Karlsruhe Institute of Technology

Customizable Route Planning in External Memory

Bachelor Thesis of

Janis Hamme

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Dipl.-Inform. Julian Dibbelt
	Moritz Baum, M.Sc.

Time Period: 1st Aug 2013 – 30th Nov 2013

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 30th November 2013

Abstract

Route planning is a popular application for mobile devices, but still some problems have not been solved for road networks of continental size. Major constraints are the limited resources, that do not allow entire road networks to be held in main memory. Only a fraction of the complete road network can be held in main memory at once – most of the data must reside in external memory. To allow efficient computation of shortest paths, the data must be arranged in a way that allows efficient access.

Computed routes must be optimal for a specific metric, e.g., have an optimal distance or travel time. These metrics are likely to change frequently, e. g., as traffic data or driving preferences have to be considered.
However, current approaches to mobile route planning, are based on metric-dependent preprocessing of data, that is too expensive for execution on mobile devices. Hence, they can not comply with real-world requirements, that include dynamic and user defined metrics.

Customizable Route Planning [1, 2], is a speedup technique, that separates preprocessing in two stages. As a result of metric-independent graph partitioning in a first stage, fast metric customization is possible for arbitrary metrics afterwards.

Based on the results of Customizable Route Planning, we develop a proof-of-concept application for mobile devices. It demonstrates, that metric customization of road networks in external memory is possible in a few minutes. Then, the precomputed data can be used to compute exact shortest paths.

Zusammenfassung

Straßennavigation auf mobilen Geräten ist eine beliebte Anwendung, dennoch sind noch nicht alle Probleme für Straßennetzwerke kontinentaler Größe gelöst. Hautprobleme sind die begrenzten Ressourcen mobiler Geräte, welche es nicht zulassen, dass komplette Straßennetzwerke in den Hauptspeicher geladen werden können. Zur Berechnung von Routen auf mobilen Systemen müssen daher Techniken anwendet werden, welche das Problem so unterteilen, dass nur wenige Teile der Daten gleichzeitig aus externem Speicher geladen werden.

Berechneten Routen müssen bezüglich einer Metrik, z.B. hinsichtlich ihrer Länge oder Reisezeit, optimal sein. In der Realität sind Metriken vielfältig, da beispielsweise die aktuelle Verkehrssituation oder benutzerspeziefischen Fahrgewohnheiten berücksichtigt werden müssen. Aktuelle Lösungen zur Berechnung von optimalen Routen auf mobilen Geräten beruhen allerdings auf rechenintensiven, metrik-abhängigen Vorberechnungen, die nicht auf mobilen Geräten durchführbar sind. Sie können daher den realen Anforderungen, welche dynamische und benutzerdefinierte Metriken beinhalten, nicht gerecht werden.

Customizable Route Planning [1, 2], ist eine Beschleunigungstechnik, welche die nötigen Vorberechnungen zweiteilt. Durch geschickte Partitionierung, können metrikabhängige Vorberechnungen in einem zweiten, wesentlich schnelleren Schritt erfolgen.

Aufbauend auf den Ergebnissen von Customizable Route Planning zeigen wir, dass metrik-abhängige Vorberechnungen, durch das effiziente Nutzen von externem Speicher, auch auf mobilen Geräten in wenigen Minuten durchführbar sind. Die vorberechneten Daten können dann genutzt werden um, optimale, kürzeste Wege zu berechnen.

Contents

1. Introduction

The calculation of shortest paths in road networks is a well studied problem in computer science. The basic approach is to run Dijkstra's algorithm to calculate the shortest paths from one vertex to all other vertices [3]. Dijkstra's algorithm runs in almost linear time, depending on the data structures used [4]. However, as the graph instances used to represent road networks of continental size are large, the runtime on such graphs is still in the magnitude of seconds on modern hardware.

For many popular applications like car and pedestrian navigation it suffices to only calculate the shortest path between a start and an endpoint, known as the point-to-point shortest path problem. Several speed up techniques have been developed that solve the problem for realistic metrics orders of magnitude faster than Dijkstra's algorithm but still maintain correctness. Common approaches are to exploit the hierarchical nature of road networks or to direct the search towards the goal by pruning parts of the search space. Usually the problem is separated in an expensive metric-dependent preprocessing stage to gather auxiliary data that will speed up shortest path queries on the other hand. Examples are ALT [5], Arc Flags [6] or Contraction Hierarchies [7]. For an overview of research refer to [8].

For real world applications, however, metrics are likely to change frequently, as real time traffic data (e. g., traffic jams) is considered or when metrics should adapt to end users' driving preferences. Moreover different metrics should be supported simultaneously (e.g travel time and distance). To solve this particular problem Delling et al. introduced Customizable Route Planning [1, 2, 9]. One key feature is the separation into three stages: metric-independent preprocessing, metric-dependent customization and shortest path queries. The relatively slow metric-independent preprocessing parts the graph into a hierarchical multilevel partition – only topological data is used. In the customization stage a metric-dependent overlay is computed from the partition in several seconds. Afterwards, shortest path queries can be accelerated using a partition-aware multilevel implementation of Dijkstra's algorithm: whenever possible the algorithm switches to the sparse overlay on a higher level and thus heuristically speeds up the computation. Metric dependent customization data is kept separately from the graph and partition to support multiple metrics with one graph instance.

Often routing algorithms are designed with a client-server model in mind, but in reality navigation systems must support offline navigation, e. g., when no data connection is available. Even today not every region is covered by cellular networks, still the navigation

system must be able to compute routes. Moreover, online navigation raises additional costs for a mobile data plan so that many navigation systems do not use an online connection at all.
When algorithms are targeted for a stand-alone implementation on mobile devices instead of server applications additional requirements must be taken into account. Usually, mobile devices have limitations on main memory usage and processing speed. Therefore, only a fraction of the complete road network can be held in main memory at once – further required data must be read from slower external memory. External memory properties dictate that data from external storage devices – mainly flash memory – must be read blockwise. The data must be arranged with respect to its locality and the access patterns of the used algorithms to avoid unnecessary reads of blocks.

In this work we adapt Customizable Route Planning to provide a mobile routing implementation that is not only capable of answering shortest path queries but also to perform the metric customization in a reasonable amount of time from external memory.

1.1 Related Work

Some of the speedup techniques for road networks have been used in an external memory implementation for mobile devices. The earliest attempt we are aware of is an implementation of the ALT algorithm on a Pocket PC by Goldberg and Werneck in 2005 [10], nevertheless random queries on a North America graph take several minutes to complete in their scenario. In 2008 Sanders, Schultes and Vetter introduced a mobile version of Contraction Hierarchies [11], bringing query times down to the dimension of milliseconds. They use more sophisticated compression techniques to compress the stored graph and exploit locality in the data to reduce the number of blocks read. In his diploma thesis, Vetter showed that it is possible to build a complete routing application for mobile devices that meets many requirements of navigation systems today [12]. In a later work mobile CH is covered in a dynamic scenario. However, random query times increase dramatically when the weight of many edges is changed [7]. The existing implementations have in common that preprocessing with respect to time and memory consumption and therefore cannot be done on mobile devices.
Another approach to mobile routing covers mobile route planning in a server scenario with limited connectivity [13].

Beside work in the field of road planning, theoretical research on shortest paths in external memory has been done for planer digraphs [14], [15] and Meyer et al. presented an external memory variant of Dijkstra's Algorithm along with an experimental evaluation in [16]. However, these results are less applicable for our scenario as they are not exploiting the special characteristics of road networks.

A compact representation of graphs has been studied in [17], proposing some compression techniques that are related to those we use for our external memory graph. Further, external memory characteristics of flash storage and their impact on algorithm design have been examined by Meyer et al. [18] and [19] analyzes the impact of storage on the performance of mobile applications.

1.2 Outline

In this work, we adapt Customizable Route Planning, to allow *exact* shortest path queries as well as metric dependent customization to be done from external memory on a mobile device. We build a proof of concept implementation evaluating customization and queries with respect to time, block reads and memory consumption for the metrics *distance* and *travel time* on a mobile device.

After the basic preliminaries in Chapter 2, we first give a detailed overview of Customizable Route Planning in Chapter 3.

In Chapter 4 we present the details of our Customizable Route Planning variant in external memory: After discussing the characteristics of external memory storage, we engineer an external memory representation for the graph, partition and customization data designed to support efficient customization and queries. Therefore, we introduce an algorithm to reorder the original graph with respect to the multilevel partition and use compression techniques to keep the stored size small.

We compare different representations for the customization overlay graph and present a new algorithm to build skeleton graphs with respect to space consumption. Moreover, we propose overlay representations that allow efficient unpacking of shortcuts. We discuss distance queries and two different versions of shortcut unpacking along with their features in the external memory scenario.

Afterwards, we provide an experimental evaluation of our implementation in Chapter 5 and discuss the different variants towards their usefulness in real world applications.

2. Preliminaries

In this chapter we introduce the formal representation of road networks and and explain how to compute shortest paths in such a network using Dijkstra's algorithm. It provides the basis for Customizable Route Planning in Chapter 3.

2.1 Shortest Paths in Road Networks

Road networks are *directed, positively weighted graphs* $G = (V, E, \text{len})$, where V is a set of *vertices* representing intersections or points on the road, $E \subseteq V \times V$ is a set of *edges* representing road segments between the source vertex u and the target vertex v. The length function $\text{len} : E \to \mathbb{R}_{\geq 0}$, also referred to as *metric* of G, assigns a positive *edge weight* to all edges $(u, v) \in E$. Examples for metrics in road networks are *distance* or *travel time*.

We call G *directed* as edges $(u, v) \in E$ are ordered tuples. The graph G is called *undirected* if for each edge the reverse edge (v, u) is in E and $\text{len}(u, v) = \text{len}(v, u)$.

For a directed weighted graph $G = (V, E, \text{len})$ the corresponding reverse graph is $G^R = (V, E^R = \{(u, v) | (v, u) \in E\}, \text{len}^R(u, v) = \text{len}(v, u))$.

Shortest Paths

Paths in a graph are ordered lists of vertices $p = (v_1, v_2, \ldots, v_n), v_1 \ldots v_n \in V, (v_i, v_{i+1}) \in E$. The length of p is $\text{len}(p) = \sum_{i=1}^{n-1} \text{len}(v_i, v_{i+1})$, the sum of the weights of edges between vertices in p. If for $s, t \in V, p = (s, \ldots, t)$ is an s-t-path and for all other $p' = (s, \ldots, t)$ the length of p is smaller or equal to p', p is a *shortest path*. The length of shortest s-t-paths is denoted by the distance $\text{dist}(s, t)$. Although a shortest path is not necessarily unique it suffices to find one solution to solve the point-to-point shortest path problem.

2.2 Dijkstra's Algorithm

Dijkstra's algorithm was introduced in 1959 by Edsger W. Dijkstra [3] to calculate shortest paths in positively weighted graphs. The algorithm scans all vertices $u \in V$ in the order of their distance $\text{dist}(s, u)$ for a given start vertex s. It builds the foundation of many speedup techniques for the point-to-point shortest path problem.

To process vertices in the order of their distances it keeps track of a *tentative distance* $d[u]$ for each vertex, initially assumed to be ∞ for $u \neq s$, $d[s] = 0$. A *priority queue* holds

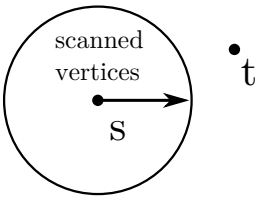
a set of vertices having a tentative distance $d[u] \neq \infty$. It is initialized to contain s at the beginning. As long as the queue is not empty a vertex u is *scanned*: the the vertex with the current minimum tentative distance is removed from the queue. Afterwards, outgoing edges of u are *relaxed*: if for $(u, v) \in E$ the tentative distance of v is larger than $d[u] + \text{len}(u, v)$, v is updated with its new tentative distance. Note that the distance of u is correct as $d[u]$ was minimal when u was removed from the queue and edge weights are defined to be positive, in particular $d[u] = \text{dist}(s, u)$. If not only the distance but shortest paths should be computed, one saves the parent $parent[v] = u$ whenever a vertex gets updated. Afterwards $parent[\cdot]$ defines a shortest path tree. If only a target vertex t is relevant, processing can be stopped as soon as t was scanned, as $\text{dist}(s, t)$ is known to be correct.

Algorithm 2.1: DIJKSTRA

Input: Graph $G = (V, E, \text{len})$, source vertex s
Data: Priority queue Q
Output: Distances d $[v]$ for all $v \in V$, shortest-path tree given by parent $[\cdot]$

```
   // Initialization
 1 forall v ∈ V do
 2     d[v] ← ∞
 3     parent[v] ← null
 4 Q.INSERT(s, 0)
 5 d[s] ← 0
   // Main loop
 6 while Q is not empty do
 7     u ← Q.DELETEMIN()                    // u is scanned
 8     forall (u, v) ∈ E do                 // relax edges
 9         if d[u] + len(u, v) < d[v] then  // update v
10             d[v] ← d[u] + len(u, v)
11             parent[v] ← u
12             if Q.CONTAINS(v) then
13                 Q.DECREASEKEY(v, d[v])
14             else
15                 Q.INSERT(v, d[v])
```

The performance of Dijkstra's algorithm mainly depends on the type of priority queue used. For Fibonacci heaps the runtime is in $O(|V| \cdot log(v) + |E|)$. As road network graphs are usually sparse, many implementations use binary heaps to avoid the constant overhead introduced through Fibonacci heaps. Binary heaps yield a runtime in $O((|V| + |E|) \cdot log(|V|))$ [20].



