

Praktikum im Hauptstudium

Sommer 2002

Geometrische Optimierungen für Dijkstras Algorithmus

Jasper Möller
mailto:moellerj@fmi.uni-konstanz.de
University of Konstanz
computer & information science

28th October 2002

Abstract

In diesem Praktikum wurden verschiedene Möglichkeiten untersucht, auf Basis geometrischer Informationen eines Graphen den Algorithmus von Dijkstra zur Suche kürzester Wege in einem Graphen zu optimieren. Ziel war es, verschiedene Typen von geometrischen Containern auf ihre Tauglichkeit als Selektionskriterium für die Auswahl von möglichen Kanten auf einem kürzesten Weg zu untersuchen. Zur Realisierung des Projekts wurde unter anderem ein spezielles Programmieridiom, die so-genannte "MixIn-Programmierung" eingesetzt, um die verschiedenen Optimierungsmöglichkeiten möglichst flexibel kombinieren zu können.

1 Problemstellung

In vielen praktischen Anwendungen ist es nötig, einen "kürzesten Weg" zwischen zwei Punkten zu finden. So ist man z.B. bei einem Routenplaner u.a. an einer Route interessiert, die Start und Ziel möglichst direkt verbindet, oder an einer Route, die eine möglichst kurze Fahrzeit aufweist. Es sind aber auch andere Kriterien für die **Distanz** zweier Punkte denkbar, so z.B. für eine Eisenbahnverbindung die Anforderung, möglichst billige Fahrpreise zu haben. Es gibt universelle Algorithmen, die für all diese Probleme eine Lösung finden (z.B. der Algorithmus von Dijkstra oder der von Bellmann-Ford). Allerdings nutzen diese Algorithmen i.A. keine Zusatzinformationen, die in dem Problem noch enthalten sind, so dass man für bestimmte Problemklassen eine beträchtliche Einsparung an Laufzeit erreichen kann, wenn man entsprechende Optimierungen vornimmt.

In diesem Praktikum sollte speziell folgende Problemstellung untersucht werden:

Problem 1. Kürzeste Wege auf gewichteten Graphen in der Ebene

- **Gegeben:** Ein (gerichteter) Graph $G = (V, E)$ mit Kantengewichten $l : E \rightarrow \mathbb{R}_0^+$ in der Ebene, d.h. jeder Knoten hat Koordinaten $(x, y) \in \mathbb{R}^2$.
- **Gesucht:** Für jedes Paar von Knoten $v_1, v_n \in V$ ein kürzester Weg in G zwischen v_1 und v_n , d.h. ein Weg $W = v_1 v_2 \dots v_n \subset V$ mit der Länge $L(W) := \sum_{i=1}^{n-1} l(v_i, v_{i+1})$ so, dass $L(W)$ minimal ist unter der Länge aller $v_1 - v_n$ -Wege.

Problem 1 ist ein Beispiel eines sogenannten *all pairs shortest paths Problems*.

Im Folgenden wird für die Längenfunktion immer der gewöhnliche euklidische Abstand der Endpunkte der Kante gewählt, d.h. $\forall e = (v_1, v_2) \in E : l(e) = |v_2 - v_1|$. Weiter fassen wir ungerichtete Graphen als gerichtete Graphen auf, indem wir jede Kante durch ein Paar von Hin- und Rückkanten gleicher Länge ersetzen.

Für Graphen mit nichtnegativen Kantengewichten lassen sich kürzeste Wege z.B. mittels Dijkstras Algorithmus finden:

Algorithm Dijkstra Input: Graph $G=(V,E)$, Knoten: s, t , Kantengewicht $l : E \mapsto \mathbb{R}_0^+$

```

1. PriorityQueue PQ
2. Knotenarray distance, Knotenarray predecessor
3.
4. für alle Knoten LoopNode in G
5.   distance[LoopNode] := infinity
6.   predecessor[LoopNode] := NULL
7. distance[s] := 0
8.
9. PQ.insert(s, 0)
10. solange PQ nicht leer ist
11.   v := PQ.extractMin()
12.   falls v != t
13.     für alle von v ausgehenden Kanten (v,w)
14.       falls distance[w] == infinity
15.         distance[w] := distance[v] + l(v,w)
16.         PQ.insert(w, distance[w])
17.         predecessor[w] = v
18.     sonst
19.       falls distance[w] > distance[v] + l(v,w)
20.         distance[w] := distance[v] + l(v,w)
21.         PQ.decreaseKey(w, distance[w])
22.         predecessor[w] = v
23.   sonst Ende

```

Der Algorithmus berechnet ausgehend von einem Startknoten s einen kürzesten Weg zu dem Zielknoten t , falls ein solcher existiert, ansonsten wird abgebrochen, wenn die PriorityQueue leer ist. Die Laufzeit liegt, wenn eine effiziente PriorityQueue verwendet wird, bei $\mathcal{O}(|V| \log|V| + |E|)$. Offenbar erhält man ein Lösung für das *all pairs shortest path* Problem, wenn man Dijkstras Algorithmus auf jedes Paar von Knoten $s, t \in G$ anwendet.

2 Optimierungen

Für viele Anwendungen, z.B. ein interaktives Abfrage-System, ist der Algorithmus zu langsam. Was für Möglichkeiten der Beschleunigung gibt es aber? Für allgemeine Graphen bestehen sicher wenig Chancen zur Verbesserung. In unserer Problemstellung werden aber spezielle Graphen betrachtet. Sollte es nicht irgendwie möglich sein, die dort vorhandene geometrische Zusatzinformation über die Lage der Knoten in der Ebene mit auszunutzen? Im folgenden werden zwei Verfahren vorgestellt, die genau das tun.

