Pierre Smits

Bewegungsplanung für Roboter

Ergänzung zur Vorlesung
"Algorithmische Geometrie"
von Winfried Kurth
Sommersemester 2003

Ausarbeitung

BTU Cottbus
Pierre Smits
Matrikelnummer: 9807740
Pierre.Smits@web.de

Cottbus 2004

Inhalt:

1. Bewegen eines konvexen Polygons

Angenommen, der Roboter sei nicht mehr nur eine Scheibe, sondern ein konvexes Polygon, so kommt es zu ernsten Komplikationen:
Es könnte notwendig werden, den Roboter zu drehen, um ihn von seiner Position auf eine andere zu bewegen. Zunächst betrachten wir jedoch nur die Bewegungen, die mittels Translationen durchgeführt werden können. Obwohl diese Aufgabe schon komplizierter ist als bei einer Scheibe, kann man weiterhin mit Minkowski-Summen als Ansatz weiterarbeiten.

1.1 Beispiel (Minkowski-Summe)

Sei der Roboter R ein Quadrat mit einem festgelegten Bezugspunkt r in seiner linken unteren Ecke (Gegensatz zur Scheibe). Zur Betrachtung wird ein einfaches Polygon P (ein Pentagon) herangezogen (siehe Abb. 1.1).
Während R um ∂P fährt, zieht r die Grenzen von P^+, der Region, in die r nicht eindringen kann:
Durch die Wahl des Bezugspunkts wächst P je nach Kante um einen anderen Betrag. So ist es beispielsweise möglich, dass, wie entlang der Kante e_0, P und P^+ zusammenfallen, weil r e_0 berührt.
Entlang e_1 und e_4 wirkt sich R unterschiedlich aus: Bei e_1 horizontal über die Breite von R und bei e_4 vertikal durch die Höhe von R.
Besonderes Augenmerk sollte man auf das Verhalten von r in der konkaven Ecke legen.
Hier überschneiden sich die von r gezogenen Grenzen, wobei nur deren Äußeres die wahren Grenzen für R in der Annäherung an P widerspiegelt.

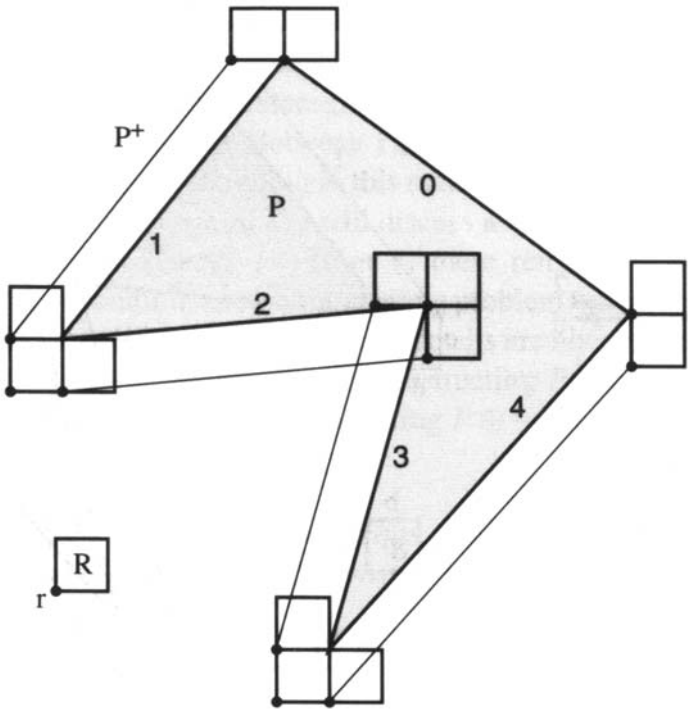


Abb.1.1: Beispiel Minkowski-Summe

1.2 Minkowski-Addition

Hinweis: Verwendung des Begriffs „Minkowski-Addition", da „Minkowski-Subtraktion" bereits für ein anderes Konzept (Guggenheimer 1977) verwendet wird.
Bei einer Scheibe galt noch $P^+ = P \oplus R$. Hier läge jedoch $P \oplus R$ außerhalb von e_0, was bei P^+ nicht der Fall ist. Eine passende Berechnung ist deshalb die Minkowski-Summe von P mit einer Spiegelung von R am Bezugspunkt r. Sei die Spiegelung von R an r gleich –R. Somit ist es relativ einfach möglich, r von ∂P weg zu halten (siehe Abb. 1.2).
Denn so ist $P^+ = P \oplus -R$. Bei einer Scheibe ist $R = -R$ (Symmetrie bzgl. des Mittelpunkts von R). Allgemein kann all dies wie folgt notiert werden.

