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

Praktikum 5 - Rapidly Exploring Random Trees (RRT)

David Welsch, Ken Hasenbank

17. Januar 2018

Beschreibung der Aufgabe

Das Ziel dieses Praktikums war es, auf Basis eines RRT's einen Weg zwischen einer Start- und einer Zielkonfiguration zu finden.

Das Konzept von RRT's ist es, eine zufällige Konfiguration an den bissherigen RRT anzuschließen. Hierfür wird der Punkt auf dem Graphen gesucht, der die der neuen zufälligen Konfiguration am nächsten ist. Nun wird von dem gefundenen Punkt aus in Richtung der neuen Konfiguration gegangen und -vorrausgesetzt es wurde kein Hindernis getroffen- der entsprechende Weg als Edge in den Graphen gespeichert. Wird jedoch ein Hindernis getroffen kann der weg auch nur zu Teilen in den Graphen aufgenommen werden um so nah wie möglich an den Rand des Hindernisses herran zu kommen. Kann man Anstelle der zufälligen Konfiguration die Zielkonfiguration direkt an den RRT anschließen kann das ëxploring"beendet werden und über einen Graph-Such-Algorithmus wie beispielsweise dem A* oder Dijkstra ein Pfad zwischen der Start- zu Zielkonfiguration gefunden werden.

In den gegebenen Aufgaben soll zunächst ein simpler Zweidimensionaler RRT im Einheitsquadrat ohne Hindernisse generiert werden. Danach soll ein Wurm mit 5 Freiheitsgraden (zwei davon translatorisch, drei rotatorisch), wie auch schon im *Praktikum 4*, durch einen Parcours mit 5 Hindernissen bewegt werden.

Beschreibung der Lösung

Zur Lösung haben wir den Code-Rahmen des PRM-Praktikums verwendet und um die funktion des Tools GnuPlot erweitert. Alle Berechnungen erfolgen in der Datei *main.cpp* und dort in der Methode *_tmain*. Wir nutzen zudem eine Methode um die Kanten zum

Graphen hinzuzufügen.

Ab hier wird durch die Beschreibung der Datei *main.cpp* unser Vorgehen erläutert:
Zunächst sind sowohl die Aufgabe 1 als auch die aufgabe 2 (inklusive des freiwilligen Teils) in einem Projekt. Zwischen den Aufgaben kann über einen define in Zeile 236 *define AUFGABE 1|2* gewechselt werden.

- Aufgabe 1 (Zeilen 233 - 320)
  - Aufgabe 1 (Zeilen 233 - 320)
    Zur Lösung dieser Aufgabe mussten wir den vorgegebenen Coderahmen zunächst so umbauen, dass er einfache 2-Dimensionale Konfigurationen entgegen nimmt. Anschließend haben wir als initiale Konfiguration den Mittelpunkt *{0.5, 0.5}* in den Graphen aufgenommen. Anschließend Generieren wir mittels eines Zufallsgenerators zufällige Konfigurationen und suchen unter zuhilfenahme eines rTrees den nächsten punkt auf dem Graphen um diese beiden Punkte dann miteinander zu Verbinden. Die Punkte werden allerdings nicht direkt verbunden sondern in einer kleinen Schrittgröße zerstückelt. Dieser Vorgang wird 1000 mal wiederholt (definiert in Zeile 238), wobei alle 10% der aktuelle Graph in eine GnuPlot-Datei abgespeichert wird. Die zwischenschritte können, wie in Abbildung 1 zu sehen, über das Tool GnuPlot gerendert werden. In Abbildung 1 gut zu erkennen ist, dass der Baum den gesamten Raum sehr