Vertices that have been scanned are also referred to as the *search space*. During execution the search space grows circular around s as the distance of scanned vertices increases.
A simple way to speed up Dijkstra for the point-to-point shortest path problem is *bidirectional search*. A second Dijkstra starting starting from t calculates distances on the reverse graph G^R. The search can be stopped as soon as the common tentative distance of forward and backward search $\mu = \overrightarrow{d}[u] + \overleftarrow{d}[u]$ exceeds the sum of the minimum keys of the Queues $\overrightarrow{Q}$.MINKEY + $\overleftarrow{Q}$.MINKEY [20].

3. Customizable Route Planning

Customizable Route Planning by Delling et al. [1, 2, 9] is a technique for fast computation of exact shortest paths in road networks which focuses on fast customization for arbitrary metrics. Therefore, CRP is separated into three stages:

1. *Metric-independent preprocessing:* find a multilevel partition for G, dividing G in *nested cells* on a fixed number of levels. Take only the graph topology as input.
2. *Metric-dependent customization:* use the partition to build a *multilevel overlay graph* for a given metric. Store the *customization data* independently to support multiple metrics for the same road network at the same time.
3. *Queries*: use *multilevel Dijkstra* to calculate shortest paths on G for a given metric. Use the customization data containing sparse overlay cells to speed up the computation over Dijkstra when possible.

3.1 Multilevel Partition

As nested multilevel partitions build the core of *Customizable Route Planning* we give a definition for such a partition of G.

Let $P = \{C_1, \ldots, C_k\}$ be an ordered set of disjoint *Cells* $C_i \subset V$ such that $\bigcup_{i=1}^{k} C_i = V$, then P is a partition of V. Equivalent to that definition is a mapping $\text{cell} : V \to \{1, \ldots, k\}$ assigning a *cell id* to each $v \in V$ defining the Cells C_i.

A multilevel partition $MLP = \{P^1, \ldots, P^L\}$ is an ordered family of partitions having L levels. The cell id for a vertex on level $\ell \in \{1 \ldots L\}$ is given by $\text{cell}_\ell(v)$. MLP is a *nested* multilevel partition under the constraint that

$$\forall u, v \in V, l \in \{1 \ldots L-1\}, \text{cell}_\ell(u) = \text{cell}_\ell(v) \Rightarrow \text{cell}_{\ell+1}(u) = \text{cell}_{\ell+1}(v)$$

which means that vertices that are in the same cell on a given level must be in the same cell on all higher levels. This further implies that for each Cell $C_i^\ell, \ell \neq L$ a *supercell* $C_j^{\ell+1}$ with $C_i^l \subseteq C_j^{\ell+1}$ exists and that $C_j^{\ell+1}$ is the union of *subcells*.

Cell graph

Based on the multilevel partition, a cell graph $CG_i^\ell = (C_i^\ell, E_i^\ell)$ can be defined. $E_i^\ell \subseteq E$ is the set of edges $\{e = (u, v) | e \in E, u, v \in C_i^\ell\}$ between vertices of that cell. Associated to

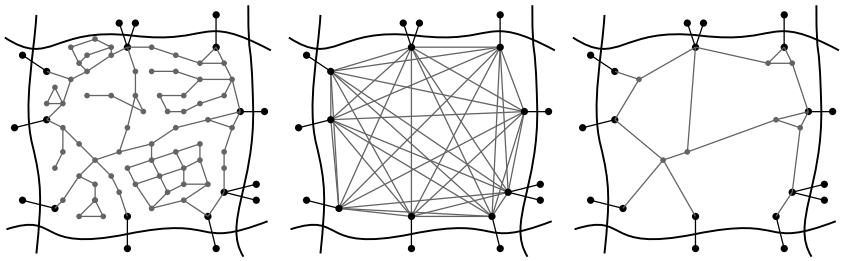
such a cell graph CG_i^ℓ is a set of incoming and outgoing *cut edges* $\hat{E}_i^\ell = \{(u,v),(v,u) \in E | \, \text{cell}_\ell(u) \neq \text{cell}_\ell(v)\}$ (edges that lead in and out of the cell).
A vertex $u \in C_i^\ell$ is a *boundary vertex* of CG_i^ℓ if an outgoing or incoming *cut edge* $(u,v) \in \hat{E}_i^\ell$ or $(v,u) \in \hat{E}_i^\ell$ exists. We denote the set of boundary vertices by $B_i^\ell \subset C_i^\ell$. Whenever the level ℓ it not required, we drop it from notation for better readability.

3.2 Customization

Given a source and target vertex s, t and a cell graph CG_i^ℓ: if neither s nor t are in CG_i^ℓ and the shortest s-t-path crosses CG_i^ℓ, the path will enter and leave CG_i^ℓ over a *cut edge* visiting *boundary vertices* u, v on cell entry and exit (note that $u = v$ is a possible option and that the path might cross a cell multiple times). Therefore, to find the correct distance $\text{dist}(s,t)$ it suffices to know the correct $\text{dist}(u,v)$ between all boundary vertices u, v of CG_i^ℓ and internals of the cell graph can be ignored.

This can be achieved by usage of an *overlay graph* of the actual cell graph in the query. An *overlay graph* contains a subset of the original vertices and preserves their distances. The goal of the customization is to calculate such an overlay graph for each cell and to preserve the distance between their boundary vertices. The simplest way to build such an overlay is to use a clique: add an edge (u,v) for all boundary vertices u, v having the edge weight $\text{dist}(u,v)$. These edges are *shortcuts* as they they skip vertices of the original graph.

Although the formal definition of overlay graphs requires that the distances of all its vertices are preserved [21], it is sufficient if the distance between the boundary vertices is preserved to maintain correctness in our scenario[1, 2]. A skeleton graph that preserves some internal vertices but reduces the number of edges can be built. Figure 3.1 gives an example for a clique or skeleton overlay cell. We denote such a *customization cell graph* by $\overline{CG}_i^\ell = (\overline{C}_i^\ell, \overline{E}_i^\ell)$.



(a) original **(b)** clique **(c)** skeleton

Figure 3.1: Original cell graph and customization cell after building a clique or a skeleton.

The *metric dependent customization* builds these cells in a bottom up fashion. It calculates distances between boundary vertices using cell local queries. As the the multilevel partition is nested, upper level cells are the union of subcells linked by their cut edges. To speed up computation, the overlay cells from the lower level can be used for the upper levels. As building the overlay is independent for each cell, the customization is highly parallelizable. It can be done in a few seconds on modern computers for road networks of continental size [1, 2]. We have a detailed look into the generation of customization cells in Section 4.3.2.

Partition Features

An algorithm to find such a partition as described in 3.1 is PUNCH [22]. It uses natural cut heuristics to find cells with a low number of cut edges. Examples of natural cuts are rivers or country borders that are typically crossed by a low number of edges.

As the size of overlay cells $\overline{CG}_i^\ell$ mainly depends on the number of their boundary vertices, a low number of boundary vertices is preferred. A maximum number of vertices U^ℓ for cells on a partition level characterizes the multilevel partition. It is set as a parameter to PUNCH, the algorithm partitioning the graph. The number of cells per level is not fixed in advance.

3.3 Multilevel Dijkstra

Multilevel Dijkstra (MLD) is a modification of Dijkstra's algorithm, that is aware of the multilevel partition and uses the precomputed overlay cells to heuristically speed up the computation. Switching to the overlay breaks down to the question which edges of a vertex should be relaxed:

Depending on the start and target vertex s, t every vertex $u \in V$ has a distinct level. It is the highest level on which it does not share a cell with neither s nor t.

$$\text{uncommonLevel}(u,v) = \begin{cases} 0 & \text{if } \text{cell}_\ell(u) = \text{cell}_\ell(v) \\ \max\limits_{\ell \in \{1..L\}} \text{cell}_\ell(u) \neq \text{cell}_\ell(v) & \text{else} \end{cases}$$

$$\text{level}(u) = \min(\text{uncommonLevel}(s,u), \text{uncommonLevel}(t,u))$$

Algorithm 3.1: MULTILEVEL DIJKSTRA

Input: Graph $G = (V, E, \text{len})$, source and target s, t, customization data for len, cell mapping cell_ℓ

Output: Distances $\mathsf{d}[v]$, shortest-path tree containing overlay edges given by $\mathsf{parent}[\cdot]$

```
// Initialization
forall v ∈ V do
    d[v] ← ∞
    parent[v] ← null
Q.INSERT(s, 0)
d[s] ← 0
// Main loop
while Q is not empty do
    u ← Q.DELETEMIN()                          // u is scanned
    level ← level(u)
    cell ← cell_level(u)
    forall (u,v) ∈ Ē^level_cell, (u,v) ∈ Ê^level_cell do   // relax overlay and cut edges
        if d[u] + len(u,v) < d[v] then                      // update v
            d[v] ← d[u] + len(u,v)
            parent[v] ← u
            Q.UPDATE(v, d[v]) // combined CONTAINS, INSERT, DECREASEKEY
```

For some vertex u, let ℓ be level(u) an c be the corresponding cell id $\text{cell}_\ell(u)$. The shortest path between s and t might cross the level-ℓ-cell C_c^ℓ of u, but as neither s nor t are in that cell it is save to use the overlay, otherwise $level(u) = 0$ and we are using the original graph. More precisely it is save to relax u's level-ℓ overlay edges in $\bar{E}_i^\ell$ plus the cut edges of u in $\hat{E}_i^\ell$.

Note that the level level(v) of vertices in the same cell equals to $level(u)$ due to the nested property of the partition. Internal vertices are automatically skipped as they cannot be reached by any edge.

We consider the original graph to be "level 0" and assume that the overlay edges for a level-0 cell $\overline{E}_i^0 = E_i^1$ equal the original edges of the corresponding level-1 cell and that the cut edges $\hat{E}_i^0 = \hat{E}_i^1$ are the same as for level 1. It avoids a case distinction between the original graph and the customization overlay in the pseudocode notation.

All the cells that can be reached by the query are referred to as the *search graph*. It is distinct for a given s, t pair. Figure 3.2 provides an illustration: the level of the cells around s and t grows whenever the supercell is left. In other words the lower level cells are bounded by the supercells of s and t. The search space grows circular around s, but it gets sparse for regions far from s and t.

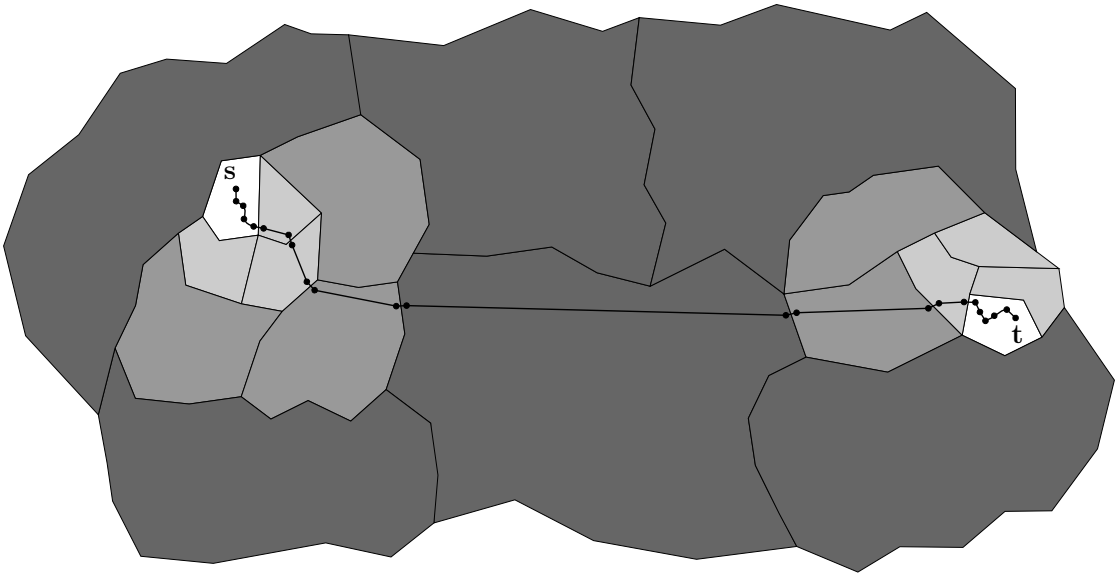


Figure 3.2: MLD search graph illustrated. Cells are darker for higher levels.

As the shortest path tree computed by MLD contains shortcut edges from the overlay, additional unpacking is required to give a complete shortest path response. The general approach to unpacking of shortcuts is to recursively use the lower level cells and perform unpacking queries down to original graph.

Running the query and unpacking of shortcuts takes a few milliseconds on a road network of continental size on a modern computer. Beside fast metric customization CRP is also suited to handle advanced requirements like turn costs or alternative routes. For an in-depth analysis of CRP in a client-server scenario refer to [2].

4. External CRP

In this chapter, we design an external memory variant of Customizable Route Planning targeted at mobile devices. First, it is essential to have a look at the characteristics of external memory and its impact on algorithm design.