Theorem 1.1:
Sei R eine Region (der Roboter) und $r \in R$ ein Bezugspunkt.
Sei P ein (beliebiges) Hindernis.
Dann ist die Region $P^+ = P \oplus -R$ die Menge von Punkten, die für r verboten sind:
Wenn R so bewegt wird, dass …

- … r im Inneren von P^+ ist, dann dringt R in P ein.
- … r auf ∂P^+ liegt, dann berührt ∂R ∂P.
- … r ausschließlich außerhalb von P^+ liegt, gilt: $R \cap P = \varnothing$.

Wie man hier sieht, müssen R und P nicht konvex sein. Sie müssen noch nicht einmal Polygone sein. Trotzdem betrachten wir im Folgenden weiterhin den einfachen Fall, dass P und R Polygone sind und R konvex ist.

1.3 Konstruktion der Minkowski-Summe

Folgend wird zunächst die Methode zur Konstruktion der Minkowski-Summe von zwei Polygonen skizziert (siehe Code 1.1).
In Abb. 1.2 sieht man bereits die jeweils gegen den Uhrzeigersinn benannten Kanten von $P^+ = P \oplus -R$. 0 bis 4 steht für die Kanten von P und a bis d für die Kanten von –R. Somit können alle Kanten von P^+ mit {0, 1, 2, 3, 4, a, b, c, d} benannt werden, je nachdem, ob die Kante von P oder –R erzeugt wird.

