Bachelor Thesis
Computer Science

Radboud University Nijmegen

Analysing Multicriteria Route Planning

Author:
Matthijs Hendriks
mhendriks@science.ru.nl

First supervisor/assessor:
Frits Vaandrager

Second assessor:
David Jansen

5 July 2013

Abstract

Whereas regular path finding is about finding an optimal path between two locations, given one criterion – such as travel time – multicriteria path finding takes multiple of such criteria into account. This thesis analyses this type of path finding using realistic example scenario's. We prove that the theoretical complexity of any multicriteria path finding problem is $\mathcal{NP}$-complete, and propose two methods to solve such a problem. The first method is able to solve an adapted version of the multicriteria path finding problem, by restricting the criteria aggregation function in such a way that we are able to treat the original problem as a regular path finding problem. To solve such a problem without any restrictions, we propose an algorithm that is a combination of existing algorithms, and improves upon them.

Contents

1 Introduction

Path finding is a technique that has many applications in all sorts of fields and is still researched extensively. One of its applications lies in navigation systems: logically, since these systems are developed to guide a user from one point to an other, path finding is used to find an optimal path between these points. Calculating such a path is an algorithmic task for which many algorithms have been created. Examples of algorithms are *Dijkstra's Algorithm*, *A**, *SHARC*, *CH*, etc.[4] These algorithms all assume that the user simply wants to reach the given destination without having multiple requirements. This is fine for most users, however, path finding scenarios that require multiple criteria to be taken into account can be thought of. For example, a user might not want to spend more than a certain amount on gas. In these cases the current path finding algorithms used by navigation systems cannot meet the user's wishes. At this time, such problems can be solved by algorithms such as the *Label Setting Algorithm*, *NSGA-II*, and *SSMOSP*, but these algorithms are either unsuitable for use on mobile devices due to their high complexity, or do not output optimal paths, whilst that is what we are looking for.[3][11][17].

In this thesis, we will propose a new algorithm, based upon existing efficient algorithms, that is able to solve problems such as ours more efficiently. We will first show an example scenario to illustrate the problem. Then we will extract criteria from this scenario and enumerate some other similar, realistic criteria. We will then move on to the actual problem, the section in which we will give an accurate definition of the exact problem, using the earlier and new scenarios. Following up on our definitions, we will analyze the problem and determine its theoretical complexity using the well-known Knapsack-problem, and suggest a method to solve an adapted version of our problem with a efficiently. After that, we will move on and have a look at some algorithms and propose a combination of these algorithms that should be able to solve our (unadapted) problem efficiently. Finally, in our conclusion, we will retrospect on these algorithms, discuss some more related work, and make suggestions for future work on this subject.

1.1 Scenarios

Some scenarios are not supported by current navigation systems. We can place each of the algorithms that can support these scenarios in one the following categories:

1. Dynamic path finding
2. Intelligent path finding
3. Multicriteria path finding

These three categories cover most advanced path finding problems. Algorithms belonging to category one are those that take e.g. traffic jams and holidays into account; those in category two are algorithms that, among other things, register and track the user's driving style or speed and use that data to calculate an optimized route for that

specific user; and category three algorithms allow the user the specify multiple criteria which the calculated path should meet.

The first two categories exceed the scope of this thesis. We will focus on algorithms from the third category instead, *multicriteria path finding*.

Multicriteria path finding is path finding with the possibility to provide more than one criteria the given path should meet. This is already possible, but only to a certain extent. For example, TomTom has navigation systems that ask the user if he wants to use highways or not, or if the system should avoid roads that require toll to be paid. However, these criteria are all so called 'hard criteria', meaning that the system wields an all-or-nothing approach: it is not possible to use as little highway as possible, combined with some other similar restriction. Next this, if no route without highways exists, the system ignores the option. For some users this will be sufficient, but some will require more possibilities. To illustrate this, we have written an example scenario.

1.1.1 Road trip

Two friends are planning a road trip through Europe. They determined a global route from their home back to their home, Nijmegen, via the biggest cities in Europe. Both of them enjoy nature and therefore they want a route that uses as many 'green roads' (i.e. roads through nature, roads with a nice view) as possible. This is a criterion they value more important than any other. Also, since they are on a road trip, they would prefer a route close to touristic attractions above routes that are further away from them, providing that the detour is within certain limits. Furthermore, they prefer a route as short as possible. However, this is the least important requirement, so they are willing to make a detour for a nicer view or touristic attraction. Finally, they want to avoid all toll roads.

Current navigation systems fail to meet these requirements because they can not handle multiple criteria. The two friends would manually have to search for touristic attractions, nature-rich roads and calculate for themselves what routes are preferred above others. It would take them hours or even days to do so and determining a sufficient route – assuming they at all know how to do this – but due to the high complexity of this problem, it is unlikely that this route would be optimal. Current systems know a road's length, but probably don't know about nature or touristic attractions on or close to it. It is true that TomTom offers *Points of Interest* lists[16], which helps our friends determining a route to their wishes, but TomTom's navigation systems do not automatically calculate a route close to such points. This results in the friends finding and following a sub-optimal path, given their criteria, and thus they are not optimally satisfied.

1.2 Criteria

Some criteria can be extracted from this scenario, but obviously more can be thought of and placed in similar scenarios. These extracted criteria and some other examples are listed below, classified by criterion type: hard or soft. Hard criteria are those that should be met under all circumstances, whilst soft criteria are criteria which should be met as far as possible. This means that if only one path between two locations can be found and this path violates a hard criterion, the algorithm should return no path at all – a hard criterion is violated. However, if a this path violates a soft criterion instead, the path should be returned as an optimal solution.

Hard criteria:

1. Bridge heights
2. Slope gradients
3. Maneuverability
4. Road restrictions (i.e.: max x kg heavy, max y meters wide)

Soft criteria:

1. Travel time
2. Route length
3. Fuel expenses
4. Travel expenses (i.e.: toll, vignets)
5. Surroundings (i.e.: forest, countryside)
6. Nearby tourist attractions

Above lists only show static criteria. Other criteria, such as traffic jams and actual redirects are dynamic criteria. Since our problem is already complex enough with static criteria only, for a bachelor thesis, these criteria are outside of this thesis' scope and will therefore not be treated any further here.

2 Problem

In the previous section we have seen a scenario in which the shortest path is not per definition the best path, and we identified a number of criteria that can be part of a scenario. In this section we will define the exact problem that we are facing.

2.1 Scenarios

In addition to the *Road Trip*-scenario, other, similar scenarios are realistic.

Vacation A family in The Netherlands has planned a vacation in the southern regions of France. They are going by car and would like to get to their camping site as soon as possible. Unfortunately, the family has limited budget for their trip and France is known for its many toll roads. To prevent spending half of this budget to cross such roads, they would like to avoid them as much as possible. However, in case using a toll road reduces travel time significantly, compared to not using it, they do want to use this road. Avoiding toll roads is important to them, but the increased duration and fuel cost of the detour are factors weighing against it. To calculate an optimal route, they want to use their navigation system.

This scenario can mostly be handled by modern navigation systems, except for the fact that such a system will either come up with a route with no toll roads at all, or with one that wants the user to use all toll roads that decrease travel time by any time - including just one minute. The system's algorithm is not able to take multiple criteria into account; all but one of the criteria are discarded. Moreover, the chosen criterion is treated as being hard, rather than soft. This results in two paths, of which neither meet the user's requirements and therefore are not optimal solutions to their problems.

Touring bus A touring bus wants to travel from Amsterdam to Berlin using the fastest possible route, with respect to the following requirements: the maneuverability of the roads; busses are too big to be able to make every turn in every road, or roads can be to small for some buses, and the nature richness of the roads; all travelers in the bus are going on a holiday and want to be able enjoy their surroundings during the journey as much as possible.

This is a major problem with current path finding algorithms. The transport-sector may have solved this, but similar limitations apply to cars with trailers.

These scenarios obviously are all possible without the use of a new type of navigation system, but it requires a lot of manual research to determine the route to drive. This is something you don't want as a navigation system manufacturer. Next to these three scenarios a lot of other, similar scenarios exist and all such scenarios can be deduced to one general problem.

Vehicle X has to go from location A to location B. Crucial requirements are H_1 and H_2 and if possible, the calculated route should meet soft criteria S_1, S_2 and S_3. For this, it is relevant to know that S_2 is more important than the other soft criteria.

Current navigation systems are unable to handle these scenarios and no algorithms that are able to determine an optimal route given such criteria without the user having to wait too long time exist at this time.

2.2 Definitions

Before looking for a solution, the problem requires some introduction and a definition.

Finding an optimal route in a road network, given a set of criteria, is a multi-objective discrete and combinatorial problem, meaning that the optimal solution exists in a finite set of possible solutions[5]. This problem can be defined mathematically and we will do so using definitions given by Ehrgott[5]. First, some introduction and necessary definitions are given. In section 2.2.1 we will give a definition of the shortest path, assuming no criteria are provided, other than that the path should be the shortest. In the subsections that follow, we will expand this definition to include multiple criteria and weights. The section will be concluded with a definition of a global function that calculates the best path. To illustrate the definitions, we will use a simplified version of the *Road Trip*-scenario as a guideline. Figure 2.1 shows a graph representing this scenario and Table 2.1 the costs to travel the roads. *Length* is measured in kilometers, *Nature* and *Touristic* on a 0-100 scale. Note that, in our definitions, we assume that there exists at least one path, whilst this may not be the case in real life scenarios.

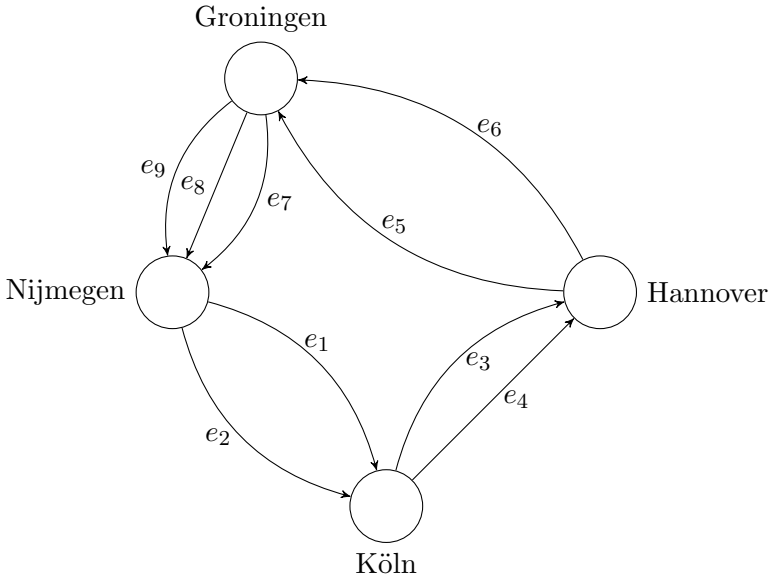

Figure 2.1: A graph of the simplified *Road Trip*-scenario.