2.1 Zielgerichtete Suche

Dem ersten Verfahren liegt folgende Idee zugrunde:

Modifiziere die Kantenlängen geeignet, so dass Kanten, die in Richtung Ziel weisen, bevorzugt werden.

Dies entspricht der intuitiven Vorstellung, dass man, wenn man versucht ein Ziel möglichst schnell zu erreichen, sich immer möglichst direkt in Richtung des Ziels bewegen wird. Konkret bedeutet das, dass man für eine Suche von s nach t für jede Kante (u, v) die modifizierte Länge $l'(u, v)$ folgendermaßen berechnet:

$$l'(u, v) = l(u, v) + |t - v| - |t - s|$$

Dadurch erhalten Kanten, die eher Richtung Ziel weisen, eine niedrigere Länge. Kanten, die vom Ziel weg weisen, werden dagegen benachteiligt. Knoten, die über eine der richtig gerichteten Kanten erreicht werden, gelangen somit weiter nach vorne in die PriorityQueue. Auf diese Weise wird versucht, zuerst Wege über diese Knoten zu finden. Man kann zeigen, dass Dijkstras Algorithmus auch mit dieser **zielgerichteten Suche** korrekt arbeitet. Nachteilig ist, dass das Verfahren nicht für beliebige nichtnegative Gewichtsfunktionen verwendbar ist.

2.2 Geometrische Beschränkungen

2.2.1 Prinzip

Eine andere Möglichkeit zur Optimierung basiert darauf, dass man in Zeile 13 des Algorithmus nicht alle ausgehenden Kanten betrachtet, sondern solche Kanten ausschließt, über die kein kürzester Weg zum Ziel führt. Praktisch heißt das:

1. Berechne in einem *Preprocessing* einmal für jeden Knoten s alle kürzesten Wege zu allen anderen Knoten.
2. Für jede ausgehende Kante (s, u) bestimme alle Knoten t_i , für die ein kürzester $s - t_i$ -Weg über (s, u) führt.

Der naive Ansatz, einfach für jede Kante die in Schritt 2 erhaltene Liste von Zielknoten zu speichern, ist allerdings aus mehreren Gründen nicht sinnvoll: Zum einen braucht das Durchlaufen einer solchen Liste i.A. auch eine nichtkonstante Laufzeit, zum anderen hat man im schlimmsten Fall einen Speicheraufwand von $\mathcal{O}(|V||E|)$. Dann kann man aber auch gleich mit Speicheraufwand $\mathcal{O}(|V|^2)$ sämtliche kürzesten Wege speichern und die Anfragen ohne Dijkstras Algorithmus in konstanter Laufzeit ausführen¹. Man wird sich vielmehr die zugrundeliegenden geometrischen Informationen über die Knoten zunutze machen und für jede Kante ein Gebiet (Container) angeben, in dem sicher alle Zielknoten (unter Umständen aber noch mehr Knoten) liegen. Diese Berechnung eines Containers für jede Kante findet anstatt oder zusätzlich zu Schritt 2. im Preprocessing statt. Beim anschließenden Suchen kürzester Wege wird Dijkstras Algorithmus dann modifiziert:

```

2. Knotenarray distance, Knotenarray predecessor
3.
...
13.   für alle von v ausgehenden Kanten (v,w)
14.       falls distance[w] == infinity

```

wird ersetzt durch:

```

2. Knotenarray distance, Knotenarray predecessor
3. Kantenarray Container
...
13.   für alle von v ausgehenden Kanten (v,w)
13a.   falls Zielknoten t in Container[(v,w)] liegt
14.       falls distance[w] == infinity

```

Ein geeigneter Container sollte also den Test in 13a. ausführen können.

2.2.2 Anforderungen an Container

An die Container, die einer Kante zugeordnet werden, sind folgende Anforderungen zu stellen:

1. Konstanter Speicherplatz für jede Kante, vorzugsweise so gering wie möglich
2. Konstante Laufzeit des Tests, ob ein Punkt in einem Gebiet liegt, möglichst effizient.

¹Verwende hierzu eine Matrix, in der für jeden Knoten der Vorgänger auf einem kürzesten Weg gespeichert wird

