 we measure the relative cost increase from returning w_0-optimal paths instead of w-optimal paths. The average difference between the two metrics is approximately 7.2%, with a standard deviation of 26%. Notice that 33% of paths are not affected by congestion while 1% of queries have double the w-optimal cost on average. In the worst case we report a 20x increase in travel time.

Experiment #2: Bounded Suboptimal Search

When computing driving directions we may not need to return the optimal path. Indeed, driving is inherently chaotic, with many stops and small unexpected delays. In these cases we can aim for faster throughput by running a bounded suboptimal search. In this setup we terminate Oracle Search as soon as the gap between the best feasible incumbent and the best lower-bound (as represented by the f-value of the current node) satisfies some criteria. Figure 6 shows *per worker* results for a range of multiplicative-factor-guarantees.

We see that when paths with larger suboptimality are acceptable the search terminates sooner and results in higher throughput: from 35K queries per second for optimal solutions to 212K queries per second for solutions up to 30% bounded suboptimal. In Figure 7 we report the percentage difference in solution cost between the optimal path and the path returned by suboptimal search. Results are presented as percentiles, with one plot representing one suboptimality guarantee. For example, when the cut-off is set at 10%, around 91% of paths returned are within 5% of the optimum.

Experiment #3: Budgeted Search

Another way to tackle search is to use a fixed budget, such as for time. In this setup we terminate Oracle Search after the maximum allowable time period has elapsed. This is useful in cases where we want to provide a response-time guarantee – e. g., "we will provide a route in under 1 s." This works because ours is a *realtime* system where we do not have to complete a search to return a solution. Figure 8 shows *per worker* results using different time cut-offs.

We see that the lower the cut-off the higher the throughput. Moreover, as shown in Figure 9, allowing more time produces solutions of higher quality. Indeed, when running the search for 1000 µs, we achieve performance almost equal to a full search. Notice how the change in throughput is not a direct function of the time limit. For example, moving from 30 µs to 300 µs does not yield a tenfold decrease in throughput. This is because not all queries exhaust their budget: for some queries with little or no congestion the optimal response response can be returned in less than 10 µs. Only a very small number of queries receive a timeout. Perhaps the best example is for 300 µs, where less than 0.00004% of queries have no solution but throughput (vs. optimal search) increases almost 50%.

Experiment #4: COST

The COST of a distributed system is defined in McSherry, Isard, and Murray (2015) as the Configuration that Outperforms a Single Thread. This metric measures the impact on performance from unavoidable overheads which systems such as ours introduce to distribute work and collect results. The COST of a system therefore is measured as the number of cores (or machines) required to outperform a single-threaded solver that tackles the same workload. In our case we measure speedup as compared to a single-threaded implementation. Figure 11 shows the COST of the system for different workloads and number of cores. For workloads smaller than 10K queries, the system does not distribute queries to workers but does all routing on the driver.

When the workload is very small – e. g., 100 queries – we do not manage to scale up, the communication costs[6] being larger than simply running the queries. With 1K queries, we can see that the performance does not increased with more than 2 cores. The curve looks identical on both sides because there is no distribution yet. Once we exceed 10K queries, our system is able to perform better than the single threaded configuration. After that, we are at least as fast (speedup = 0.98) as the single threaded implementation with a single core. The COST of our system is thus 2, the number of cores needed to outperform the single threaded implementation.

[6]The working thread still needs to read the queryset.

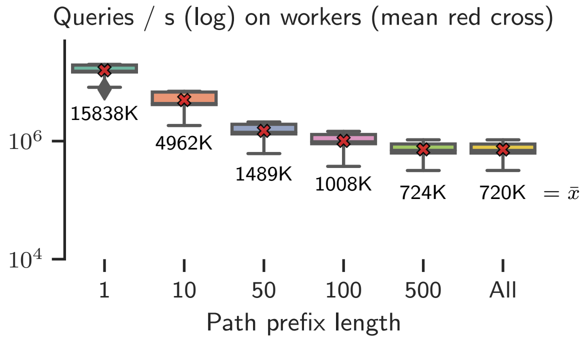

Figure 4: Throughput per worker when computing k-length prefixes. We do not perform search, we only extract the k first move using the CPD. The numbers below, in black, represent the mean throughput recorded across partitions.