Edge	From	To	Length	Nature	Touristic	Toll
e_1	Nijmegen	Köln	160km	30	50	No
e_2	Nijmegen	Köln	190km	60	60	Yes
e_3	Köln	Hannover	330km	30	50	No
e_4	Köln	Hannover	290km	10	20	No
e_5	Hannover	Groningen	340km	50	20	No
e_6	Hannover	Groningen	300km	30	70	No
e_7	Groningen	Nijmegen	210km	40	20	No
e_8	Groningen	Nijmegen	190km	30	30	No
e_9	Groningen	Nijmegen	230km	80	60	No

Table 2.1: The cost of the edges of the graph in Figure 2.1.

Every road map can be represented by a graph. In this graph, all intersections are represented as nodes and all traversable roads as edges. A path in this graph is an ordered sequence of nodes.

Definition 2.1. *A directed graph is a pair $G = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is a finite set of nodes and $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is a set of edges. A directed graph is a weighted directed graph if at least one cost is assigned to each edge. A path in G is defined to be a sequence of nodes $P = (n_0, n_1, ..., n_l)$, for $l \geq 0$, such that $\forall_{n \in P}[n \in \mathcal{N}]$ and $\forall_{i<l}[(n_i, n_{i+1}) \in \mathcal{E}]$. We refer to l as the length of the path, and write $\mathcal{P}$ as the set of all paths in graph G.*

To conveniently determine a path between two arbitrary chosen nodes, we'll have to define a relation between two paths to indicate if one path is preferred above an other. In case we are only interested in the path that is the shortest path, we should be able to do so by their length and all we would need is a function to calculate the length of a path: φ. However, our scenario requires us to take multiple criteria into account. Even though we still are able to calculate the travel cost of this path in this case – each path would have as many costs as criteria – it is harder to compare paths. To solve this problem, we will need a second function, to combine several costs to one single cost: θ. This allows us to use to compare paths in a natural way.

Definition 2.2. *Let $\theta : \mathbb{R}_+^k \to \mathbb{R}_+$ be a monotonic aggegration function from a given tuple of any size to a single value in domain $\mathbb{R}_+$, where $k \geq 1$ is the number of criteria and $\mathbb{R}_+$ is defined as $\mathbb{R}_{\geq 0}$. Let $\varphi : \mathcal{E} \to \mathbb{R}_+^k$ be the cost function that specifies the cost to traverse an edge $e \in \mathcal{E}$. We lift this function to paths inductively as follows: $\varphi(n, n) = 0^k$; $\varphi(n_1, ..., n_{l-1}, n_l) = \varphi(n_1, ..., n_{l-1}) + \varphi(n_{l-1}, n_l)$. Then we can define a binary relation $\prec$ on $\mathcal{P}$ as $\forall_{P_1, P_2 \in \mathcal{P}}[P_1 \prec P_2 \leftrightarrow \theta(\varphi(P_1)) < \theta(\varphi(P_2))]$.*

In previous definitions we assumed that the path we are looking for is the shortest path between two points in the graph. However, it is very well possible that our ideal route is not the shortest path, but the one that scores best on criteria x and y instead. Therefore, we need to generalize our problem and will refer to the shortest path (if applicable) as the optimal path. The shortest and optimal path are guaranteed to be identical in case the only criterion is path length.

2.2.1 Defining the optimal path

Let us consider the two friends planning a road trip. As displayed in the graph, they start their road trip in Nijmegen and Köln, Hannover and Groningen are the major cities they intend traveling to, before arriving back home again. The shortest path is probably not identical to the optimal solution, e.g. depending on the importance of the criteria e_2 may be preferred above e_1, but we will assume it is for now. In that case, the goal is to determine the shortest path and we can define this as follows.

Definition 2.3. *For graph G, let $\mathcal{P}'$ be the set of all paths P from node $n_{start} \in \mathcal{N}$ to $n_{target} \in \mathcal{N}$, where $P_i = (n_{i1}, ..., n_{il})$ and every $n_{ij} \in \mathcal{N}$. Let $\theta : \mathbb{R}_+^k \to \mathbb{R}_+$ be a function that returns its first argument, i.e. $\theta(c_1, ...c_k) = c_1$, for $c \in \mathbb{R}_+^k$. Now we can calculate an optimal solution. A solution $P^* \in \mathcal{P}'$ is called optimal if $\neg\exists_{P \in \mathcal{P}'} [P \prec P^*]$.*

A resulting optimal path is noted as $P*$. Since the only criterion that, at this time, matters in our example is the path length, we have a graph in which every edge has one relevant cost only, being the length of the edge. Because of this, this optimal path P^* then is guaranteed to be the shortest path. The optimal path in Figure 2.1's graph would be (e_1, e_4, e_6, e_8), which has a total cost (length) of 940km.

Definition 2.4. *Let G be a graph, φ a cost function, and θ an aggregation function. Let n_{start} and n_{target} be two nodes of G. An optimal path in G is a path for which the total travel cost between n_{start} and n_{target} is minimized. The problem of finding such a path is referred to as a path finding problem.*

2.2.2 Adding criteria

Now we have given a definition of the shortest path for the road trip. However, this path does not take any of the other listed criteria into account, so those should be added.

Criteria can be divided in two types: soft criteria and hard criteria. Hard criteria are those that should be met no matter the cost, and soft criteria should be taken into account as much as possible. Therefore, all edges not meeting any hard criterion should be removed from the graph. In the road trip example, this is e_2, as it is a toll road[1].

Definition 2.5. *Let $\mathcal{K}$ be the set of all relevant hard criteria. Let $\mathcal{E}_\mathcal{K}$ be the set of all edges that do not meet these criteria. Then, we define $\mathcal{E}' = \mathcal{E}/\mathcal{E}_\mathcal{K}$ as the set of all edges that meet the hard criteria. Now, let graph $\mathcal{G}' = (\mathcal{N}, \mathcal{E}')$ be the graph $\mathcal{G}$ with all the edges that do not meet all of our hard criteria excluded.*

This leaves us with the same road map graph as we had before, excluding all edges that represented toll roads. Although the following is not the case in the road trip example, it is possible that a path between two nodes existed in $\mathcal{G}$, but no longer does in $\mathcal{G}'$, since an edge that did not meet all hard criteria previously connected these nodes. For example, it may no longer be possible to reach an island if you exclude ferries.

[1] Actually, there are no toll roads on the entire route, but for this example, we assume e_2 is one.

Now that we have a graph $\mathcal{G}'$ that does not contain any edges $e \in \mathcal{E}_{\mathcal{K}}$, soft criteria can be added and existing definitions can be expanded to include these criteria. Three soft criteria apply in our example: nature richness of the route, the amount of touristic attractions near the route, and total length of the route. These criteria do not all have the same importance level in practice, so we will add weights later on.

Definition 2.6. *Let $\mathcal{C}$ be the set of all relevant soft criteria and let $k = |\mathcal{C}|$ be the size of $\mathcal{C}$. Then $\mathbb{R}_+^k \ni (c_{i1}, ..., c_{ik})$ is a vector defining the cost of all criteria $c \in \mathcal{C}$ for an edge $e_i \in \mathcal{E}$. All edges $e \in \mathcal{E}$ have such a cost vector.*

In theory, we could say that these definitions would already allow for determining the optimal path given a set of (unweighted) criteria. However, in practice not all criteria are of interchangeable format. For example, nature richness of the route cannot be simply compared to the route's length. Prior to be able to compare values in different units, we should convert them to one generic unit. A score or rating for example.

Definition 2.7. *Let $\alpha_x : \mathbb{R}_+ \to \mathbb{R}_+$ be a valuation function that converts values in any arbitrary unit x to a generic unit and let $\mathcal{A}$ be the set of all such valuation functions. Let $\theta : \mathbb{R}_+^k \to \mathbb{R}_+$ be defined as $\theta(c_1, ..., c_k) = \sum_{i=1}^{k} \alpha_i(c_i)$, where $\alpha_i \in \mathcal{A}$ is the valuation function for a value in unit c_i.*

The α functions may be hard to determine. One possibility is to convert all costs to a cost on a scale of 1 to 100, where 1 is the best cost to have and 100 the worst. In the case of nature richness of the route, we can assume that if this is known for all roads, it is rated on some scale. Thus we just need to multiply and inverse the result, since a higher 'cost' is better in this case. But if a criterion would be e.g. the fuel cost however, one way to rate this is estimating the minimum, maximum and average fuel consumption of the used type of vehicle, for common circumstances (e.g. on a slope, or at a certain speed, or in the city). Then, it is possible to interpolate between these values for each road segment and scale the cost. A similar strategy can be applied in rating the number of touristic attractions. We can convert the road lengths by normalizing them and multiplying the result by 100. At this point it is possible to correctly and consistently find the optimal path $P^* \in \mathcal{P}'$, with respect to unweighted criteria. Table 2.2 shows the costs table, after applying alpha to all edge costs. Edge e_2 is left out because it is a toll road, and since now all toll roads have been removed, the *Toll*-column has been left out.

2.2.3 Adding weights

After the costs per criteria for the full path have been calculated, the criteria should be weighted according to the user's wishes. This means that one criterion can be regarded as more important than an other, by assigning a higher weight to it, or vice versa. To do so, we would need k weights; one for each criterion. Also, φ needs a redefinition, so that it will take weights into account.

Edge	From	To	Length	Nature	Touristic
e_1	Nijmegen	Köln	47	70	50
e_3	Köln	Hannover	97	70	50
e_4	Köln	Hannover	85	90	80
e_5	Hannover	Groningen	100	50	80
e_6	Hannover	Groningen	88	70	30
e_7	Groningen	Nijmegen	62	60	80
e_8	Groningen	Nijmegen	56	70	70
e_9	Groningen	Nijmegen	68	20	40

Table 2.2: The converted cost of the edges of the graph in Figure 2.1.

Definition 2.8. *Given a tuple of weights $\mathcal{W} = (w_1, ..., w_k)$, where each w_i corresponds to one c_i and k is the size of the soft criteria (and weights) set. We will extend our function $\varphi : \mathcal{E} \to \mathbb{R}_+^k$ to include these weights as follows: $\varphi(e) = (w_1 \cdot e_{c_1}, ..., w_i \cdot e_{c_i})$, where e_{c_i} is the cost of the ith criterion of edge e.*

Using this definitions, we can find the path that scores optimal given some criteria and weights. For example, if our weights would range from zero to one, nature richness of the route would get weight 0.9, whilst nearby touristic attractions and route length would get a weight factor of respectively 0.5 and 0.2 for example. These weights favor nature rich roads the most, then touristic attractions, and finally route length. This will give us an optimal route that meets the friends' requirements as far as possible. This gives us the table below.