4.1 External Memory

In a simplified model, memory can be classified into internal memory (RAM) and external memory storage. In general, RAM can be considered to be fast, volatile and as its name indicates, it is optimized for random access. On the other hand external memory is permanent and can store data that exceeds the size of internal memory. Together, they form a memory hierarchy: The CPU can directly operate on data in internal memory while data in external memory must be copied to internal memory before. Access to external memory is slow and dominates the actual computation in terms of runtime for many applications.

If the internal memory is not large enough to hold all the required data, data must be swapped in an out of external memory. While internal memory is usually accessible at byte addresses, external memory is divided into larger blocks. A file system abstracts from the raw devices and the operating system offers an interface to access these files. If an entity that is smaller than the minimum block size, the complete block must still be fetched. But if a larger fraction of the block is relevant, that additional data adds no additional cost. Therefore, one principle of external memory algorithm design is to *arrange data locally* to minimize the number of block reads. A fixed number of blocks can be held in internal memory and builds the *cache*. To avoid the costly read of blocks it is preferred to group data that is likely to be used together to *exploit cache effects*.

Storage of mobile devices is usually flash based, e. g., SD cards or internal flash memory. The impact of flash memory for algorithm design [18] and effects of flash memory and access patterns for application performance on mobile devices [19] have been studied: Although flash memory is known to offer fast random reads, block access patterns are still crucial for application performance. In fact, random access performance for both read and write operations is significantly slower for small block sizes. To improve performance, operating systems use a read ahead buffer combined with algorithms to detect sequential access patterns [23]. Data that is likely to be required next is fetched in the background. To take advantage of these effects it is recommended to *access blocks sequentially* if possible.

An alternative is to use unbuffered access to files with slightly lower overhead for small blocks on one hand, but no speed up for sequential access on the other. We provide a short comparison in our experimental evaluation in Chapter 5.

4.2 External Graph and Overlay Data

The goal of our application is to support efficient metric customization using an external graph, and to use the generated metric-dependent overlay to compute shortest paths afterwards. The graph and overlay data must be efficiently accessible in external memory. Building the external memory graph, which is available initially to the application, is part of the metric-independent preprocessing stage and does not have to be done on a mobile device.

First of all, consider the *access patterns* of customization and queries. Based on the patterns, the data can be reordered to improve *locality*. To build a customization cell either the corresponding level-1 cell of the original graph or all already customized subcells are required. Similar patterns apply to the shortest path queries: In most cases cells are visited completely, only cells that are located on the border of the search space might be visited only partially. Section 3.3 showed that cells in the search graph on a certain level are bounded by their supercell. Hence, it is likely that adjacent cells on the same level are required simultaneously. The level can change only at the cut edges in case the region of the supercell is left. To respect the locality for both query and customization, we store single cells as a contiguous memory block, but also store cells that have common a supercell consecutively in external memory as they are likely to be required together.
When cells are traversed by the query, the traversal is always leading over a cut edge. To avoid random block access, cut edges should be stored together with the cells on every overlay level, although that adds a small amount of redundancy. Further, the cell ids $\text{cell}_\ell(u)$ of a target vertex are required to find the level $\text{level}(u)$ for such a vertex.

4.2.1 Reordering

To arrange our input data according the access patterns, we reorder the original graph such that the vertices of all cells on every level have a consecutive ids. We further reorder the cell ids of the nested multilevel partition, such that cells, that build a common cell on the next higher level, follow each other.

Therefore, we are using a modified undirected breadth-first search algorithm, that processes entire cells first, before it continues with the next cell. Essential part is the $\text{uncommonLevel}(u, v)$ function defined in Section 3.3. Instead of using a single queue, our algorithm uses $L + 1$ queues to hold vertices according to their uncommon level. A Vertex u is taken from the lowest queue containig vertices and reached vertices v are inserted in the queue for $\text{uncommonLevel}(u, v)$, e. g., vertices of the same level-1 cell are inserted in queue 0 and vertices that do not have a cell in common with u on any level go to queue L. When a queue was empty, a cell is completed. The new order of cells and vertices, is then given by the order, in that vertices are processed and cells are completed. Algorithm 4.1 provides pseudocode notation. The algorithm is simplified, as vertices of a cell are not guaranteed to be connected: for the actual implementation a check is required if the cell is completed after a queue was empty – if not, the search must continue with an unprocessed vertex of that cell. We omit that routine for simplicity.
Note that vertices can be in multiple queues at the same time: if multiple vertices v_1, v_2 of the same cell are in a higher level queue and v_1 gets processed, v_2 is reached at some point and inserted in Queue 0 as the uncommon level has changed. We remove processed vertices at the front of the queues before a new vertex is taken out. Figure 4.1 provides an illustration of the new cell and vertex order after the algorithm was applied.

Algorithm 4.1: REORDER GRAPH (SIMPLIFIED)

Input: Graph $G = (V, E)$, start vertex v, multilevel partition
Output: Mapping for vertex ids vertexMap[·], Mapping for partition cells partitionMap[·][·]

```
   // counter used to assign indexes
1  vertexCounter ← 0
2  cellCounter[L] ← {0, ..., 0}
3  activeCell[L] ← {cell_1(v), ..., cell_L(v)}
   // L + 1 queues, queue 0 for vertices of the same cell, queues
      1,..,L according to their highest uncommon level
4  Queue[L + 1]
5  Queue[0].pushBack(v)
6  finished ← false
7  while not finished do
       // loop over levels
8      for level ∈ {0, ..., L} do
           // remove processed vertices from queue
9          while processed(Queue.front()) do
10             Queue.popFront ()
           // process vertex:  assign new id
11         if not Queue[level].queueEmpty() then
12             u ← Queue[level].front()
13             vertexMap[u] ← vertexCounter++
               // the current cell changes on lower levels
14             for (i ← 0; i < level; ++i) do
15                 activeCell[i] ← cell_{i+1}(u)
               // add adjacent vertices to queue for their uncommon level
16             forall e ∈ E : e = (u, v) ∨ e = (v, u) do
17                 Queue[uncommonLevel(u, v)].pushBack(v)
18         break
           // queue was empty ⇒ cell completed
19         else
20             if level < L then
21                 partitionMap[level][activeCell[level]] ← cellCounter[level]++
22             else
23                 finished ← true
```

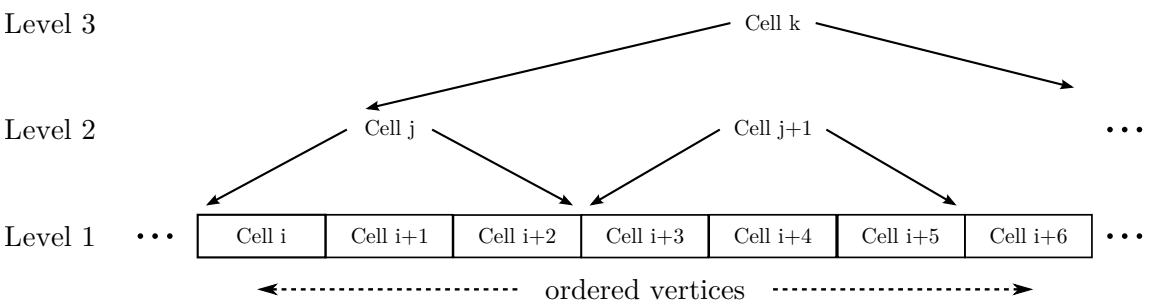


Figure 4.1: Vertex and cell order after reordering.

4.2.2 Data Structures

As complete cells are required in most cases, we serialize entire cell graphs to store them in external memory. The drawback of this approach is that random access on a vertex or edge level is not possible, but compression is straightforward as the size of data fields can be changed dynamically.

Our external graph data structure stores the original graph as serialized adjacency arrays of level-1 cells. We can exploit the order we have obtained to store the cells in a compact way. Every vertex in the original graph is identified by a global id that is given by the vertex order. As vertices in a cell have a consecutive ids, it is sufficient to know the global id of the first vertex, the remaining ids are given implicitly by their order. To encode the partition, every vertex is associated with the cell id for each level – but since that information is equal for each vertex in the same cell it needs to be stored only once per cell. The cells ids can be efficiently encoded into a single data field by using a fixed number of bits for each level. The the uncommonLevel(u, v), level(u), and cell$_\ell(u)$ functions can then be implemented efficiently by using bit operations. Basic graph and partition information, e. g., the number of vertices, edges, levels and the number of cells per level are encoded in the graph header. Vertices are not associated with any distinct features in our scenario and can be encoded implicitly: edges have a flag that indicates if they are the last edge of a vertex. Data fields with dynamic size can significantly reduce the stored size. In many cases, edge weights are small and fit into a single byte field. Due to the BFS ordering of vertices, ids of adjacent vertices have a small difference. Whenever possible we store the target vertex id as offset to the source vertex id in a single byte. Real-world navigation applications would probably not encode edge weights for multiple metrics, but include additional data fields or flags to classify roads into different types and to indicate restrictions, e. g., speed limits. In combination with the edge length, various metrics, e. g. travel time, can be derived at runtime. For our purpose, however, a proof of concept implementation, it suffices to include a distance and a travel time metric.
A similar approach to our graph reordering and compression of vertex ids was taken by Blandfort et al. in their experimental analysis of a compact graph representation [17]. They use a seperator tree to reorder vertices and store offsets of vertex ids in combination with sophisticated encoding methods to keep the representation compact.

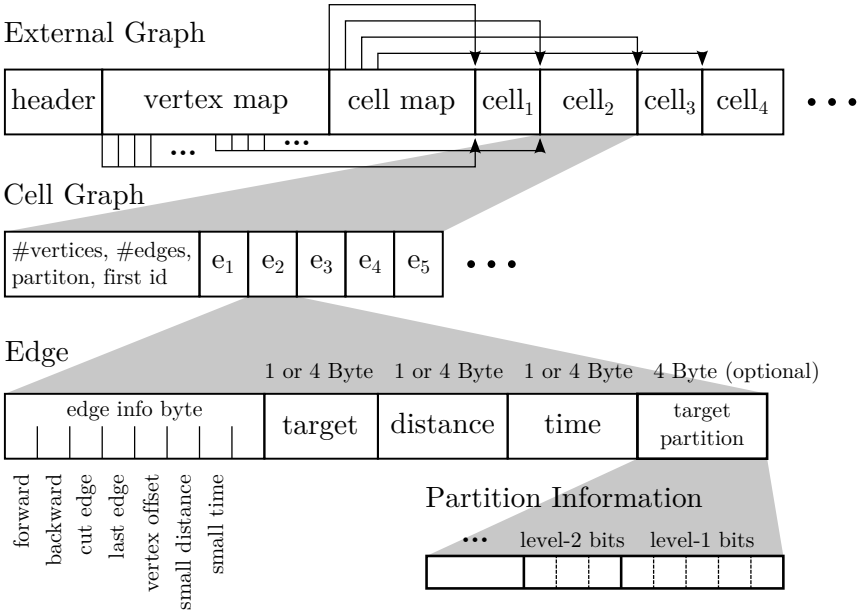


Figure 4.2: External graph data structure overview.

Cut edges include the cell ids of target vertices as they are required to traverse into another cell. Random accessible maps for vertex and cell ids point to the memory addresses of corresponding cells. The vertex map is required to find start and target cells for random queries. Real-world applications require more complex lookup data structures, e. g., address mappings or spatial indexes. The cell map is required for customization and cell traversal. Alternatively, the addresses of adjacent cells could be stored locally at the cut edges, but that would require a overlay data structure of fixed layout. Recall that the graph data is metric-independent and must not change for different metrics. To preserve the option of using arbitrary size overlay data and as the cell map is small and cache efficient we decided to use the map to locate adjacent cells instead of fixed memory addresses. The overlay data structure contains a similar cell map for multiple levels.

Unfortunately, we cannot fit cells into a fixed number of aligned blocks as the size of cells varies and is given by the partition. In Mobile Contraction Hierarchies [11], Sanders et al. separate their graph instance into independent accessible segments that fit exactly into one memory block. In our scenario, the separation of cells in exactly fitting segments is unsuited as the complete cell is required in most cases. Additionally, blocks that overlap cell boundaries can be accessed from cache in case the next cell is required. An overview to the entire external graph data structure is shown in Figure 4.2.

The overlay data structure is build similarly to the external graph. No vertex map is required as cells are only addressed by their id and level. Serialization of cells works differently: global vertex ids must be stored explicitly as only a subset of the original vertices is included. Local indexes of the stored vertices, at least all boundary vertices of the cell, are used to encode the internal structure. Only one edge weight must be stored per edge as the overlay data is metric-dependent.

Loading cells. For deserialized cell graphs in internal memory we use a standard adjacency array. We make a distinction between the *local index* of a vertex and the *global id* that is unique for the complete graph. Multiple cells can be deserialized or copied into in a single adjacency array by appending cells at the end, see Figure 4.3. Edge and vertex indexes are offset by the number of vertices and edges, that currently exist in the adjacency array. This way we can easily join subcells into a single graph for the customization or build the search graph during the query.

For cut edges the target vertex index is unknown. It is resolved using a hash map, holding a mapping from global id to a local vertex index for all boundary vertices – if the target cell is loaded yet. For simplicity, the representation of vertices and edges is verbose: each vertex stores its global id and its cell ids, the same information for target vertices is held by edges.