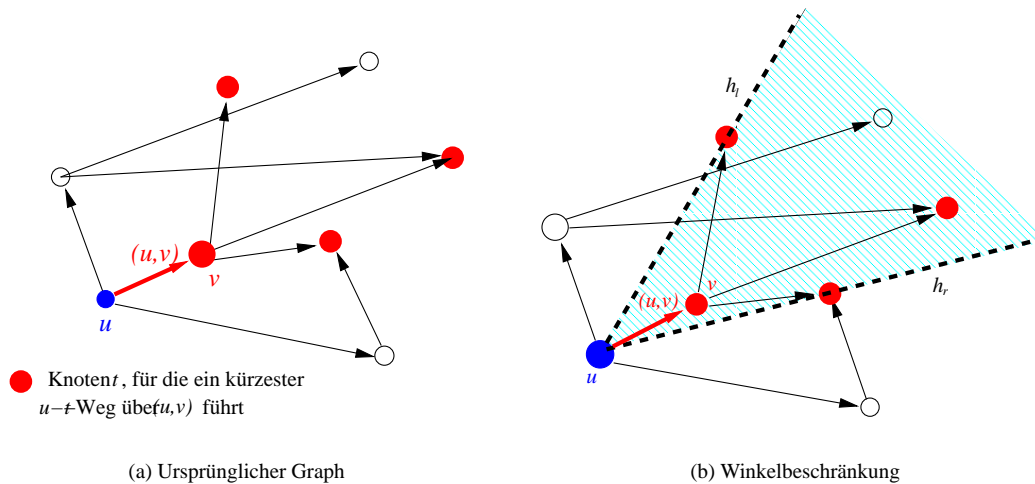


Figure 1: Geometrische Optimierungen, 1

3. Möglichst “minimaler” Container, d.h. es sollten möglichst wenig Punkte im Gebiet enthalten sein, für die trotzdem kein kürzester Weg über die entsprechende Kante führt.
4. Erträgliche Zeiten für das Preprocessing

Es ist klar, dass die Bedingungen z.T. gegeneinander wirken. So liefert z.B. die konvexe Hülle einer 2-dimensionalen Punktmenge Container mit vergleichsweise geringem Verschnitt, aber die Bedingungen 1. und 2. sind im Allgemeinen verletzt. Das umschließende Rechteck mit minimaler Fläche braucht vergleichsweise hohen Aufwand für das Preprocessing, ebenso Kreise und Ellipsen minimaler Fläche.

2.2.3 Untersuchte Container

Zur Diskussion standen hier folgende Container, wobei im folgenden für eine Kante $(u,v) \in E$ immer $\mathcal{T}_{(u,v)} := \{t \in V \mid \text{es existiert ein kürzester } u-t\text{-Weg in } G \text{ über } (u,v)\}$ die Menge der möglichen Zielknoten für eine Kante, sowie $C_{(u,v)}$ den dieser Kante zugeordneten Container bezeichne. Außerdem unterscheiden wir der Einfachheit halber nicht zwischen Knoten und den ihnen zugeordneten Punkten in der Ebene.

Winkelbeschränkung Hier ist für jede Kante $(u,v) \in E$ der Container $C_{(u,v)}$ durch ein Paar von Halbgeraden h_l und h_r , die beide im Punkte u beginnen, bestimmt, wobei

$$h_l \text{ und } h_r \text{ beginnen beide in } u$$

$$\forall t \in \mathcal{T}_{(u,v)} : t \text{ liegt rechts von } h_l \text{ und links von } h_r$$

und der Winkel zwischen h_l, h_r minimal unter allen Paaren von Halbgeraden mit diesen beiden Eigenschaften ist (s. Abb. 1(b)). Da u schon implizit durch die zugehörige Kante gegeben ist, müssen lediglich die beiden Halbgeraden bzw. zwei Punkte als Repräsentanten gespeichert werden, also konstanter Speicheraufwand. Der Test auf Inklusion erfolgt wieder durch Überprüfen der zweiten Bedingung, ist also auch in konstanter Laufzeit durchführbar.

Achsenparallele Rechtecke, sog. Bounding Box Dies ist wahrscheinlich der einfachste Container, es gilt für eine Kante (u,v) :

$$C_{(u,v)} = [x_{min}, x_{max}] \times [y_{min}, y_{max}]$$

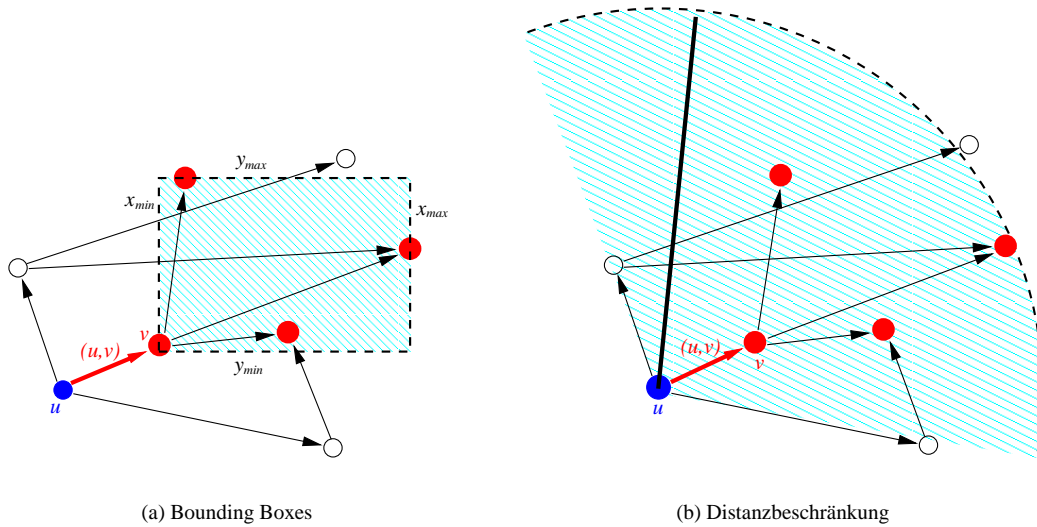


Figure 2: Geometrische Optimierungen, 2

wobei

$$x_{min} = \min_{t \in \mathcal{T}(u,v)} \{x_{coord}(t)\}$$

analog für die anderen Eckpunkte (s. Abb. 2(a)). Eigentlich würde man zur Speicherung der Eckpunkte vier Punkte $p_1 = (x_{min}, y_{min})$, $p_2 = (x_{max}, y_{min})$, $p_3 = (x_{max}, y_{max})$ und $p_4 = (x_{min}, y_{max})$ benötigen. Man kann allerdings die Koordinaten von p_2 und p_4 aus denen der anderen beiden Punkte berechnen, so dass lediglich Speicherplatz für zwei Punkte (d.h. vier Fließkommazahlen) benötigt wird. Die Bestimmung der vier Extrema von $\mathcal{T}(u,v)$ kann in Linearzeit sowohl online wie offline durchgeführt werden. Zur Überprüfung, ob ein gegebener Punkt $p_0 = (x_0, y_0)$ in $C_{(u,v)}$ liegt, muss lediglich überprüft werden, ob