Edge	From	To	Length	Nature	Touristic	θ
e_1	Nijmegen	Köln	9	63	25	97
e_3	Köln	Hannover	19	63	25	107
e_4	Köln	Hannover	17	81	40	138
e_5	Hannover	Groningen	20	45	40	105
e_6	Hannover	Groningen	18	63	15	96
e_7	Groningen	Nijmegen	12	54	40	106
e_8	Groningen	Nijmegen	11	63	35	109
e_9	Groningen	Nijmegen	14	18	20	52

Table 2.3: The converted and weighted cost of the edges of the graph in Figure 2.1.

Table 2.3 shows us the weighted costs and the aggregated costs (the sum) per edge. If we compare this table with Table 2.1, we see that for the route from Köln to Hannover, e_3 is preferred above e_4, whilst it is 40km longer. However, both the *Nature* and *Touristic*' of e_3 score significantly better than those of e_4. From Hannover to Groningen however, the shorter route is taken, because next to being shorter, there are a lot more touristic attractions near e_6. It is true that e_5 has a higher nature richness, but it is not high enough to compensate. On the final part of the road trip,

e_9 scores over two times better than then e_7 and e_8; which is exactly as we expected. Since this route has a high score on the two most important weighted criteria, its total costs drops rapidly, compared to the others. Now that we have all required information, we can determine the optimal path for our friends to follow: (e_1, e_3, e_6, e_9).

For the sake of this example, our aggregation function simply combines all costs by summing them up, and we have done this per edge instead of per path. However, θ may be defined as a function that returns ∞ if the total length is above a certain limit, e.g. 1000km, and the sum of the costs otherwise. In that case, the path we just found would not be the optimal path, since it has a total length of 1020km and its cost would increase to infinity.

2.2.4 Optimal path

Now that we have all definitions to find an optimal path in a graph, we can compose a partial function $optimal\_path : (G, \mathcal{K}, \mathcal{C}, \mathcal{W}, \mathcal{A}, \theta, n_{start}, n_{target}) \to \mathcal{P}$ that calculates the optimal path given a graph $\mathcal{G}$, a set of hard criteria $\mathcal{K}$ and soft criteria $\mathcal{C}$ with linked weights $\mathcal{W}$, a set of valuation functions $\mathcal{A}$, an aggregation function θ, and a start and target node $n_{start}, n_{target} \in \mathcal{N}$. The path P determined by this function meets our requirements, meaning that no path $P' \in \mathcal{P}$ exists, such that $P \preceq P'$ and P does not contain any of the edges that fail to meet the hard criteria. Sub-optimal paths or paths that do not meet all hard criteria are not given.

Definition 2.9. *Function* $optimal\_path : (G, \mathcal{K}, \mathcal{C}, \mathcal{W}, \mathcal{A}, \theta, n_{start}, n_{target}) \to \mathcal{P}$ *finds a path* P *such that (with* $k = |\mathcal{C}|$*) holds:* $\theta(\varphi(P)) = \min_{P' \in \mathcal{P}}(\theta(\varphi(P')))$*, and* $\forall_{e \in \mathcal{E}_P}\, [e \notin \mathcal{E}_\mathcal{K}]$*, where* φ *is the cost function,* $\mathcal{E}_P$ *the set of all edges in* P *and* $\mathcal{E}_\mathcal{K}$ *the set of all edges that do not meet the hard criteria.*

Note that multiple optimal paths can be found in the same graph, for the same start and target node. This was not the case in the *Road Trip*-example, but, especially in larger graphs, this is very well possible. Therefore, *optimal_path* gives us *an* optimal path, rather than *the* optimal path.

3 Complexity

Determining the optimal path given multiple criteria is $\mathcal{NP}$-complete. We can prove this by polynomially reducing the Knapsack decision problem to a simple optimal path problem and will do so after theorem 3.1. First, however, we will introduce the Knapsack problem. Finally, we will see that it is possible to reduce the path finding problem's complexity drastically by putting some restrictions on θ, allowing for reducing the multicriteria problem to a single-criterion problem.

The Knapsack problem is a well known problem. It involves maximizing the value whilst not exceeding a (predefined) maximum cost. It is called the Knapsack problem because of its analogy with a real-life Knapsack problem: when filling your knapsack, you cannot take all desired items with you since the knapsack has a limited size. So, you want to put as many valuable items in your knapsack. Each item has a different size and value, so there are many (im)possible combinations of items.

Definition 3.1. *Let $\mathcal{O}$ be a finite set of objects (items). We define the size function $s : \mathcal{O} \to \mathbb{R}_+$ and the value function $v : \mathcal{O} \to \mathbb{R}_+$. The Knapsack problem describes the question whether or not subset $\mathcal{O}' \subseteq \mathcal{O}$ exists, such that $\sum_{o\in\mathcal{O}'} s(o) \leq C$ and $\sum_{o\in\mathcal{O}'} v(o) \geq V$.*

In short, the problem is finding a combination of items, such that the sum of the size of all items is below the knapsack's maximum capacity C, but above the desired minimum value V. Finding an answer to this question is is $\mathcal{NP}$-hard[6].

Theorem 3.1. *Determining the optimal path in a graph $G \in \mathcal{G}$, given $k > 1$ criteria is NP-complete.*

Proof. We can prove that our path finding problem is $\mathcal{NP}$-complete by polynomially reducing the Knapsack problem to a rather trivial optimal path problem. To do so, we will need to construct a graph that represents a Knapsack problem.
Let $\mathcal{O}$, V, C, s, v be an instance of the Knapsack problem as defined in Definition 3.1. Let

$$
\begin{aligned}
\mathcal{N} &:= \{n_0, ..., n_m\} \\
\mathcal{E} &:= \bigcup_{i=1}^{m} \left((n_{i-1}, n_i) \cup (n_{i-1}, n_i)'\right) \\
s(e) &:= \begin{cases} n_i^1 & \text{if} \quad e = (n_{i-1}, n_i) \\ 0 & \text{if} \quad e = (n_{i-1}, n_i)' \end{cases} \\
v(e) &:= \begin{cases} 0 & \text{if} \quad e = (n_{i-1}, n_i) \\ n_i^2 & \text{if} \quad e = (n_{i-1}, n_i)' \end{cases} \\
\theta(s, v) &:= \begin{cases} 1 & \text{if} \quad s > C \vee v < V \\ 0 & \text{if} \quad s \leq C \wedge v \geq V \end{cases} \\
n_{start} &:= n_0 \\
n_{target} &:= n_m \\
m &\geq 2
\end{aligned}
$$

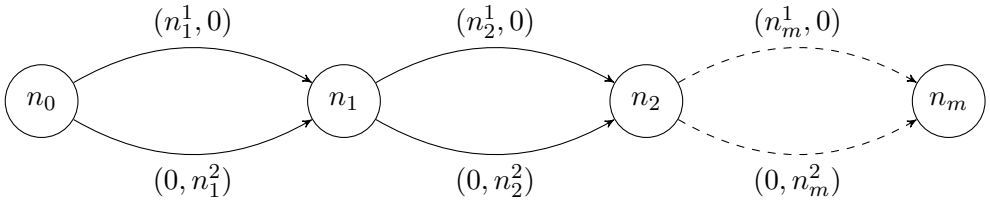

Figure 3.1: A visual representation of the graph of the proof of Theorem 3.1.

This graph is shown in figure 3.1. We have m nodes, that are all connected to the next node (if any) by two edges. Each node $n \in \mathcal{N}$ represents an 'item' in the knapsack problem and each edge $e \in \mathcal{E}$ represents the decision to either include or exclude the item. We have two criteria, being the size s and the value v. Our model map θ is designed in such a way that it returns a cost of 1 if the total size exceeds a maximum capacity C or the total value is below V, and a cost of 0 if none of these conditions are violated. Both C and V are chosen to represent the cost respectively value of an other path in our graph.

Let $P \in \mathcal{P}$ be a path from n_{start} to n_{target}. Let $P^s = \varphi(P)^1$ the total size of all 'items' in the path and $P^v = \varphi(P)^2$ the total value, where we take φ as defined in Definition 2.2. Then

$$f(P) = \theta(P^s, P^v) \leq \theta(C, V) \tag{1}$$

if and only there exists $x \in \{0,1\}^m$ such that $\theta\left(\varphi\left(n^1 \cdot x^T, n^2 \cdot (e-x)^T\right)\right) \leq \theta(C, V)$, where $e = \{1\}^m$. Note that $x_i = 1$ if $(n_{i-1,}, n_i) \in P$ and $x_i = 0$ if this is not the case, i.e. $(n_{i-1}, n_i)' \in P$. Since the number of paths that satisfy (1) is equal to the number of solutions to corresponding Knapsack problem, finding the optimal solution for a multicriteria path finding problem is $\mathcal{NP}$-complete. □

Above proof is an adaptation of Ehrgott[5], who has his proof based upon a proof from Serafini[14]. Ehrgott also proves $\mathcal{NP}$-completeness for acyclic directed graphs, but his definition of an optimal path is different from ours. Whilst an optimal path for him is a path that scores better at one criterion and at least as good on all criteria as an other path does, our definition states that an optimal path has a lower cost than any other path, using some function θ that combines all costs to one cost.

3.1 Criteria reduction

The prove given in previous section allows function θ to be non-additive, like averaging or something similar. For example, it could be defined as a function that sums the path costs of all k criteria and returns that as total cost, unless the cost of criteria x passes a certain threshold. In that case, the total cost that is returned could be e.g. positive infinite (if a maximum cost is exceeded) or negative infinite (if other paths don't matter anymore). Such functions would give users the option to provide maximum or minimum values for criteria, but requires k total costs for a path. This is exactly what we have proven in previous section to be $\mathcal{NP}$-hard.

Ideally, we would be able to pre-process any graph G and 'merge' the costs of each edge into one cost, instead of doing so on path level, as Tarapata does with a so called metacriterion function[15]. This allows usage of existing, very fast path finding algorithms designed to process one criterion. Henig claims that not all attributes can be aggregated. Multiple attributes have different representations and not al these representations can be combined. This is especially the case for cardinal (e.g. travel duration) versus ordinal (e.g. nature richness) attributes[9]. However, in definition 2.7 we have introduced the function set $\mathcal{A}$ that contains all functions that convert an arbitrary unit to a generic unit. This includes ordinal 'units'. We then can combine multiple criteria by using θ and the α's as our metacriterion function.