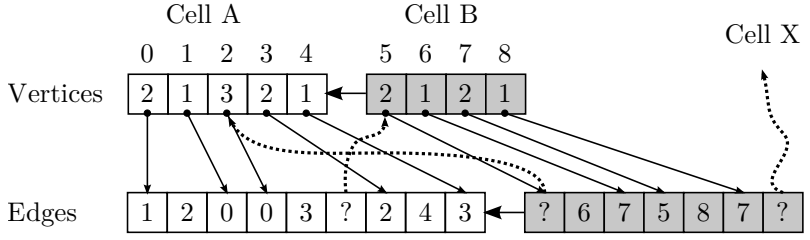


Figure 4.3: Adjacency array of cells in internal memory. Multiple cells can be loaded into a single adjacency array, cut edges must be resolved using the global vertex ids.

4.3 Customization

During the customization stage the metric-dependent overlay data structure is built from the external graph. Overlay cells have to be built for cells on each level such that the distance dist(u, v) for *boundary vertices* u, v is preserved (refer to Section 3.2). This process must be repeated for every metric that should be used in the following query stage.

Building a customization cell consists of the following steps:

1. Obtain origin cell: for the lowest level use the cells from the graph, for all upper levels join already customized subcells.
2. Run shortest-path queries for each boundary vertex.
3. Create a distance-preserving customization cell from the query results.

4.3.1 Parallelization

For optimal run times, the customization of cells must be parallelized. Even mobile devices feature multi-core processors with up to four cores today. In general, the customization is well parallelizable, as the overlay for each cell can each be computed independently. The only constraint is that subcells of a cell are customized first, as using the overlay cells speeds up the customization on the upper levels. The common approach is to customize cells bottom-up and to complete lower levels first. Customization can either be parallelized per cell or per query for each boundary vertex. But as creating the overlay cells from the query results can take a significant amount of time, depending on the overlay representation used, it pays to parallelize on a cell level.

As the customization reads and writes cells from and to external memory, efficient parallelization is non-trivial in our scenario. I/O operations, reading and writing cells, must be synchronized between threads and cells must be written in the order of their cell id.
The order of cells guarantees that subcells are stored consecutively. If subcells are loaded to customize their supercell, these cells can be read sequentially. Moreover, the complete graph or entire levels of the can be processed sequentially as we customize cells in the order of their cell ids (recall Figure 4.1). We further do not want to change the stored order to maintain locality for shortest path queries.
Threads should not block and waste time due to I/O (assuming that the customization is not I/O dominated). Blocking might occur in two cases. First a thread might require a new cell but another thread is blocking I/O, secondly a thread might wait to write a customized cell as either the previous cell has not been written yet by another thread or I/O is currently blocked.

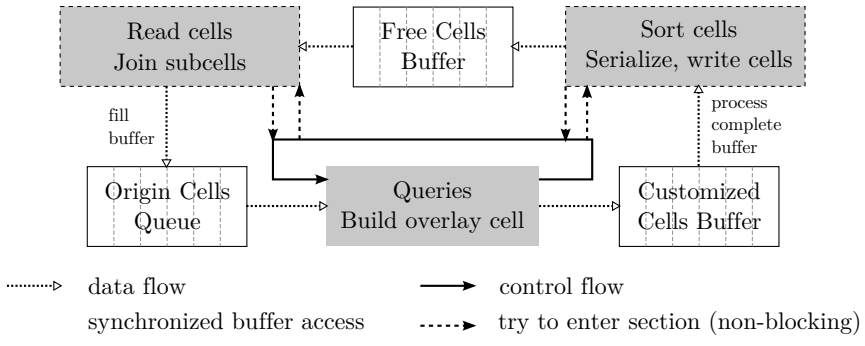


Figure 4.4: Parallelization of the customization: data and control flow per thread. The cell buffers are shared among all threads.

To overcome these limitations we use a queue that holds origin cells ready to be customized and a buffer to temporarily hold the customized cells. I/O operations are not necessarily done before and after customizing a cell. Each thread attempts to write back the customized cells or load new origin cells, but if that fails, it continues to customize another cell. Before cells are written they must be ordered by their cell id. Cells that are out of order remain in the buffer. We reasonably limit the overall number of cells being either in the work queue or waiting for write-back to keep the memory requirements low. Therefore, we use a fixed number of containers that hold the cell graphs. They are reused for different cells to avoid fragmentation of internal memory. An intermediate buffer holds currently unused containers. Figure 4.4 visualizes the data and control flow of the customization.

4.3.2 Building the Overlay

Let $CG_i = (C_i, E_i)$ be a cell graph that should be customized and $B_i \subseteq *C_i$ the subset of boundary vertices. A customization cell graph $\overline{CG}_i = (\overline{C}_i, \overline{E}_i)$ that preserves distances $\text{dist}(u, v)$ for $u, v \in B_i$ should be found. (Recall the definitions from Chapter 3)

Fist of all we run a shortest path query form each boundary vertex $b \in B_i$. The queries yield a shortest path tree $\mathsf{parent}_b[\,\cdot\,]$ and distances $\mathsf{d}_b[\,\cdot\,]$ for each boundary vertex. The distance between boundary vertices $u, v \in B_i$ is given by $\text{dist}(u, v) = \mathsf{d}_u[v]$. If no path between u and v exists the distance equals infinity $\mathsf{d}_u[v] = \infty$.

The obvious approach to construct $\overline{CG}_i$ is to build a *clique*. Set $\overline{C}_i = B_i$ and $\overline{E}_i = B_i \times B_i$ with $\text{len}(u, v) = \mathsf{d}_u[v]$ for $(u, v) \in \overline{E}_i$. For cliques, the number of edges $|\overline{E}_i| = |B_i|^2$ is quadratic in the number of boundary vertices. Cliques have been used as default representation by Delling et al. for in CRP [1, 2]. They have a major advantage over other overlay types: They are trivial to build and and the topology of cell graphs stays the same for all metrics. As the topology is fixed, cliques provide performance guarantees for any metric. Instead of an adjacency array representation, matrices can be used to store edge weights, allowing a lightweight implementation.
However, cliques have a major drawback in the external memory scenario. A quadratic number of edges results in a relative large stored size of the cell. I/O is expected to be the bottleneck for shortest path queries and it pays to use more sophisticated representations that yield a smaller stored size.

For realistic metrics like *distance* or *travel times*, road networks exhibit a strong hierarchy: some vertices are more import than others, as they are more likely to be part of shortest paths, e. g., highways are likely to be used for distant s and t. The same applies for cell graphs: some of the internal vertices are part of many shortest paths between the boundary vertices. A huge part of the vertices is not covered by any shortest paths between boundaries at all, see Figure 4.5 for an example. The number of edges can be reduced by preserving some of the important internal vertices.

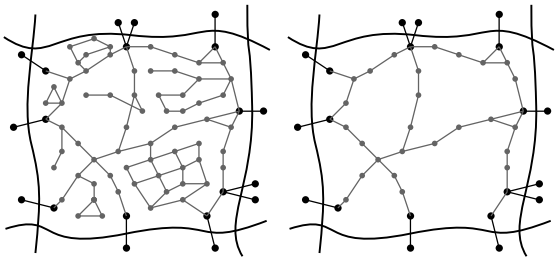


Figure 4.5: Original cell graph (left) and shortest paths between boundary vertices (right).

4.3.2.1 Creating Cell Graphs for Preserved Vertices

Customization cell graphs $\overline{CG}_i = (\overline{C}_i, \overline{E}_i)$ of the overlay must preserve the distances $\text{dist}(b, c)$ between their boundary vertices $b, c \in B_i$. Beside the boundary vertices B_i, preserved vertices $\overline{C}_i \subseteq C_i$ in the overlay cell can contain *internal vertices* $C_i \setminus B_i$ of the original cell. In the following we discuss, how to find the overlay edges $\overline{E}_i$ for a given set of preserved vertices $\overline{C}_i$, such that the customization cell graph $\overline{CG}_i$ preserves the distances between boundary vertices. We are only considering vertices on shortest paths $p_{b,c}$ between boundary vertices $b, c \in B_i$.

Definition and Theorem. Let $p_{b,c} = (b, \ldots, c)$ be the shortest paths between *boundary vertices* $b, c \in B_i$ given by the shortest path trees $\mathsf{parent}_b[\cdot]$. We add an edge (u, v) with $\text{len}(u, v) = \text{dist}(u, v)$ to $\overline{E}_i \subseteq \overline{C}_i \times \overline{C}_i$ if a subpath $p' = (u, \ldots, v)$ exists for any $p_{b,c}$ and p' contains no other preserved vertices $a \in C_i, a \neq u, v$. Then $\overline{CG}_i = (\overline{C}_i, \overline{E}_i)$ preserves distances $\text{dist}(b, c)$ for boundary vertices $b, c \in B_i$.

Proof. First of all shortest b-c-paths cannot have a smaller $\text{dist}(b, c)$ in $\overline{CG}_i$ as only edges (u, v) of length $\text{dist}(u, v)$ in CG_i were added and the concatenation of shortest paths cannot yield a shorter path.
We must consider two cases. Either an edge (b, c) exists in $\overline{E}_i$, but then the length of the edge is $\text{dist}(b, c)$ by construction. Or the shortest path $p_{b,c} = (b, \ldots, a_0, \ldots, a_n, \ldots, c)$ in CG_i contains a finite number preserved vertices $a_i \in C_i$. It can be split in subpaths containing no further preserved vertices (these subpaths are also shortest paths). An edge was added for each subpath and the sum of the edge lengths corresponds to $\text{dist}(b, c)$. □

A similar approach to overlay graphs has been taken by Schulz et al. [24] and the proof of correctness for this customization cell graph resembles the edge-min overlay graph defined by Holzer et al. [21]. However, our customization cell graph is technically not an overlay, as we do not preserve the distance between all preserved vertices $\overline{C}_i^\ell$ but only *between boundary vertices*. The generated graph is not necessarily edge-minimal as the shortest paths between boundary vertices in $\mathsf{parent}_b[\cdot]$ might include different shortest subpaths between preserved vertices. In this case redundant edges are added to $\overline{E}_i$. The effect should be negligible for road networks, as in most cases edge weights differ enough to make shortest paths unique.

Algorithm. For a given set of vertices $\overline{C}_i$, Algorithm 4.2 constructs the edges directly from the shortest path trees $\mathsf{parent}_b[\cdot]$ and the distances $\mathsf{d}_b[\cdot]$. The shortest paths trees $\mathsf{parent}_b[\cdot]$ contain the reverse shortest paths from any boundary vertex to boundary vertex b. We run an outer loop over all preserved vertices and an inner loop over all shortest path trees: if a preserved vertex $a \in \overline{C}_i$ is in a shortest path tree $\mathsf{parent}_b[\cdot]$, we find the reverse subpath from a to the next preserved vertex $a' \in \overline{C}_i$. We use a map for all preserved vertices to mark found subpaths from a to a'. As all reverse subpaths starting from a are found successively, we can use an increasing index to mark preserved vertices a'. If a subpath was not marked yet, an edge (a', a) is added. The edge length is given as difference of the distances in the shortest path trees $\mathsf{d}_b[a] - \mathsf{d}_b[a']$.

Before the cell can be serialized and stored in external memory, an adjacency array based cell graph must be built. To built such a graph representation the number of edges must be known in advance for each vertex and edges with identical source vertex must be stored consecutively. Therefore, we sort the edges by their cell local source vertex index. Cut edges $\hat{E}_i$ must be included in the cell graph.

Algorithm 4.2: FIND CELL EDGES

Input: Cell graph $CG_i = (C_i, E_i)$, Boundary vertices $B_i \subseteq C_i$, preserved vertices $\overline{C}_i$, shortest path trees $\mathsf{parent}_b[\cdot]$ and distances $\mathsf{d}_b[\cdot]$ for all boundaries
Output: Customization cell edges $\overline{E}_i$ and edge lengths $\text{len}(e)$ for $e \in \overline{E}_i$

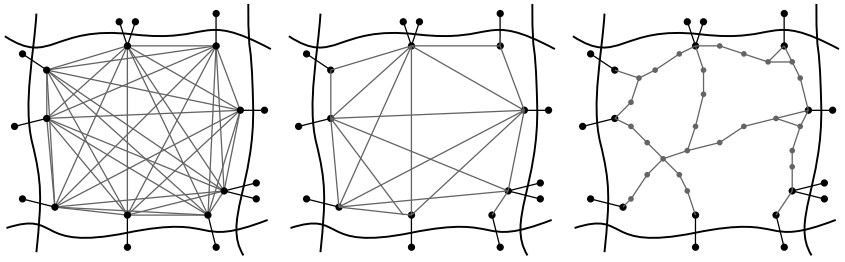
```
// map to mark existing edges between preserved vertices
edgeMarker ← 0
edgeMap[|C̄_i|] ← {0, ..., 0}
// loop over preserved vertices
forall a ∈ C̄_i do
    edgeMarker++
    // loop over boundaries
    forall b ∈ B_i do
        if (a ≠ b and a is on shortest path in parent_b[·]) then
            v ← parent_b[a]
            // find subpaths that do not contain other preserved vertices
            while v ∉ C̄_i do
                v ← parent_b[v]
            if edgeMap[v] ≠ edgeMarker then
                Ē_i ← Ē_i ∪ {(v, a)}          // add an edge
                len(v, a) ← d_b[a] − d_b[v]    // set length
                edgeMap[v] ← edgeMarker
```

4.3.3 Overlay Types

Algorithm 4.2 provides a general approach to define a set of overlay types beside cliques. We examine various variants in the context of external memory CRP. Our goal is to keep the stored size of the overlay small, since small overlay data reduces the number of blocks that must be loaded during shortest path queries. Hence, a speedup in run times is expected by using more sophisticated overlay representations.
Using Algorithm 4.2 the topology of a cell graph is defined by the vertices $\overline{C}_i$ that are preserved. At least all boundary vertices must be included.