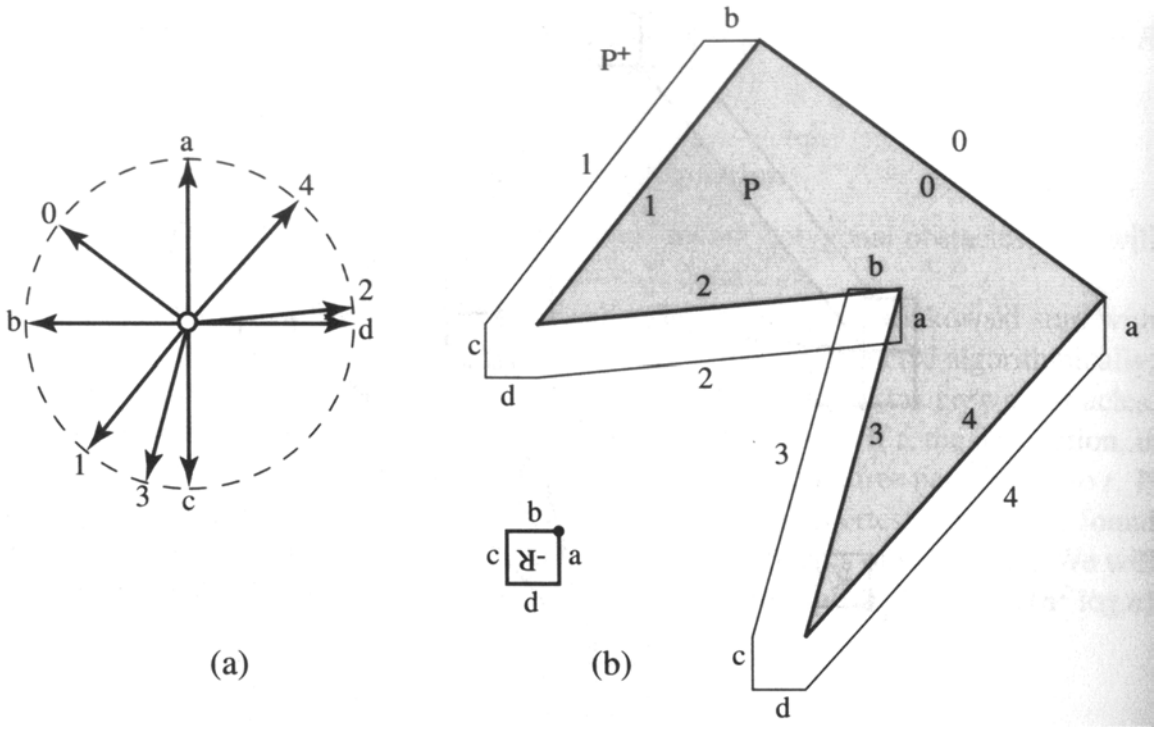


Abb.1.2: Stern-Diagramm und Minkowski-Addition

Wenn R entlang Kante „2" von P fährt, folgt der Bezugspunkt r einer parallelen Kante von P^+ die ebenfalls „2" genannt wird.
Wenn R mit „c" an der Ecke von „1" und „2" entlangfährt, bezeichnet man die durch den Bezugspunkt erzeugte Kante ebenfalls mit „c". So entsteht ein sich selbst schneidender Pfad (τ), dessen Außengrenzen P^+ umschließen. Dieser Pfad (τ), auch *Windung* von P und –R (vgl. Guibas, Ramshaw u. Stolfi 1983) genannt, ist ein einfaches Verfahren, um P^+ zu finden.
Hinweis: Wie zu sehen, können Teile des Pfades τ innerhalb von P^+ liegen.

Das Muster der Benennung der Kanten von τ kann leicht anhand des Stern-Diagramms (siehe Abb. 1.3), der Randvektoren von P und –R, verstanden werden. Falls P groß und R kein ist, kann man auch vereinfacht sagen, dass die Kanten von τ mit denen von P^+ korrespondieren (vermischt mit einzelnen Kanten von –R). Man sieht, dass Bezeichnungen von τ der Reihe nach (0, b, 1, c, d, 2, a, b, 3, c, d, 4, a) die Teilfolge (0, 1, 2, 3, 4), also P enthalten. Mit Hilfe des Stern-

Diagramms ist es möglich, die beigemischten Kanten von –R vorauszusagen.

Im Stern-Diagramm wird jeder Kante ein Vektor, entsprechend seiner Richtung beim Durchlauf gegen den Uhrzeigersinn mit einheitlicher Länge, zugeordnet. Alle Vektoren werden am gleichen Ursprung abgetragen (siehe Abb. 1.2).
Beginnend bei 0, dreht man gegen den Uhrzeigersinn um das Stern-Diagramm.
Zwischen den Indizes i und i+1 der Kanten von P notiert man alle dazwischen liegenden Indizes von –R.

Beispiel:

- Zwischen 0 und 1 wird b angetroffen ➔ die Teilfolge (0, b, 1) entsteht.
- Zwischen 1 und 2 werden c und d angetroffen ➔ die Teilfolge (1, c, d, 2) entsteht.
- usw.

Um die Minkowski-Summe P^+ von τ zu erhalten, müssten noch die verbleibenden Selbstüberschneidungen von τ gefunden werden (siehe hierzu Guibas, Ramkumar). Es sollte klar sein, dass hierzu ein nicht weiter beschriebenes Verfahren existiert.

Wie kann hieraus die Komplexität der Berechnung von $P \oplus -R$ bestimmt werden?

Theorem 1.2:
Wenn P genau n und R eine feste (konstante) Anzahl von Ecken hat, dann ergeben sich folgende Komplexitäten für die Konstruktion von $P \oplus -R$.

