HAL
open science

Evolution and Evaluation of the Penalty Method for Alternative Graphs

Moritz Kobitzsch, Marcel Radermacher, Dennis Schieferdecker

To cite this version:

Moritz Kobitzsch, Marcel Radermacher, Dennis Schieferdecker. Evolution and Evaluation of the Penalty Method for Alternative Graphs. ATMOS - 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems - 2013, Sep 2013, Sophia Antipolis, France. pp.94–107, 10.4230/OASIcs.ATMOS.2013.94 . hal-00871738

HAL Id: hal-00871738

https://inria.hal.science/hal-00871738v1

Submitted on 10 Oct 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Evolution and Evaluation of the Penalty Method for Alternative Graphs

Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker

Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
{kobitzsch,schieferdecker}@kit.edu, marcel.radermacher@student.kit.edu

Abstract

Computing meaningful alternative routes in a road network is a complex problem – already giving a clear definition of a best alternative seems to be impossible. Still, multiple methods [1, 2, 4, 17, 18] describe how to compute reasonable alternative routes, each according to their own quality criteria. Among these methods, the *penalty method* has received much less attention than the via-node or plateaux based approaches. A mayor cause for the lack of interest might be the unavailability of an efficient implementation. In this paper, we take a closer look at the penalty method and extend upon its ideas. We provide the first viable implementation –suitable for interactive use– using dynamic runtime adjustments to perform up to multiple orders of magnitude faster queries than previous implementations. Using our new implementation, we thoroughly evaluate the penalty method for its flaws and benefits.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases Alternatives, Routing, Shortest Paths, Penalties, Parallelization

Digital Object Identifier 10.4230/OASIcs.ATMOS.2013.94

1 Introduction

Finding shortest paths in a road network is a well studied problem. Modern speed-up techniques can compute routes in a split second. These algorithms are usually based on an asymmetric approach: Exploiting the uniqueness of the shortest path distance, the road network is processed and augmented in advance. This –potentially– time-consuming preprocessing step then allows for fast subsequent queries. The approach inherently assumes a static nature of the input, though.

In contrast to the shortest path, alternative routes do not need to be optimal. Existing techniques to compute alternative routes either avoid speed-up techniques completely [2, 4], or try to relax the computational methods used during preprocessing or at query time [1, 18]. Methods that bypass speed-up techniques are only suitable for quality evaluations or offline usage. Thus, algorithms like the *penalty method* [2] are not explored to their full potential, lacking an efficient implementation.

While the authors of [2] hope for an efficient implementation to be feasible, they fail to provide any details on how to achieve this. In fact, until recently it simply was not possible to deal with the amount dynamic changes to the graph required by the penalty method – even though some techniques already existed that could handle small search spaces quite efficiently [5, 21]. By now, techniques such as *Customizeable Route Planning* [6] are available that can be extended to allow for preprocessing of entire continental networks within less than a second. This (near) real-time processing is achieved by extensive use of parallelism and vectorization and brings an efficient implementation of the penalty method within reach.

In this paper we show how to achieve an efficient implementation of the penalty method, providing speed-ups up to multiple orders of magnitude above previous implementations [2].

© Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker;
licensed under Creative Commons License CC-BY
13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13).
Editors: Daniele Frigioni, Sebastian Stiller; pp. 94–107
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Based on this implementation, we thoroughly evaluate the potential of the penalty method. The paper is structured as follows: Our terminology is introduced in Section 2, followed by related work in Section 3. Our contribution to the penalty method is discussed in Section 4. Supporting experiments are presented in Section 5, before a conclusion is drawn in Section 6.

2 Preliminaries

Every road network can be viewed as a directed and weighted graph:

▶ **Definition 1** (Graph,Restricted Graph). A weighted **graph** $G = (V, A, c)$ is described as a set of vertices $V, |V| = n$, a set of arcs $A \subseteq V \times V, |A| = m$ and a cost function $c : A \mapsto \mathbb{N}_{>0}$. We might choose to restrict G to a subset $\widetilde{V} \subseteq V$. This **restricted graph** $G_{\widetilde{V}}$ is defined as $G_{\widetilde{V}} = (\widetilde{V}, \widetilde{A}, c)$, with $\widetilde{A} = \{a = (u, v) \in A \mid u, v \in \widetilde{V}\}$.

We define paths and the associated distances as follows:

▶ **Definition 2** (Paths,Length,Distance). Given a graph $G = (V, A, c)$: We call a sequence $p_{s,t} = \langle s = v_0, \ldots, v_k = t\rangle$ with $v_i \in V, (v_i, v_{i+1}) \in A$ a **path** from s to t. Its **length** $\mathcal{L}(p_{s,t})$ is given as the combined weights of the represented arcs: $\mathcal{L}(p_{s,t}) = \sum_{i=0}^{k-1} c(v_i, v_{i+1})$. If the length of a path $p_{s,t}$ is minimal over all possible paths between s and t with respect to c, we call the path a **shortest path** and denote $p_{s,t} = \mathcal{P}_{s,t}$. The length of such a shortest path is called the **distance** between s and t: $\mathcal{D}(s, t) = \mathcal{L}(\mathcal{P}_{s,t})$. Furthermore, we define $\mathcal{P}_{s,v,t}$ as the **concatenated path** $\mathcal{P}_{s,v,t} = \mathcal{P}_{s,v} \cdot \mathcal{P}_{v,t}$.