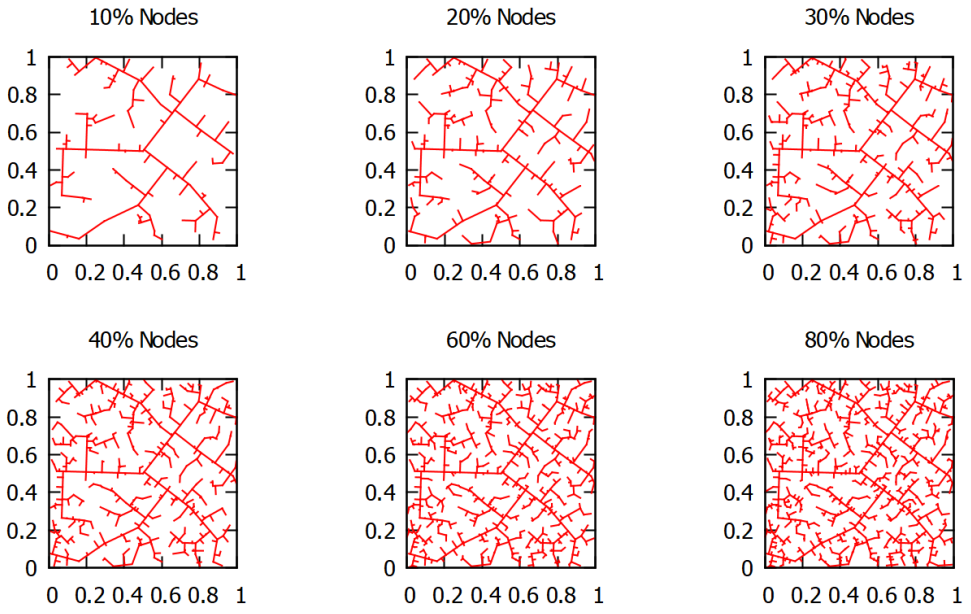

Abbildung 1: Zwischenschritte

schnell erkundet und anschließent nurnoch feinkundiger untersucht. So sieht

man in den ersten 50% die größten unterschiede zwischen den einzelnen stufen und später wird der Graph nurnoch um sehr kurze verzweigungen erweitert. Abbildung 2 zeigt nochmal das Ergebnis nachdem alle 100% (1000 Nodes) hinzugefügt wurden.

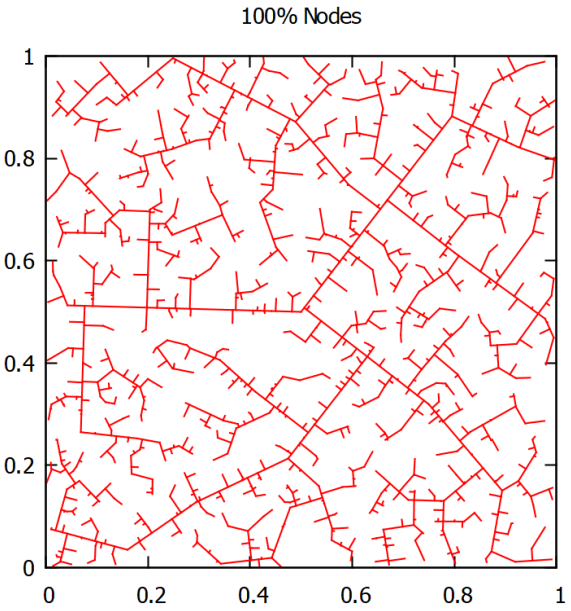

Abbildung 2: Endergebnis

- Aufgabe 2 (Zeilen 102 - 223 und 322 - 567)
  - Funktionen (Zeilen 102 - 223)
    - *getVector*
      Diese Funktion gibt die Konfiguration einer angegebenen Position aus einem Graphen zurück.
    - *addVertex*
      Diese Funktion fügt eine Konfiguration in zwei unabhängigen Graphen und den dazugehörenden rTrees hinzu. Dies wird verwendet um Zwei separate Graphen für Start und Zielkonfiguration pflegen zu können und trotzdem am Ende alle Daten in einem Globalen Graphen vorliegen zu haben.
    - *findNearest*
      Sucht mittels eines rTree die n nächsten Nachbarn zu einem angegebenen Punkt aus einem Graphen und gibt diese zurück.
    - *doStep*
      Diese Funktion geht, von einen gegebenen Startkonfiguration, eine Schritt-

länge entgegen der angegebenen Zielkonfiguration. Ist die Entfernung zwischen der Start-/ und Zielkonfiguration geringer als die angegebene Schrittlänge wird die Zielkonfiguration zurück gegeben.