Zeitkomplexität:

R	P	Größe der Summe	Zeitkomplexität
konvex	konvex	O(n)	O(n)
konvex	nicht konvex	O(n)	O(n² log n)
nicht konvex	nicht konvex	O(n²)	O(n² log n)

Tabelle 1.1: Zeitkomplexität in Abhängigkeit von n, der Anzahl der Ecken des Hindernisses P (Tabelle ermittelt durch Giubas, Toussaint, Sharir und Kaul, O`Connor u. Srinivasan)

1.4 Implementierung der Minkowski-Windung (Pfad τ)

Die meisten Bemühungen bei diesem Ansatz erfordert die Konstruktion des Stern-Diagramms. Mit diesem kann man leicht, durch wiederholte Addition der korrespondierenden (siehe Abb. 1.2) Kantenvektoren, die Minkowski-Windung herstellen. Die Konstruktion des Stern-Diagramms ist im Wesentlichen ein winkelweises Sortieren von Vektoren. Man vergleiche mit VL Algorithmische Geometrie: Algorithmus zur Berechnung konvexer Hüllen nach Graham. Analog dazu wird jeder Kantenvektor in einem Array von entsprechenden Kantenvektor-Strukturelementen abgespeichert und mittels `qsort` (Quicksort) sortiert (siehe Code 1.1).

```
typedef struct tPointStructure tsPoint;
typedef tsPoint *tPoint;
struct tPointStructure{
    int     vnum;
    tPointi v;
    bool    primary;
};

#define PMAX 1000    /*max # of points*/
typedef tsPoint tPointArray[PMAX];
static tPointArray P;

int m;   /*total number of points in both polygons*/
int n;   /*number of points in primary polygon*/
int s;   /*number of points in secondary polygon*/

main()
{
    tPoint p0 = { 0, 0 };
    int j0;                    /*index of start point*/

    j0 = ReadPoints( p0 );
    Vectorize();
    qsort(
        &P[0],                 /*pointer to 1st element*/
        m,                     /*number of elements*/
        sizeof( tsPoint ),     /*size of each element*/
        Compare                /*-1,0,+1 compare function*/
    );
    Convolve( j0, p0 )
}
```

Code 1.1: Datenstruktur und Hauptprogramm zur Berechnung der Minkowski-Windung

`bool primary` zeigt an, ob ein Punkt zu P (`TRUE`) oder zu R (`FALSE`) gehört. Dieser Wert wird ebenso wie die Folge der Punkte von P, sowie R (➔ `vnum`) von `ReadPoints` ermittelt. Da man nur –R benötigt, wird bereits beim Einlesen das Polygon R gespiegelt und als –R weiterverwendet. Die Gesamtzahl der Ecken m ergibt sich aus m = n + s (mit n = #Kanten in P und s = #Kanten in R).
Der erste Schritt zur Berechnung ist nun, aus den Eckpunkten die Kantenvektoren „ohne" Normalisierung (→ vereinfacht Konstruktion von τ, vgl. Abb. 1.2) mittels `Vectorize` (Code 1.2) zu bestimmen.

```
void vectorize( void )
{
    int i;
    tPointi last;    /*holds last vector difference*/

    SubVec( P[0].v, P[n-1].v, last );
    for( i=0; i<n-1; i++ )
        SubVec( P[i+1].v, P[i].v, P[i].v );
    P[n-1].v[X] = last[X];
    P[n-1].v[Y] = last[Y];
    SubVec( P[n].v, P[n+s-1].v, last );
    for( i=0; i<s-1; i++)
        SubVec( P[n+i+1].v, P[n+i].v, P[n+i].v );
    P[n+s-1].v[X] = last[X];
    P[n+s-1].v[Y] = last[Y];
}
```

Code 1.2: Vectorize

Festlegungen zur Reihenfolge der Kantenvektoren, welche in alle Richtungen zeigen können (siehe Code 1.3):