In the context of multiple graphs or paths, we denote the desired restriction via subscript. For example $\mathcal{D}_G(s, t)$ denotes the distance between s and t in G, with $\mathcal{D}_{p_{s,t}}(a, b)$ we denote the distance between a and b when following $p_{s,t}$.

Our metric of choice is the average travel time. Therefore, we might choose to omit the cost function c when naming a graph and simply write $G = (V, A)$.

3 Related Work

Both shortest paths and alternative routes computation are important for our work. Following, we give a short overview of the techniques most relevant to our contribution.

Shortest Paths

Algorithms for computing exact shortest paths have come a long way. Starting back in the late 1950s with Dijkstra's algorithm [12], incredible progress was made in this area – especially during the last decade. By now, we can answer distance queries on a road network over a million times faster than Dijkstra's algorithm.

All relevant *speed-up techniques* to Dijkstra's algorithm share an asymmetric approach: In a preprocessing step, auxiliary information is generated once and then used in all subsequent queries. This approach is effective if arc costs do not change, but it becomes a major bottleneck if not prohibitive, if preprocessing has to be repeated multiple times. [3, 9, 23] give a general overview on speed-up techniques as we focus on the following two techniques:

Contraction Hierarchies [14] (CH) is probably the most studied speed-up technique. During preprocessing, the graph representing the road network is augmented by carefully chosen (shortcut) arcs while arcs not required for correctness are removed. This results in a sparse directed acyclic graph (DAG). A query corresponds to a bidirectional variant

of Dijkstra's algorithm on this modified graph. Road networks of continental size can be preprocessed within minutes and distance queries run in the order of one hundred microseconds. Reconstructing the complete shortest path requires roughly the same time. This technique is most suited for static settings in which graphs do not change.

Customizeable Route Planning [6] (CRP, also known as Multi Level Dijkstra) is the current pinnacle in a long list of multi-level separator based techniques, [8, 16, 22] to name a few. In a first preprocessing step, a multi-level partition is generated. The boundary vertices of this partition induce an overlay graph at each level. To maintain shortest paths, each cell is connected in a clique. Computing this representation relies only on the structure of the graph. A metric is incorporated in a second step, when correct costs are computed for all arcs within the cliques. The (bidirectional) query traverses arcs like Dijkstra's algorithm. When a boundary vertex is reached, the query switches to the next higher level and continues to traverse only arcs in the respective overlay graph[1]. CRP profits from using *PUNCH* [7] to find tiny separators. The best variant uses a combination of up to 5 levels and an additional set of guidance levels (or *shadow levels*) for preprocessing. This setup allows for distance queries in about one millisecond and updates of the entire metric in less than a second.

Alternative Routes

Abraham et al. [1] were the first to formally introduce alternative routes in road networks, even though related methods like the k-shortest path problem [13, 24] have been introduced before. We do not cover these methods as no suitable implementation exists for continental sized road networks and because some of the earlier methods do not produce good alternative routes due to a plethora of short detours available in road networks. By now, two most common approaches found in the literature are via-node alternative routes or the related plateaux method [1, 2, 4, 17, 18], and penalty-based approaches [2] (among others). Their following description is taken partly from [17]:

Via-Node Alternative Routes

Within a graph $G = (V, A)$, a via-node alternative to a shortest path $\mathcal{P}_{s,t}$ can be described by a single vertex $v \in V \setminus \mathcal{P}_{s,t}$. The alternative route is described as $\mathcal{P}_{s,v,t}$. As this simple description can result in arbitrarily bad paths, for example paths containing loops, Abraham et al. [1] define a set of criteria to be fulfilled for an alternative route to be *viable*. A viable alternative route provides the user with a real alternative, not just minimal variations (*limited sharing*), is not too much longer (*uniformly bounded stretch*) and does not contain obvious flaws, i.e. sufficiently small sub-paths have to be optimal (*local optimality*). Formally, these criteria are defined as follows:

▶ **Definition 3** (Viable Alternative Route). Given a graph $G = (V, A)$, a source s, a target t, and a via-node v as well as three tuning parameters γ, ϵ, α; v is a viable via-node and defines a **viable via-node alternative route** $\mathcal{P}_{s,v,t} = \mathcal{P}_{s,v} \cdot \mathcal{P}_{v,t}$, if following criteria are fulfilled:

1. $\mathcal{L}(\mathcal{P}_{s,t} \cap \mathcal{P}_{s,v,t}) \leq \gamma \cdot \mathcal{D}(s,t)$ *(limited sharing)*
2. $\forall a, b \in \mathcal{P}_{s,v,t},\ \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,a) < \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,b) :$
   $\mathcal{D}_{\mathcal{P}_{s,v,t}}(a,b) \leq (1+\epsilon) \cdot \mathcal{D}(a,b)$ *(uniformly bounded stretch)*
3. $\forall a, b \in \mathcal{P}_{s,v,t},\ \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,a) < \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,b),$
   $\mathcal{D}_{\mathcal{P}_{s,v,t}}(a,b) \leq \alpha \cdot \mathcal{D}(s,t) :\ \mathcal{D}_{\mathcal{P}_{s,v,t}}(a,b) = \mathcal{D}(a,b)$ *(local optimality)*

[1] For efficiency, the query may descend to lower levels when close to target/source. See [6] for details.