4.3.3.1 Cliques, Reduced Cliques and Full Overlay

For *cliques* the number of edges $|\overline{E}_i| = |B_i|^2$ is the squared number of boundary vertices. The main advantage of cliques are their metric-independent topology and their efficient matrix graph representation. Therefore, we use matrices to store clique cell graphs of metric-independent size in external memory. All other types use an adjacency structure to serialize cell graphs.



(a) clique **(b)** reduced clique **(c)** full

Figure 4.6: Cliques, reduced cliques and full overlay

Two trivial ways to define $\overline{C}_i$ are either to set the preserved vertices to the boundary vertices $\overline{C}_i = B_i$, or to preserve all vertices that are covered by shortest paths between boundaries. The outcome is either a *reduced clique* or a *full* overlay (see Figure 4.6).

Reduced cliques are a sparse variant of cliques. They contain shortcut edges between boundary vertices as cliques do, but only if the the path between two boundary vertices b, c is not leading over another boundary vertex in between.

Reduced cliques for CRP have also been examined in [1] but their approach is different. Delling et al. build cliques first, and run local Dijkstra Queries on the complete overlay to find redundant edges afterwards. However, for our approach to customization in external memory, it is essential that each cell is built independently.

The *full* overlay preserves complete paths between boundary vertices. In other words parts of the graph that are not relevant for shortest paths between boundaries are removed. Our experimental evaluation in Chapter 5 show, that such a verbose overlay is not suited for external memory CRP. Nevertheless, the resulting number of vertices and edges provide a good indicator for the hierarchical properties of a metric.

4.3.3.2 Skeleton Graphs

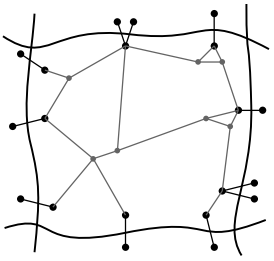


Figure 4.7: Skeleton cell graph

To further reduce the number of edges we propose a method to create *skeleton* graphs. When internal vertices beside the boundary vertices are preserved in the customization cell, the number of edges changes (see Figure 4.7) compared to cliques.

Skeleton graphs have also been examined by Delling et al. in the original CRP publication [1], but the algorithm we use for their generation is new. The original approach was to start with *full* cell graphs as defined in 4.3.3.1 and to greedily contract low degree internal vertices. Instead of removing vertices bottom up, we propose adding them top-down. To find a good set of vertices we formalize the change in the number of edges if an additional vertex is preserved:

Let $\overline{C}_i$ be the set of preserved vertices and $v \in C_i$ an internal vertex on a shortest path between boundary vertices. Further v should not be preserved $v \notin \overline{C}_i$ yet.
$\overline{E}_i$ is the set of required edges for $\overline{C}_i$ and $\overline{E}'_i$ is the set for $\overline{C}_i \cup \{v\}$. We define the reduction of edges

$$\text{edgeReduction}(v) = |\overline{E}_i| - |\overline{E}'_i|$$

as a measure of v's importance. Taking the edge reduction, we can greedily set vertices to be preserved until the number of edges cannot be reduced further. To calculate the edge reduction for each vertex, without running Algorithm 4.2 for each, we must specify the edge reduction in greater detail.

To calculate the edge reduction for a vertex v, we consider the shortest paths $p_{b,c}$ between boundary vertices $b, c \in B_i$. For a subpath $p' = (a_1, \ldots, v, \ldots, a_2)$ of any $p_{b,c}$ such that $a_1, a_2 \in \overline{C}_i^{\ell}$ are the only preserved vertices in p', an edge is added to $\overline{E}_i$ (see Section 4.3.2.1). If we further consider *all* such subpaths p' of any $p_{b,c}$ having a distinct tuple (a_1, a_2), then for each subpath an edge is created. We denote the set of all p' leading over v by *ShortestSubpaths*(v).
But if v would be a preserved vertex, an edge from each distinct a_1 to v and an edge from v to each distinct a_2 would be added. Those vertices build the neighborhood of v, in particular

$$\overleftarrow{N}(v) = \{a \mid (a, \ldots) \in \textit{ShortestSubpaths}(v)\}$$
$$\overrightarrow{N}(v) = \{a \mid (\ldots, a) \in \textit{ShortestSubpaths}(v)\}$$

for vertices that are adjacent to v over an incoming edge or an outgoing edge. The edge reduction is then given by

$$\text{edgeReduction}(v) = |\mathit{ShortestSubpaths}(v)| - |\overleftarrow{N}(v)| - |\overrightarrow{N}(v)|$$

Fortunately the edge reduction can be calculated by traversing the shortest path trees $\mathsf{parent}_b[\,\cdot\,]$ for all relevant vertices. For each vertex $a_1 \in \overline{C}_i$ (outer loop) and each boundary vertex $b \in B_i$ (inner loop) Algorithm 4.3 traverses shortest paths in $\mathsf{parent}_b[\,\cdot\,]$ from a_1 until an other preserved vertex $a_2 \in \overline{C}_i$ is found. Subpaths $p' = (a_1, \ldots, a_2)$ starting at a fixed a_1 are processed consecutively. If the path's target vertex a_2 was not in a path with source a_1 yet, the path $p' = (a, v_0, \ldots, v_i, t)$ is a distinct shortest subpath for all vertices v_i in between. We use counters to calculate the number of distinct subpaths and their distinct source and target vertices. To avoid double counts we must mark vertices for sources a_1 and targets a_2.

Afterwards, the vertex yielding the highest reduction of edges is added to the set of preserved vertices. We repeat calculating the reduction of edges and preserve more vertices until the number of edges cannot be reduced any further. Serialized vertices take less space than edges in our case. Else the routine would have to stop as soon as the reduction falls below a certain threshold.

4.3.3.3 Preserving Boundary Vertices on Upper Levels

With regard to edge unpacking in the following Section 4.4, we define an overlay representation that makes guarantees about preserved vertices. For CRP, the shortest paths, that are found by the query, contain shortcuts from the overlay. To obtain a full path description, cells must be unpacked recursively from top level cells down to level-1 on the origin graph.

To avoid recursive unpacking, boundary vertices of level 1 are preserved on all upper levels. Therefore, we store a flag to indicate that these vertices must be preserved on all upper levels. Many of the level-1 boundaries do not remain on higher levels as they do not appear on shortest paths.
We can then unpack shortcut edges directly on the origin graph and reduce the number of blocks we need to read.

Our experimental evaluation in Chapter 5 shows that level-1 boundaries are a good choice for the preserved vertices $\overline{C}_i$ and that the resulting size of the overlay data compares to the other overlay representations. We consider two variants with either *skeletons* or *reduced cliques* on level-1 and preserved boundary vertices on upper levels.

Algorithm 4.3: FIND SKELETON VERTICES

Input: Cell graph $CG_i = (C_i, E_i)$, initially preserved vertices $\overline{C}_i = B_i$, shortest path trees $\mathsf{parent}_b[\cdot]$
Output: Customization vertices $\overline{C}_i$

```
1  nextRound ← true
2  while nextRound do
       // initialize counter
3      distinctSubpaths[|C_i|] ← {0...0}
4      distinctSource[|C_i|] ← {0...0}
5      distinctTarget[|C_i|] ← {0...0}
       // maps to avoid double count
6      sourceVisited[C_i|] ← {0...0}
7      targetVisited[|C_i|][|C̄_i|] ← {false...false}
8      sourceMarker ← 0
       // loop over preserved vertices
9      forall a ∈ C̄_i do
10         sourceMarker++
           // loop over boundaries
11         forall b ∈ B_i do
12             if (a ≠ b and a is on shortest path in parent_b[·]) then
                   // traverse path to find other preserved vertex (target)
13                 t ← parent_b[a]
14                 for t ∉ C̄_i do
15                     t ← parent_b[t]
                   // if (a,t) was not processed ...
16                 if sourceVisited[t] ≠ sourceMarker then
17                     sourceVisited[v] ← sourceMarker
18                     v ← parent_b[t]
                       // ...  traverse again, update vertex counter
19                     for v ∉ C̄_i do
20                         distinctSubpaths[v]++
21                         if sourceVisited[v] ≠ sourceMarker then
22                             distinctSource[v]++
23                             sourceVisited[v] ← sourceMarker
24                         if not targetVisited[v][t] then
25                             distinctTarget[v]++
26                             targetVisited[v][t] ← true
27                         v ← parent_b[v]
       // find vertex with max edge reduction (loop)
28     edgeRed ← max v ∈ C_i: distinctSubpaths[v] − distinctSource[v] − distinctTarget[v]
       // if the best vertex reduces edges, set it preserved, stop else
29     if edgeRed > 0 then
30         C̄_i ← C̄_i ∪ {v}
31     else
32         nextRound ← false
```

4.4 Query and Edge Unpacking

Shortest path queries are almost straightforward: We build the search graph in internal memory during the query and load cells, from external memory, whenever a required vertex is not yet in the search graph.

Building the search graph is done as explained in Section 4.2.2: Every vertex has a local index in the search graph, but the local index is initially unknown for target vertices of cut edges. Therefore, we manage a hash map to map global ids to the local index for all boundary vertices. The mapping is updated for new boundary vertices whenever a cell is loaded.

We run unidirectional Multilevel Dijkstra starting at a vertex s and stop the search as soon as the target t was scanned. For given s, t, the corresponding level-1 cells from the graph must be initially loaded. The cells are located in external memory using the vertex map of the graph data structure (see Section 4.2.2). The cell of t is required from the start as we need the cell id $\text{cell}_\ell(\cdot)$ on each level ℓ for both s and t to calculate the highest uncommon level $\text{uncommonLevel}(v)$ for vertices v in the search graph. For details on Multilevel Dijkstra recall Section 3.3.

Loading cells into the search graph takes place when cut edges are relaxed. The local index of the target vertex v must be resolved in case it is unknown. Cut edges store the global id and the cell id $\text{cell}_\ell(v)$ for each level ℓ of the target vertex. The cell required for v is the cell on the highest uncommon level $\ell = \text{uncommonLevel}(v)$. We load the cell if it is not in the search graph yet. Afterwards the local index of v can be resolved with the hash map. We then store the index at the cut edge to avoid querying the hash map multiple times.

Naturally, the search graph is larger for distant s and t. However, final size of the search graph not only depends on the number of cells that were required. The size also depends on the overlay representation used, as their cells have a different number of vertices (at least all boundary vertices) and edges per cell. Requirements on internal memory increase for larger search graphs – we will evaluate that effect in our experimental evaluation in Chapter 5.

We use a binary heap as priority queue. Vertex labels hold the distances $\mathsf{d}[\cdot]$ and parent pointers $\mathsf{parent}[\cdot]$ for each vertex. These data structures can be implemented as for any static adjacency array graph due to the local vertex indexes: the local indexes run from 0 to N (N increases during the query). Data fields can be addressed directly by the index.

4.4.1 Edge Unpacking

The shortest path $p_{s,t}$ found by the query contains shortcut edges from the overlay. If not only the distance but a full path description is required these edge must be unpacked. We consider two different variants of edge unpacking: *Recursive* unpacking for the overlay types *clique, reduced cliques* and *skeleton* overlays. It was proposed for unpacking in the original variant of CRP [1, 2]. The alternative is *direct* edge unpacking on the graph. It is possible when boundary vertices of level 1 are *preserverd* on upper levels.

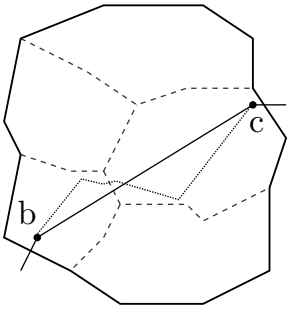


Figure 4.8: Unpacking of a cell using the subcells.

Recursive Edge Unpacking. The shortest path $p_{s,t}$ found by the query contains subpaths $p_{b,c}$ between boundary vertices b, c of a customization cell $\overline{C}_i$. As internal vertices of that cell were not (all) preserved, we run an unpacking query on its subcells to recover the vertices that are missing on that level. The subpath

$p_{b,c}$ must be replaced by the more detailed path $p'_{b,c}$ found by the unpacking query. If the cell is a level-1 cell, the query must be run on the original cell to recover all original vertices.

This is done recursively, top down from the highest level to the cells of the original graph, as unpacking a cell only restores the boundary vertices of the lower level. We unpack all cells on a fixed level first before we start unpacking the lower level.

Delling et al. use a shortcut cache to speed up recursive unpacking in their server implementation of CRP [2]. However, in a mobile scenario edge unpacking should be fast from the beginning, when the cache could not yet be initialized. Therefore, we propose direct unpacking to avoid additional block reads from the overlay data.