- Ein Vektor entlang der negativen x-Achse hat den kleinstmöglichen Winkel (0-Winkel);
- Ein Vektor a wird als „Nachfolger" eines Vektors b bezeichnet, wenn er gegen den Uhrzeigersinn vom 0-Winkel aus gesehen nach b aufgetragen wird.
- Bei Vektoren mit gleichem Winkel bezeichnet man den Längeren als Nachfolger.
- Gleicher Winkel + Gleiche Länge ➔ Vektoren gleich ➔ kein Nachfolger

```
int compare ( const void *tpi, const void *tpj )
{
    int a;           /*AreaSign result*/
    int x, y;        /*projections in 1st quadrant*/
    tPoint pi, pj;   /*recasted points*/
    tPointi Origin = { 0, 0 };
    pi = (tPoint)tpi;
    pj = (tPoint)tpj;
```

```
    /*A vector in the open upper halfplane is after a vector
      in the closed lower halfplane*/
    if      ( (pi->v[Y] > 0) && (pj->v[Y] <= 0) )
        return 1;
    else if ( (pi->v[Y] <= 0) && (pj->v[Y] > 0) )
        return -1;

    /*A vector on the x axis and one in the lower halfplane
      are handled by the Left computation below*/
    /*Both vectors on the x axis require special handling*/
    else if     ( (pi->v[Y] == 0) && (pj->v[Y] == 0) ){
        if      ( (pi->v[X] < 0) && (pj->v[X] > 0) )
            return -1;
        if      ( (pi->v[X] > 0) && (pj->v[X] < 0) )
            return 1;
        else if ( abs(pi->v[X]) < abs(pj->v[X]) )
            return -1;
        else if ( abs(pi->v[X]) > abs(pj->v[X]) )
            return 1;
        else
            return 0;
    }/*else if*/



    /*Otherwise, both in open upper halfplane, or both
      in closed lower halfplane, but not both on x axis*/
    else {
        a = AreaSign( Orign, pi->v, pj->v );
        if      ( a > 0 )
            return -1;
        else if ( a < 0 )
            return 1;
        /*begin collinear*/
        else{
            x = abs(pi->v[X]) - abs(pj->v[X])
            y = abs(pi->v[Y]) - abs(pj->v[Y])
            if      ( (x < 0) || (y < 0) )
                return -1 ;
            else if ( (x > 0) || (y > 0) )
                return 1;
            /*points are coincident*/
            else
                return 0;
        }
        /*end collinear*/
    }
}
```

Code 1.3: Compare

Nachdem alle Vektoren mittels `qsort` in die richtige Reihenfolge gebracht wurden, muss diese Liste nur noch (mehrmals) durchlaufen (`i=(i+1)%n`), analog zum Kreisen um das Stern-Diagramm, und die entsprechenden Vektoren gezeichnet werden (siehe Code 1.4). Begonnen wird an einem vordefinierten Punkt (`p`), mit einem festgelegten Vektor. In diesem Codebeispiel wird der kleinste Vektor (ohne Vorgänger) des Polygons P verwendet. Alternativ könnte man auch in der rechten oberen Ecke des Polygons beginnen.

```
void Convolve( int j0, tPoint p )
{
    int i;          /*index into sorted edge vectors P*/
    int j;          /*primary polygon index*/

    MoveTo_i( p );
    i = 0;       /*start an angle –pi, rightward vector*/
    j = j0;      /*start searching for j0*/
    do {
        /*advance around secondary edges until next
          j is reached*/
        while ( !( P[i].primary && P[i].vnum == j ) ) {
            if ( !P[i].primary ){
                AddVec( p, P[i].v, p );
                LineTo_i( p );
            }
            i = (i+1) % m;
        }
        /*advance one primary edge*/
        AddVec( p, P[i].v, p );
        LineTo_i( p );
        j = (j+1) % n;
    } while ( j != j0 );

    /*finally, complete circuit on secondary/robot polygon*/
    while ( i != 0 ) {
        if ( !P[i].primary ) {
            AddVec( p, P[i].v, p );
            LineTo_i( p );
        }
        i = (i+1) % m;
    }
}
```

Code 1.4: Convolve (fettgedruckt: vgl. Postscript-Befehle)

Hinweis:

- `j=(j+1)%n;` für Schleife über Vektoren von P.

Im folgend gegebenen Beispiel (siehe Abb. 1.3) werden 10 Schleifendurchläufe über die m (mit m = n + s = 22 + 8 = 30) Vektoren (siehe Abb. 1.4) benötigt. Die 10 Durchläufe werden auch anhand der 10 Schleifen der Minkowski-Windung τ sichtbar.