The usual choice for the tuning parameters is to allow for at most $\gamma = 80$ % overlap between $\mathcal{P}_{s,v,t}$ and $\mathcal{P}_{s,t}$. Furthermore the user should never travel more than $\epsilon = 25$ % longer than necessary between any two points on the track, and every subpath that is at most $\alpha = 25$ % as long as the original shortest path should be an optimal path.

These criteria require a quadratic number of shortest path queries to be fully evaluated. Therefore, Abraham et al. propose an approximation [1]. For example, their *T-test* for local optimality is achieved by a single query between two vertices close to the via-node. Due to properties of this T-test, the criteria (and thus also the numbers in Section 5) are usually evaluated only for the part of the via-route that is distinct from the shortest path.

Definition 3 can be directly extended to allow for a second or third alternative (alternative routes of *degree* 2, 3 or even n), as only the limited sharing parameter has to be tested against the full set of alternatives already known.

Abraham et al. [1] give multiple algorithms to compute alternative routes. The reference algorithm (X-BDV) is based on a bidirectional implementation of Dijkstra's algorithm and is used as the gold standard. To avoid the long query time of Dijkstra's algorithm on continental-sized networks, they also give techniques based on Reach [15] and Contraction Hierarchies [14]. Due to the strong restrictions of the search spaces caused by the speed-up techniques, they present weakened query criteria which they call relaxation. For example, in the Contraction Hierarchy they allow to look downwards in the hierarchy under certain conditions. The relaxation can be applied in multiple intensities. Commonly used is the 3-relaxation which we refer to by X-CHV.

Luxen and Schieferdecker [18] improve the algorithm of Abraham et al. [1] in terms of query times by storing a precomputed small set of via-nodes for pairs of regions within the graph. This is the fastest method to compute via-node alternative routes as of now.

The Penalty Method

Bader et al. [2] describe an entirely different approach. Their main focus is on computing a full alternative graph to present as a whole, or possibly to extract alternative routes from. The construction of the graph, however, relies on the iterative computation of multiple shortest paths. After a single path is computed, they apply a penalty to the path and to every arc directly connected to it, thus potentially finding a different path in the next iteration. By lowering successive increases in penalty, they propose to stop iterating when no penalty is applied to the extracted path anymore. The publication itself does not give exact numbers, but according to one of the authors 20 iterations are performed to generate paths. From this set of computed paths, the best ones are selected and combined into an alternative graph. To make the graph more readable, the authors present two filters that can be applied to the graph. Again, they do not specify any details on how to select the paths for the final graph. The quality of the graph is evaluated using two measures (total and average distance) while at the same time limiting the complexity by putting a hard bound on vertices of degree higher than two (*decision vertices*). It is defined as follows:

▶ **Definition 4** (Alternative Graph Quality)**.** The quality of an **alternative graph** $H = (V_H, A_H)$, also called *target function*, is given by $totalDistance - (averageDistance - 1)$ with $averageDistance \in [1.0, 1.1]$. The upper limit is enforced during graph construction. Given start s and target t, the contributing values are defined as:

1. $totalDistance := \sum_{a=(u,v)\in A_H} \frac{c(a)}{\mathcal{D}_H(s,u)+c(a)+\mathcal{D}_H(v,t)}$ *(indicates sharing)*
2. $averageDistance := \frac{\sum_{a\in A_H} c(a)}{\mathcal{D}(s,t)\cdot totalDistance}$ *(indicates stretch)*

Intuitively speaking, the total distance describes how many distinct paths can be found in the graph, while the average distance describes how much longer such a path is on average. The authors themselves do not provide an efficient implementation of the penalty method but claim the implementation to be possible with [21]. With single arc updates taking several milliseconds this claim seems excessive.

Recently, Paraskevopoulos and Zaroliagis published a new paper [19] on the penalty method. They suggest modifications to the penalization scheme to obtain higher quality alternative graphs. Additionally, they introduce pruning techniques that allow for faster query times than the original work by Bader et al. [2].

4 Alternative Graph Computation

The basic setup of our algorithm follows the ideas from [2], as described in Section 3. We compute a shortest path, potentially add the obtained path to our output, penalize the arcs on the path as well as the adjoined arcs and repeat. The general process is illustrated below as Algorithm 1:

Algorithm 1 CRP-π

```
1   original_path = path = computeShortestPath();
2   H = original_path;
3   while L(path) ≤ (1 + ε) · L(original_path) do
4   begin
5       applyPenalties(path);
6       path = computeShortestPath();
7       if isFeasable(path) do
8       begin
9           H = H ∪ path;
10      end;
11  end;
12  return H;
```

Following, we explain the meaning of `isFeasable(path)` and the stopping criterion of our algorithm. We describe the modifications we make to the algorithm proposed by Bader et al., which we also refer to as *classcial* method. We show how to achieve an efficient implementation and how to extract single alternative routes.

4.1 Path Selection

Algorithm 1 utilizes the procedure `isFeasible(path)` to decide whether to keep an alternative route or not. The original paper [2] does not specify how to exactly choose the paths that form the alternative graph. According to personal conversation with one of the authors, their algorithm computes up to 20 paths and performs a selection based on some priority terms afterwards.