$$x_0 \in [x_{min}, x_{max}] \quad \text{und} \quad y_0 \in [y_{min}, y_{max}]$$

erfüllt ist, d.h. es werden maximal vier Vergleiche benötigt, die Laufzeit ist also konstant.

Kreise mit Startknoten als Zentrum/Distanzbeschränkung Hier ist einer Kante (u, v) ein Kreis mit Zentrum u zugeordnet, wobei gilt (s. Abb. 2(b)):

$$C_{(u,v)} = B_u(r_{max}(u, v))$$

mit

$$r_{max}(u, v) = \max_{t \in \mathcal{T}(u,v)} \{\|t - u\|\}$$

Wieder kann $r_{max}(u, v)$ in Linearzeit bestimmt werden. Da u bereits durch die Kante implizit gegeben ist, muss lediglich $r_{max}(u, v)$ für jede Kante gespeichert werden.

Zur Überprüfung, ob ein gegebener Punkt p_0 in $C_{(u,v)}$ liegt, muss lediglich überprüft werden, ob

$$\|p - u\| \leq r_{max}(u, v)$$

erfüllt ist, die Laufzeit ist also wieder konstant.²

²Um mehrfaches Wurzelziehen zu vermeiden, wird man in der Praxis alle Berechnungen nicht mit der euklidischen Distanz, sondern mit ihrem Quadrat und r_{max}^2 durchführen, was auf den meisten Rechnerarchitekturen sehr viel effizienter sein dürfte.

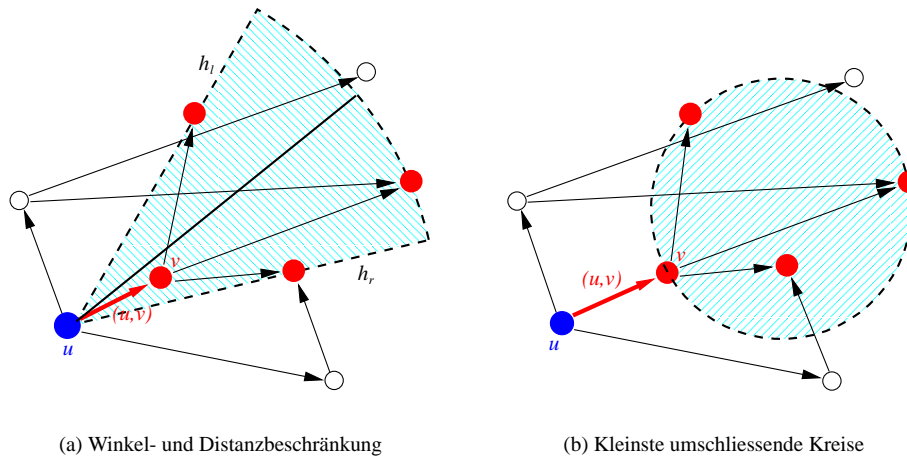


Figure 3: Geometrische Optimierungen, 3

Winkel mit Distanzbeschränkung Dies ist einfach die Kombination aus der gewöhnlichen Winkelbeschränkung und einem u -zentrierten Kreis (s. Abb. 3(a)), es müssen also pro Kante drei Werte gespeichert werden, und für den Test werden maximal drei Berechnungen benötigt.

Umschliessende Kreise minimaler Fläche Hierbei handelt es sich um einen Kreis (s. Abb. 3(b))

$$C_{(u,v)} = B_{x_0}(r) \quad \text{mit} \quad \forall t \in \mathcal{T}_{(u,v)} : t \in B_{x_0}(r)$$

und

$$A(B_{x_0}(r)) = \pi r^2 \quad \text{minimal unter allen Kreisen, die ebenfalls } \mathcal{T}_{(u,v)} \text{ überdecken}$$

Dieser so harmlos aussehende Container ist überraschend schwer zu bestimmen. Von E. Welzl gibt es einen Algorithmus [1], der diesen Kreis offline mit einem randomisierten Verfahren und durch geschicktes Ausnutzen der Abarbeitungsreihenfolge des Algorithmus in annähernd linearer Laufzeit (in der Anzahl der Zielpunkte) berechnet. Im allgemeinen wird für diesen Kreis dann nicht das Zentrum x_0 und der Radius r gespeichert, sondern stattdessen drei Punkte auf dem Kreisrand, und damit dann der Test auf Inklusion durchgeführt.

3 Implementation

3.1 Softwareumgebung und Klassenhierarchie

Das Projekt wurde komplett in C++ realisiert, als Compiler kam der GNU C-Compiler in der Version 2.95 unter Linux zum Einsatz. Weiter wurde als Klassenbibliothek LEDA in der Version 4.2 verwendet.