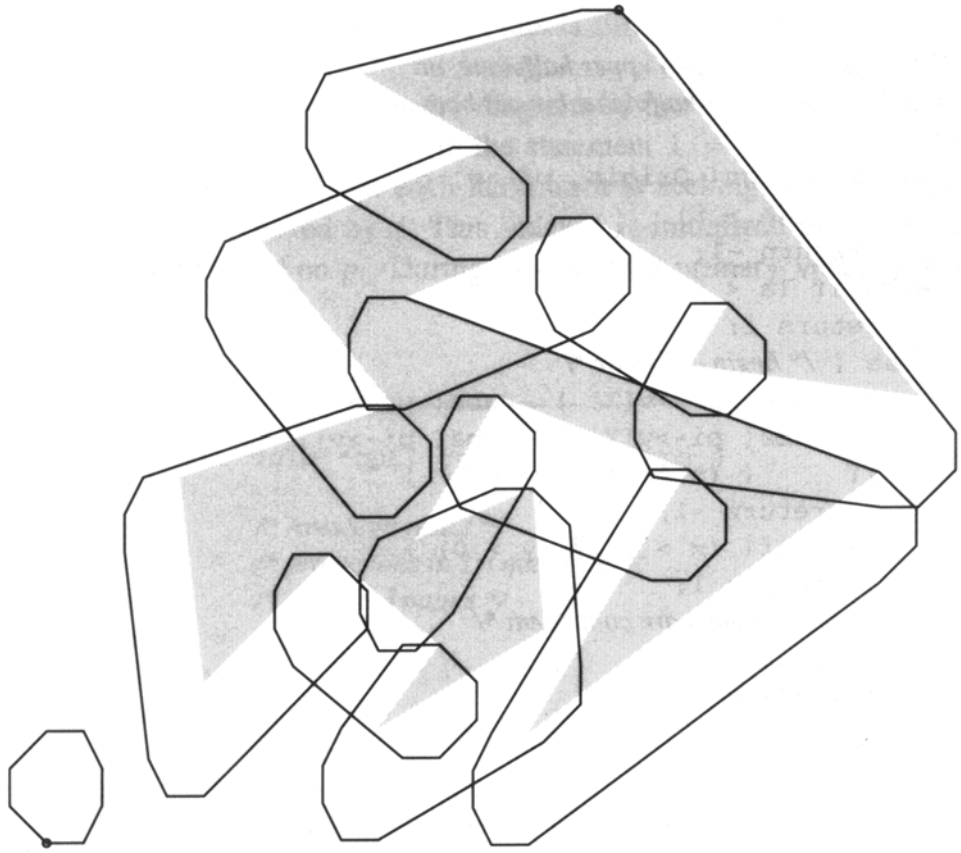


Abb.1.3: Minkowski-Windung eines nicht konvexen Polygons P und eines konvexen 8-Ecks R