Now that we are able to aggregate multiple criteria, we need to restrict θ. Since this function should now be applied on an edge instead of on a path, threshold-based mapping functions are no longer meaningful. A single edge will generally not exceed such a threshold and even if it would, it would make no real sense. An edge cost exceeding some threshold would also be removed if this threshold would have been translated to a hard criterion.

Definition 3.2. *We restrict $\theta : \mathbb{R}_+^k \to \mathbb{R}_+$ to any function for which holds that it is a homomorphism, so: $\forall_{r_1^k, r_2^k \in \mathbb{R}_+^k} \left[\theta\left(r_1^k\right) + \theta\left(r_2^k\right) = \theta\left(r_1^k + r_2^k\right)\right]$. We will refer to a multicriteria path finding problem that is restricted by such a θ as a restricted multicriteria path finding problem.*

This rests us only to process graph G and apply θ on all edges, prior to determining the optimal path. The complexity of doing this can be calculated easily: if we have a total of m nodes in our graph and all nodes can be connected twice – we have a directed graph – to each other node, there are m^2 edges. However, nodes cannot be connected by an edge to itself, so the maximum number of connections is $m^2 - m$. We have to apply the mapping function on all edges, giving us a complexity of $O(m^2 - m) = O(m^2)$. In addition to this, we only have the complexity of the algorithm used to find the 'shortest' path in a standard directed graph. This means the overhead for restricted multicriteria path finding problems is $O(m^2)$, which is significantly better than non-restricted path finding problems. We can even improve this by combining the criteria 'on the flow' (i.e. only when the algorithm reaches that edge), instead of processing the entire graph prior to executing a path finding problem. As the complexity of our θ function is $O(1)$, this approach has the same complexity as that of the path finding algorithm used.

4 Algorithms

For variants of the problems such as described in this thesis, many efficient algorithms have been developed. Furthermore, we have suggested a way to reduce a multicriteria path finding problem to a single criteria path finding problem, by restricting θ, as explained in Definition 3.2. However, we want to be able to solve non-restricted multicriteria path finding problems. In this section we will have a look at two well known path finding algorithms: the Label Setting algorithm and Contraction Hierachies[11][7]. After having introduced these algorithms, we the will propose a combination of these two algorithms that should be able to solve such path finding problems more efficiently.

4.1 Label Setting Algorithm

The basic Label Setting Algorithm is an algorithm that does not calculate an optimal path from a start node n_{start} to a target node n_{target}, but rather calculates an optimal path from the start node to every other node in the graph. After it has done this, it returns an optimal path from n_{start} to n_{target} – providing there is any – and discards the others. So, let us illustrate this, using the graph of our *Road Trip*-scenario, in Figure 2.1. If we would ask the optimal path between Nijmegen and Hannover, it would calculate the optimal path from Nijmegen to Köln, from Nijmegen to Hannover, and from Nijmegen to Groningen, even though we don't need two of those paths. This is the case, since we do not know for sure if a path is optimal, before we have found all paths, because of the possibility that a path of length 1000 is optimal whilst a path of length 10 is not.

The algorithm is named the Label Setting algorithm because it determines all optimal paths using so called labels. These labels are created by the algorithm and assigned to exactly one node per label. A node may have multiple labels. Every label L of a node n references a label p, that is the predecessor label of n, on a path containing n. This means that if we have a path (v, u, w), the label of w references u's label as its predecessor label, and the label of u references v's label as its predecessor node. So if we have determined the labels of all nodes, a simple backtracking algorithm can determine an optimal path.

A label $L = (c_1, ..., c_k, n, p)$ is a tuple with $k+2$ attributes, with k being the number of criteria. Every label contains the total cost (per criterion) c_i of the path $(n_{start}, ..., n)$, the node n itself, and the label's predecessor p. To access the node a label L belongs to, we write L_n, and we refer to the other attributes in a similar way. For the predecessor label we write L_p, for the cost tuple of a label L_c, and for the cost of the ith criterion we write L_{c_i}. As we can see in Definition 4.1, we can compare labels the same way as we compare paths.

Definition 4.1. *We write $\mathcal{L}$ as the set of all labels. We define a binary relation $\prec$ on $\mathcal{L}$ as $\forall_{L,L' \in \mathcal{L}}[L \prec L' \leftrightarrow \theta(\varphi(L_c)) < \theta(\varphi(L'_c))]$. If $L \prec L'$, we say L dominates L'.*

The Label Setting algorithm starts at node n_{start} and spreads out through the entire graph. Two lists are created initially: a list of temporary labels (TL) and a list of

permanent labels (PL). A label in PL will never be removed, whilst a label in TL will be. An initial label, belonging to n_{start}, is created and added to the temporary label list. We then take and remove the only non-dominated label L from the temporary label list and add it to the permanent label list. For each node n' that is directly connected to the current node L_n, the total cost to reach n' is calculated and a new label L' is created. This label contains its predecessor (which is L), the total cost to reach n', and of course n' itself. If L' is not dominated by any label in TL that represents the same node as L' does, L' is added to TL and, if it dominated a label, that label is removed. This means that if a node is reached multiple times, the labels with the lowest costs are kept and the others are discarded. Above is repeated as long as the temporary label list is not empty, causing the algorithm to stop as soon as there are no more nodes to check. At this time, all nodes n then have a label assigned to it, containing the cost to reach n – starting at n_{start} – and its predecessor. All optimal paths can then be retrieved by backtracking the path from n to n_{start} using the predecessor labels of the labels in PL. Algorithm 4.1 shows the pseudo code for the Label Setting algorithm[5].

Algorithm 4.1 Multicriteria Label Setting algorithm

Input: A directed graph $G = (\mathcal{N}, \mathcal{E})$ with k criteria per edge; nodes n_{start}, n_{target}

```
L := (0, ..., 0_k, n_start, nil)                  # Initialize L on n_start
TL := {L}                                         # Initialize TL with L
PL := ∅                                           # Initialize PL empty
while TL ≠ ∅ do                                   # While TL is not empty:
  L := L ∈ TL, with ∄_{L'∈TL}[L' ≺ L]             #   Let L be the lowest cost label
  n := L_n                                        #   Let n be the label's node
  TL := TL/{L}                                    #   Remove L from TL
  PL := PL ∪ {L}                                  #   Add L to PL
  for all n' ∈ N | (n, n') ∈ E do                 #   For all connected nodes:
    c := φ(n, n')                                 #     Let c be the cost of (n, n')
    L' := (L_{c_1} + c_1, ..., L_{c_k} + c_k, n', L)  #   Let L' be a new label for n'
    if ∄_{L''∈TL∪PL}[L''_n = n' ∧ L'' ≺ L'] then  #     If this label is not dominated:
      if L' ≺ L'' then                            #       If this label dominates:
        TL := TL/{L''}                            #         Remove L'' from TL
      end if                                      #       End if
      TL := TL ∪ {L'}                             #       Add L' to TL
    end if                                        #     End if
  end for                                         #   End for
end while                                         # End while
Backtrack from n_target to n_start using the predecessor labels of the labels in PL
```

Output: All optimal paths from n_{start} to n_{target}

To illustrate this, let us have a look at the graph displayed in Figure 4.1. This graph contains 6 nodes and 9 edges, and each edge has $k = 4$ (comparable) costs. The graph is created by Martins E. Q. V., but its weights are slightly adjusted to better suit the examples in this thesis[11]. Let the aggregation function θ be the sum function, i.e.

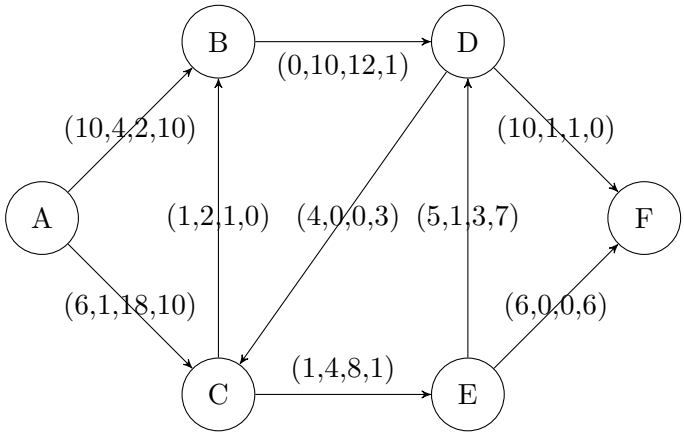

Figure 4.1: An example graph to explain the Label Setting algorithm.

$\theta(c_1, c_2, c_3, c_4) = c_1 + c_2 + c_3 + c_4$. We will use the Label Setting algorithm to find the optimal path from node $n_{start} = A$ to $n_{target} = F$, for readability reasons we will only once fully write down a label and refer to it afterwards as L_{index}. All labels are listed in Table 4.1.

After initialization of the algorithm, $L_0 = (0, 0, 0, 0, A, nil)$, $TL = \{L_0\}$, PL is empty, and we enter the `while`. We pick L_0, set n to A, empty TL and add L_0 to PL. Two nodes are directly connected to A, being B and C. The order in which these nodes are handled does not matter, so let us start with B. After looking up the costs of edge (A, B), we create a new label (L' in the algorithm) $L_1 = (10, 4, 2, 10, B, L_0)$ and, can add it to TL. Since TL is empty at this time, there was no dominating label, so we did not have to discard L_1 or any other label. We reached the end of the for loop and can now do the same for node C: we create a new label $L_2 = (6, 1, 18, 10, C, L_0)$ and add it to TL. Note that we do not see L_1 as a label dominating L_2, since they belong to different nodes. TL does now contain labels L_1 and L_2, and $PL = \{L_0\}$.

At the next iteration, we select the label that is not dominated. Label L_1 has a total cost of $10 + 4 + 2 + 10 = 26$ and L_2 has a total cost of $6 + 1 + 18 + 10 = 35$, so we continue with the former. We remove L_1 from the temporary labels list and add it to the permanent labels list. The only directly connected node of $L_{1_n} = B$ is D and its cost is $(0, 10, 12, 1)$, so the label for D will be $L_3 = (10 + 0 = 10, 4 + 10 = 14, 2 + 12 =$

Label	Costs	Node	Pre	Label	Costs	Node	Pre
L_0	(0,0,0,0)	A	*nil*	L_1	(10,4,2,10)	B	L_0
L_2	(6,1,18,10)	C	L_0	L_3	(10,14,14,11)	D	L_1
L_4	(7,5,26,11)	E	L_2	L_5	(7,3,19,10)	B	L_2
L_6	(14,14,14,14)	C	L_3	L_7	(20,15,15,11)	F	L_3
L_8	(12,6,29,18)	D	L_4	L_9	(13,5,26,17)	F	L_4

Table 4.1: The labels Algorithm 4.1 will produce for the graph of Figure 4.1.

$14, 10+1 = 11, D, L_1$). This label is not dominated, so we add it to TL, now $\{L_2, L_3\}$.