While their implementation does not focus on query times but on evaluating multiple different approaches, we have to consider the cost of performing too many iterations. Thus, we take a different approach. Instead of applying penalties as many as 20 times, we consider the true length –without penalties– of the computed path. Whenever this length exceeds $(1 + \epsilon) \cdot \mathcal{D}(s, t)$, we stop our algorithm (see Definition 3, uniformly bounded stretch).

Every path we find during the execution is evaluated by the aforenamed procedure for its potential *value* to the alternative graph. We postulate that a path must offer at least one deviation to the current alternative graph of length $\delta \cdot \mathcal{D}(s, t)$ or more, with δ usually chosen

as $\delta = 0.1$ (compare Definition 3, limited sharing). The detours satisfying this requirement are checked against the current alternative graph H for stretch. If one of these detours between vertices a, b is not longer than $(1+\epsilon) \cdot \mathcal{D}_H(a,b)$, its containing path is added to the alternative graph. All other paths are rejected.

4.2 Changes to Penalization

As described above, the classical penalty method penalizes the arcs along the shortest path by adding a fraction of the arcs' original length. This fraction is called the *penalty factor* π_f and is usually chosen as $\pi_f = 0.04$. Additionally, the adjoined arcs of the path are penalized by adding $\pi_r = \pi_f \cdot 0.002 \cdot \mathcal{D}(s,t)$, the so called *rejoin penalty*. These changes to the metric are persistent during the computation of an alternative graph.

The penalty method, as suggested by Bader et al. [2], requires a significant amount of iterations. Even when using our stopping criterion from the previous section, we experience a similar behavior, requiring about 20 iterations and more on average. Therefore, we take a slightly different approach to penalization and modify the way penalties are introduced to the graph.

In contrast to the choice of Bader et al., we multiply the current lengths of the arcs on the shortest path by $1 + \pi_f$. Thus, we penalize arcs which are used often more strongly than others, and significantly increase the rate at which penalties grow (geometrically growing). This is in contradiction to Bader et al.'s preference to lower successive increases in penalty. Our choice is not only motivated by its beneficial impact on the number of iterations, but also favors detours to segments of the alternative graph that are already covered by multiple paths. But of course, this may result in our algorithm missing some promising path candidates due to the higher increase in penalties.

The higher penalization has influence on the choice of the rejoin penalty as well. While Bader et al. propose a relatively small penalty π_r, this choice proved to be not high enough for our faster growing multiplicative penalties to prevent short detours. We therefore change the additive penalty to $\pi_r = \alpha \cdot \sqrt{\mathcal{D}(s,t)}$, with $\alpha = 0.5$ as a typical value. This change is motivated by the following observation: Consider $\mathcal{P}_{s,t}$ and a new path $p_{s,t}$ found on the penalized graph which deviates from $\mathcal{P}_{s,t}$ between vertices a, b. Due to penalization, $\mathcal{D}_{p_{s,t}}(a,b) + 2 \cdot \pi_r \leq (1+\bar{\pi}) \cdot \mathcal{D}(a,b)$, with $\bar{\pi}$ the average penalization along $\mathcal{P}_{a,b}$. In other words, for a new detour to be found between vertices a, b, it has to be shorter, including rejoin penalties, than the current (penalized) shortest path between a, b. This condition gets easier to fulfill, the further s, t are apart as the rejoin penalty grows much slower than the path length. Therefore, we allow for larger detours to be found on longer paths. But locally, we only want short detours (compare Definition 3, local optimality).

4.3 Fast Computation

Our most significant change to the classical method is the introduction of CRP as speed-up technique. While there are other methods for dynamic shortest path computation [5, 21], none of them is sufficient for the high amount of dynamic behavior required for our cause. As recent developments have shown, close to real-time processing of entire graphs is possible with the CRP technique [10].

While we initially took a different approach, the general methods –making extensive use of vector instructions and parallelism– remain the same. Even though it does not seem intuitive to perform more work than necessary for a pure update of a shortest path, the

locality properties, vectorization and suitability for parallel processing allow a CRP based implementation to outperform other approaches that consider only the changed arcs.

As our set of updating techniques –while much simpler to implement– is outperformed by the methods recently described by Delling and Werneck [10], we do not focus on our exact implementation of CRP. Instead, we concentrate on an essential modification to CRP, required to achieve maximal performance of the penalty method:

Dynamic Level Selection

Applying a multi-level technique is always a balancing act between fast queries of long routes and overhead for short routes. In our case, namely the reiterated computation of a shortest path and the update of affected arcs between the same source and target, we also have to weigh update costs against query times. While beneficial for long range queries, large cells in the upper levels have a high update cost in comparison to the smaller cells further down the hierarchy. These updates soon dominate the runtime of our algorithm as even rather short paths can touch many high level cells. As we perform multiple computations between the same source and target pair, we can alleviate this problem. After an initial computation of the shortest path, we can analyze the path regarding its length and the cells the path touches on different hierarchy levels. For the cost of storing some additional mapping information, we use the obtained information to dynamically adjust our implementation of CRP to work only on a fixed number of levels. Restricting to a subset of levels allows for faster updates of CRP as not all levels have to be updated, at the cost of higher query times.