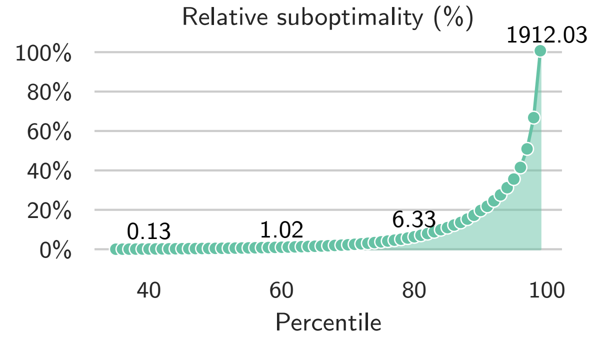

Figure 5: We measure the quality of the unbounded suboptimal path found by blindly following the oracle vs. optimal. We plot the mean suboptimality for each percentile, and, for every tick mark, the maximum in black.

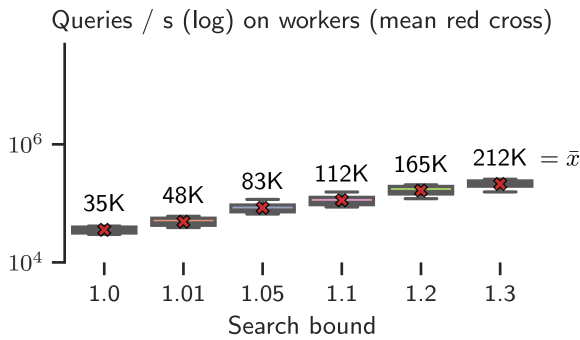

Figure 6: Throughput per worker when running a bounded suboptimal search with different quality guarantees – 1.0 means an optimal search. The numbers above, in black, represent the mean throughput recorded across partitions.

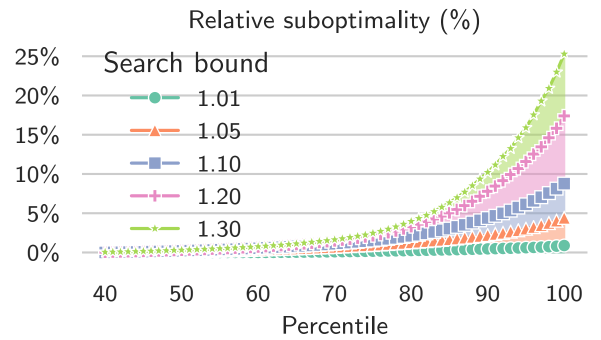

Figure 7: Percentage difference between the optimal path and suboptimal paths found for different admissibility percentage (% sub-optimal).

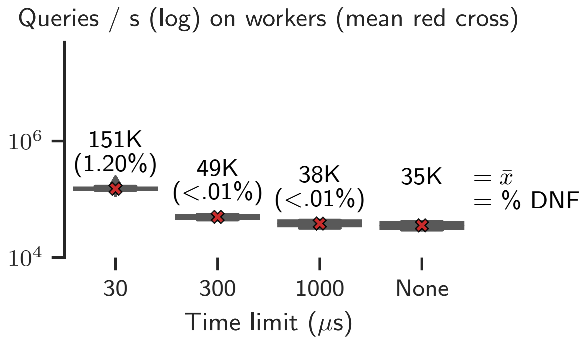

Figure 8: Throughput per worker when running a time-bounded search for different time budgets. The numbers above, in black, represent the mean throughput recorded across partitions.

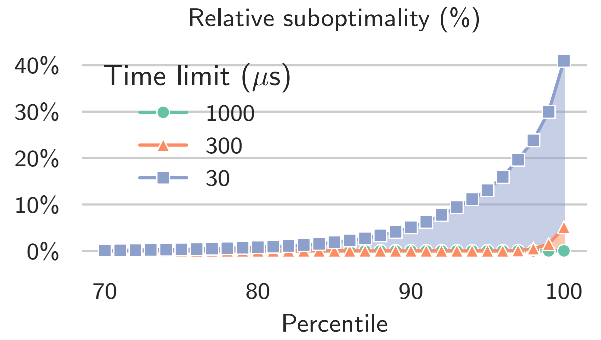

Figure 9: Percentage difference between the optimal path and suboptimal paths found for different time limits (µs).

Discussion

Our experimental results show that we scale well in a variety of scenarios. However most results are on the workers and with full queryset (651K). This is an *ideal setup* where we do not have to worry about communication overheads. The gap between the ideal throughout and the achieved throughput on the driver is shown in Figure 10.

Notice that as the suboptimality tolerance increases the

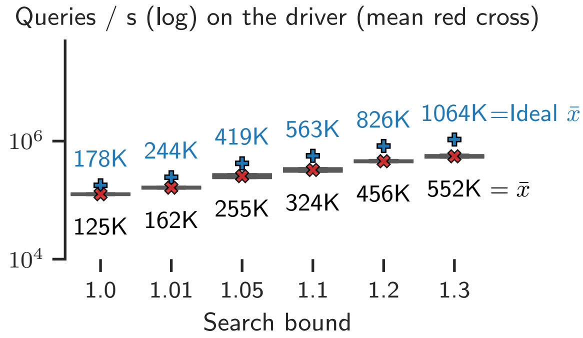

Figure 10: Throughput measured on the driver. The blue marks and numbers represent throughput on an *ideal system* – one with no communication overheads.

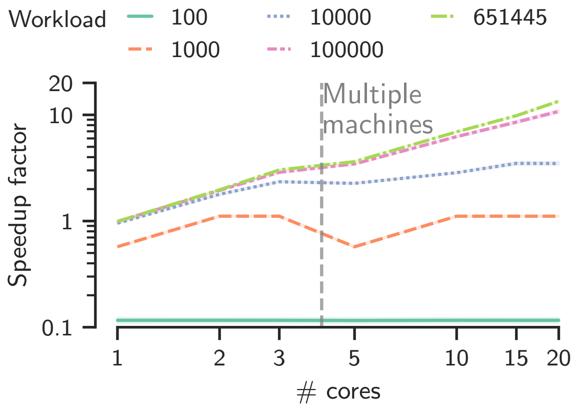

Figure 11: Speedup of our system vs. single-threaded search for different workloads. We measure on the driver, from sending out the queries to receiving the response from all workers.

gap between our system and the ideal widens. This is because the ideal setup only considers the time spent doing search in parallel whereas in practice one incurs a number of unavoidable overheads, from network latency to reading queries. When the workload is small (or easy) the overheads can dominate total time, resulting in workers becoming *starved*. Figure 11 shows that when there is sufficient work to distribute our system is up to 14 times faster than an equivalent single thread implementation. This means a speedup of 2.8 for each (4-core) machine in the system.

Workload balancing. In the experiments, we use a naïve allocation for the workers: we allocate an equal number of rows of the oracle to each. This leads to workers sometimes getting starved because the queries are not distributed evenly across the nodes. Because we know the queryset *a priori*, we can ensure similar workloads across workers. Doing this results in a 10% throughput gain, on the workers and on the driver. This is due to a smaller makespan – i. e., the system waits less for the slowest worker to finish.

Leveraging caches. We consider two optimisations that can improve cache behaviour. First, we *sort the queries by target*. After solving the first query, all subsequent queries with the same target will share the heuristic cache. When the w_0 paths overlap, the next query can be solved faster. We also *allocate targets to specific threads*. In this case queries can not only share a heuristic (software) cache but also the CPU can have first-move data already loaded in (hardware) cache.

When sorting the queryset beforehand and running the same experiments, we measured a throughput increase on the workers of 50% (52K query / s) when sorting and 60% (59K query / s) when adding thread allocation. This translated to an increase of 25% (153K query / s) on the driver when sorting and 62% (201K query / s) when adding thread allocation. Again, the significant increase when measuring on the driver is due to a reduced makespan on the workers.

Revisiting CCH

In Figures 1 and 2 we compared Customizable Contraction Hierarchies (CCH, Dibbelt, Strasser, and Wagner 2014) with Oracle Search and Dijkstra search to highlight the potential drawbacks of customising auxiliary data online. We now give a more detailed discussion of these experiments.

Our CCH implementation is from *RoutingKit*,[7] a freely available C++ library with overlapping authors to CCH. We used same graph and queryset as in our main experiment with free-flow travel time being the initial metric (w_0).

We experimented with a variety of changesets, from one modification per update (CCH-1) to every modified edge in the congested graph (CCH-all). In each case we pick as the changeset the top n perturbed edges from the congestion model where edges are ordered by the frequency with which they appear on a shortest path in the queryset. We notice that the cost of customising CCH data, even for a comparatively small (100 edges) changeset, is almost as expensive as modifying *all graph edges*, even when the repair is performed in parallel.[8] Figure 1 shows that for a modest number of customisations (< 1715 per queryset) CCH can be faster than Oracle Search, on average. As more customised queries are added, it becomes advantageous to switch. Eventually even Dijkstra search is faster than CCH, on average.

The largest speedups for Oracle Search (approximately 6x) are in the zero updates case – i. e., for queries where w_0 is the graph metric. In this setting all problems can be optimally solved using only the path oracle. Figure 2 shows an experiment where some proportion of queries are solved with w_0 and all the rest have a customised metric w. Here we find that Oracle Search is always faster, on average.

Our analysis shows that CCH improves on Oracle Search in one setting only: when the cost of customisation can be amortised over a large number of subsequent queries. When the metric changes often, such as per query, CCH performance drops quickly and no-repair methods are better.

[7] https://github.com/RoutingKit/RoutingKit (commit: `613b725`)

[8] The RoutingKit implementation we use supports parallel customisation only for the all-graph-edges setting.

Conclusion

In this paper, we consider new ideas for the design of centralised routing oracles that provide customised driving directions. We propose a family of anytime algorithms, called Oracle Search, which can identify feasible routes quickly and optimal routes eventually. We combine Oracle Search with a distributed workload management system and experiment with a congestion model for the city of Melbourne, Australia. Using a small commodity cluster we show that our approach scales to hundreds of thousands of simultaneous queries per second and more. We achieve these results in a range of settings including optimal, bounded-suboptimal and time-budgeted search. We further show compelling performance advantages vs. CCH (Dibbelt, Strasser, and Wagner 2014), a leading method from the recent literature.

As future work one may consider heuristics to determine a good/performant division of available work to worker machines. Also for dynamically adjusting search parameters during a budgeted/bounded setting. Another promising direction involves computing several different oracles, each with a different w_0 metric. This may improve performance in cases where the customised w metrics have different cost floors (cf. the current assumption which assumes w_0 is always freeflow travel-time).

Acknowledgements

Research at Monash University is funded by the Australian Research Council under grants DP190100013, DP200100025 and by a gift from Amazon.

References

Abraham, I.; Delling, D.; Goldberg, A. V.; and Werneck, R. F. F. 2012. Hierarchical Hub Labelings for Shortest Paths. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, 24–35. doi:10.1007/978-3-642-33090-2_4.

Aimsun. 2019. Aimsun Next 8.4 User's Manual. qthelp://aimsun.com.aimsun.8.4/doc/UsersManual/Intro.html. Accessed on: 2020-02-15.

Arz, J.; Luxen, D.; and Sanders, P. 2013. Transit Node Routing Reconsidered. In *SEA*, 55–66.

Australian Bureau of Statistics. 2016. 2016 Census QuickStats. https://quickstats.censusdata.abs.gov.au/census_services/getproduct/census/2016/quickstat/2GMEL?opendocument. Accessed: 2021-02-15.

Bono, M.; Gerevini, A. E.; Harabor, D. D.; and Stuckey, P. J. 2019. Path planning with CPD heuristics. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 1199–1205. AAAI Press.

Botea, A. 2011. Ultra-Fast Optimal Pathfinding without Runtime Search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

Botea, A.; and Harabor, D. 2013. Path planning with compressed all-pairs shortest paths data. In *Twenty-Third International Conference on Automated Planning and Scheduling*.

Delling, D.; Goldberg, A. V.; Pajor, T.; and Werneck, R. F. 2017. Customizable Route Planning in Road Networks. *Transportation Science* 51(2): 566–591. doi:10.1287/trsc.2014.0579.

Delling, D.; Kobitzsch, M.; and Werneck, R. F. 2014. Customizing driving directions with GPUs. In *European Conference on Parallel Processing*, 728–739. Springer.

Department of Transport, Victoria. 2016. Victorian Integrated Survey of Travel and Activity . https://transport.vic.gov.au/about/data-and-research/vista.

Dibbelt, J.; Strasser, B.; and Wagner, D. 2014. Customizable Contraction Hierarchies. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Proceedings*, 271–282.

Florian, M.; Mahut, M.; and Tremblay, N. 2008. Application of a simulation-based dynamic traffic assignment model. *European Journal of Operational Research* 189(3): 1381–1392.

Funke, S.; and Storandt, S. 2015. Personalized route planning in road networks. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 1–10.

Geisberger, R. 2015. Route planning. US Patent 9,175,972. Accessed: 2021-02-15.

Geisberger, R.; Sanders, P.; Schultes, D.; and Delling, D. 2008. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *WEA*, 319–333.

Geisberger, R.; Sanders, P.; Schultes, D.; and Vetter, C. 2012. Exact routing in large road networks using contraction hierarchies. *Transportation Science* 46(3): 388–404.

McSherry, F.; Isard, M.; and Murray, D. G. 2015. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

OpenStreetMap contributors. 2017. Planet dump. https://planet.osm.org. Accessed on: 2020-02-15.

Schultes, D.; and Sanders, P. 2007. Dynamic Highway-Node Routing. In *International Workshop on Experimental and Efficient Algorithms*, 66–79. Springer.

Strasser, B.; Botea, A.; and Harabor, D. 2015. Compressing Optimal Paths with Run Length Encoding. *Journal of Artificial Intelligence Research* 54: 593–629.

Zhao, S.; Chiari, M.; Botea, A.; Harabor, A. G. D.; Saetti, A.; and Stuckey, P. J. 2020. Bounded Suboptimal Path Planning with Compressed Path Databases. In Beck, C.; Karpas, E.; and Sohrabi, S., eds., *Proceedings of the 30th International Conference on Automated Planning and Scheduling*, 333–341. AAAI Press.