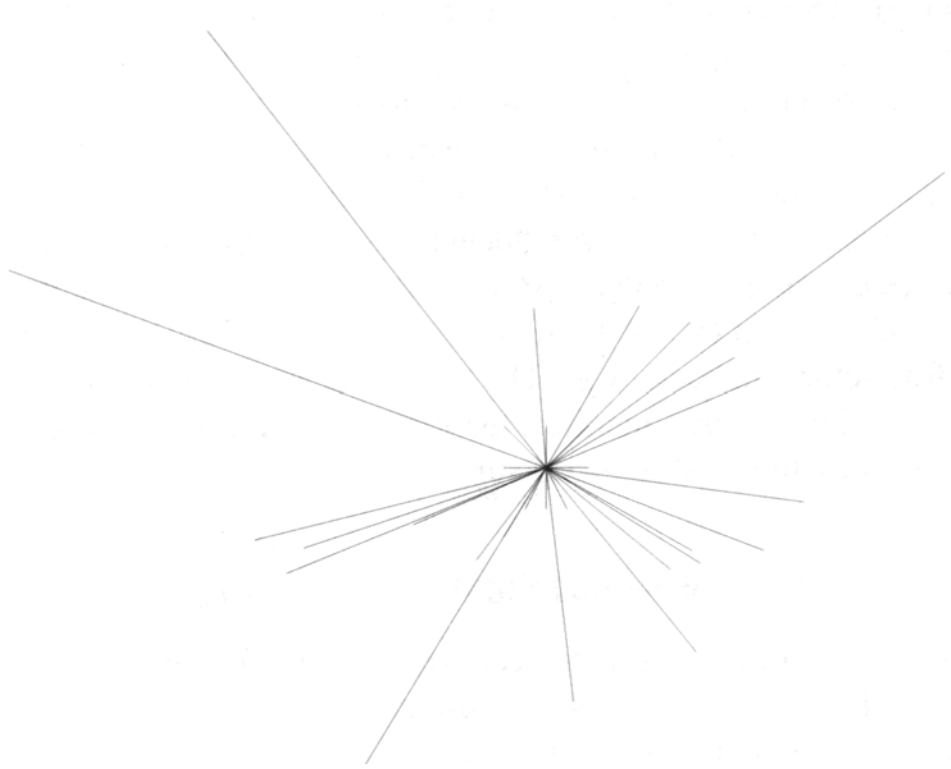


Abb.1.4: Sterndiagramm der Kantenvektoren von P und –R

1.5 Algorithmus der Bewegungsplanung (konzeptuell)

Gegeben:

- konvexes Polygon R
- Hindernispolygone P_1, P_2, …, $P_m \in \mathbb{P}$ mit insgesamt n Ecken.

Grob-Algorithmus:

1. Erweiterung $\forall P_i \in \mathbb{P}$: $P_i^+ = P_i \oplus -R$.
2. Bilde Vereinigung: $P^+ = U_i P_i^+$.
3. Finde die zusammenhängende Fläche (Komponente) C, die den Startpunkt s und den Zielpunkt t enthält.
4. Finde einen Weg von s nach t innerhalb von C.