4.4 Alternative Route Extraction

When presented with an alternative graph, multiple ways exist to extract alternative routes. To compare ourselves to other methods in terms of success rate, we apply a two-step approach. In a first step, we perform X-BDV on our alternative graph without checking for local optimality. This test is omitted as the alternative graph does not provide enough information to compute shortest paths between arbitrary vertices with respect to the underlying graph. We call the result of this first step *CRP-π-via*. After having searched exhaustively for via-node alternatives, we run a simplified penalty method to extract further routes. We do not apply rejoin penalties as the alternative graph is sparse and only contains meaningful junctions. The X-BDV alternatives are used to initialize penalization, and the extracted paths have to adhere to the same stretch and overlap criteria as imposed by X-BDV. The full algorithm is named *CRP-π*. When omitting the first step, we call the results *CRP-π-penalty*.

5 Experiments

We first provide a detailed overview of our experimental setup and a short outline of the quality measurements used during the evaluation of the penalty method. This is followed by an extensive experimental evaluation of our techniques as introduced in Section 4.

Setup

We run our algorithm on four Intel Xeon E5-4640, clocked at 2.4 GHz with 32 cores in total and 512 GB of RAM – the actual space consumption of CRP-π is much less though. The machine is running Ubuntu 12.04. We apply the C++ compiler of the GNU Compiler Collection (GCC), version 4.6.1, with parameters `-std=c++0x -fopenmp -O3 -msse4.1 -mtune=native`. For parallelization we use OpenMP. Our implementation is based on a

partition generated by PUNCH, comparable to the one used in [6] with 5 levels, including a shadow level. The road network of Western Europe supplied by PTV AG for the 9th Dimacs Implementation Challenge [11] is used in our experiments. It contains 18 million nodes and 24 million edges and uses the travel time metric as arc costs. The graph consists of a single strongly connecteed component. We present numbers based on random queries and on rank queries. The Dijkstra rank of a vertex is defined as the step in which it is settled during the execution of Dijkstra's algorithm. For both variants we choose 1 000 queries uniformly at random, unless said otherwise.

For comparison, we apply our own implementations of the competing via-node and penalty approaches, with results similar to the original papers. Tuning parameters are chosen at their usual values as introduced before. Note that our implementation of the classical penalty method corresponds to CRP-π while using Dijkstra's algorithm and the classical penalization scheme. In particular, we apply our stopping criterion instead of running 20 iterations straight and add each viable path to the alternative graph as soon as it is discovered. This is due to Bader et al. not providing details on the path selection process in their paper.

5.1 Runtime

One of the most important characteristics in routing applications is the query time. Therefore, we first evaluate the runtime of our algorithm before turning to the quality of the computed routes.

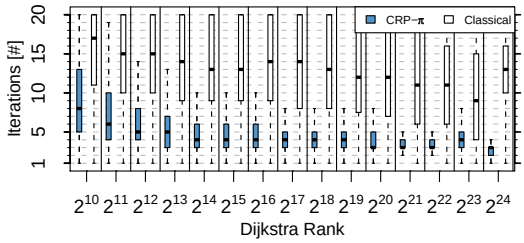

Figure 1 Number of iterations required by the penalty method until the stopping criterion holds.

The number of iterations performed is likely the most influential factor to our query time. Figure 1 shows that the number of iterations is much higher for queries of lower rank. Therefore, it is important to perform updates very fast for short queries. We also see that without our modifications to the penalization, we experience an average of 12 iterations and higher across the full range of queries. Remember, the original implementation by Bader et al. always computed 20 iterations. Considering the cost to perform a single iteration (Figure 2), the classical penalization would not be suitable for an efficient implementation, even when using CRP. We also see that update costs remain small as only affected cells are recomputed.

These findings are further confirmed by Figure 3, which gives sequential runtimes for using Dijkstra's algorithm with classical penalization. Note the logarithmic scale of the y-axis. We see that query times already become impractical for very short ranged queries. This is expected behavior and comparable to the original algorithm [2].

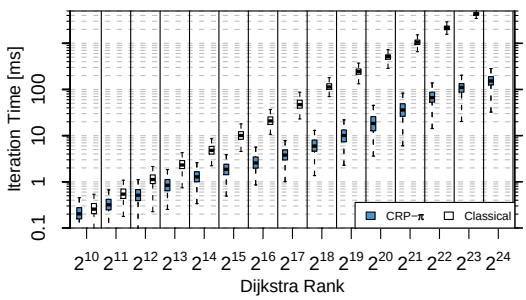

Figure 2 Sequential runtime of a single iteration of the penalty method.

Using a bidirectional implementaion of Dijkstra's algorithm could reduce runtimes of the classical approach by a small constant factor, but they would remain prohibitively high. This becomes evident in particular when looking at long range queries, the classical approach requires up to 5 seconds per iteration while CRP-π only takes between 100 and 200 milliseconds. Not even goal directed methods as in the new method of Paraskevopoulos and Zaroliagis [19] seem to suffice for reaching the query performance of CRP-π.

Dynamic Level Selection

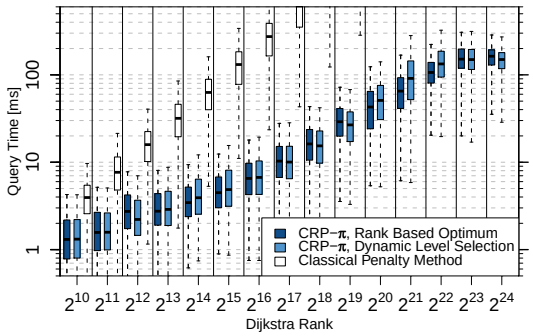

Figure 3 Runtime of the penalty method with dynamic level selection and rank based optima. CRP-π uses 16 cores, the classical method 1 core.

Figure 4 gives an impression on the impact of both parallel execution of our algorithm as well as the overhead introduced by using a fixed number of levels for CRP. Especially for short routes, the negative effects of using many levels of the partition become obvious. This is as the cells in the higher levels are very large and the costly updates dominate the benefits in query time. Using all 5 levels is actually only beneficial in CRP-π for finding shortest paths in the original graph. Moreover, we see that using more than 12 cores is hardly effective for any type of query. In fact, parallel execution only starts to pay off for long range queries, with 12/16 cores performing up to 4 times faster than sequential execution. For short to medium range routes, it can be deemed a waste of resources.

Following the results of Figure 4, we propose to use only two levels for short routes, switch to three levels for the medium range, and finally apply four levels on long range queries. We base the decision on how many levels to use on the hop count of the shortest path (which is always computed using all 5 levels of the partition). To tune this selection process, we generate a different set of routes for every Dijkstra rank and compute the average number of hops on the shortest path.

The performance of the resulting dynamic algorithm is shown in Figure 3. We see that our dynamic algorithm is sometimes even slightly faster than the best values of the respective Dijkstra rank queries. This is, as the dynamic choice allows us to better adjust the performance of our algorithm to the actually required workload whereas a forced number of levels does not represent the required work as accurately. For comparison, Figure 3 shows the respective best values from Figure 4. The values were extracted on Dijkstra-rank basis. Due to very long arcs, i.e. ferry connections, we experience some erratic parallelization behavior when dynamically selecting the levels based on the hop-count alone. A selection based on affected cells in each level might provide better results in the future.

Avoiding dynamic level selection, the best choice is obviously using three levels of hierarchy. Only for very short and very long range queries we see detrimental effects. But in the worst case, there is a slowdown by a factor of up to 4.

Although, at the current state, the required number of cores for a viable execution of our algorithm might be considered high for long range queries, the algorithm performs queries efficiently and with a low number of cores for all reasonable distances. To improve workload, dynamic selection of cores can be introduced similar to dynamic selection of levels.

Figure 3 compares only our best results, obtained by parallel processing, to the classical approach, running sequentially. Though, we clearly outperform this approach even when using a single core. Figure 4 demonstrates the performance benefits of our general approach over the classicial method. As seen in the plot depicting dynamic level selection, CRP-π takes at most 600 milliseconds on one core for the longest queries whereas the classical approach requires beyond 100 seconds, which is off the scale in Figure 3.

For completeness, we state the runtimes of the via-node approaches introduced during the following quality analysis. X-BDV requires 14.1 seconds on average to compute three alternatives, where possible. X-CHV takes 4.95 milliseconds for the same task.