Die Grundversion von Dijkstras Algorithmus für eine s, t -Wegsuche wurde in eine Klasse `TargetDijkstra` gekapselt, die insbesondere eine Methode `run(leda_node s, leda_node t)` bereitstellt. Außerdem sind insbesondere folgende `protected`-Methoden vorhanden, die von spezielleren Ableitungen überschrieben wurden:

- `double length()`: liefert die Gewichtsfunktion für eine Kante zurück, dies wird für die zielgerichtete Suche von der Klasse `GoalDijkstra` überschrieben.
- `void preModifyingEdge(leda_edge)`: diese Methode wird vor der *Relaxation* oder wenn ein Knoten das erste Mal gefunden wird, ausgeführt. Dies wird insbesondere für die geometrischen

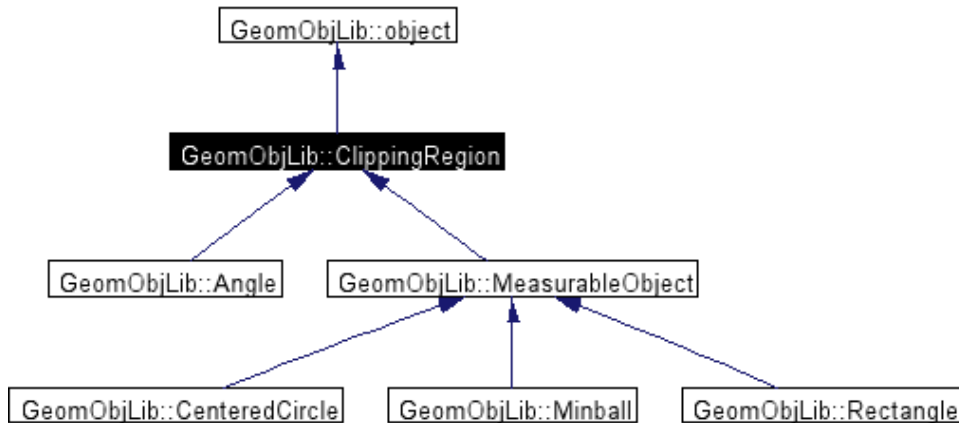


Figure 4: Klassenhierarchie für geometrische Container

Optimierungen verwendet, um im Preprocessing einem Zielknoten eine Ausgangskante des Startknotens zuzuordnen.

- `bool processEdge(leda_edge, leda_node, leda_node)`: wird aufgerufen, wenn in Schritt 13a. des modifizierten Dijkstra-Algorithmus (s. 1) überprüft werden soll, ob eine Ausgangskante verarbeitet werden soll. In der Basisklasse wird immer `true` zurückgeliefert, in den geometrischen Optimierungen wird überprüft, ob der Zielpunkt im zur Kante gehörigen Container liegt.

Für die Containerobjekte wurde eine gemeinsame abstrakte Basisklasse `ClippingRegion` erstellt, die insbesondere die Methoden

- `void addPoint(leda_point)`: vergrößert den Container so, dass der übergebene Punkt auch noch in diesem Container liegt
- `bool contains(leda_point)`: prüft, ob ein gegebener Punkt in dem Container liegt

bereitstellt, die von den konkreten Implementationen eines Containers dann implementiert werden müssen (s. Abb. 4). Das Preprocessing und die geometrische Optimierung wird von einer Klasse `SelectionDijkstra` durchgeführt, in der im Konstruktor für jedes Paar s und t von Knoten in einem gegebenen Graphen ein kürzester Weg gesucht wird, wobei zu jedem Zielknoten die Ausgangskante gespeichert wird und für diese Ausgangskante die Methode `preprocessEdge()` ausgeführt wird, in der dann z.B. der zu dieser Kante gehörige Container vergrößert werden kann:

```

virtual void preprocess()
{
    // initialise edges
    ...
    // run Dijkstra's algorithm for every node
    ...
    leda_node Source;
    forall_nodes(Source, Graph)
    {
        // instantiate Dijkstra
        ...
        // run Dijkstra
        DijkstraAlgo.run(Source);
    }
}

```

```

// adjust edge attributes for this source node
leda_node Node;
forall_nodes(Node, Graph)
{
=>   if ( DijkstraAlgo.hasSourceEdge(Node))
      { // preprocess source edge
        // virtual:
=>     preprocessSourceEdge(DijkstraAlgo.sourceEdge(Node) ,
                             Node);
...

```

Damit dieses Verfahren funktioniert, müssen alle Container die Möglichkeit haben, inkrementell durch Hinzufügen einzelner Punkte erzeugt zu werden, wozu eben die Methode `addPoint` dient.

3.2 MixIn-Programmierung

Bei der Implementation des Projektes ergab sich das Problem, dass die normale, insbesondere einfache, Vererbung u.U. nicht flexibel genug war, um alle Anforderungen abzudecken. So musste z.B. für jede Kombination von *Aspekten* eines Algorithmus (z.B. Instrumentierung, zielgerichtete Suche, mit Pfadbestimmung, mit Optimierung etc.) jeweils komplett neu abgeleitet werden, was nicht nur fehlerträchtig ist, sondern auch nicht im Sinne einfacher Code-Wiederverwendung ist. Das Ziel sollte ein Satz von Klassen sein, die jede für sich einen interessanten Teilaspekt implementieren, und zwar unabhängig davon, ob überhaupt abgeleitet worden ist bzw. von welcher Klasse. Der Anwender dieser Klassen sollte dann nur noch spezifizieren müssen, welche Aspekte für sein Problem kombiniert werden sollen, ohne dass erneut ein kompletter Vererbungspfad in der Klassenhierarchie erzeugt werden muss.

Mehrfachvererbung als Möglichkeit, verschiedene Aspekte von Basisklassen zu kombinieren, scheidet vor allem deshalb aus, weil sie für den Anwender meist zu fehleranfällig in der Anwendung ist, insbesondere was Namenskonflikte und speziell in C++ die Schwierigkeiten mit `virtual` Basisklassen anbelangt. Erfolgreicher ist der Ansatz, jede Klasse, die einen Aspekt implementiert, über die Basisklasse zu parametrisieren, also z.B. nicht einfach:

```
class GoalDijkstra: public IrgendeinBasisDijkstra {
```

sondern:

```
template <typename BaseDijkstra>
class GoalDijkstra: public BaseDijkstra {...
```

so dass bei der **Verwendung** von `GoalDijkstra` einfach angegeben werden kann, von welcher Klasse `GoalDijkstra` denn nun tatsächlich abgeleitet werden soll, d.h. mit den Aspekten welcher Klasse die von `GoalDijkstra` implementierten Aspekte kombiniert werden sollen, z.B.

```
GoalDijkstra<CountingDijkstra> Algo;
```

erzeugt eine Kombination aus `GoalDijkstra` und `CountingDijkstra` (einer instrumentierten Variante des Basisalgorithmus. Es ist klar, dass man dieses Verfahren iterieren kann, so dass man z.B.

```
BoundingBoxDijkstra<SelectionDijkstra<GoalDijkstra
    <CoordinateDijkstra<CountingDijkstra<Dijkstra>>>>> Algo;
```

als Kombination aus diversen Aspekten verwenden kann.

Bei diesem Ansatz macht einem allerdings die starke Typbindung von C++ zuerst einen Strich durch die Rechnung: Dadurch, dass für eine Klasse, die einen Aspekt vorhanden implementiert, im Normalfall ein Konstruktor mit Argumenten nötig ist, ergibt sich folgendes Problem: In der speziellen Klasse muss ein Konstruktor vorhanden sein, der alle für eine konkrete Parametrisierung nötigen Argumente verarbeiten kann. Wenn also z.B. die Klasse `FoodDijkstra` zwei Konstruktorargumente für sich benötigt und z.B. die Klasse `CoordinateDijkstra` drei Argumente, dann müsste für die folgende Parametrisierung:

```
FooDijkstra<CoordinateDijkstra>
```

in der Klasse `FooDijkstra` u.a. ein Konstruktor der Form

```
FooDijkstra(arg1, arg2, arg3, arg4, arg5):
    BaseDijkstra(arg1, arg2, arg3)...
```

vorhanden sein. Wenn man aber über die Basisklassen keinerlei Voraussetzungen machen will, kann man diese Konstruktoren nicht sinnvoll für alle denkbaren Möglichkeiten für Basisklassen definieren.

Eine mögliche Lösung für dieses Problem besteht darin, die Parameter in einer eigenen Klasse zu kapseln, so dass für jeden Parameter der Zugriff auf seinen Wert möglich ist, oder aber ein Verweis auf die nachfolgenden Parameter erfolgt. In den Aspektklassen entnimmt der jeweilige Konstruktor dann nur die für ihn nötigen Argumente aus dem Parameterobjekt, konstruiert daraus den für den speziellen Aspekt nötigen Teil und übergibt an die unbekannte Basisklasse einfach den kompletten Rest der Liste von Argumenten³. Die Basisklasse verfährt dann entsprechend mit dem Rest der Argumentliste.

Für Parameterobjekte mit einem oder zwei Argumenten könnte die Implementation so aussehen:

```
template<typename T, typename NEXT>
struct Param {

    T &value;
    NEXT next;

    // Constructor with 1 argument
    Param(T &t):value(t),next(Nil){}

    // Constructor with 2 arguments
    template<typename T2>
    Param(T2 &t2,
          T &t):
        value(t),
        next(t2)
    {}
    ...
}
```

Bei der Definition eines neuen Aspektes würde man dann z.B. folgendes verwenden

```
template<typename BaseDijkstra>
class CoordinateDijkstra:public BaseDijkstra {
    // Create parameter type
    typedef Param<leda_node_array<leda_point>,
                 typename BaseDijkstra::ParamType >
        ParamType;

    CoordinateDijkstra(ParamType & P):
        BaseDijkstra(P.next),
        Coordinates(P.value)
    { };
};
```

4 Auswertung

Für die Auswertung wurden mit LEDA zufällige planare Graphen mit Grössen zwischen 5 und 1000 Knoten und jeweils unterschiedlichen durchschnittlichen Knotengraden erzeugt jeweils eine komplette *all pairs shortest path* Suche auf jedem dieser Graphen für die folgenden Verfahren durchgeführt:

³Dieses Verfahren besitzt eine gewisse Ähnlichkeit mit der Handhabung variabler Argumentlisten in C

- Dijkstras Algorithmus ohne weitere Optimierungen (s. 1)
- Dijkstra mit zielgerichteter Suche (s. 2.1)
- Dijkstra mit Winkelrestriktion (s. 2.2.3, Abb. fig:angle)
- Dijkstra mit Bounding Boxes als Container (s. 2.2.3, Abb. 2(a))
- Dijkstra mit Distanzbeschränkung (s. 2.2.3, Abb. 2(b))
- Dijkstra mit kombinierter Distanz- und Winkeleinschränkung (s. 2, Abb. 3(a))
- Dijkstra mit kleinsten umschließenden Kreisen als Containern (s. 2, Abb. 3(b))

Für jeden Algorithmus wurde die Gesamtzahl der im Laufe der kompletten Wegsuche verwendeten Kanten und Knoten bestimmt. Die Messungen erfolgten unter Linux auf einem PC mit 800MHz Athlon CPU und 256MB Arbeitsspeicher. Dabei wurden folgende Messergebnisse erhalten, wenn man die Anzahl der Operationen in Bezug zur Gesamtgrösse der Graphen betrachtet. Dabei wurde getrennt für Graphen mit durchschnittlichem Knotengrad 2 und Grad 6 ausgewertet:

Knotengrad zwischen 1 und 2, Gesamtzahl der Operationen

Knoten	dijkstra	BBox	Winkel	Distanz	Winkel+Dist.	Minballs	zielger.
10	224	118	127	144	125	118	217
15	825	285	326	479	313	292	806
20	1581	540	664	865	598	577	1576
25	3168	956	1193	1893	1084	1046	3038
75	23708	6181	8118	11539	7116	7488	23614
100	55939	11785	16038	24650	13583	15416	55603
150	86284	24904	28361	46670	26589	29212	86013
200	294649	50602	69114	139189	62278	82167	292957
250	562080	97268	148546	255400	120548	151042	559170
750	9090066	820456	1536692	3368930	1210389	1823747	9046727

Knotengrad zwischen 5 und 6, Gesamtzahl der Operationen

Knoten	dijkstra	BBox	Winkel	Distanz	W.+D.	Minballs	zielger.
15	3936	727	995	2266	978	770	3046
20	9454	1466	2068	5152	2000	1677	7243
25	18601	2757	3811	10088	3628	3370	14048
50	155621	16271	26176	79426	22639	25030	115962
75	535416	50478	85246	272939	70969	88880	397116
100	1275713	110834	187406	641887	157375	212335	947631
150	4398436	338252	618770	2276926	507908	785978	3236083
200	$10,3 \cdot 10^6$	$0,741 \cdot 10^6$	$1,43 \cdot 10^6$	$5,31 \cdot 10^6$	$1,12 \cdot 10^6$	$1,91 \cdot 10^6$	$7,56 \cdot 10^6$
250	$20,7 \cdot 10^6$	$1,46 \cdot 10^6$	$2,85 \cdot 10^6$	$10,9 \cdot 10^6$	$2,28 \cdot 10^6$	$4,11 \cdot 10^6$	$1,52 \cdot 10^6$
500	$167 \cdot 10^6$	$8,77 \cdot 10^6$	$19,7 \cdot 10^6$	$88,9 \cdot 10^6$	$15,4 \cdot 10^6$	$32,6 \cdot 10^6$	$121 \cdot 10^6$
750	$555 \cdot 10^6$	$26,9 \cdot 10^6$	$64,9 \cdot 10^6$	$298 \cdot 10^6$	$51,8 \cdot 10^6$	$116 \cdot 10^6$	$406 \cdot 10^6$

Bezogen auf den Standard-Dijkstra mit 100% der besuchten Knoten/Kanten erhält man dann die Ergebnisse in Abb. 5, wenn man die übrigen Knotengrade ebenfalls mit einbezieht: Man sieht also, dass die Verwendung von Bounding Boxes mit Abstand den grössten Gewinn bringt. Ebenfalls noch recht effizient ist die Kombination von Winkel- und Distanzbeschränkung. Alle anderen Verfahren, insbesondere die recht aufwendige Verwendung von umschließenden Kreisen minimaler Fläche, erbringen vor allem bei grösseren Graphen deutlich schlechtere Ergebnisse. Es zeigte sich auch, dass die Container-Verfahren, insbesondere die Bounding Boxes, aber auch die anderen Typen, gut mit der Grösse der Graphen skalieren, d.h. der Effizienzgewinn durch die Optimierungen wird um so grösser, je grösser die Graphen sind.

Betrachtet man noch zusätzlich die Abhängigkeit vom durchschnittlichen Knotengrad, so scheinen die Verfahren auch mit diesem Parameter zu skalieren, allerdings fällt der Trend nicht so eindeutig aus. Um einen

eindeutigeren Trend zu erhalten, müssten noch mehr Messungen in Abhängigkeit von diesem Parameter durchgeführt werden.

5 Fazit und Ausblick

Es hat sich gezeigt, dass sich auch mit relativ einfachen Containern, nämlich den Bounding Boxes einerseits und der Kombination aus Winkel- und Distanzbeschränkung andererseits, eine beträchtliche Optimierung der Abarbeitungszeit von Dijkstras Algorithmus erreichen lässt. Besonders angenehm dabei ist, dass sich für diese Container der Aufwand im Preprocessing wie auch der verwendete Speicherplatz durchaus im Rahmen halten und die Tests dadurch, dass nur Vergleichsoperationen (auch für die Lagetests) nötig sind, sowohl effizient wie auch numerisch stabil sind.

