Hochschule Darmstadt

Lehrveranstaltung: Simulation von Robotersystemen
Dozent: Prof. Dr. Thomas Horsch
Laboringenieur: Rudi Scheitler
Versuch: Pfadsuche im Konfigurationsraum
Datum: 24.6.2016

Name	Matrikelnummer
Fabian Alexander Wilms	735162

Studiengang: Mechatronik

Abgabedatum: 24.6.2016

Inhaltsverzeichnis

Konfigurationsräume geben an, für welche Gelenkstellungen ein Roboter mit seiner Umgebung kollidiert. Dies kann nun zur Bahnplanung genutzt werden. Sind für Start- und Zielpose alle Gelenkstellungen bekannt, so können beide Posen durch je einen Punkt im Konfigurationsraum dargestellt werden. Vom Startpunkt aus soll jetzt ein Pfad zum Zielpunkt gesucht werden, wobei Kollisionen zu vermeiden sind.

1 Kollisionsfreie Bahnplanung im Konfigurationsraum (Breitensuche)

Es sind die Konfigurationsräume des letzten Praktikums sowie ein neuer gegeben. Da die Konfigurationsräume durch Stichproben entstanden sind, sind diese räumlich diskret und können als Bitmaps gespeichert werden. Das heißt, dass die Information, ob der Roboter mit der Umgebung kollidiert oder nicht nur mit einer endlichen Auflösung bekannt ist.

Jeder Konfigurationsraum stellt nun einen Graphen dar. Ein Graph ist eine Struktur, die aus einzelnen Knoten besteht, die untereinander verbunden sind. Jeder Knoten steht für eine Gelenkstellung, d.h. einen Pixel. In Fall zweidimensionaler Konfigurationsräume hat jeder Knoten 8 direkte Nachbarn. Der zu implementierende Algorithmus zur Bestimmung eines Pfads zwischen Start- und Zielpunkt nennt sich Breitensuche.

Es folgt eine Beschreibung des Algorithmus:

Der Startknoten wird in einer Warteschlange gespeichert (Schritt 1) und als bearbeitet markiert (Schritt 2). Falls die Warteschlange nicht leer ist, wird der älteste Knoten in dieser ausgelesen und aus ihr gelöscht (Schritt 3). Ist dieser der Zielknoten ist der Algorithmus beendet (Schritt 4). Anschließend untersucht man dessen direkte, noch nicht bearbeitete Nachbarn: Sie werden zunächst als bearbeitet markiert und, wenn die zugehörige Gelenkstellung keine Kollision verursacht, werden die Koordinaten der ursprünglichen Zelle in der neuen Zelle gespeichert und diese in der Warteschlange eingereiht (Schritt 5). Als nächstes wird zu Schritt 3 gesprungen.

Die folgenden Abbildungen illustrieren, dass zunächst alle direkten Nachbarn des aktuell betrachteten Knotens durchsucht werden, bevor man in die Tiefe geht.

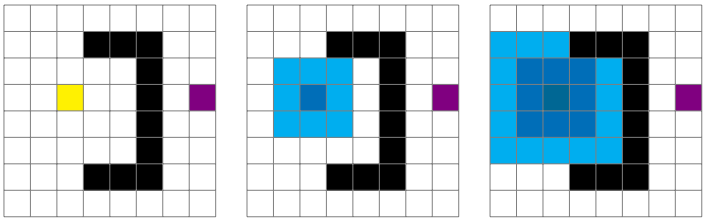

Abbildung 1: Konfigurationsraum: Die ersten drei Schritte einer BFS

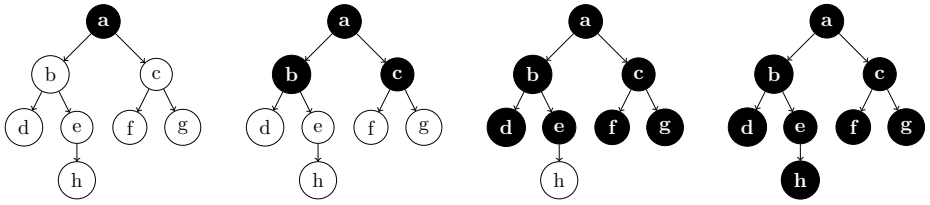

Abbildung 2: Graph: Die ersten vier Schritte einer anderen BFS

Implementiert wird die Breitensuche wie folgt:

Zu Beginn des Programms wird eine Bitmap-Datei eingelesen und im 2D-Array `char **cspace` gespeichert. Höhe und Breite der Datei werden ebenfalls gespeichert und verwendet, um ein weiteres 2D-Array zu erzeugen, diesmal vom Typ `Cell`. `Cell` ist ein `struct` und enthält folgende Elemente: `bool bMarked`, `int nLastX` und `int nLastY`.