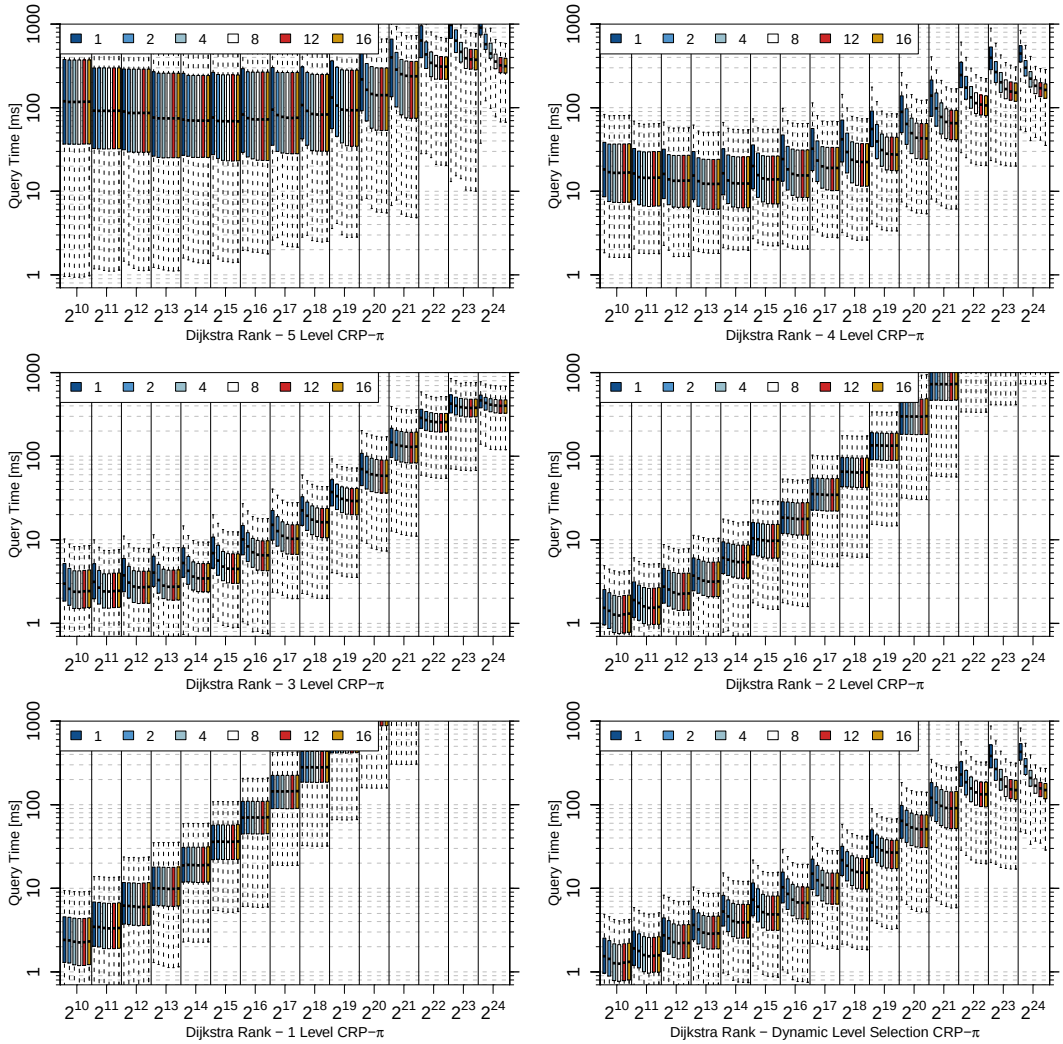

Figure 4 Runtime of the penalty method. Each illustration shows the runtime for a range of cores. The number of levels of the underlying CRP implementation is either fixed to a given set of levels, or dynamically adapted after the initial shortest path query.

5.2 Quality

First, we evaluate the results of our algorithm with regards to the quality as defined in the original paper of Bader et al. [2]. We note that without access to their path selection criteria, we were unable to reproduce the numbers listed in their work. In addition, their numbers stem from a very sparse test set of 100 queries. Another implementation [20] faced similar difficulties, but at least conducted more extensive measurements. They report an average value of 2.89 for the alternative graph quality of Definition 4, compared to the 3.21 in [2]. Without filtering, our algorithm yields alternative graphs with an average quality rating of 3.32 with 17.4 decision vertices on average. As this is above the proposed hard limit in the original paper, we can filter the graph by removing all arcs that are only contained in paths longer than the allowed maximum stretch. The additional overhead is negligible, well below 100μs on average, as the alternative graphs are tiny. Filtering reduces the quality to 2.89 and the decision vertices to 9.53. While limiting the number of decision vertices is beneficial

Table 1 Success rates and average path quality numbers for the first through third alternatives in terms of Definition 3. Compared to the results found in [1], local optimality is not strictly enforced. See text for a discussion on maximum/minimum path quality values.

#	algorithm	success [%]	stretch [%]	sharing [%]	optimality [%]
first	X-BDV	96.0	10.0	41.8	75.4
	X-CHV	89.6	80.4	40.6	68.1
	CRP-π-via	95.2	42.8	31.6	27.1
	CRP-π-penalty	96.3	40.6	40.8	24.4
	CRP-π	96.3	42.9	31.9	26.9
second	X-BDV	87.6	13.8	59.5	65.1
	X-CHV	72.5	269.0	57.6	57.2
	CRP-π-via	79.8	47.1	44.7	22.9
	CRP-π-penalty	83.1	60.5	36.8	10.8
	CRP-π	84.0	47.6	45.9	22.1
third	X-BDV	75.5	17.2	65.6	54.6
	X-CHV	51.4	214.0	63.6	46.8
	CRP-π-via	52.7	66.5	49.3	18.0
	CRP-π-penalty	53.0	65.9	32.0	5.6
	CRP-π	62.9	67.4	51.8	15.9

for the readability of an alternative graph, it reduces the potential for extracting multiple viable alternatives. Thus, we opt to not apply the filter for the following analyses.

Now, we take a look at the well established via-node approach. For comparison, we choose the Dijkstra based (X-BDV), and the Contraction Hierarchy based (X-CHV) variants introduced in [1]. Table 1 summarizes the results, giving numbers for success rates, i.e. how often we can extract one to three viable alternatives from our alternative graph, and for the quality measures introduced in Definition 3. As we do not strictly enforce local optimality, we disabled this criterion for X-BDV and X-CHV to allow for a fair comparison. We see that success rates of CRP-π are well above X-CHV for all alternatives and even on par with X-BDV for the first route. For second and third routes our algorithm fares slighty worse compared to X-BDV. Note though that X-BDV obtains its high success rates at the cost of prohibitively slow query times of about 14 seconds. Average path quality measures seem reasonable with our uniformly bounded stretch and local optimality values being worse but our sharing values being better than those of the via-node approaches. This is an expected compromise as lower stretch comes with higher overlap and vice versa. The overall high stretch values are due to none of the algorithms enforcing uniformly bounded stretch explicitly, only total stretch of each path is enforced.