Trotzdem bieten sich natürlich eine ganze Menge von Erweiterungen und Verallgemeinerungen an:

- Die Verwendung von weiteren Containern (Ellipsen, kantenparallele Rechtecke, Quadrilaterale minimaler Fläche...), sowie von Schnitten mehrerer Container ist am naheliegendsten. Sinnvoll wäre hierbei die Verwendung einer bereits getesteten Geometriebibliothek, wie z.B. CGAL.
- Mehr Messungen, insbesondere auch auf realen Graphen (Eisenbahnnetze, Verkehrswege allgemein).
- Erweiterungen auf:
 - Mehr Dimensionen. Hierbei nimmt allerdings u.U. der Aufwand für die nötigen Berechnungen zu sehr zu
 - Andere Metriken. Sowohl die zielgerichtete Suche als auch die Container-Restriktion sollten weiterhin funktionieren, wenn man die Verfahren entsprechend anpasst.
 - Beliebige (nichtnegative) Gewichtsfunktionen. Die Containerrestriktion sollte prinzipiell auch damit zurechtkommen, denn für sämtliche Tests und Berechnungen geht die Gewichtsfunktion überhaupt nicht ein. Fraglich ist allerdings, ob dann immer noch ein Beschleunigungseffekt auftritt, da bei der gewöhnlichen Längenfunktion die Zielknoten zu einer Kante typischerweise relativ nahe beieinander liegen, ein Container also tatsächlich eine spürbare Selektion vornimmt. Dies muss im allgemeinen Fall natürlich nicht gegeben sein. Eventuell ist der Gewinn dann höher, wenn man zuvor die Koordinaten der Knoten modifiziert.
- Ein grundlegendes Prinzip hinter den Optimierungen über Container ist die Selektion geeigneter Kanten oder Knoten. Neben den hier untersuchten geometrischen Verfahren könnte man natürlich auch andere Heuristiken heranziehen, die nicht zwingend an zugrundeliegende geometrische Informationen gebunden sind, z.B. bevorzugte Behandlung von Knoten mit hoher Betweenness-Zentralität o.ä.

References

- [1] Emo Welzl. *New Results and New Trend in Computer Science*, volume 555 of *Lecture Notes Comput. Sci.*, chapter Smallest enclosing disks (balls and ellipsoids), pages 359–370. Springer-Verlag, 1991.

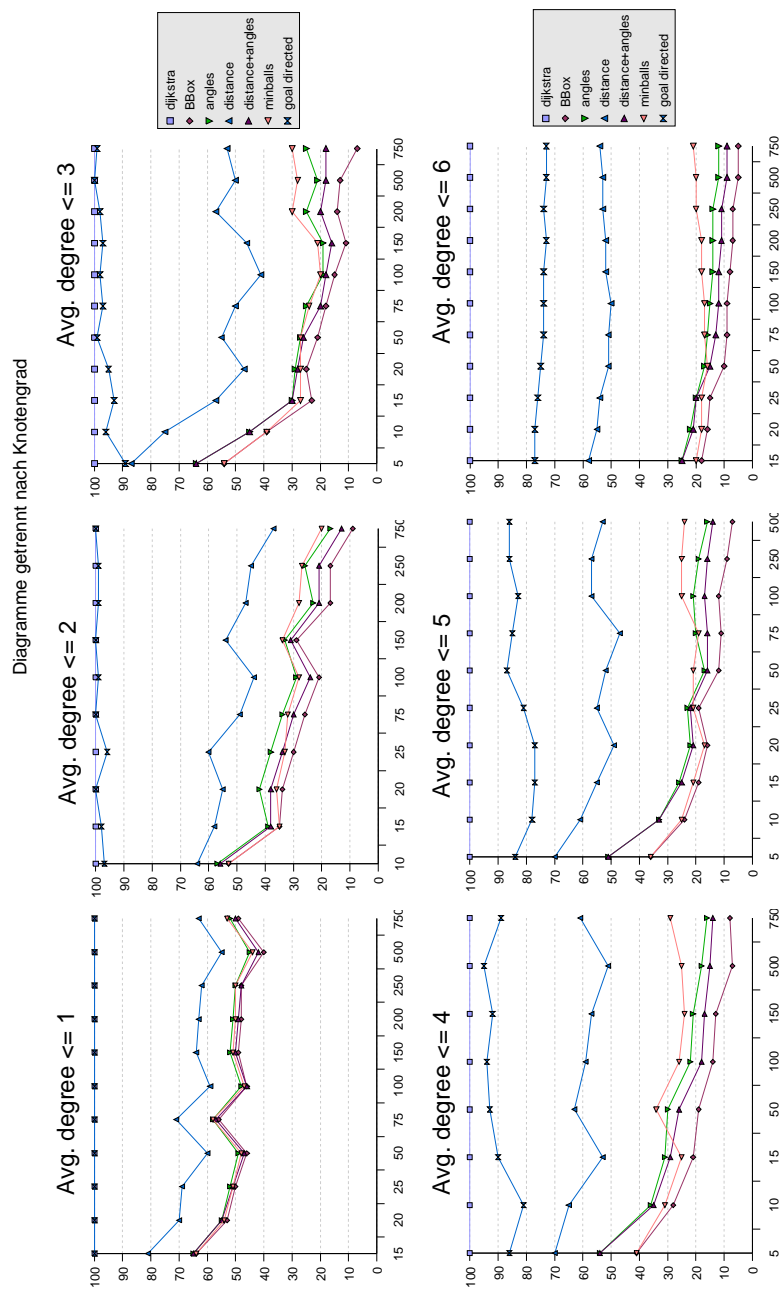


Figure 5: Ergebnisse getrennt nach Knotengraden: Es wird die relative Anzahl der Operationen (besuchte Knoten und Kanten), bezogen auf den Standard-Dijkstra mit 100% der Operationen, in Abhängigkeit von der Knotenzahl des Graphen betrachtet. Die Graphen wurden zufällig generiert und getrennt nach den durchschnittlichen Knotengraden untersucht (s. 4)