The total cost of L_3 is $10+14+14+11 = 49 > 35$, so we continue with label L_2 and add it to PL. The directly connected nodes are B and E; let us first handle E. Similarly as we did for nodes B and D, we create a new label $L_4 = (7, 5, 26, 11, E, L_2)$ and add it to TL. Now for B, after having created a new label $L_5 = (7, 3, 19, 10, B, L_2)$, we see that it is dominated by L_1: both labels belong to node B and $7+3+19+10 > 10+4+2+10$. Therefore, we discard L_5 and move on with the next label in $TL = \{L_3, L_4\}$.

We look for the label that is not dominated, and find that $10 + 14 + 14 + 11 = 7 + 5 + 26 + 11$, so the label we pick does not matter. Let us choose L_3; the first things we then do are removing L_3 from the temporary label list and adding it to the permanent label list. This leaves us with $TL = \{L_4\}$ and $PL = \{L_0, L_1, L_2, L_3\}$. Node D is directly connected to C and F. For C, we create label $L_6 = (14, 14, 14, 14, C, L_3)$ and after seeing that this label is dominated by L_2, so we discard it. We did not yet encounter node F, so we create label $L_7 = (20, 15, 15, 11, F, L_3)$ and add it to TL.

This iteration, we pick the non-dominated label L_4, removing it from the temporary labels list and add it to the permanent labels list. This label's node (E) is directly connected to D and F. The label we create for D is $L_8 = (12, 6, 29, 18, D, L_4)$ and this label is clearly dominated by L_3, so we discard it. However, F's label, $L_9 = (13, 5, 26, 17, F, L_4)$ does not dominate and is not dominated by L_7, so we leave L_7 in TL, but add L_9.

Finally, we execute the last two iterations: we take the last remaining labels (L_7 and L_9) from TL and add them to the permanent labels list. None of these label nodes have outgoing connections to any other node in the graph, so we reach the end of the `while` without entering the `for`, and with $TL = \emptyset$, the algorithm exit the while with $PL = \{L_0, L_1, L_2, L_3, L_4, L_7, L_9\}$.

Now we can construct a path from A to F by backtracking, using the nodes of the labels in the permanent label list. We search for the label L with $L_n = F$, then we recursively select a predecessor label of L, until all predecessor labels are *nil*, in which case we reached the start node. In our example, we find that F has two labels, L_9 and L_7. This means that there are two optimal paths to this node. After backtracking, the label paths we get are (L_7, L_3, L_1, L_0) and (L_9, L_4, L_2, L_0) (the rest of the labels have one predecessor). All that rests us to do now, is reverse these lists and take the node of each label, this gives us all optimal paths: $P_1 = (A, B, D, F)$ and $P_2 = (A, C, E, F)$.

4.2 Contraction Hierarchies

The basic idea of a Contraction Hierachy (CH) is to create hierarchic levels of the graph, in which each level contains shortcut edges between two nodes, nodes that were not directly connected on the previous level. Figure 4.2 shows an example of this: we have a graph G with nodes v, u, w and edges (v, u) with cost a and (u, w) with cost b. The left graph in the figure shows this situation. If we would then apply the Contraction Hierarchy construction algorithm, we would get the shortcut edge (v, w), with cost $a + b$, providing that it is an optimal path between v and w. In case there already exists an edge between the shortcut nodes, the algorithm assigns the lowest of

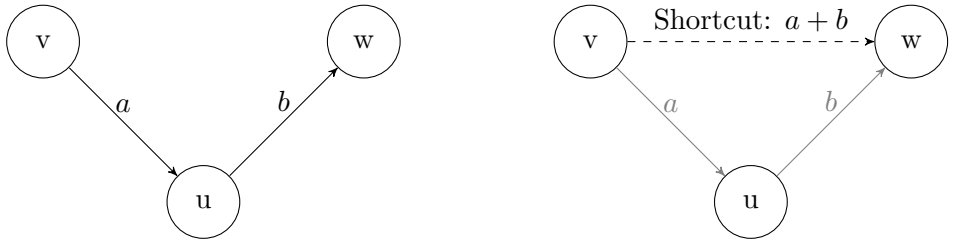

Figure 4.2: An illustration of a contraction. Left before, right after contraction.

the two costs to the existing edge[7]. The algorithm removes per level all redundant edges after creating the shorcut edges. In this case, this would be edge a and edge b, since they no longer need to connect u to v and w, because of the shortcut edge. The right side of the figure shows the graph after contraction: the shortcut edge created is dashed and the original edges are displayed as removed, i.e. grayed out.
Contraction Hierarchies are constructed by executing two main algorithms. The first algorithm is responsible for sorting all nodes in graph G by importance, which is required to be able to let the second algorithm optimally contract the graph. We will not discuss the first algorithm in detail, as it is a complex procedure and independent of the search algorithm. For details on the sorting algorithm and more explanation on the importance of nodes, see Geisberger's Diploma Thesis on Contraction Hierarchies[7].

Definition 4.2. *To be able to conveniently compare two nodes by importance, we define a binary relation $<$ on $\mathcal{N}$ as $\forall n_1, n_2 \in \mathcal{N}[n_1 < n_2 \leftrightarrow n_2$ is 'more important' then $n_1]$. Similarly, we define a binary relation $>$ on $\mathcal{N}$ as $\forall n_1, n_2 \in \mathcal{N}[n_1 > n_2 \leftrightarrow n_1$ is 'more important' then $n_2]$. We say an edge $(n_1, n_2) \in \mathcal{E}$ is 'important' if $n_1 < n_1$.*

With the importance relations from Definition 4.2, we can move on to the second algorithm. This algorithm involves removing redundant edges and creating shortcut edges, i.e. creating the contraction hierarchy. In Algorithm 4.2 the pseudo code of the basic algorithm to contract the graph is shown. As deleting nodes from the graph involves a more complex algorithm and is not required to find an optimal solution, the pseudo code abstracts from this action.

As we can see, it is a rather simple algorithm with no confusing or complex steps. However, the `if` in the algorithm involves some complex calculations and will consume most of the total execution time. To check if a path $P = (v, u, w)$ is an optimal path from v to w, we need to perform a so called *local search*. Technically, this means that, for each execution of the `if`, we need to execute a path finding algorithm with $n_{start} = v$ and $n_{target} = w$. This algorithm has to discard all paths P' for which holds that its cost exceeds the cost of P, i.e. $\theta(\varphi(P')) > \theta(\varphi(P))$. This seems trivial, but path P can have a very high cost, e.g. if one of the edges involves a highway without exits for over a hundred kilometers. If P's cost is high enough, we may end up searching large parts of the graph; this is very time consuming. That is why we need to limit the local searches, to prevent them from becoming excessively expensive. To do so, we can either limit the number of nodes of a potential better path, and discard it if this

Algorithm 4.2 Basic Contraction algorithm

Input: A directed graph $G = (\mathcal{N}, \mathcal{E})$

```
for all u ∈ N, sorted by <, ascending do   # For all nodes in the graph, sorted:
  for all (v,u) ∈ E, v > u do              #   For all u's not important edges:
    for all (u,w) ∈ E, w > u do            #     For all u's important edges:
      if ∄_{P(v,...,w)∈P} [P ≺ (v,u,w)] then   #   If (v,u,w) is an optimal
        e := (v,w)                         #     path from v to w:
        E := E ∪ {e}                       #       Add a shortcut edge to E,
        φ := φ ∪ {(e, φ(v,u,w))}           #       with weight φ(v,u) + φ(u,w)
      end if                               #     End if
    end for                                #   End for
  end for                                  # End for
end for                                    # End for
```

number is exceeded, or limit the number of edges on such a path: a hop limit. The former method leads to more dense contracted graphs and does not speed up actual contraction (if used in algorithm 1), so we will use a hop limit[7].

There are two different hop limit searches: *Fast Local 1-Hop Search*, and *1-Hop Backward Search*. The 1-Hop Search assumes most edges in the graph are an optimal path between the two nodes they connect, which is usually the case in road networks. The backward search is an a-Hop Search, with $a \geq 2$. In this search, a path finding algorithm explores all nodes that are within reach of $a - 1$ edges. In this process, it assigns – like in the Label Setting algorithm – a label to each visited node n, containing the nodes predecessor, and the cost to travel from v to n. Afterwards, a backward 1-Hop Search is performed, starting at the target node, w. This search looks for all surrounding nodes that have a label and selects the node with the lowest total cost. The resulting path $P' = (v, ..., w)$ is an optimal path between v and w, of length $l \leq a$.

If $P = P'$, i.e. if we did not found a path better than P, we assume that P is optimal. Note that it might occur that, although we assumed P is optimal, it is not. This is unfortunate, but does not prevent us from finding an optimal path[7]. We then create a shortcut edge (v, w), and store the middle node u of P at w, so that we can reproduce the lowest level (original) graph from the contracted graph. We refer to this node as w_u.

After the graph has been contracted (which can be done multiple times), we can perform optimal path searches. We do so by using an interleaved, bidirectional search. Two path finding algorithms start from respectively the start node n_{start} and the target node n_{target}. So the former performs a forward search, whilst the latter performs a backward search, following all edges in opposite direction. If multiple edges have the same total cost, the searches respectively prioritize the edges towards the more important, and from the less important nodes. The algorithms only relaxes edges that are directed toward a respectively higher or lower node in the hierarchy. They terminate if their paths meet and cannot find a better path. Exactly how it is determined if there is a better path depends on the path finding algorithm used.

Now that we have found a path in the contracted graph, we need to unpack the shortcut edges to extract the actual path. This can be done by a simple recursive routine Geisberger presented[7]. Earlier this section, we said we store the middle node of a path (v, u, w) in the last node. This means we can always construct the original path from a shortcut edge (v, w), namely (v, w_u, w). If we do this recursively, we will eventually end with the original graph. After unpacking all edges of the path is complete, we have found an optimal path from n_{start} to n_{target}, using contraction hierarchies.

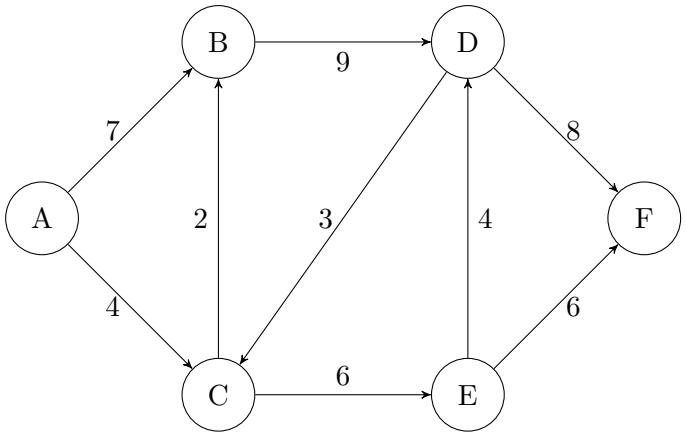

Figure 4.3: An example graph to explain Contraction Hierarchies.