  * *exploreRandom*
    Diese Funktion wird verwendet um einen Graphen um eine zufällige Konfiguration zu erweitern.
    Es wird zunächst eine zufällige Konfiguration im Konfigurationsraum erzeugt (unabhängig ob diese im Freiraum liegt oder nicht). Anschließent wird auf dem Graphen der Punkt gesucht welcher der zufälligen Konfiguration am nächsten ist. Von diesem Punkt aus wird eine Schrittweite der zufälligen Konfiguration entgegen gegangen. Ist die neu entstandene Konfiguration im Freiraum und von dem Grafen ohne Kollision auf direktem wege zu erreichen, so wird der Punkt sowie eine Kante in den Graphen eingefügt.

  * *expandTowards*
    Diese Funktion wird verwendet um einen Graphen in richtung einenr angegebenen Zielkonfiguration zu erweitern.
    Dabei wird auf dem Graphen nach der Konfiguration gesucht, welche der Zielkonfiguration am nächsten ist. Von dieser wird schrittweise der Zielkonfiguration entgegen gegangen. Jeder Schritt, der Kollisionsfrei durchgeführt werden kann, wird als Punkt und als Kante in den Graphen aufgenommen. Wird die Zielkonfiguration erreicht, wird dies mit dem Rückgabewert *true* an den Aufrufer signalisiert. Bei einer Kollision hingegen wird mit dem rückgabewert *false* signalisiert, dass der Graph den Zielpunkt nicht erreichen konnte.

- Implementierung des Algorithmus (Zeilen 322 - 567)
  Der Ablauf des Algorithmus kann mittesl der oben beschriebenen Funktionen sehr einfach implementiert werden. Zu beginn werden die Start-/ sowie Zielkonfigurationen in die entsprechenden Graphen eingetragen. Die folgenden 4 Schritte werden nun solange wiederholt bis die Ziel-/ und Startgraphen verbunden sind (die Graphen sind verbunden wenn der Start-/ oder Zielgraph eine Konfiguration des anderen erfolgreich erreichen konnte):

  * Erweitere den Zielgraphen um eine zufällige Konfiguration.
  * Versuche die hinzugefügte Konfiguration vom Startgraphen aus zu erreichen.
  * Erweitere den Startgraphen um eine zufällige Konfiguration.
  * Versuche die hinzugefügte Konfiguration vom Zielgraphen aus zu erreichen.

  Anschließend wird der Graph-Such-Algorithmus Dijkstra auf unseren globalen Graphen (beinhaltet sowohl den Start-/ als auch den Zielgraphen) angewendet und das gefundene Ergebnis in einer EasyRob-Datei ausgegeben.

Was wir gelernt haben

Wir haben durch Aufgabe 1 schnell gemerkt wie gut der RRT einen Raum erkunden kann und uns dazu entschieden auf die erhöhte Komplexität der Parallelisierung (wie in Praktikum 4 verwendet) verzichtet. Nachdem wir unseren Code währende der Entwicklung mehrmals getestet haben ist uns schnell Klar geworden, dass dies eine gute entscheidung war. Nach erfolgreicher Implementierung und Testens der angegebenen Start-/ und Zielkonfigurationen (alle ergebnisse unter 3 Sekunden) waren wir sehr erstaunt, wie viel schneller ein Zielgerichteter Algorithmus wie der RRT doch sein kann.