Direct Edge Unpacking. The overlay variants defined in Section 4.3.3.3 preserve boundary vertices of level-1 on all higher levels if necessary. Hence, shortcut edges can be unpacked directly using the graph cells. However, the internal vertices' cell and vertex ids have not been preserved. We do not need them as the distances suffice to find the correct path. We unpack level-1 cells in the order they appear on the shortest path: the global id and cell of the entry boundary vertex b is then available at the cut edges. To find the entry boundary vertex d of the next level-1 cell we compare their distance found by a local query to the distances in $p_{s,t}$ we already know – in most cased the distance yields a distinct solution. On rare occasions the distance of multiple boundary vertices in adjacent cells equals (see Figure 4.9). In these cases the search must recurse into the adjacent cells until all but one search terminate, as the distances do not match. Our experimental evaluation in Chapter 5 shows that these recursions are rare. The number of cells loaded from the graph increases only marginally (in fact, the observed depth of recursion does not exceed 1 cell in our tests).

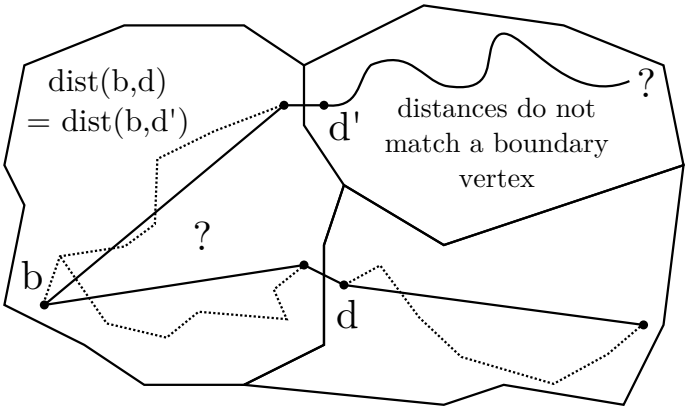


Figure 4.9: Direct edge unpacking: on rare occasions the search must recurse into multiple adjacent cells as no distinct solution was found.

5. Experimental Evaluation

In this chapter, we present an in-depth evaluation of External Memory Customizable Route Planning in all variants we have have proposed in the previous chapter. *Metric customization* and *shortest path queries* in an external memory scenario build the core of our work. We rate all possible options, which include different block and cache sizes, buffered or unbuffered I/O, and the different overlay types we have proposed. *Cliques*, *reduced cliques* and *skeletons* with *recursive* edge unpacking and *reduced cliques*, *skeletons* with *preserved boundary vertices* on higher levels and *direct* unpacking.

As graph instance we use the European road network made available by PTV AG for the *9th Dimacs Implementation Challenge* [25]. We use it along with a partition kindly provided by Microsoft Research and computed by PUNCH [22]. Statistics on the graph instance, our external graph data structure and the partition it includes are listed in Tables 5.1 and 5.2.

Table 5.1: Graph instance and the resulting size of our external graph data structure.

instance	vertices	edges (bidirectional)	vertex map [MB]	cell map [MB]	cells [MB]	total [MB]
Europe (EU)	18 010 173	44 435 092	68.70	0.08	200.56	269.36

Table 5.2: Multilevel partition of the graph.

	level 1	level 2	level 3
cell size limit U	2^{10}	2^{15}	2^{20}
cells	20 481	623	20
boundary vertices	328 626	34 782	2 753
cut edges	360 207	37 736	2 990

We build the external graph data structure as described in Section 4.2. We reorder the vertices and cells in a first step and store the graph in form of serialized level-1 cells. We include the metrics *travel time* in seconds and *distance* in meters. The distance is computed from vertex coordinates as it was not included in the input graph data. Real world applications would include the distance together with a road type and various restrictions (e. g., speed limits) to derive multiple metrics at run time. Table 5.1 shows the stored size

of our graph. The vertex and cell map are required to access the cells efficiently – the actual graph data including both metrics is contained in the 'cells' section requiring 200.56 MB. The average cell size is just under 10KB (including 864 vertices and 2132 edges on average). The entire data structure is contained in a single file – no additional data is required to run the customization for one of the metrics.
The original partition contains 4 levels with cell sizes limited to 2^5, 2^{10}, 2^{15} and 2^{20} vertices. We do not use the lowest level as we currently can not take advantage of cells with a maximum of 32 vertices. The cells would take 241.6 MB instead of 200.56 MB with the additional level included. The resulting increase in data generated per metric by the customization is more significant (refer to Section 5.4 for more details).

To simulate a realistic scenario, we use a regular smart phone for most of our experiments. It features a dual core ARMv7 processor with 1 Ghz clock speed (Mediatek MTK5677) and 512 MB RAM. The operating system is Android 4.0.4. We implemented external memory CRP with C++11 and compile it with the GNU Compiler Collection in Version 4.7.2 (Optimization Level 3). The application is statically linked against the C and C++ standard libraries glibc/gnu-libstdc++ and pthreads to parallelize the customization. We further use the `time` command from the GNU Project to measure the *maximum resident size* for queries and customization – the maximum amount of physical memory used by an application during its runtime.

We read and write external memory data from a SanDisk 32 GB Class 10 micro SDHC Card. The manufacturer promises sequential read speed up to 30 MB/s, but we measure the actual characteristics under various conditions in Section 5.2. These characteristics are essential for a proper rating of our algorithms performance.

5.1 Methodology

For both queries (optional with edge unpacking) and customization experiments we are using an internal block cache that can hold a fixed number of memory blocks of fixed size from the external graph or customization data. The cache uses a Least Recently Used replacement strategy. We are only using block aligned reads/writes of a fixed size, to load data from the external files (graph and metric-dependent data) into the cache. We count a block access as block read if it could not be accessed from the cache.
The customization reads the entire graph and reads/writes the overlay levels in sequential fashion. Hence, we are not providing explicit I/O statistics for the customization, but we make measurements to determine the overall impact on run times. Unless otherwise stated we run 1 000 random queries or 10 customization rounds and average the results. The customization uses both cpu cores of our device but queries are single-threaded.
To obtain accurate measurements of I/O despite buffered file access, we flush the system cache before each customization or query run. To verify the correctness of our implementation, we compared the distances of 10 000 random queries against Dijkstra's Algorithm on the original graph data for each overlay type.

We distinguish different types of queries, similar to the query types used for Mobile CH [11]. Whenever we compare different configurations (e. g., *cliques* vs. *skeletons*) we run the same set of queries for each.

'Cold' queries start with an initially empty cache (we clear the cache before each run). *Cold* queries are representative for expected run times and block reads immediately after the program was started.

'Warm' queries run on an initialized cache. We initialize the cache with 1 000 queries in a first step and measure run times and block reads of a different set of 1 000 random queries in a second step.

'No I/O' queries are used to measure the performance if all data is loaded from the internal cache. Therefore, we run each query twice. The first query initialized the cache with all relevant blocks. We repeat the query and measure the run time. The results indicate the impact of I/O on run times for *cold* queries.

'Static' queries give indication on processing time for the actual query without the overhead required for our external memory implementation (e. g. cell deserialization). We run a query to build the search graph first (see Section 4.4). Then the same query on the initialized search graph provides results comparable to the performance of Dijkstra on a static graph.

Measurements for queries include the reinitialization of all required data structures (e. g., search graph, vertex labels and queries) but not the time to flush caches as it is only done for the purpose of accurate measurements. We do not measure initialization times for the complete application, e. g., allocating memory for the cache or opening graph and customization data files, as we do not regard it as part of the actual query. On application start a one block header is read for both graph and customization data. It contains information about the data structure layout and basic information about the graph instance (e. g., number of vertices).

The performance of edge unpacking is always measured along with shortest path queries to obtain a benchmark of the overall performance. Calculating the distance is not sufficient for the most realistic use cases on mobile devices.

5.2 External Memory Characteristics

We compare the read performance of buffered and unbuffered block reads for random and sequential pattern in Figure 5.1. Buffered file access uses intermediate buffers of the operating system: the operating system tries to recognize random or sequential access patterns and prefetches data (in larger blocks) to increase the throughput. The prefetching of data can happen asynchronously in the background and allows other computations to be done in the meanwhile. For information on the underlying algorithms refer to [23]. Data that was read once might remain in the system cache to speed up future reads. Unbuffered access reduces the overhead by omitting the operating system buffers. Low level system APIs must be used for unbuffered access as common file interfaces like <fstream> for C++ or <stdio.h> from the C library treat files as data streams and use buffered access internally. For better comparison, we use the low level APIs for buffered file access as well as for unbuffered access.

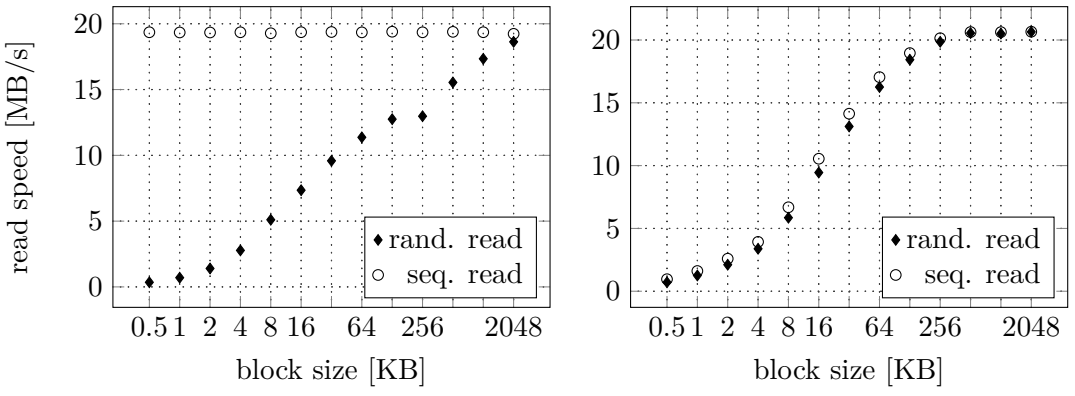


Figure 5.1: Buffered block read (left) vs. unbuffered block read (right).

For our measurements we created a 2 GB file and read a total of 64 MB in either random or sequential fashion for different block sizes. The results show that the maximum throughput of 20 MB/s (not the advertised 30 MB/s) is only reached for large block sizes or sequential access patterns with buffered reads. Therefore, buffered access might have a significant advantage for sequential reads. The speed difference for random and sequential reads is negligible for unbuffered access, but results from [18] show that this does not hold for write access due to the internals of flash devices. Our measured sequential write speed is 7.4 MB/s.

block size [KB]	0.5	1	2	4	8	16	32	64
buffered [ms]	1.42	1.42	1.43	1.43	1.55	2.13	3.22	5.52
unbuffered [ms]	0.73	0.79	0.94	1.20	1.36	1.68	2.40	3.86

Table 5.3: Latency for random block reads.

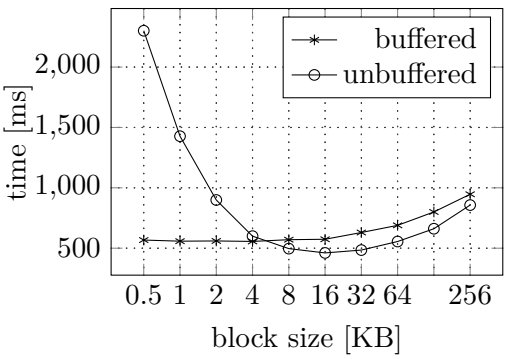
Table 5.3 shows, that latency for single block reads is smaller for unbuffered access. This might lead to reduced run times for random access patterns. As the throughput increases, it might pay to read larger blocks if the locality of the data is good enough. The latency for blocks up to 4KB equals for buffered reads. It seems to bee the minimal access entity for buffered access. We examine the impact of block size and access method on run times and the amount of data read in the following section.

5.3 Impact of Block and Cache Size

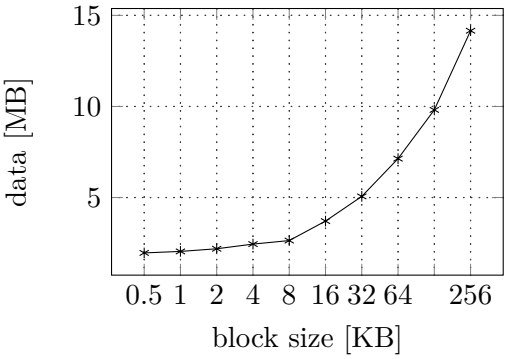
Before examing all customization types in detail, we determine how different *block sizes* with *buffered* or *unbuffered* access and different *cache sizes* affect the customization and query. Recall from Section 4.2.2 that we do not align cells to block boundaries, as their size is given by the partition. The size of our external graph and the metric-dependent overlay data remains unaffected for different block sizes.

Customization. The customization processes the cells bottom up and completes the lowest levels first – due to our reordering of cells (Section 4.2.1) it can read and write data sequentially. It takes full advantage of sequential read and write speeds using buffered access. We are not noticing any changes in run times for different block sizes. Unbuffered file access is unsuited for the customization as sequential speed is not reached without using very large block sizes – but large block sizes result in long periods of blocking due to I/O and reduce the performance gain through parallelization (recall Section 4.3.1). Due to the sequential processing, the customization does not profit from large cache sizes. We set the cache size to 2 MB for all our experiments.