We will give an example of the construction of a Contraction Hierarchy using the graph in Figure 4.3. In this example, we will abstract from the algorithm that is used to determine optimal paths, and just provide an optimal path instead. Furthermore, we assume that the importance ordering of the nodes is A, B, C, D, E, F. The path we are looking for has $n_{start} = A$ and $n_{target} = F$. For the local searches, we use a *1-Hop Backward Search* with $a = 3$.

With the first iteration of the first `for`, we select the most important node, being A. Then we pick the first node that is less important then A, and is directly connected to this node. In practice, 'less important' means that the node has not yet been used by the algorithm. For A, these nodes are B and C, and since $B > C$, we start with B. Now we do the same for this node and select D, since that is the only node directly connected to B. Now we can construct a path (A, B, D) and we need to check if this path is optimal, or a better path exists. As we can see, there is one other path from A to D that has a lower cost then $7 + 9 = 16$. Path $P = (A, C, B, D)$ has a cost of 15. Our hop limit is 3 and that is also the length of P, so this path is found and we do not add a shortcut edge (A, D). After this, we jump back to the beginning of the second loop, to handle node the next node, C. This node is directly connected to both B and E, but B is more important then C, so we move on to E. We now need to check if there is a better path from A to E, then (A, C, E), which is not the case, so we add a shortcut edge (A, E) with cost 12.

Back in the main loop, node B is next. This node has only one directly connected node, D, and this also accounts for node D, which has F. So we search for a path from B to F that has a cost lower then 17, and see there is one: $P = (B, D, C, E, F)$. However, the local search algorithm will not find this algorithm, since we are using a *1-Hop Backward Search*, that has a hop limit of 2. Since P is not found, we add a second shortcut edge: (B, F). The cost of this edge is 17.

The next node is C, and the only node that is less important and directly connected is E. For E, this holds for F, so we are looking for a path that is better then (C, E, F) and we see there is none, so a shortcut edge (C, F) is added, with cost 12.

In the fourth iteration, we are at node D, which has F as a node that meets the importance and connection requirements, but there are no nodes that are less important then F, so the third `for` will not be entered and no shortcut edges will be added. The same accounts for node E at the next iteration. The final iteration handles F and encounters the same situation. Figure 4.4 shows the resulting graph. All shortcut edges are dashed, and edges (A, C), (B, D), (C, E), (D, F) and (E, F) have been grayed out, since they are removed.

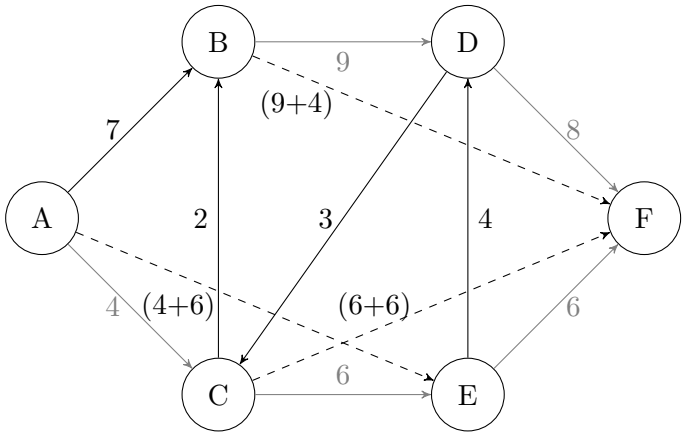

Figure 4.4: The example graph of Figure 4.3 after contraction.

We can repeat this procedure multiple times, this gives us an even more dense graph every time. Note that the node importance order of this graph is chosen arbitrarily, so the order is not optimal. Therefore, the graph in Figure 4.4 may seem a little awkward.

To determine an optimal path from A to F in this graph, we can apply any (bidirectional) path finding algorithm, as long as we modify the algorithm in such a way that it only relaxes edges that are directed towards a higher level when performing a forward search, and only relazes edges that are directed towards a lower level when performing a backward search. Geisberger uses Dijkstra's search, so we will use that as well here[7]. We start two searches: a forward search that starts at A (in the highest level graph A appears – which is the contracted graph in this case) and a backward search starting at F (in the lowest level graph). At first, the forward search follows the edge (A, B), as it has a lower cost then the shortcut edge (A, E), and the edge (A, C) does not exist

at this point. At the same time, the backward search follows the edge (E, F). It then traverses edge (D, F), whilst the forward search travels (A, E). At the next iteration, the forward search relaxes (A, E), as there can not be a better path from A to E and E appears in a lower level graph, and the backward search relaxes (E, F), as there can not be a better path from E to F and E appears in a higher level graph. Now we have found two paths: (A, E) and (E, F). We concatenate these paths to (A, E, F), unpack the shortcut edge (A, E), and extract the optimal path (A, C, E, F).

4.3 Multicriteria Contraction Hierarchies

The Label Setting and Contraction Hierarchy algorithms can be combined to one algorithm. In this section, we will combine these algorithms and give a proof sketch that our new algorithm is correct. Providing an extensive mathematical proof will be too time-consuming and is therefore outside the scope of this thesis.

The main idea of the combined algorithm is that we have a hierarchical Label Setting algorithm. To do this, we will use Contraction Hierarchies, which we create and use to find an optimal path, using the Label Setting algorithm. However, this requires a few adaptations to the algorithm:

1. The original Contraction Hierarchies use a modified Dijkstra version to contract the graph and find optimal paths. This is fine for a regular path finding problem, one that only takes path length into account. However, we have multiple criteria and Dijkstra's algorithm cannot handle this (efficiently). To handle this, we will use an adapted version of the Label Setting algorithm.

2. For construction of the hierarchies, the contracting algorithm uses an a-Hop Search. In most cases, $a = 1$ will suite the purpose, but because of our multiple criteria it is more likely it will not, since most edges (v, w) are not an optimal path from v to w. This makes 1-Hop Search unsuitable for our purpose. We could try to overcome this problem by choosing a bigger a, but it is very well possible that, even with $a = 10$, we will not find a path better then (v, u, w), although there is one. Experiments should tell us what a is best in a real life road network.

Now that we know this, we can change the Label Setting algorithm such that it takes a Hop Search into account. For this, the algorithm needs to return 'no path' if it cannot find a path within a hops. Algorithm 4.3 shows the updated pseudo code.

A total of three changes are introduced with this new algorithm. First of all, we have appended the label with an extra value, which we refer to as L_a. This value is the $k + 3$th attribute of a label L, and stores the path length from n_{start} to L_n. Note that the path length is the edge count of the path, and is unrelated to the cost of the path. The second change involves the `if` statement. This statement now includes an extra check $\theta(\varphi(L'_c)) < K$, so that the total cost of a found path does not exceed a given maximum cost K. Next to this, it also includes an extra check $L'_a < a$, that ensures the newly created label does not belong to a node that is the ath node of a path, i.e.

Algorithm 4.3 Multicriteria Label Setting algorithm with a-Hop Search

Input: A directed graph $G = (\mathcal{N}, \mathcal{E})$ with k criteria per edge;
start node n_{start}; hop limit $a \geq 1$; max (aggregated) cost K

$L := (0, ..., 0_k, n_{start}, nil, 0)$ # Initialize L on n_{start}
$TL := \{L\}$ # Initialize TL with L
$PL := \emptyset$ # Initialize PL empty
while $TL \neq \emptyset$ **do** # While TL is not empty:
  $L := L \in TL$, with $\nexists_{L' \in TL}[L' \prec L]$ # Let L be the lowest cost label
  $n := L_n$ # Let n be the label's node
  $TL := TL/\{L\}$ # Remove L from TL
  $PL := PL \cup \{L\}$ # Add L to PL
  for all $n' \in \mathcal{N} \mid (n, n') \in \mathcal{E}$ **do** # For all connected nodes:
    $c' := \varphi(n, n')$ # Let c' be the cost of (n, n')
    $L' := (L_{c_1} + c'_1, ..., L_{c_k} + c'_k, n', L, a + 1)$ # Let L' be a new label for n'
    if $\nexists_{L'' \in TL}[L''_n = n' \wedge L'' \prec L']$ # If this label is not dominated
     and $\theta(\varphi(L'_c)) < K$ # and the cost is lower then K
     and $L'_a < a$ **then** # and (n, n') is not the ath edge:
      if $L' \prec L''$ **then** # If this label dominates:
        $TL := TL/\{L''\}$ # Remove L'' from TL
      end if # End if
      $TL := TL \cup \{L'\}$ # Add L' to TL
    end if # End if
  end for # End for
end while # End while

Output: PL: The labels of all nodes reachable with $x < a$ steps from n_{start}

that the path does not exceed length a. Finally, at the end of the outer `while`, we no longer backtrack the path from any of the nodes in PL. Instead, we output a list of all labels that plausibly represent an optimal path from n_{start} to any target node n_{target}.

In addition to above version of the Label Setting algorithm, we need to construct an other adaptation. Since we are using a-Hop Search with $a > 1$, and as the adapted algorithm shows, we only search $a-1$ steps forward. The ath step needs to be performed backwards, for this we require an algorithm that does a backward search and matches the found nodes with all resulting nodes from the forward searching algorithm. The result of this change is shown in Algorithm 4.4.

Parts of the code have been removed from Algorithm 4.3 algorithm to get to this new one, and different input is required. Instead of a start node, it now requires a target node. Next to this, a list of labels is expected; this list should contain all values obtained by executing Algorithm 4.3. As for the code changes, we are only looking one step ahead in the graph, rendering the list of labels that plausibly represent an optimal path obsolete. Therefore we removed the `while` and all other TL- and PL-related code. The for loop has been replaced by a loop that accepts all labels that represent a node that is directly connected to the target node. If such a label L' represents a path

Algorithm 4.4 Multicriteria Label Setting algorithm with 1-Hop Backward Search

Input: A directed graph $G = (\mathcal{N}, \mathcal{E})$ with k criteria per edge;
labels PL; target node n_{target}; max (aggregated) cost K

```
L := (0, ..., 0_k, n_target, nil, 0)          # Initialize L on n_target
for all L' ∈ PL | (L'_n, n_target) ∈ E do      # For all connected nodes in PL:
  c' := L'_c + φ(L'_n, n_target)              #   Let c' be the total cost to n_target
  if θ(c') < L'_c and L_p ≠ nil               #   If this is the best cost
   and θ(c') < K then                         #    and the cost is lower then K:
    L := (c', n_target, L', L'_a + 1)         #     Let L be a new label for L'_n
  end if                                      #   End if
end for                                       # End for
Backtrack from n_target to the start node using the label L
```

Output: All optimal paths from a start node to n_{target}

that does not exceed the maximum cost and has a lower cost then a previously found path (if any), we replace L by L'. After all labels have been processed we can use the resulting label to find an optimal path to the start node, providing such a path exists.