We further find that not enforcing local optimality has little impact on the average path quality values. Only the uniformly bounded stretch of X-CHV increases significantly. Due to the structure of CH search spaces, computed alternatives often exhibit an overlapping subpath at the via-node. This would imply infinite stretch values, filtering these overlaps we obtain the listed values of above 200. Maximum stretch and minimum optimality values degrade dramatically without enforcing local optimality, though. This leads us to look more closely into how poor bounded stretch and local optimality values arise. As they represent averages of the worst values on each path, it is easy to see that even tiny suboptimalities compared to the full path length lead to poor quality values. We find stretch values over

100% for about 10% of all alternatives and local optimality values below 1% for about 20% of all alternatives of CRP-π. This is about twice as often as for X-CHV, with X-BDV only showing single poor values. We further checked that this is always caused by a single subpath of small length (about 1% - 3% of the full path length). Thus, we conjecture that these problems are repairable with very local searches at low costs.

Finally, we compare the results of running only one step of our alternative route extraction to the full process. We see that the different extraction methods compliment each other. On their own, they offer comparable success rates on par or even better than X-CHV. When combined, the success rates increase – especially for higher degree alternatives. This leads us to the conjecture that both approaches provide structural different routes.

Encouraged by this observation on the small scale, we study the structural differences offered by our penalty approach compared to via-node based approaches on the whole. We evaluate the extracted alternative routes with regards to the additional information they provide that cannot be obtained by via-node based approaches. For this analysis we consider all extracted routes $p_{s,t}$, compute a via-node alternative $\mathcal{P}_{s,v,t}$ for each vertex v on that route, and compute the overlap between these two paths. For a fair comparison, we only consider vertices that yield a viable via-node alternative with respect to the stretch and overlap criteria. Furthermore, we do not consider overlapping subpaths that are also part of the shortest path $\mathcal{P}_{s,t}$. We find that that maximum overlap is well below 80% and getting smaller for higher degree alternatives – 77.9%, 72.7%, 65.5% for the first through third alternative, respectively. This implies that our approach offers a meaningful addition to the established via-node approaches.

6 Conclusion

The extensive use of vectorization and modern multi-core machines has enabled us to provide the first implementation of the penalty method that is suitable for interactive applications. We have shown the results to provide meaningful additions to the world of alternative routes. Some open problems remain though. Most interesting would be to find an (approximable) set of criteria to classify good alternatives, not tailored to one specific approach. Furthermore, the runtime of our implementation remains high, especially when compared to shortest path queries. We want to find ways to improve upon this implementation and compute alternative graphs even quicker. The recent work by Paraskevopoulos and Zaroliagis [19] seems to be promising in this respect. Their approach is orthogonal to ours and should integrate well.

Acknowledgements We would like to thank Daniel Delling of Microsoft Reasearch, Sillicon Valley for providing a PUNCH partition of our European road network.

References

1 Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative Routes in Road Networks. *ACM Journal of Experimental Algorithmics*, 18(1):1–17, 2013.

2 Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. Alternative Route Graphs in Road Networks. In *International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems (TAPAS'11)*, volume 6595 of *LNCS*, pages 21–32. Springer, 2011.

3 Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, 2010.

4 Cambridge Vehicle Information Tech. Ltd. Choice Routing. `http://camvit.com/camvit-technical-english/Camvit-Choice-Routing-Explanation-english.pdf`, 2005.

5 Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, and Camillo Vitale. Fully Dynamic Maintenance of Arc-Flags in Road Networks. In *International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, pages 135–147. Springer, 2012.

6 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In *International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *LNCS*, pages 376–387. Springer, 2011.

7 Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph Partitioning with Natural Cuts. In *International Symposium on Parallel and Distributed Processing (IPDPS'11)*, pages 1135–1146. IEEE, 2011.

8 Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. High-Performance Multi-Level Routing. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 73–92. AMS, 2009.

9 Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*, pages 117–139. Springer, 2009.

10 Daniel Delling and Renato F. Werneck. Faster Customization of Road Networks. In *International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 2013.

11 Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. AMS, 2009.

12 Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

13 David Eppstein. Finding the k Shortest Paths. In *Symposium on Foundations of Computer Science (FOCS'94)*, pages 154–165. IEEE, 1994.

14 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, 2012.

15 Ronald J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.

16 Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, 2008.

17 Moritz Kobitzsch. An Alternative Approach to Alternative Routes: HiDAR. In *European Symposium on Algorithms (ESA'13)*. Springer, 2013.

18 Dennis Luxen and Dennis Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. In *International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *LNCS*, pages 260–270. Springer, 2012.

19 Andreas Paraskevopoulos and Christos Zaroliagis. Improved Alternative Route Planning. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13)*, OASIcs. Dagstuhl Publishing, 2013.

20 Marcel Radermacher. Schnelle Berechnung von Alternativgraphen. Bachelor's thesis, Karlsruhe Institute of Technology, Fakultät für Informatik, 2012.

21 Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In *Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *LNCS*, pages 66–79. Springer, 2007.

22 Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *LNCS*, pages 43–59. Springer, 2002.

23 Christian Sommer. Shortest-Path Queries in Static Networks, 2012. submitted. Preprint available at `http://www.sommer.jp/spq-survey.htm`.

24 Jin Y. Yen. Finding the k Shortest Loopless Paths in a Network. *Management Science*, 17(11):712–716, 1971.