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

Praktikum 1 - Bug Algorithmus

David Welsch, Ken Hasenbank

31. Oktober 2017

Beschreibung der Aufgabe

Das Ziel dieses Praktikums war es, eine Version des Bug-Algorithmus zu implementieren. Bei der Klasse der Bug-Algorithmen handelt es sich um Algorithmen, die genutzt werden, um einen Weg zwischen zwei Punkten in einem 2D-Raum zu finden. Hindernisse werden dabei je nach Ausprägung des Bug-Algorithmus unterschiedlich umgangen. Alle Algorithmen der Klasse haben gemeinsam, dass sie nicht vorausschauend arbeiten, sondern erst auf ein Hindernis reagieren, wenn sie darauf treffen (ausgenommen der Tangent-Bug Algorithmus, welcher hier aber nicht betrachtet werden soll.) Die verschiedenen Ausprägungen arbeiten wie folgt:

(a) Bug-0 Algorithmus: Nach dem Auftreffen auf ein Hindernis läuft der Algorithmus in eine vorbestimmte Richtung am Hindernis entlang, bis die Sicht auf das Ziel wieder frei ist. Dann läuft er wieder in die Richtung des Ziels. Dies wird so lange wiederholt, bis der Zielpunkt erreicht ist. Es können noch Variationen in der Laufrichtung beim Auftreffen auf ein Hindernis definiert werden, damit Endlosschleifen ausgeschlossen werden.

(b) Bug-1 Algorithmus: Bei diesem Algorithmus wird jedes Hindernis nach dem Auftreffen komplett umlaufen. Während des Umlaufens wird der Abstand zum Ziel an jedem Punkt gemessen und der Punkt mit dem niedrigsten Abstand gespeichert. Nach einem kompletten Umlauf wird dieser Punkt wieder angelaufen und von dort Richtung Ziel weiter gegangen. Dieses Verhalten wird bei jedem Hindernis fortgeführt, bis das Ziel gefunden wurde.

(c) Bug-2 Algorithmus: Beim Bug 2 Algorithmus wird zu Beginn eine imaginäre *M-Linie* vom Start zum Zielpunkt gezogen. Trifft der Algorithmus auf ein Hindernis, verhält er sich ähnlich wie Bug 0, allerdings läuft er so lange am Hindernis entlang bis er die M-Linie in einem geringeren Abstand als beim Auftreffen auf das Hindernis wieder erreicht. Dann läuft er wieder entlang der *M-Linie*. Dieses Verhalten wird bei jedem Auftreffen auf ein Hindernis wiederholt, bis das Ziel gefunden wurde.

Wir haben uns dafür entschieden, den Bug-0 Algorithmus zu implementieren. Hierfür haben wir den vorgegebenen Code-Rahmen genutzt und unsere Lösung in C++ geschrieben.

Beschreibung der Lösung

Zur Lösung haben wir den vorgegeben Code-Rahmen verwendet. Für unseren Code haben wir die Klasse *Bug0* erstellt, welche von der Klasse *BugAlgorithm* erbt. Dann haben wir die Funktionen *update*, sowie *obstacleInWay* bearbeitet.

- In der Funktion *update (Zeile 25-63)* setzen wir zuerst den Punkt *robotPos* auf die geerbte aktuelle Position. Dann prüfen wir in Zeile 29, ob wir bereits am Ziel angekommen sind. Dies wird geprüft, indem der aktuelle Abstand zwischen dem Ziel und dem Roboter mit einem Mindestwert (ebenfalls geerbt) verglichen wird. Ist dies der Fall, wird der aktuelle Punkt auf die Zielposition gesetzt und die Funktion wird mit dem Return-Wert *true* beendet. Wenn nicht, wird als nächstes mit der Funktion *obstacleInWay* geprüft, ob die Sicht auf das Ziel frei ist.
  Wenn nein, wird eine boolean-Variable *getroffen* auf *true* gesetzt. Diese bewirkt im weiteren Verlauf, dass ein Counter hochgezählt wird, der den Algorithmus spätestens nach 10.000 Schritten abbricht, um eine Endlosschleife zu vermeiden.
  Ist kein Hindernis im Weg, wird die Richtung des Roboters wieder in Richtung des Ziels gesetzt. Zudem wird das Hochzählen des Counters wieder gestoppt.

  In Zeile 60 findet dann die Bewegung des Roboters um einen Schritt statt, während in Zeile 61 die interne Position des Roboters auf den neuen Punkt gesetzt wird.

- In der Funktion *obstacleInWay (Zeilen 65-86)* ist die Funktionalität implementiert, um das Auftreffen des Roboters auf ein Hindernis zu ermitteln. Zu Beginn der Funktion erstellen wir zuerst eine 3×3-Matrix. Diese wird dann so befüllt, dass sie zu einer 90°-Rotationsmatrix (gegen den Uhrzeigersinn) wird. Anschließend wird in einer for-Schleife für alle vorhandenen Hindernisse zuerst die Distanz zwischen Roboter und Hindernis (Variable *dist*), sowie der Vektor, der auf vom Roboter aus auf den nächsten Punkt des Hindernisses zeigt, ermittelt. Dieser Vektor (Variable *out*) zeigt senkrecht auf das Hindernis. Ist die Distanz kleiner als ein vorgegebener Wert (Konstante *DIST_MIN*), wird der Vektor *out* mit der Rotationsmatrix multipliziert und normiert. Der resultierende Vektor zeigt parallel entlang des Hindernisses. Anschließend wird die Richtung des Roboters auf diesen neuen Vektor (Variable *out2*) gesetzt. Zudem wird die Nummer des gefundenen Hindernisses per *return*-Befehl zurückgegeben.

Was wir gelernt haben

Es war sehr interessant, einen solchen Algorithmus zu implementieren, da wir beide noch keinerlei Erfahrungen mit Wegfindungsalgorithmen gesammelt hatten.

Wir haben gemerkt, dass wir durch das Nachvollziehen der Arbeitsweise des Algorithmus diesen sehr gut implementieren konnten. Nach anfänglichen Schwierigkeiten konnten wir den gegebenen Coderahmen gut nutzen.