Now that we have our Multicriteria Contraction Hierarchies algorithms complete, we will provide a proof sketch that they indeed work as intended and will find an optimal path if such a path exists. The proof assumes correctness of the Label Setting and Contraction Hierarchies algorithms, which has been proved for both[11][7].

Theorem 4.1. *The modified algorithms are correct.*

Proof (sketch). Let us consider Contraction Hierarchies. The original algorithms use a modified Dijkstra search to construct and use these hierarchies[7]. For Multicriteria Contraction Hierarchies, we replaced Dijkstra's algorithm by a modified Label Setting algorithm, but did not change anything else. This means that if we assume our modified Label Setting algorithms are correct, the Contraction Hierarchies algorithms also are.

To prove the modified Label Setting algorithms correct is less trivial, since the algorithm is split up into two parts. The first part, Algorithm 4.3, is an exact copy of the original algorithm, except that it needs to comply to two criteria: the total cost of a path may not exceed a certain cost K and the total path length may not exceed a certain hop limit $a - 1$. In practice, the maximum cost K is the cost to traverse $P = (v, u, w)$. At the time this algorithm is executed, we are looking for a path that is shorter then P. Thus, we can discard any label of which the total cost exceeds $\theta(\varphi(P))$. This is enforced by the second line of the `if` statement. As for the hop limit, each time a node appears in the `for`, we increase the hop count L_a by one. We use the same principle as for the cost to do this: take the total cost (length, hops) so far, and add to it the additional cost (1) to reach this node. The third line of the `if` ensures that paths that reached length a are discarded, by not adding the new label to TL. If a label is not added to TL, it can not be added to PL, thus can not be returned. The algorithm ends after returning all labels PL that meet the requirements and represent

a path that starts with a node n_{start}. Note that the check reads $L'_a < a$, rather then $L'_a \leq a$. Although the second of these checks actually limits the length to a and the first limits it to $a-1$, we search for the last edge of the path using a backward search.

Algorithm 4.4 is more extensively adapted. The goal of this algorithm is that it scans, starting at the target node n_{target}, all nodes around it, that are directly connected to n_{target} and have been labeled by the first algorithm. For each node, the total cost from n_{start} to n_{target} is calculated just as it is done in the original Label Setting algorithm. Since do not want to return paths with a higher cost then (v, u, w), we check if this total cost, aggregated, exceeds the maximum cost K, which is the same as in previous algorithm. We do not need to check if the label's hop count exceeds a hop limit a, since this algorithm is only inputted the output PL from the first algorithm, and this output is guaranteed to only output labels with a maximum hop count of $a-1$. Since the second algorithm does not append multiple labels to these labels, the hop count can not exceed a. Keep in mind that we only want to store labels that represent a path that is better then one already known – if any. If this is the case, then we re-assign label L in such a way that the cost is the total cost from n_{start} to n_{target}, the node the label belongs to is the target node, the predecessor node is the current node, and the hop count is increase by one. If this is not the case, we skip this label and continue with the next label in PL. In short, labels that do not help us construct a path that is better then any other path we know, are discarded and will never overwrite an other label. After the `for` has ended, we have a label L that is either the same label as it was at time of initialization, or represents an optimal path from n_{start} to n_{target}. In the first case, we can conclude there is no optimal path within a hops. In the second case, we know that we have found an optimal path of length L_a, and we can backtrack to n_{start} to construct this optimal path. Thus, our modified algorithms are correct. □

We will verify our algorithms by executing it on the example graph in Figure 4.1. We assume the node importance ordering A, B, C, D, E, F and contract the graph once. As in our example of the Label Setting algorithm, we want an optimal path from A to F and we use the same aggregation function θ, so we expect to get the same two paths again: (A, B, D, F) and (A, C, E, F). We will use a *1-Hop Backward Search* with $a = 3$. Since the nodes are ordered the same as in our example of Contraction Hierarchies, and we have not changed the Contraction Hierarchy creation algorithm, we will not give a detailed explanation again and skip to the shortcut edge creation of all node combinations (v, u, w). These combinations are (A, B, D), (A, C, E), (B, D, F), and (C, E, F). Table 4.2 shows all labels that are created during executing of our algorithm, where L^i_j is the jth label of the ith execution of our modified algorithms.

First, we will see if we can find a shortcut path for (A, B, D). We calculate the cost from A to D via B and find that is is $\theta(10+0, 4+10, 2+12, 10+1) = 49$. Now we execute the adapted Label Setting algorithms with $a = 3$ and $K = 49$. Initially, label $L^0_0 = (0, 0, 0, 0, A, nil, 0)$ is created and added to TL. In the first iteration, we remove it from TL and add it to PL. The nodes that are directly connected to A are B and C, so we create the labels $L^0_1 = (10, 4, 2, 10, B, L^0_0, 1)$ and $L^0_2 = (6, 1, 18, 10, C, L^0_0, 1)$ respectively. Note that L^0_0 has 0 hops and both L^0_1 and L^0_2 have 1 hop. Both labels are

not dominated, the cost of both labels is below 49 and the hop count of both label's path is below $a = 3$. So we move on with label L_1^0 (since it dominates L_2^0).

We remove it from TL, add it to PL, and handle its only directly connected node: D. For this node, we create new label $L_3^0 = (10, 14, 14, 11, D, L_1^0, 2)$ and see that its total cost is 49. This cost is not lower then the max cost, so we discard the label. The node that has label L_2^0, C has two directly connected nodes: B and E. For E, we create label $L_4^0 = (7, 5, 26, 11, E, L_2^0, 2)$, and for B label $L_5^0 = (7, 3, 19, 10, B, L_2^0, 2)$. As we have seen in our Label Setting example, the latter is dominated by L_1^0 and is discarded. E's label is not dominated by an other label, but its total cost is $7+5+26+11 = 49$ is not below K, so it is discarded anyway. We remain with an $TL = \emptyset$ and $PL = \{L_0^0, L_1^0\}$, of which the later is outputted by the algorithm.

This output we use as input for Algorithm 4.4, together with K and $n_{target} = D$. Let us execute the algorithm. This algorithm initializes a label $L_6^0 = (0, 0, 0, 0, D, nil, 0)$ and loops through all labels in PL, that have a node to which is D directly connected, i.e. there is a connection from a node n, *to* node D, instead of the other way around. The only label for which is this is L_3^0, so we calculate the new cost $c = (10, 14, 14, 11)$. This cost is (aggregated) not lower then K, so label L_6^0 is not updated. There is no other node that D is directly connected to, so we exit the `for` with the initial label. This label has no predecessor, so we cannot backtrack any path and the algorithm outputs $\emptyset$. This means there is no path from A to D that is better then (A, B, D), and we create our first shortcut edge (A, D), with cost $(10, 14, 14, 11)$.

Now we have checked one out of the four we need to check, so we need to apply the same procedure for the other three paths. Describing the procedure for these paths as extensively as we have done for the first would be very time consuming and adds little to this example, so we will only tell which labels and shortcut paths are created.

As B is more important then C, we will start with this node and see if we can create a shortcut path for (B, D, F), which has cost $K = 10+11+13+1 = 35$. During execution of the first algorithm, the following labels are created: L_0^1 for B, L_1^1 for D, L_2^1 for C, L_3^1 for B, L_4^1 for E, and L_5^1 for F. After execution, $PL = \{L_0^1, L_1^1, L_2^1\}$. Label L_3^1 is discarded because its hop count of 3 is not smaller then the hop limit $a = 3$.The

Label	Costs	Node	Pre	Hops	Label	Costs	Node	Pre	Hops
L_0^0	(0,0,0,0)	A	*nil*	0	L_1^0	(10,4,2,10)	B	L_0^0	1
L_2^0	(6,1,18,10)	C	L_0^0	1	L_3^0	(10,14,14,11)	D	L_1^0	2
L_4^0	(7,5,26,11)	E	L_2^0	2	L_5^0	(7,3,19,10)	B	L_2^0	2
L_6^0	(0,0,0,0)	D	*nil*	0					
L_0^1	(0,0,0,0)	B	*nil*	0	L_1^1	(0,10,12,1)	D	L_0^1	1
L_2^1	(4,10,12,4)	C	L_1^1	2	L_3^1	(5,12,13,4)	B	L_2^1	3
L_4^1	(5,18,20,5)	E	L_2^1	3	L_5^1	(10,11,13,1)	F	L_1^1	2
L_6^1	(0,0,0,0)	F	*nil*	0					

Table 4.2: The labels Algorithms 4.3 and 4.4 will produce for the graph of Figure 4.1.

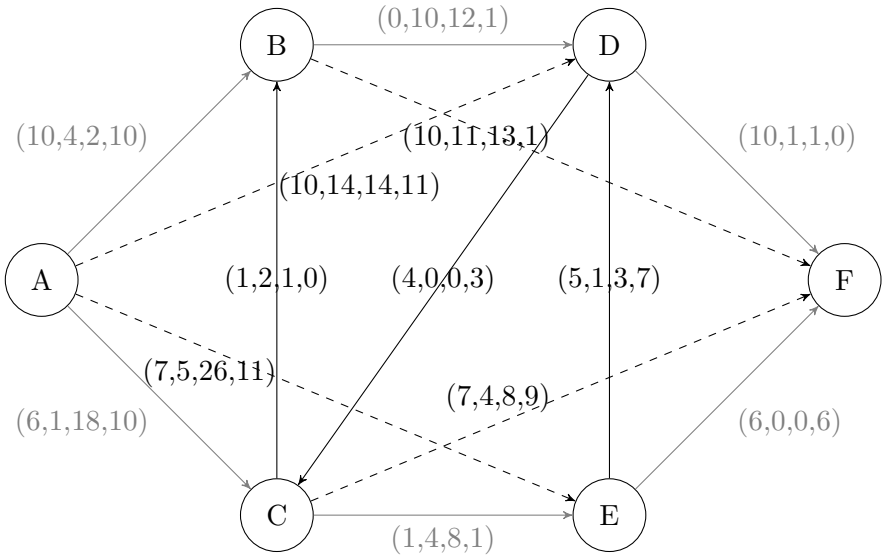

Figure 4.5: The example graph of Figure 4.1 after contraction.

algorithm discards the remaining two labels because their costs exceed the maximum cost of 35. After this, the second algorithm creates the initial label L_7^1 finds that L_0^1 and L_2^1 are not directly connected to the target node F, and that L_1^1 has a total cost equal to K. This means no shorter path is found and the Algorithm 4.2 creates a shortcut edge (B, F) with costs $(10, 11, 13, 1)$.