Block size for queries. Figure 5.2 (a) shows the impact of block sizes on run times for queries with unpacking, using a *skeleton* overlay and *travel times*. The best run times are achieved by 16 KB blocks and unbuffered file access, but the slow down for buffered reads is only small. Buffered reads perform better for small block sizes, as unbuffered reads can not take advantage of the locality in the data (note that block sizes below 4 KB yield equal run times for buffered reads as the the system always fetches a 4 KB block – subsequent reads within the same block hit the operating system cache).
In the skeleton overlay we use for the measurements, cells have an averaged size of only 700 bytes (averaged over all levels) – but it still pays to read larger blocks: Our reordering of cells guarantees that cells, that have a common super cell, are contiguously stored in memory. Due to the characteristics of the Multilevel Dijkstra search graph, these cells are likely to be required together: blocks containing parts of two contiguous cells can be



(a) Runtime for queries with unpacking

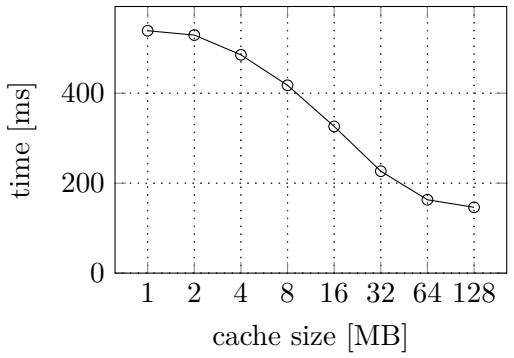


(b) Read data for queries with unpacking

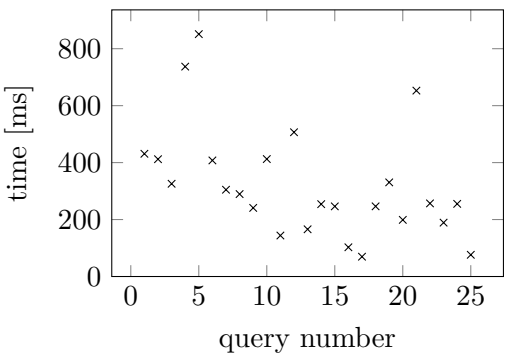
Figure 5.2: Impact of the block size on run times and the the amount of data read for *cold* queries with unpacking. Customization type is *skeleton*, unpacking is recursive. The results are averaged for 500 random queries. The cache size was set to 2 MB.

accessed from cache when the second cell is required. The average size of 10 KB for graph cells might also be a dominant factor for the optimum of 16 KB blocks, but 16 KB blocks yield the best performance for plain queries with the skeleton overlay as well.

The overall amount of data read increases for larger block sizes (Figure 5.2 b), as it is more likely to read data, that is irrelevant for the computation. The effect is less pronounced for small block sizes, as the locality of the data is high enough. The amount of data read is minimal for 512 Byte blocks (345.6 KB for the plain query, 2008.8 KB for the query with unpacking). Although the data doubles for 16 KB blocks (563.1 KB for the query and 3797.3 KB with unpacking), the run time is optimal, as the decrease in locality is lower, than the increase in I/O throughput. It shows that, minimizing the amount of read data does not always yield optimal solutions for external memory applications.



(a) Warm queries for different cache sizes.



(b) Run times for 25 queries on initially empty cache.

Figure 5.3: Impact of cache size on *warm* queries with unpacking. For the left plot, the cache was initialized with 1 000 queries, then the run times were are averaged for 1 000 additional queries. The right plot shows query times for 25 random queries, starting with an initially empty cache of 64 MB. Reads are buffered and the block size was set to 4 KB.

Unfortunately 16 KB blocks are not optimal in all cases, e. g., queries using a *clique* overlay yield a significant better performance for buffered access (277 ms buffered vs. 355 ms unbuffered). The reason is that clique overlay cells take significant more space and the prefetching algorithms of the operating system can speed up sequential reads.

We restrict ourselves to block sizes of 4 KB and buffered file access for further experiments, as the optimum block size for unbuffered reads highly depends on the specific case. 4 KB buffered reads might not offer optimal performance in all cases, but the performance is on a high level for any case. Future work might consider the use of adaptive block sizes, to exploit the locality in an optimal way.

Cache size for queries. The block cache holds recently used blocks in internal memory. In case a new block is read, the least recently used block is replaced. Figure 5.3 (a) shows the impact of different cache sizes for *warm* queries with unpacking. Naturally, the query times and the amount of data read decreases for larger cache sizes. In general, a larger cache reduces run times but caching is a trade-off between the main memory requirements of an application and performance. If the performance of cold queries is sufficient, a small cache is enough to hold the data that is required for a single query.
The right plot (Figure 5.3 b) indicates that even a small number of queries that fill the block cache yield a speedup, as some particular important parts of the data (e. g., the cell maps or the highest level of the overlay) are immediately available for all following queries.

For further experiments, we're using a default cache size of 64 MB for *warm* and 2 MB for *cold* queries.

5.4 Customization

We compare the overlay types from Section 4.3.3 for the customization of *travel time* and *distance* metrics. Summarized results are given in Table 5.4 and per level statistics are listed in Table 5.5. The time required to build the overlay ranges from just under 2 minutes for *clique* variants to 4 minutes for *skeletons* with travel time metric and almost 6 min for *skeletons* with distance metric.

We provide the total number of vertices and edges per level for each overlay type. The *full* overlay is included as indication for the hierarchical properties of the metrics. It preserves all vertices on shortest paths boundary vertices. Only 328 995 vertices and 531 448 edges are relevant for travel times on the highest level. The hierarchy is less pronounced for distances, as 1 439 840 vertices and 2 490 031 edges remain.

Table 5.4: Customization total run times, stored size of the overlay and maximum resident set size during the customization

	Time			Distance		
Type	time [s]	space [MB]	RSS [MB]	time [s]	space [MB]	RSS [MB]
full	–	87.33	1255.8	–	169.7	1732.5
clique	107.9	43.6	56.9	109.4	43.6	58.7
red-clique	110.0	39.0	48.1	104.4	42.4	48.3
skeleton	228.2	14.8	42.0	326.1	22.5	45.1
red-clique-pres	102.8	30.5	81.5	106.7	36.9	175.7
skeleton-pres	137.2	16.3	81.6	156.4	26.0	142.1

Reduced cliques take only slightly more time to compute than *cliques*, as the overlay contains about one third fewer edges on all levels. This indicates, that the overhead of Algorithm 4.2, which is used to find the required edges for all overlay types, except for cliques, is small. The stored size for cliques is 43.6 MB equally for both metrics as the topology is metric-independent. Clique edges are stored as matrices. Due to the increased overhead for adjacency graphs over matrices, reduced cliques can not significantly reduce the stored size.

Skeletons reduce the size of the overlay significantly. Only 2 425 additional vertices (compared to cliques) and 23 682 total edges are needed to preserve distances on the highest level for travel times. The number of additional vertices per level indicates the total number of iterations that were needed to find the the vertices for the skeleton graphs (Algorithm 4.3). The algorithm greedily preserves internal vertices and terminates when no further vertex can be preserved to reduce the number of edges. Skeletons exploit the hierarchy of metrics, but even for artificial metrics that do not exhibit any hierarchy at all the resulting overlay would never be worse than for reduces cliques. More vertices must be preserved for distances, but the resulting number of edges is still very low compared to cliques. The run time increases disproportionally on higher levels but the overall run times of 4 minutes for travel time and 6 minutes for distances are still acceptable. The stored size reduces to only 14.8 MB or 22.5 MB.

Preserving the boundary vertices of level-1 on all higher levels (*red-clique-pres* and *skeleton-pres*) yields fast computation times when cliques are used on the lowest level. Skeletons on level 1 have a slightly increased run time, but the stored size of the overlay remains almost as small as for skeletons on all levels. The number of boundary vertices that remain on the highest level is significantly larger for the distance metrics as the hierarchy is less pronounced compared to travel times. That number might further increase for metrics that

Table 5.5: Customization: overlay graph vertices, edges and the runtime required per level.

		Time			Distance		
Type	level	vertices	edges	[s]	vertices	edges	[s]
full	1	5 167 502	10 201 986	–	7 202 699	14 791 407	–
	2	1 916 286	3 513 944	–	4 194 156	7 942 225	–
	3	328 995	531 448	–	1 439 840	2 490 031	–
clique	1	328 626	6 999 558	73.1	328 626	6 999 558	73.1
	2	34 782	2 560 436	18.8	34 782	2 560 436	20.5
	3	2 753	594 247	15.9	2 753	594 247	15.8
red-clique	1	328 626	4 222 053	83.5	328 626	4 473 322	77.8
	2	34 782	1 694 671	14.7	34 782	1 703 675	15.0
	3	2 753	395 801	11.8	2 753	378 336	11.6
skeleton	1	473 315	1 874 653	120.1	483 066	2 550 713	127.8
	2	60 378	275 494	49.1	66 029	460 436	82.4
	3	5 178	23 682	59.0	5 767	43 770	115.8
red-clique-pres	1	328 626	4 222 053	80.9	328 626	4 473 322	76.8
	2	127 997	520 144	13.4	228 996	904 667	13.9
	3	28 355	63 647	8.6	90 783	205 628	16.0
skeleton-pres	1	473 315	1 874 653	118.5	483 066	2 550 713	127.7
	2	128 011	520 076	10.1	229 013	904 677	12.6
	3	28 353	63 644	8.6	90 789	205 628	16.2

are not well-behaved like travel times or distances. In the worst case all boundary vertices of level-1 are preserved on the highest level.

For each overlay type, we measure the maximum resident set that was required to build the customization. It is a indicator for the practicability in a real environment, as main memory is a limited resource on mobile devices. Most of the customization data structures exist per thread – the memory usage is therefore only representative for two threads. A *full* overlay cannot be computed on our device as memory usage is to high. We made the measurements on a desktop computer. Recall from Section 4.3.2 that we need $|B_i| \cdot |C_i|$ (boundary vertices times cell vertices) vertex labels to store parent pointers and distances for the customization queries. Although only 8 Byte per label are required, the memory consumption is too high if many vertices are preserved on the higher levels. The same effect is significant for *red-clique-pres* with preserved boundary vertices: 175.7 MB for the distance metric is within the limits of our device but the memory consumption might be too high if considered that other mobile applications also require some of the total memory in a multitasking environment. However, the memory requirements for the other overlays types should be no be problem for mobile devices.

To measure the impact of I/O operations on run times, we run the customization from the same SD Card, that we use with the smart phone, on a desktop computer. It features a quad core AMD processor with 3.2 GHz clock speed. The times to build reduced cliques for travel times are 14.6 s, or 7.9 s if we use a ram disk instead of the SD Card. The difference is basically the time required to sequentially read the graph and to write the overlay data. It shows that the customization on mobile devices is not limited by I/O although the entire graph must be processed.

Building the overlay on the lowest level takes the most time for all types. Cells for the lowest level are built from the graph cells – higher level cells are built from the overlay cells of the lower level. Results from Delling et al. [1, 2] show, that the cost for the lowest level can be reduced when a *shadow level* with smaller cell sizes is used (e. g., the level with up to 2^5 vertices per cell we do not use). The shadow level is then only used to speed up the customization but not included as overlay for the queries.
If we include the additional level in our current implementation the customization time increases from 110 s to 159 s for reduced cliques and travel times, probably due increased I/O. The overlay size grows from 39 MB to 169 MB (even 141 MB for skeletons as the number of edges can not be significantly reduced on the lowest level). The slowdown of queries is marginal, despite the increased size. Only a small part from the lowest overlay level is required for each query.

Further optimizations to the customization on a CPU instruction level were proposed in [9]. They are less suited for mobile applications as they typically must be supported on various CPU architectures (ARM CPUs do not support the required instructions at all).

5.5 Queries and Edge Unpacking

The last section showed, that building a metric-dependent overlay is a matter of minutes on mobile devices. But in the end, the fast computation of shortest paths is most important for the usability of Customizable Route Planning in a real-world mobile application. The computation of shortest paths must be fast from a end user perspective, e. g., fast enough to start car navigation immediately after selecting the destination. For details on our query and edge unpacking refer to Section 4.4. As for the customization, we compare the *travel time* and *distance* metric.

Queries without edge unpacking may seem unrealistic in a mobile scenario at first glance, nevertheless some use cases exist. The Multilevel Dijkstra search graph (Section 3.3) contains the original cell for the source and target vertex. Hence, the found path contains all details to the point where the start cell is left. First driving directions can be generated immediately. Further, no edge unpacking is required to find the minimum distance for nearby points of interest.

Table 5.6 lists the query results for all overlay types. Queries using a *skeleton* overlay yield the best run times for *cold* queries: 89.5 ms for travel times and 107.7 ms for the distance metric. Queries using *cliques* and *reduced cliques* show only a small difference to each other: the run times for cold queries are just under 300ms. As cliques and reduced cliques do not exploit the hierarchy of metrics, their results show no significant difference for distances and travel times.
The performance gap between the distance metric and travel times is more significant for *clique-reduced-pres* and *skeleton-pres*. The performance is good for travel times but can not compare to the skeleton overlay when distances are used. The difference between a skeleton or a reduced clique overlay on the lowest level is small – cliques on the lowest level should be preferred as the customization is faster.

In general, the run times are reflected in the number of block reads. Queries using skeletons require the fewest blocks for travel times (99.1 blocks / 396.4 KB) and distances (151.9 / 607 KB). About six times more data is required for cliques with travel times, but the run time does not increase proportionally as the I/O throughput grows when larger cells are read.

The *warm* query requires approximately 10 blocks reads for all overlay types – the result is obvious as each overlay fits into the 64 MB cache and only two graph cells are required per query. The speedup for queries which would not require any I/O is small. It strikes that the run time is still high compared to the same queries on a *static* graph. The overhead required for the external memory implementation (e. g., cell deserialization) dominates the static processing.

The number of vertex scans is close to the number of vertices in the search graph for all overlay types, as our search graph grows dynamically when cells are required. The search

Table 5.6: Query statistics: number of vertex scans, average search graph vertices and edges, run times and read blocks.

					cold		warm		no I/O	static
Type	metric	vertex scans	graph vert.	graph edges	time [ms]	read blocks	time [ms]	read blocks	time [ms]	time [ms]
clique	time	**6 317**	**7 477**	665 953	285.9	688.9	110.8	11.3	102.5	35.1
	dist	**6 285**	**7 325**	648 719	272.4	671.7	108.2	11.3	99.8	34.6
clique-red	time	**6 317**	**7 477**	443 963	270.0	683.3	92.0	10.1	84.0	25.1
	dist	**6 285**	**7 325**	421 951	260.8	654.1	89.1	10.9	80.5	24.5
skeleton	time	9 997	11 657	**48 623**	**89.5**	**99.1**	**31.4**	**7.8**	**24.0**	**7.8**
	dist	10 707	12 315	**78 266**	**107.7**	**151.9**	**36.9**	**8.2**	**29.6**	**10.1**
clique-red-pres	time	26 396	31 935	101 641	133.0	168.5	53.0	8.9	45.1	20.4
	dist	68 447	81 858	228 041	225.2	379.9	107.3	9.9	100.6	55.6
skeleton-pres	time	26 762	32 338	95 021	119.5	157.9	50.6	**7.8**	43.8	20.3
	dist	68 828	82 269	222 912	220.2	371.7	105.9	8.4	100.0	56.3

graph for skeleton overlays has significant less edges than for cliques, still the number of additional vertices is low. It shows in the *static* run time for skeletons which is significant lower than for all other overlay types.

Edge Unpacking. We evaluate two different variants of edge unpacking to retrieve a full path description: the common approach is *recursive* unpacking, as it is used by Delling et al. in their implementation of CRP [2]. As a second variant we proposed *direct* unpacking on the graph for the overlay types *clique-red-pres* and *skeleton-red-pres* in Section 4.4.1. It allows unpacking of paths without recursion on the overlay, that requires only the original cells of the graph. Hence, no additional block reads for the overlay are required for *cold* queries.

Results for queries with unpacking are listed in Table 5.7. For *cold* queries, the variants with direct edge unpacking (*clique-red-pres* and *clique-red-pres*) yield the best run times: about 450 ms for travel times and 550 ms with the distance metric are required for a query with subsequent unpacking. The performance gap between both metrics is the result of different query times: compared to Table 5.6, it shows that the overhead for unpacking is almost equal for travel times and distances. *Cold* queries with unpacking using a *clique* or *reduced clique* overlay require significant more block reads, but the run times just under 1 s might still be fast enough for real-world applications. Due to the compact overlay, the *cold* run times with a skeleton overlay with about 600 ms for both metrics are still significant lower.

The advantage for direct unpacking reduces for *warm* or *no I/O* queries. The recursions on the overlay are less expensive when the corresponding blocks are available from the cache. Hence, the skeleton overlay does not perform significant worse if the cache is initialized. Due to the stronger hierarchy of travel times, the number of required graph cells to unpack edges of the lowest overlay level is slightly higher for the distance metric (93.3 vs. 105.7). Note that the number of graph cells, that are required for direct edge unpacking increases only marginally. The slight increase is caused by recursions, that are required when the

Table 5.7: Queries with edge unpacking: recursive edge unpacking is used for clique, clique-red and skeletons. Edges are unpacked directly with the graph cells using clique-red-pres and skeleton-pres overlays. The column 'graph cells' shows the number of cells that were required to unpack edges on the graph. The maximum resident set size was measured for 1 000 queries including 2 MB cache.

			cold		warm		no I/O	
Type	metric	graph cells	time [ms]	read blocks	time [ms]	read blocks	time [ms]	RSS [MB]
clique	time	93.9	946.6	1 600.0	339.4	99.0	271.1	32.6
RECURSIVE	distance	105.7	883.1	1 511.3	455.1	265.7	285.0	31.9
clique-red	time	93.9	896.6	1 550.6	298.4	83.9	238.9	22.6
RECURSIVE	distance	105.7	871.7	1 488.6	425.7	259.9	255.9	21.6
skeleton	time	93.8	566.1	613.5	170.1	36.9	144.3	**6.5**
RECURSIVE	distance	105.7	617.6	731.0	304.9	183.5	**179.5**	**8.0**
clique-red-pres	time	94.1	454.3	507.9	175.5	48.7	141.1	11.7
DIRECT	distance	106.0	551.7	736.3	319.5	167.5	209.8	18.4
skeleton-pres	time	94.1	**440.4**	**497.2**	**166.1**	**37.1**	**138.8**	10.8
DIRECT	distance	106.0	**546.9**	**728.1**	**313.2**	**156.4**	208.8	17.5

computed distances of cell internal vertices are not sufficient to find a distinct solution. However, these recursions are very rare and have no significant impact on the overall cost.

The hierarchical properties of the metrics show in the cache efficiency: although the overlay size is equal for cliques, the unpacking requires significant less blocks for *warm* travel time queries. Hence, the run times are faster, although the overlay is metric-independent. The graph cells, that are crossed by shortest paths seem to be more random for the distances metric than for travel times.

As for the customization, we measured the maximum resident set size that was required during 1 000 queries with edge unpacking, to test if our solution is suitable for real-world applications. The values include a 2 MB cache (as for *cold* queries). If the performance of *warm* queries is required the memory consumption would increase accordingly. The results in Table 5.7 show, that the memory consumption is not significant above 32 MB for all variants – the difference between the overlay types has its main reason in the different sizes of the query search graph (see Table 5.6), which is dynamically built in internal memory. Edge unpacking has only little effect on the total memory consumption as only the cells, that are required to unpack edges in the super cell must be hold in internal memory simultaneously.

Overall, it becomes apparent that unpacking on the graph is the most expensive part of edge unpacking. However, it is still fast enough for practical usage in a mobile application. In a real-world application the graph might need more space to store the required information for various metrics (e. g., speed limits or road types). To further stress the practicability of unpacking, we increased the the stored size for each edge in the graph by 8 bytes: the total size for the serialized cells of the external graph increases from a 205.56 MB to 540.37 MB. Query times without edge unpacking remains largely unaffected as only 2 cells are required per query. To retrieve a full path description with a *clique-red-preserve* overlay, the time increases from 454.3 ms (our default configuration) to 939.2 ms with larger cells. We believe, that run times just under 1 s for a query with edge unpacking, are still acceptable for most mobile applications.

6. Conclusion

We developed a proof-of-concept application, that makes use of external memory, to adapt the core features of Customizable Route Planning for a mobile device: it enables efficient *metric customization* in only few minutes, and allows subsequent *shortest path queries* in less than 150 ms on a European road network for a distance or travel time metric. Retrieving a complete path description is possible in less than 600 ms.

Therefore, we engineered an external graph data structure, that includes the nested multilevel partition required for Customizable Route Planning, and stores the graph as serialized, independent cells. To increase the locality for query and customization, we reordered partition cells with respect to the multilevel partition, such that cells, that build a supercell on the next higher level, follow each other and occupy a continuous block in external memory.

The key to fast metric customization in external memory is the completely independent processing of cells: only a small fraction of the graph instance must be hold in internal memory simultaneously to generate a metric-dependent overlay. Our reordering of partition cells, allows fast sequential read and write patterns on external memory, that do not limit the processing speed of mobile devices.
As query times are dominated by I/O, we compared different overlay types, and implemented a variant of skeleton graphs, that reduces the overlay data to only 14.8 MB for travel times. Hence, we were able to reduce the on average required data to less than 400 KB for random queries.
Further, we made sure, that the main memory consumption does not exceed reasonable limits for mobile applications. In fact, our algorithms can be used to run the customization or queries with less than 64 MB main memory.

Future Work. Further optimizations, that exceed the scope of this work, are waiting for future work: the amount of necessary data and the run time for the queries could be further reduced by using bidirectional search. Another possible optimization for future work is the implementation of a shadow level, to make the customization faster.

For our work, we only considered a travel time and a distance metric. However, the results suggest, that our proposed methods can be used to implement Customizable Route Planning in a real-world mobile, stand-alone application, that supports efficient metric-customization and shortest path queries for *arbitrary metrics*. Such metrics can be user defined metrics, metrics that adapt to user's driving preferences or metrics that include dynamic traffic information.

Bibliography

[1] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Customizable Route Planning," in *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)* (P. M. Pardalos and S. Rebennack, eds.), vol. 6630 of *Lecture Notes in Computer Science*, pp. 376–387, Springer, 2011.

[2] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Customizable Route Planning in Road Networks." Submitted for publication, 2013.

[3] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[4] A. V. Goldberg, "A Practical Shortest Path Algorithm with Linear Expected Time," *SIAM Journal on Computing*, vol. 37, pp. 1637–1655, 2008.

[5] A. V. Goldberg and C. Harrelson, "Computing the Shortest Path: A* Search Meets Graph Theory," in *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pp. 156–165, SIAM, 2005.

[6] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling, "Fast Point-to-Point Shortest Path Computations with Arc-Flags," in Demetrescu *et al.* [25], pp. 41–72.

[7] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact Routing in Large Road Networks Using Contraction Hierarchies," *Transportation Science*, vol. 46, pp. 388–404, August 2012.

[8] D. Delling, P. Sanders, D. Schultes, and D. Wagner, "Engineering Route Planning Algorithms," in *Algorithmics of Large and Complex Networks* (J. Lerner, D. Wagner, and K. A. Zweig, eds.), vol. 5515 of *Lecture Notes in Computer Science*, pp. 117–139, Springer, 2009.

[9] D. Delling and R. F. Werneck, "Faster Customization of Road Networks," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, vol. 7933 of *Lecture Notes in Computer Science*, pp. 30–42, Springer, 2013.

[10] A. V. Goldberg and R. F. Werneck, "Computing Point-to-Point Shortest Paths from External Memory," in *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pp. 26–40, SIAM, 2005.

[11] P. Sanders, D. Schultes, and C. Vetter, "Mobile Route Planning," in *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, vol. 5193 of *Lecture Notes in Computer Science*, pp. 732–743, Springer, September 2008.

[12] C. Vetter, "Fast and Exact Mobile Navigation with OpenStreetMap Data," Master's thesis, Karlsruhe Institute of Technology, 2010.

[13] D. Delling, M. Kobitzsch, D. Luxen, and R. F. Werneck, "Robust Mobile Route Planning with Limited Connectivity," in *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pp. 150–159, SIAM, 2012.

[14] D. Hutchinson, A. Maheshwari, and N. Zeh, "An external memory data structure for shortest path queries (extended abstract)," in *Computing and Combinatorics* (T. Asano, H. Imai, D. Lee, S.-i. Nakano, and T. Tokuyama, eds.), vol. 1627 of *Lecture Notes in Computer Science*, pp. 51–60, Springer Berlin Heidelberg, 1999.

[15] L. Arge and L. Toma, "External data structures for shortest path queries on planar digraphs," in *Algorithms and Computation* (X. Deng and D.-Z. Du, eds.), vol. 3827 of *Lecture Notes in Computer Science*, pp. 328–338, Springer Berlin Heidelberg, 2005.

[16] U. Meyer and V. Osipov, "Design and implementation of a practical i/o-efficient shortest paths algorithm," in *ALENEX*, pp. 85–96, 2009.

[17] D. K. Blandford, G. E. Blelloch, and I. A. Kash, "An Experimental Analysis of a Compact Graph Representation," in *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pp. 49–61, SIAM, 2004.

[18] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo, "Characterizing the performance of flash memory storage devices and its impact on algorithm design," in *Experimental Algorithms* (C. McGeoch, ed.), vol. 5038 of *Lecture Notes in Computer Science*, pp. 208–219, Springer Berlin Heidelberg, 2008.

[19] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," *Trans. Storage*, vol. 8, pp. 14:1–14:25, December 2012.

[20] K. Mehlhorn and P. Sanders, *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[21] M. Holzer, F. Schulz, and D. Wagner, "Engineering Multilevel Overlay Graphs for Shortest-Path Queries," *ACM Journal of Experimental Algorithmics*, vol. 13, pp. 1–26, December 2008.

[22] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck, "Graph Partitioning with Natural Cuts," in *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pp. 1135–1146, IEEE Computer Society, 2011.

[23] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the design of a new linux readahead framework," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 75–84, July 2008.

[24] F. Schulz, D. Wagner, and C. Zaroliagis, "Using Multi-Level Graphs for Timetable Information in Railway Systems," in *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, vol. 2409 of *Lecture Notes in Computer Science*, pp. 43–59, Springer, 2002.

[25] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, eds., *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, vol. 74 of *DIMACS Book*. American Mathematical Society, 2009.