Die Breitensuche findet in der Funktion `FindPath()` statt, der die Koordinaten des Start- und Zielpunktes übergeben werden. Diese Koordinaten beziehen sich auf das Koordinatensystem der Bitmap und nicht direkt auf die Gelenkvariablen.

Der erste Schritt ist es, die einzelnen Zellen des Konfigurationsraumes zu initialisieren. Mit dem Wert des Elements `bMarked` wird angegeben, ob eine Zelle schon besucht wurde. Zunächst werden alle Zellen auf `bMarked = false` gesetzt werden.

Um bei der anschließenden Bestimmung des Pfads in `main()` nicht für jede Zelle entlang des Pfads überprüfen zu müssen, ob X- und Y-Koordinate schon mit den Sollwerten übereinstimmen, wird `nLastX` der Startzelle auf -1 gesetzt. Somit muss nur jeweils 1 Wert abgefragt werden.

Dann wird die Startzelle in die Queue eingefügt (Schritt 1) und markiert (Schritt 2). Die erste Zelle in der Queue wird ausgelesen und aus der Queue gelöscht. Solange noch Elemente in der Queue sind, wird nun eine `while`-Schleife ausgeführt (Schritt 3). Ist die Zelle die Zielzelle, so wird die Funktion beendet (Schritt 4). Dann folgt eine Schleife, welche achtmal läuft, einmal für jede

Zelle, die potentiell an die aktuelle grenzt. Im Array `DirTable` stehen die relativen X- und Y-Koordinaten der Nachbarn, bezogen auf die aktuelle Zelle. Bei jedem der acht Schleifendurchläufe werden die relativen Koordinaten zu den absoluten aktuellen addiert (letztere sind für alle 8 Durchläufe gleich). Wurde die Zelle mit diesen neuen Koordinaten schon zuvor bearbeitet (`bMarked == 1`), so wird sie übersprungen. Andernfalls wird sie auf diese Art markiert und überprüft, ob die dazugehörige Gelenkstellung eine Kollision verursacht. Falls nicht, werden in der Zelle die vorherigen Koordinaten gespeichert (dies wird zur Rückverfolgung des Pfades benötigt) und die Zelle in der Warteschlange gespeichert (Schritt 5). Aufgrund der `while`-Schleife wird nun automatisch zu Schritt 3 gesprungen.

Abbildung 3: Beispiel-Konfigurationsraum

```
144 /*
145  *  FindPath - Breadth-First (Breitensuche)
146  *  Algorithmus zum finden eines Pfades vom Start zum Endpunkt (
         goal)
147  */
148 bool FindPath(int nStartX, int nStartY, int nGoalX, int nGoalY)
149 {
150     printf("Suche␣Pfad␣von␣x=%d␣y=%d␣nach␣x=%d␣y=%d\n", nStartX,
            nStartY, nGoalX, nGoalY);
151
152     // initalize cells
153     for (int my_y = 0; my_y < height; my_y++)
154     {
155         for (int my_x = 0; my_x < width; my_x++)
156         {
157             aCells[my_y][my_x].bMarked = false;
158         }
159     }
160
161     // create queue
162     std::queue<Coordinate> queue;
163
```

```
164     // starting cell
165     Coordinate c;
166     c.x = nStartX;
167     c.y = nStartY;
168
169     //  Folgende Zeilen demonstrieren die Manipulation von Queues
            (Warteschlange)
170     //  Einfach auskommentieren und testen
171
172     /*
173     printf("Queue leer? %d\n", queue.empty());       // Am Anfang
            ist die Queue leer
174
175     queue.push(c);                                   // Element in
             die Queue anstellen
176     printf("Queue leer? %d\n", queue.empty());       // Jetzt
            sollte die Queue nicht mehr leer sein
177
178     printf("Elemente in Queue: %d\n", queue.size()); // Anzahl
            Elemente in der Queue
179
180     Coordinate test = queue.front();                 // front()
            gibt das forderste Element in der Queue zurück,
181     printf("x: %d, y: %d\n", test.x, test.y);        // jedoch
            wird das Element dadurch nicht entfernt!
182
183     queue.pop();                                     // Entfernt
            das erste Element aus der Queue
184
185     printf("Queue leer? %d\n", queue.empty());       // Nun sollte
             die Queue wieder leer sein
186     */
187
188     // push first cell into queue
189     queue.push(c);
190
191     // mark starting cell for backtracing
192     aCells[nStartY][nStartX].nLastX = -1;
193
194     // mark starting cell
195      aCells[c.y][c.x].bMarked = 1;
196
197     int color = 1;
198
199     // while there's still cells in the queue...
200     while (!queue.empty())
201     {
202         // get the first element in the queue
203         c.x = queue.front().x;
204         c.y = queue.front().y;
205         queue.pop();
206
207         // if it has the desired coordinates we terminate
208         if (c.x == nGoalX && c.y == nGoalY)
```

```
209         {
210             printf("found: %d %d \n", c.x, c.y);
211             printf("last: %d %d \n", aCells[c.y][c.x].nLastX,
                    aCells[c.y][c.x].nLastY);
212             return true;
213         }
214 
215         Coordinate temp;
216         for (int i = 0; i < 8; i++)
217         {
218             // make sure we're still within the array's bounds
219             if (c.y + DirTable[i][1] >= 0 && c.y + DirTable[i][1]
                    < height && c.x + DirTable[i][0] >= 0 && c.x +
                    DirTable[i][0] < width)
220             {
221                 // if we haven't checked the next cell yet...
222                 if (aCells[c.y + DirTable[i][1]][c.x + DirTable[i
                        ][0]].bMarked == false)
223                 {
224                     // mark cell
225                     aCells[c.y + DirTable[i][1]][c.x + DirTable[i
                            ][0]].bMarked = true;
226 
227                     // if there's no collsion in the next cell...
228                     if (cspace[c.y + DirTable[i][1]][c.x +
                            DirTable[i][0]] == 0 || cspace[c.y +
                            DirTable[i][1]][c.x + DirTable[i][0]] ==
                            127)
229                     {
230                         // watch the BFS as it happens
231                         // cspace[c.y + DirTable[i][1]][c.x +
                                DirTable[i][0]] = 127;
232                         cspace[c.y + DirTable[i][1]][c.x +
                                DirTable[i][0]] = color;
233                          SaveAsPNG("cspace_bfs_debug.png", cspace,
                                 width, height);
234                          getchar();
235 
236                         aCells[c.y + DirTable[i][1]][c.x +
                                DirTable[i][0]].nLastX = c.x;
237                         aCells[c.y + DirTable[i][1]][c.x +
                                DirTable[i][0]].nLastY = c.y;
238                         temp.x = c.x + DirTable[i][0];
239                         temp.y = c.y + DirTable[i][1];
240                         queue.push(temp);
241                         color++;
242                     }
243                 }
244             }
245         }
246     }
247     return false; // Default Return
248 }
```

Listing 1: Bestimmung des Konfigurationsraumes

Bei der BFS in 8er-Nachbarschaft sind die abgesuchten Konfigurationen rechteckig angeordnet:

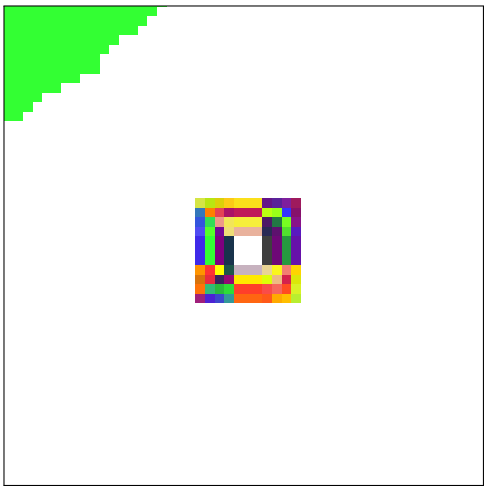

Abbildung 4: Beginn der Breitensuche (8er-Nachbarschaft)

Abgesuchte Konfigurationen sind im folgenden grün markiert:

Man sieht, dass ein – wenn auch nicht optimaler – Pfad gefunden wird:

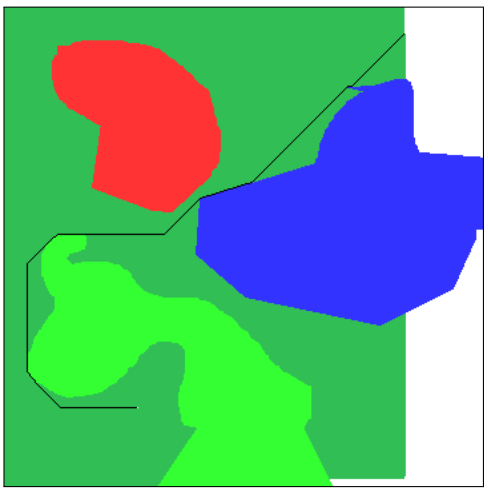

Abbildung 5: Pfad im Beispiel-Konfigurationsraum (8er-Nachbarschaft)

Für den TT-Roboter ergibt sich folgender Pfad:

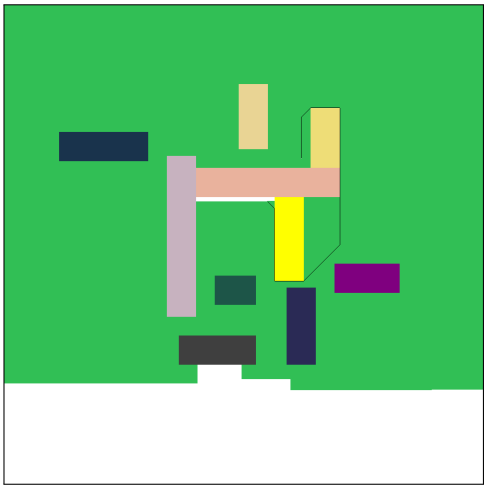

Abbildung 6: Bahn des TT-Roboters (8er-Nachbarschaft)

Führt man das erzeugte EasyRob-Programm mit aktiver TCP-Spur aus, so ergibt sich ein ähnliches Bild:

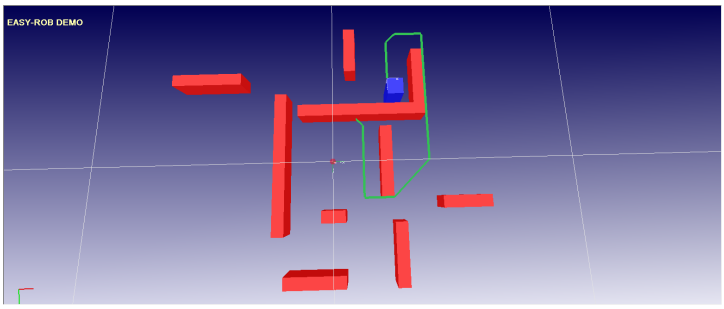

Abbildung 7: Bahn des RR-Roboters (8er-Nachbarschaft) in EasyRob

Zum Vergleich wird noch eine Pfadsuche in 4er-Nachbarschaft durchgeführt:

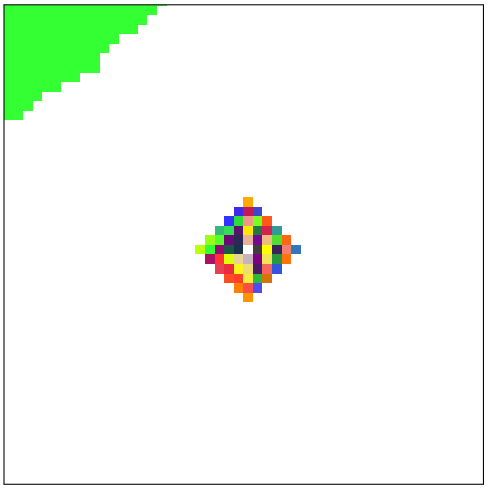

Abbildung 8: Beginn der Breitensuche (4er-Nachbarschaft)

Die abgesuchten Zellen sind in diesem Fall diamantförmig angeordnet

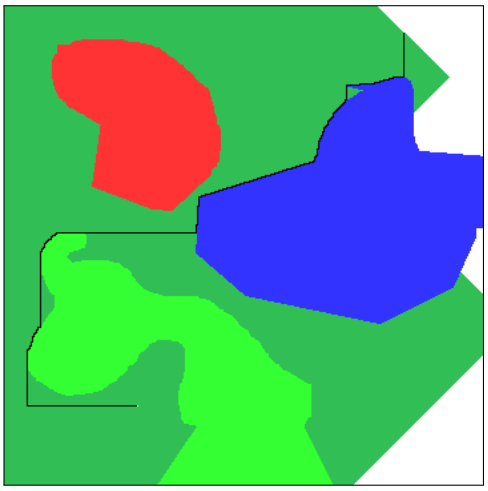

Abbildung 9: Pfad im Beispiel-Konfigurationsraum (4er-Nachbarschaft)

Es fällt auf, dass bei der Suche in 4er-Nachbarschaft nur horizontale und vertikale Pfadstücke entstehen und bei der Suche in 8er-Nachbarschaft auch Strecken, die eine Rotation von $\pm 45°$ haben.

Das bedeutet auch, dass eine 1 Pixel breite Diagonale nur bei der Suche in 8er-Nachbarschaft als Teil des Pfads in Frage kommt. Bei der Suche in 4er-Nachbarschaft wird bei folgendem Konfigurationsraum kein Pfad gefunden:

Abbildung 10: Modifizierter Beispiel-Konfigurationsraum; Q_{frei} enthält eine 45°-Diagonale

Bei der Suche in 8er-Nachbarschaft ist dies kein Problem:

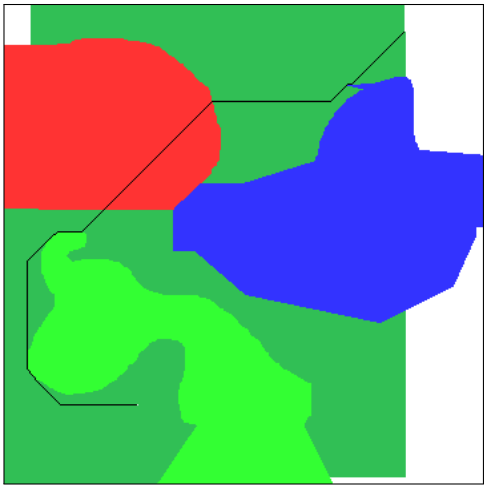

Abbildung 11: Pfad im modifizierten Beispiel-Konfigurationsraum (8er-Nachbarschaft)

Die BFS funktioniert ohne Änderungen in `FindPath()` auch für RR-Roboter:

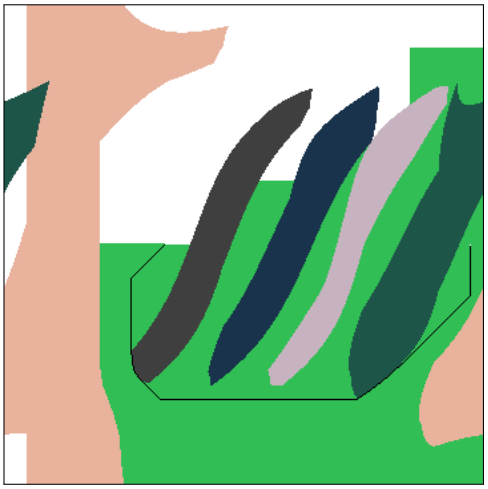

Abbildung 12: Bahn des RR-Roboters (8er-Nachbarschaft)

Beim RR-Roboter fällt ein Vergleich der Bahn im Konfigurationsraum und in der Simulation schwerer. Markante Segmente des Pfads lassen sich dennoch wiedererkennen, wie zum Beispiel der Bereich, in dem q_2 konstant ist und sich nur q_1 ändert.

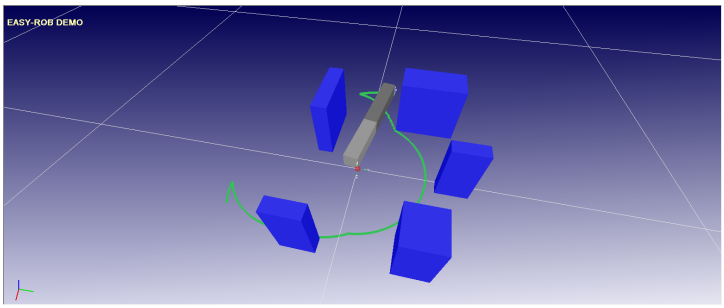

Abbildung 13: Bahn des RR-Roboters (8er-Nachbarschaft) in EasyRob

Da zu Beginn eine andere Abfragereihenfolge der Nachbarkoordinaten vorgegeben war als die hier benutzte, konnte man feststellen, dass die Güte des gefundenen Pfades deutlich von dieser Reihenfolge abhängt. Insbesondere der ursprünglich gefundene Pfad für den RR-Roboter wich stark vom oben dargestellten ab.