For the remaining paths we do the same, and find two new shortcut edges (A, E) and (C, F), with costs respectively $(7, 5, 26, 11)$ and $(7, 4, 8, 9)$. Figure 4.5 shows the contracted graph. All shortcuts are dotted, and all removed edges are grayed out.

Now that we have contracted our graph, we can use two Label Setting algorithms to determine all labels of all nodes and determine an optimal path. Let us execute the algorithm: at first, the forward algorithm creates an initial label for A and the backward search one for F. For the forward search, there are only two directly connected nodes, being D and E. Node B and C have no edge connected to A in the contracted graph. For both nodes, the algorithm creates a label. The backward search is directly connected to E and D in its graph level, so it creates labels for these nodes. At the second iteration, the forward search selects the label of node D ((A, D) and (A, E) have the same cost, but D is more important then E) and adds it to the permanent labels list. The backward search does the same for E's label ((D, F) has the same cost as (E, F), but E is less important then D). The forward and backward search repeat this for nodes E and D in the last iteration. The two searches have met and we now have found two labels of D and two labels of E, of which each is the last label of an optimal path. This is very similar to the situation Dijkstra's search would have given us: two paths to a central point represent one optimal path. If we combine these paths we get paths (A, D, F) and (A, E, F). We now unpack these paths and see that we have indeed found both optimal paths: (A, B, D, F) and (A, C, E, F).

5 Conclusion

In this thesis, we have explained the idea of multicriteria path finding using some example scenarios, and have proven that finding an optimal solution to this problem is $\mathcal{NP}$-complete. To overcome this problem, we have suggested to restrict the aggregation function in such a way that we can reduce the multicriteria path finding problem to a single criteria path finding problem. Whilst this approach may suffice for some multicriteria path finding problems, there are still problems that can not be solved this way. Therefore, we proposed an efficient algorithm that combines two existing methods: *Label Setting* and *Contraction Hierarchies*. We refer to this new method as *Multicriteria Contraction Hierarchies*. To show that this new method is correct, we have given both a proof sketch and an example.

5.1 Applications

The algorithm we proposed has many applications, of which one of the most important is route planning on mobile (navigation) devices. Major companies such as TomTom, Garmin and Google now use their own path finding algorithm. For example, TomTom uses methods they refer to as *IQ routes* and *HD routing*[13]. Unfortunately, these algorithms itself are intellectual property of TomTom International B.V. and not publicly accessible, but both algorithms solve dynamic path finding problems, rather then multicriteria path finding problems, so Multicriteria Contraction Hierarchies may be an interesting improvement to these algorithms[16]. Google however, has given a so called "Tech Talk" titled *Fast Route Planning*, in which they reveal using an adapted version of Contraction Hierarchies with Dijkstra's search[8]. As our algorithm is an 'extension' to Contraction Hierarchies, supporting multiple criteria using the method we proposed may be an interesting option for Google. This is especially the case since Google's service *Google Maps* calculates the desired paths using Google's servers, rather then that the user's computer. So even if our Multicriteria Contraction Hierarchies algorithm has a complexity too high for mobile devices, it may still perform fast enough on a server.

5.2 Discussion

Multicriteria Contraction Hierarchies are promising, but we have to take note of the fact that this method is a combination of two basic different path finding methods. Both of these methods have been improved several times, for several purposes, e.g. Contraction Hierarchies have (i.a.) been improved to minimize computation time to specialize it for mobile devices[12]. Furthermore, Geisberger himself has introduced numerous improvements to Contraction Hierarchies. For the Label Setting algorithm, this is also the case: *Label Correcting* improves Label Setting in such a way that it is able to detect negative cycles, and can therefore handle negative costs c.q. weights, whereas the Label Setting algorithm cannot and won't terminate if it enters a negative cycle[5]. All such improvements are not included in the algorithm we proposed, so it probably does not perform as well as it would if we would add all these improvements.

In addition to this, we have not implemented our method, as it would probably cost a considerable amount of time; too much for a Bachelor thesis. So it is very well possible that our algorithm does not improve on the execution time of existing multicriteria path finding algorithms, though it does theoretically. An other risk is that its space or time complexity is too high for practical use on a mobile device. These devices, such as navigation systems, often do not have a lot of memory or computation power available. If this is required for regular path finding problems, using our algorithm, then basic (unimproved) Multicriteria Contraction Hierarchies are not suitable for such devices.

5.3 Related Work

Many researchers have found path finding to be an interesting field of research. Many papers have been written on fast static algorithms, algorithms such as *Dijkstra's algorithm*. Since this algorithm, methods have been development that are up to three millions times faster[4]. One of the leading researches in the area of path finding is Peter Sanders. He has, together with others, developed a number of efficient path finding algorithms. One of these algorithms is *Contraction Hierarchies*, the method we used to base our proposed algorithm upon[7]. This algorithm has been adapted multiple times to i.a. minimize computation time to specialize it for mobile devices[12] and minimize computation space consumption[1]. Other, popular and adapted algorithms are *Highway Hierarchies* and *SHARC*[2], which both use a hierarchy system. Since this thesis is about multicriteria path finding, all here mentioned papers are about an other area of path finding. However, these algorithms can all be applied after having reduced a multicriteria path finding problem to a single criteria one.

Next to general path finding, multicriteria and multiobjective path finding is researched extensively. One of the most cited researches in this area is Ehrgott, who has written a book on multicriteria optimization and in detail discusses several multicriteria path finding algorithms. Two types of these algorithms are the *Label Setting* and *Label Correcting* algorithms. These algorithms essentially solve the original path finding problem of this thesis, but still are too time and space consuming to be well applicable in navigation systems. The same accounts for other multicriteria path finding problem solving algorithms, such as evolutionary algorithms. These algorithms 'evolve' to a solution and are rather popular. Zitzler et al. have created a list of the most promising evolutionary algorithms at that time and ranks *SPEA* and *NSGA* highest[18]. Both of these algorithms have been improved to respectively *SPEA-II* and *NSGA-II* since and these perform equally fast. However, both algorithms are not guaranteed to find the optimal solution and have a complexity of $O(k \cdot m^2)$, with m the number of nodes and k the number of criteria[3][19]. Unfortunately, this makes these algorithms unsuitable for navigation systems.

Finally, Tsaggouris & Zaroliagis have developed an FPTAS (**f**ully **p**olynomial-**t**ime **a**pproximation **s**cheme) that runs in polynomial time and is a generalization of the aforementioned label algorithms: *SSMOSP*[17]. Although this does look very promising, it has as a downside that it determines an approximation to the optimal solution rather than an optimal solution, whilst in this thesis we were looking for the latter.

5.4 Future Work

Considering the algorithm we proposed, there is still room for improvement. As we mentioned in Section sec:discussion, our algorithm does not include any of the improvements that were developed for either Label Setting or Contraction Hierarchies. Furthermore, we could think of some improvements ourself. For example, using the current algorithms we proposed, the same labels are calculated multiple times; an intelligent 'caching' mechanism – that determines if a label needs to be recalculated – in the algorithms should be able to significantly improve the performance.

Next to this, we have not proven Multicriteria Contraction Hierarchies correct, nor have we implemented the algorithm and tested it. Before an algorithm such as this will be (commercially) interesting, it has to be thoroughly tested and compared to other algorithms. Although it seems that our algorithm will perform better then existing multicriteria algorithms, investigation and more research is needed before we can conclude this is indeed the case.

We suggest that future work proves Multicriteria Contraction Hierarchies and further analyses the method. Also, an implementation should be written and tested on a large, real world road graph. In addition to this, several improvements should be considered and eventually implemented. Finally, these versions of the algorithm will need to be compared to existing algorithms, so that we are able to see if the new algorithm performs better then existing algorithms.

6 References

[1] Batz, G. V., Geisberger, R., Neubauer, S., & Sanders, P. (2010). Time-dependent contraction hierarchies and approximation. In *Experimental Algorithms* (pp. 166-177). Springer Berlin Heidelberg.

[2] Bauer, R., & Delling, D. (2008, April). SHARC: Fast and robust unidirectional routing. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX08)* (pp. 13-26).

[3] Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. A. M. T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on, 6*(2), 182-197.

[4] Delling, D., Sanders, P., Schultes, D., & Wagner, D. (2009). Engineering route planning algorithms. In *Algorithmics of large and complex networks* (pp. 117-139). Springer Berlin Heidelberg.

[5] Ehrgott, M. (2005). *Multicriteria optimization* (Vol. 2). Berlin: Springer.

[6] Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability* (Vol. 174). New York: freeman.

[7] Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms* (pp. 319-333). Springer Berlin Heidelberg.

[8] Google Tech Talks (2009). *Fast Route Planning - Youtube* (online). Retrieved June 30, 2013, from `http://www.youtube.com/watch?v=-0ErpE8tQbw`.

[9] Henig, M. I. (1994). Efficient interactive methods for a class of multiattribute shortest path problems. *Management Science, 40*(7), 891-897.

[10] Jozefowiez, N., Semet, F., & Talbi, E. G. (2008). Multi-objective vehicle routing problems. *European Journal of Operational Research, 189* (2), 293-309.

[11] Martins, E. Q. V. (1984). On a multicriteria shortest path problem. *European Journal of Operational Research, 16* (2), 236-245.

[12] Sanders, P., Schultes, D., & Vetter, C. (2008). Mobile route planning. In *Algorithms-ESA 2008* (pp. 732-743). Springer Berlin Heidelberg.

[13] Schfer, R. P. (2009, August). IQ routes and HD traffic: technology insights about tomtom's time-dynamic navigation concept. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (pp. 171-172). ACM.

[14] Serafini, P. (1987). Some considerations about computational complexity for multi objective combinatorial problems. In *Recent advances and historical development of vector optimization* (pp. 222-232). Springer Berlin Heidelberg.

[15] Tarapata, Z. (2007). Selected multicriteria shortest path problems: An analysis of complexity, models and adaptation of standard algorithms. *International Journal of Applied Mathematics and Computer Science, 17* (2), 269-287.

[16] TomTom (2013). *TomTom* (online). Retrieved June 29, 2013, from `http://www.tomtom.com/en_us/`.

[17] Tsaggouris, G., & Zaroliagis, C. (2005). Improved FPTAS for multiobjective shortest paths with applications. *CTI Techn. Report TR-2005/07/03.*

[18] Zitzler, E., Deb, K., & Thiele, L. (2000). Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary computation, 8*(2), 173-195.

[19] Zitzler, E., M. Laumanns, & Thiele, L. (2001). SPEA2: Improving the strength Pareto evolutionary algorithm. *TIK-Report No. 103.* Zurich, Switzerland: Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich.