Wintersemester 2017/2018
Motion Planning
Prof. Horsch, Rudi Scheitler

Praktikum 3 - Visibility Graph

David Welsch, Ken Hasenbank

1. Dezember 2017

Beschreibung der Aufgabe

In diesem Praktikum sollte die Wegfindung mit Hilfe des Visibility-Graphs realisiert werden.

Ein Visibility-Graph beinhaltet Punkte aus einem Konfigurationsraum sowie die Kanten zwischen diesen Punkten, welche kollisionsfrei verbunden werden können. Der Name *Visibility*-Graph kommt zu Stande, da die Kanten auch als Sichtverbindungen zwischen den Punkten gedeutet werden können. Ein Beispiel für einen Visibility-Graph ist in Abbildung 1 zu sehen.
Bei diesem Graph kann man sehen, dass die Verbindungen entlang der Sichtlinien, jeweils von den Ecken der Hindernisse zu Ecken andere Hindernisse, bzw. vom und zum Start-/Zielpunkt, gezeichnet sind. In der Abbildung wird der Startpunkt als *Agent* bezeichnet, der Zielpunkt als *Goal*.

In diesem Praktikum soll nun zuerst ein Visibility-Graph erstellt werden. Im nächsten Schritt muss dann auf diesem Graphen mit Hilfe eines Wegfindungs-Algorithmus, zum Beispiel Dijkstra oder der A* -Algorithmus, der kürzeste Pfad von Start zu Zielpunkt gefunden werden.

Beim freiwilligen Teil der Aufgabe haben wir uns darauf geeinigt, einen auf *Supporting*- und *Separating*-Lines reduzierten Visibility-Graphen zu erstellen. Ein solcher reduzierten Graph ist in Abbildung 2 zu sehen.

Beschreibung der Lösung

Im Anhang sind die von Gnuplot erstellten Plots der einzelnen Graphen zu sehen.

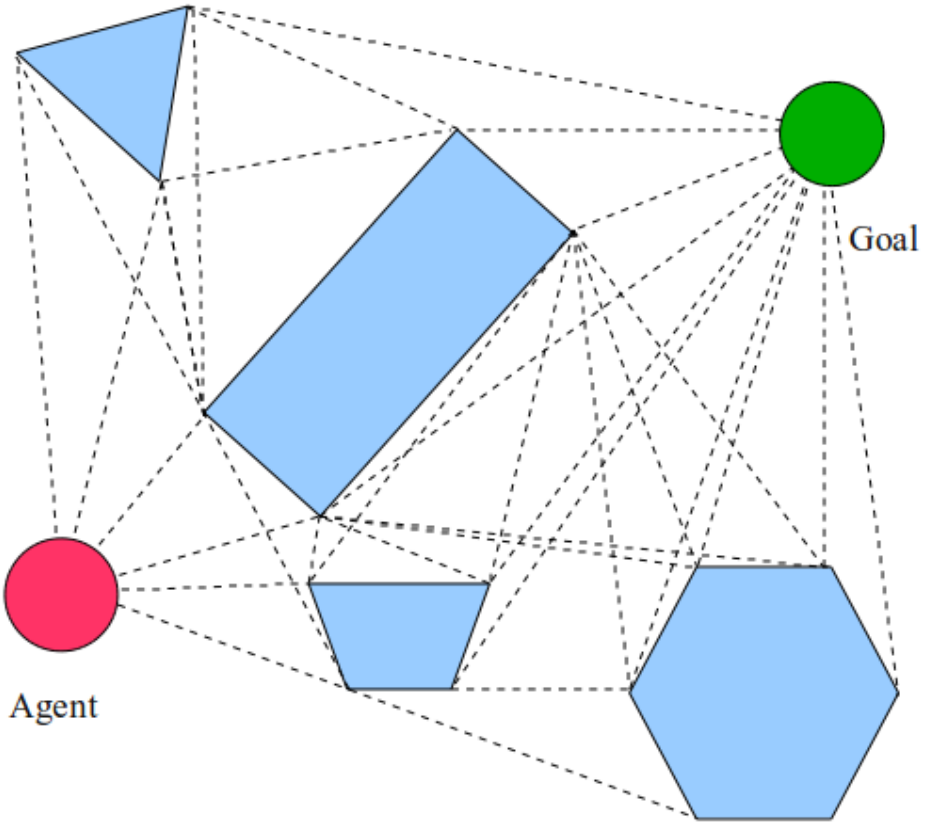

Abbildung 1: Visibility Graph

Zur Lösung haben wir den vorgegeben Code-Rahmen, jedoch auf Grund verschiedener Probleme auf unseren Entwicklungsrechnern nicht die gesamte Boost-Library, verwendet. Zur Lösung haben wir ebenfalls die Klasse *Linie* neu geschrieben, welche viele Hilfsfunktionen enthält. Die eigentliche Lösung wird mit Hilfe dieser Hilfsfunktionen in der Funktion und Klasse *VisibilityGraph* berechnet. Anhand dieser Funktion beschriebe ich nun unser Vorgehen.

- (Zeile 26) Zuerst wird ein Vektor erstellt, welcher später die Punkte auf dem kürzesten Pfad enthält.
- (35-38) Dann werden vier Graphen erstellt. Diese enthalten später die folgenden Kanten. Initial werden alle mit den Kanten aus dem Graph g befüllt.
  - obstacleGraph: Wird mit den Außenkanten der Hindernisse befüllt
  - illegalLinesGraph: Wird mit den diagonalen Verbindungslinien der Hindernisse befüllt

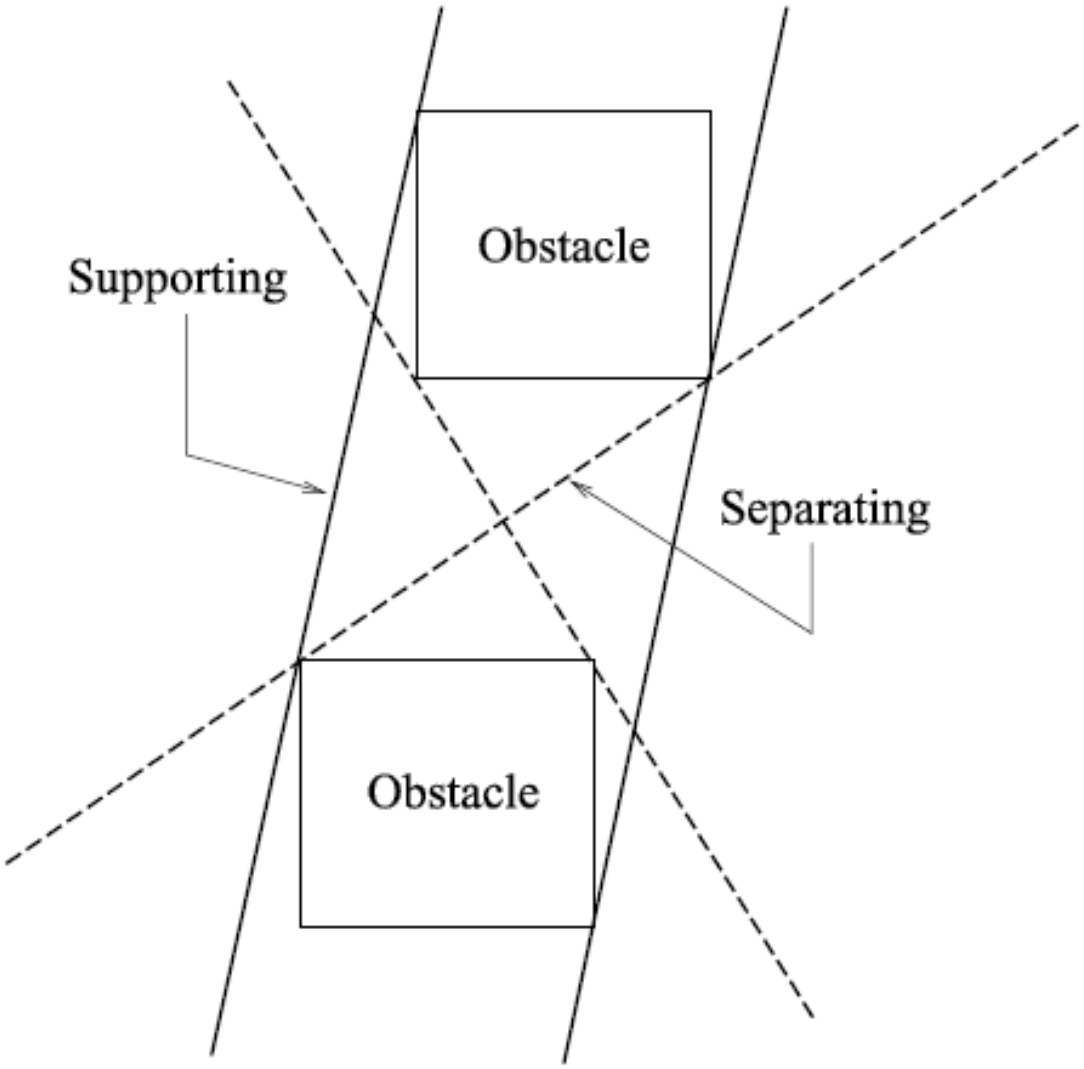

Abbildung 2: Reduced Visibility Graph

  - visibilityGraph: Dieser Graph wird nachher den berechneten Visibility-Graph beinhalten
  - pathGraph: Dieser Graph wird am Ende den als am Kürzesten berechneten Pfad enthalten

  Zusätzlich wird noch ein Vektor aus Obstacles erstellt, welcher die selbst erstellte Struktur Obstacle enthält. Diese Struktur wird später näher beschrieben.

- (Zeile 42) Im nächsten Schritt wird in der Variable *knoten* die Anzahl der im Graphen g enthaltenen Eckpunkte gespeichert.

- (Zeile 45-51) Dann wird eine Adjazenzliste in Form eines zwei-dimensionalen double-Arrays erstellt und mit der Gewichtung -1 initialisiert. -1 bedeutet, dass die Punkte nicht verbunden sind. Ansonsten enthält diese Liste die Länge der Kante zwischen den Punkten.

- (Zeile 53-54) Hier werden zwei Vektoren aus der selbst erstellten Klasse *Linie* erstellt, welche mit den Kanten der Hindernisse (*obstLines*) bzw. den Kanten innerhalb der Hindernisse (*illegalLines*) befüllt werden soll.

- (Zeiel 55-81) In dieser for-Schleife werden die zuvor erstellten Vektoren mit den Diagonalen im Hindernis bzw. den Hinderniswänden befüllt. Um dies zu realisieren wird in der äußeren Schleife über alle Hindernisse iteriert. Bei jedem Schleifendurchlauf wird zusätzlich jeweils ein mal über die beiden gegenüberliegenden Punkte (Zeile 59-67), sowie über alle Punkte (Zeile 70-79) des jeweiligen Hindernissen iteriert.
  In der ersten Schleife werden die Diagonalen erstellt, in der zweiten die Kanten der Hindernisse.

- (Zeile 84-114) In dieser for-Schleife wird der eigentliche Visibility-Graph erstellt sowie die Adjazenzliste mit den richtigen Werten für die Gewichtung gefüllt. Dies geschieht wie folgt:

  - (Zeile 84) Es wird über alle in dem Graph g enthaltenen Punkte iteriert.
  - (Zeile 85) Für jeden untersuchten Punkt wird erneut durch alle Punkte gelaufen, jedoch diesmal startend vom aktuell untersuchten Knoten, damit keine Verbindungen doppelt untersucht werden.
  - (Zeile 86-88) Die Punkte am Index i und j, sowie die Linie zwischen diesen beiden Punkten wird zwischengespeichert.
  - (Zeile 91) Anhand der Hilfsfunktionen *isIllegal* und *isVisible* (werden später erklärt) wird überprüft, ob die eben zwischengespeicherte Linie eine der Diagonalen in den Hindernissen ist (*isIllegal*) und ob sie eine Außenkante der Hindernisse schneidet (*isVisible*).
  - (Zeile 108) Ist die Linie keine Diagonale der Hindernisse und schneidet sie keine der Kanten der Hindernisse wird sie zum *visibilityGraph* hinzugefügt.
  - (Zeile 109-110) In der Adjazenzmatrix wird zusätzlich die Länge der Linie in beide Richtungen gespeichert.

- (Zeile 115) Die Adjazenzmatrix wird zu Debugzwecken auf der Console ausgegeben.

- (Zeile 117) Es wird ein Vektor des Typs *int* erstellt, welcher durch die Funktion dijkstra mit Zahlen befüllt wird. Dieser Vektor enthält dann die Nummern der Knoten, welche den kürzesten Weg darstellen.
  Die Funktion *dijkstra* enthält den Code des Dijkstra-Algorithmus und berechnet mit Hilfe dessen die entsprechenden Weg-Knoten.

- (Zeile 119-133) Anschließend werden die Kantennummern auf der Konsole ausgegeben und die Kanten des Weges in den Graphen *pathGraph* gespeichert. In den Zeilen 129 und 130 werden die Punkte aus dem ursprünglichen Graphen g in den Vektor *path* gespeichert. Dieser repräsentiert dann den Weg.
- (Zeile 140-143) Hier werden die Graphen erstellt, damit sie mit GnuPlot visualisiert werden können.

Die so berechnete Lösung ist in Abbildung 3 im Anhang ersichtlich.

Beschreibung der Hilfsklasse *Linie*

Die Hilfsklasse *Linie* enthält alle nötigen Funktionen um die oben beschriebene Lösung zu berechnen. Ich werde nacheinander alle Funktionen beschreiben.

- orientierungZurLinie(Point test)
  Mit dieser Funktion wird die Orientierung eines Punktes zu der Linie berechnet und als Integer codiert zurückgegeben. Die Rückgabewerte haben folgende Bedeutung:
  0 = Der Punkt befindet sich auf der Linie
  1 = Der Punkt befindet sich rechts von der Linie
  2 = Der Punkt befindet sich links von der Linie

- schneidetLinie(Linie test)
  Mit dieser Funktion wird berechnet, ob eine andere Linie diese Linie schneidet. Hierfür wird zuerst mit Hilfe der Funktionen *compare* und *extends* gestet, ob die Linien oder Start bzw. Endpunkte identisch sind. Anschließend werden die Orientierungen der Start- und Endpunkte der beiden Linien mit Hilfe der Funktion *orientierungZurLinie* miteinander verglichen. Sind die Orientierung der Anfangs- und Endpunkte jeweils unterschiedlich, so schneiden sich die Linien und es wird true zurückgegeben. Ansonsten wird false zurückgegeben da sie sich nicht schneiden.

- compare(Linie test)
  In dieser Funktion wird verglichen ob die Linien identisch sind. Hierbei spielt die Orientierung keine Rolle.

- extends(Linie test)
  Diese Funktion berechnet, ob die Linien in einem Anfangs- oder Endpunkt identisch sind.

- getObstOrientierung(Obstacle obst)
  Hier wird berechnet, wie ein Obstacle zu der Linie liegt. Ein Obstacle ist beschrieben durch das struct *Obstacle*, welches aus einem Vektor aus Linien besteht. Die Funktion gibt zurück, ob die Linie durch das Obstacle geht, rechts, oder links davon liegt. Die Kodierung entspricht der von *orientierungZurLinie*.

- isSeparating(Obstacle obst1, Obstacle obst2)
  Diese Funktion berechnet anhand zweier Obstacles, ob die Linie die Obstacles trennt, also ein Obstacle links und das andere rechts liegt, und die beiden Obstacles auf derselben Seite der Linie liegen. Treffen alle drei Bedingungen nicht zu, wird false zurückgegeben. Das bedeutet, dass die Linie in Bezug auf die beiden übergebenen Obstacles keine Seperating-Line ist.

- isSupporting(Obstacle obst1, Obstacle obst2)
  Hier wird berechnet, ob die Linie eine Supporting-Line ist. Hierfür werden diesselben Dinge berechnet wie in der vorherigen Funktion, jedoch werden sie anders ausgewertet:
  Die Linie darf nicht durch eines der Obstacles verlaufen, jedoch müssen beide Obstacles auf derselben Seite des Objektes sein. Dann ist die Linie eine Supporting-Line.

- length()
  Berechnet die Länge der Linie.

Beschreibung der Zusatzaufgabe (Reduzierter Visibility Graph)

Die Zusatzaufgabe wird in der Datei *VisibilityGraph.cpp* in den Zeilen 94 bis 106 gelöst. Dies geschieht wie folgt:
In jeder zuvor als mögliche Verbindung gefundenen Linie wird mit Hilfe der Funktionen *isSupporting* und *isSeparating* aus der Klasse *Linie* geprüft, ob es sich bei der gerade geprüften Linie um eine solche handelt. Nur wenn eine der beiden Funktionen *true* zurück gibt, wird die Linie auch in den *visibilityGraph* aufgenommen, ansonsten wird durch *continue* die nächste Linie überprüft.
Die vorher im Vektor *obstacles* gespeicherten Hindernisse werden hierfür genutzt.

Die Lösung ist in Abbildung 4 zu sehen.

Was wir gelernt haben

Die Erstellung des Visibility Graphs war eine große Herausforderung. Zusätzlich hatten wir große Probleme mit der Boost-Library, da benötigte imports nicht funktioniert haben. Wir haben uns deshalb entschlossen, die entsprechenden Funktionen selbst zu implementieren und hatten damit schneller Erfolg.
Die Zusatzaufgabe stellte noch mal eine Herausforderung dar, aber nach reichlicher Überlegung konnten wir auch diese erfolgreich lösen.
Insgesamt war diese Praktikumsaufgabe sehr aufschlussreich und interessant.

Anhang

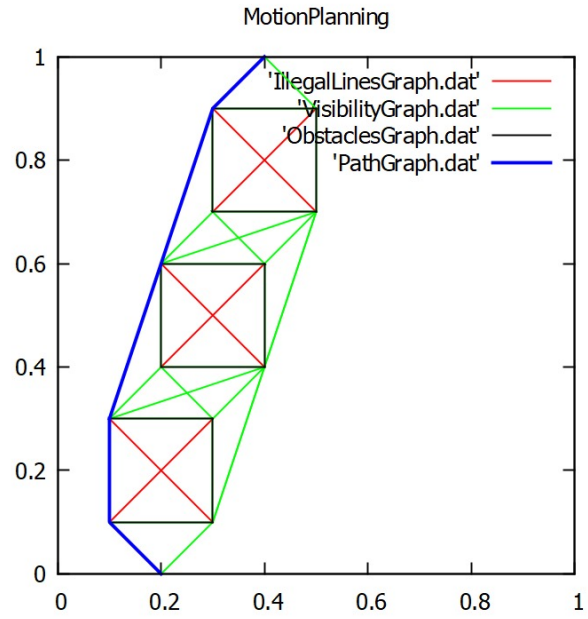

Abbildung 3: Lösung - ohne Zusatzaufgabe

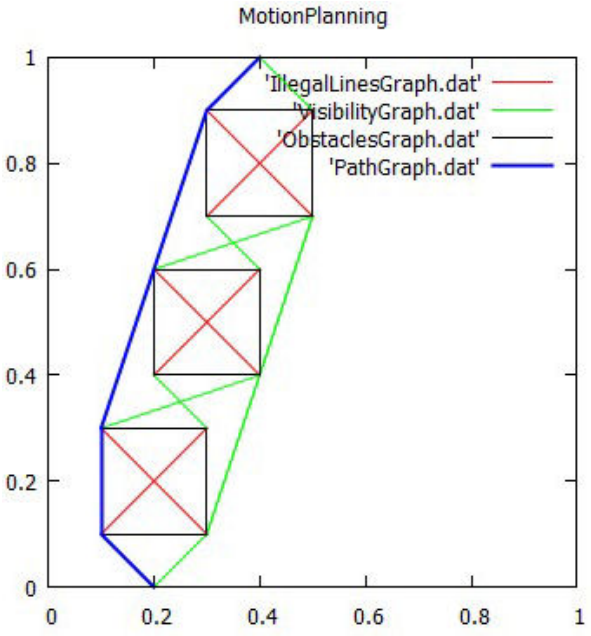

Abbildung 4: Lösung - mit Zusatzaufgabe