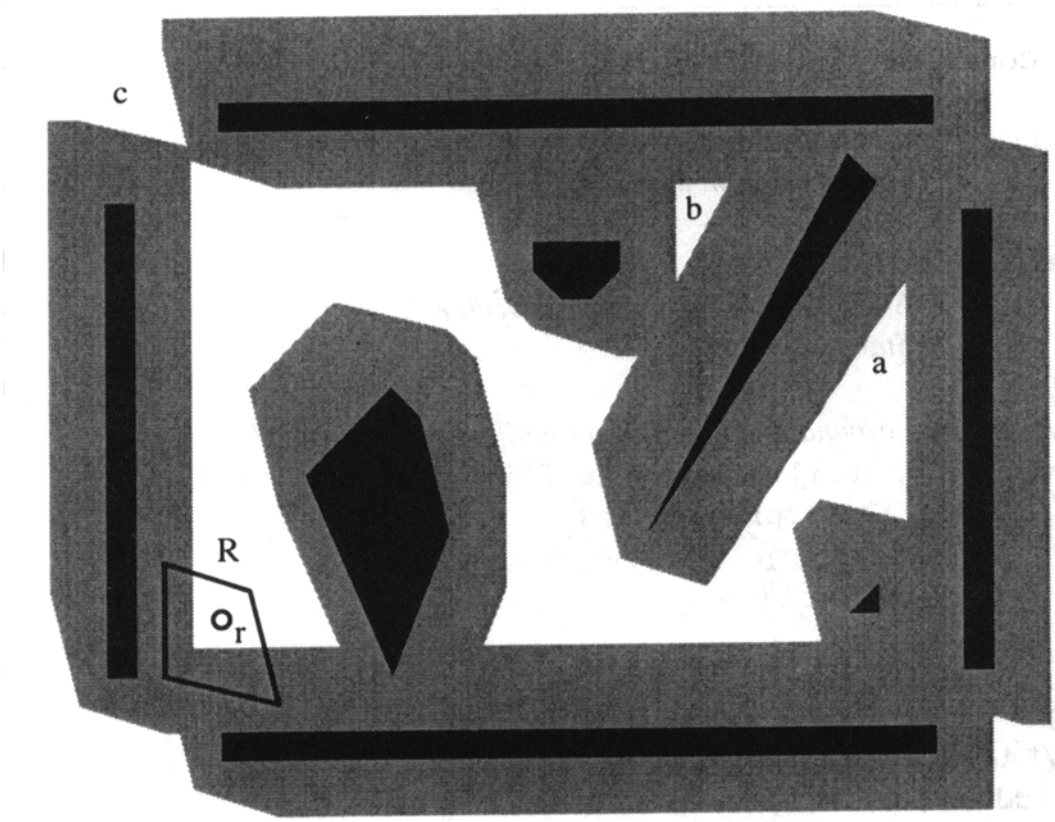


Abb.1.5: Roboter in der Ebene mit mehreren Hindernissen und Zielpunkten

Wie in Abb. 1.5 zu sehen, lassen sich hieraus einige Schlussfolgerungen ziehen. Um entscheiden zu können, ob ein Punkt von einem anderen aus erreichbar ist, muss man lediglich wissen, ob beide Punkte auf derselben **freien** zusammenhängenden Komponente liegen. Die Planung des Weges vom Start- zum Zielpunkt beschränkt sich dadurch auf diese Komponente. Die Umsetzung benötigt im bisher günstigsten Fall (nach Kedem & Shahir 1990) nur O(n log n) Rechenschritte!

Theorem 1.3:
Die Zeitkomplexität, einen Weg für einen konvexen Roboter R von einem Start- zu einem Zielpunkt zu finden, beträgt bei beliebig vielen Hindernispolygonen (die vermieden werden sollen) mit insgesamt n Ecken O(n log n). Falls der Roboter R k Ecken hat, lässt sich die Zeitkomplexität noch genauer angeben: O(kn log(kn)).

Diese Komplexität konnte von Aranov und Sharir (1997) auch für den 3-dimensionalen Raum erreicht werden!

1.6 Beispiel

Zum Zeitpunkt der Erstellung dieses Dokuments konnten unter folgendem Link die oben gegebenen Probleme anhand einer Java-Implementierung nachvollzogen werden.

http://www.diku.dk/hjemmesider/studerende/palu/start.html

2. Bewegen einer Leiter

Noch schwieriger als das vorangegangene Problem ist die Bewegungsplanung eines steifen strichförmigen Roboters unter polygonalen Hindernissen. Dieses Segment (Roboter) wird meist als Leiter (oder Stange) bezeichnet. Was es so kompliziert macht, ist, dass man hier Rotationen zulässt. Rotation gibt der Leiter drei „Freiheitsgrade" in ihrer Bewegung:

- horizontale Translation;
- vertikale Translation;
- Rotation.

Das heißt, dass es nicht möglich ist, dieses Problem in ein Problem einer Punktbewegung in zwei Dimensionen umzuwandeln, da solch ein Punkt nur zwei Freiheitsgrade besitzt. Jedoch ist es möglich, jedes Leiterproblem auf ein Problem der Bewegungsplanung eines Punktroboters zu verringern, der sich in einem dreidimensionalen Hindernisraum bewegt (Idee von Lozano-Perez und Wesley (1970)). Diese Idee ist in einem Beispiel (Abb. 2.1(a)) visualisiert. Die Leiter L ist zunächst horizontal und soll dem in (a) angezeigten Weg folgen. Das kann sie nur, wenn sie sich dreht. In (b) sieht man, wie die Hindernisse durch die Minkowski-Summe (Gegenstände und horizontale Leiter) wachsen. Dass C über A und B hinauswächst, zeigt deutlich, dass L nicht ohne sich zu drehen durch den Durchlass gelangt. (c) zeigt die gewachsenen Hindernisse, wenn die Leiter um 30° gedreht wird und in Teil (d) um 60°. In(d) ist der vertikale Durchlass zwischen A und B geöffnet, während der horizontale Durchlass zwischen A und B geschlossen ist. Diese Abbildungen zeigen, dass die Leiter einem Weg folgen kann, der in (a) angezeigt wird, indem man sie zwischen A und B mit weniger oder keiner Umdrehung bewegt und dann durch stückweise Rotation auf 60° oder mehr dreht, um sie vertikal zwischen A und C zu bewegen.

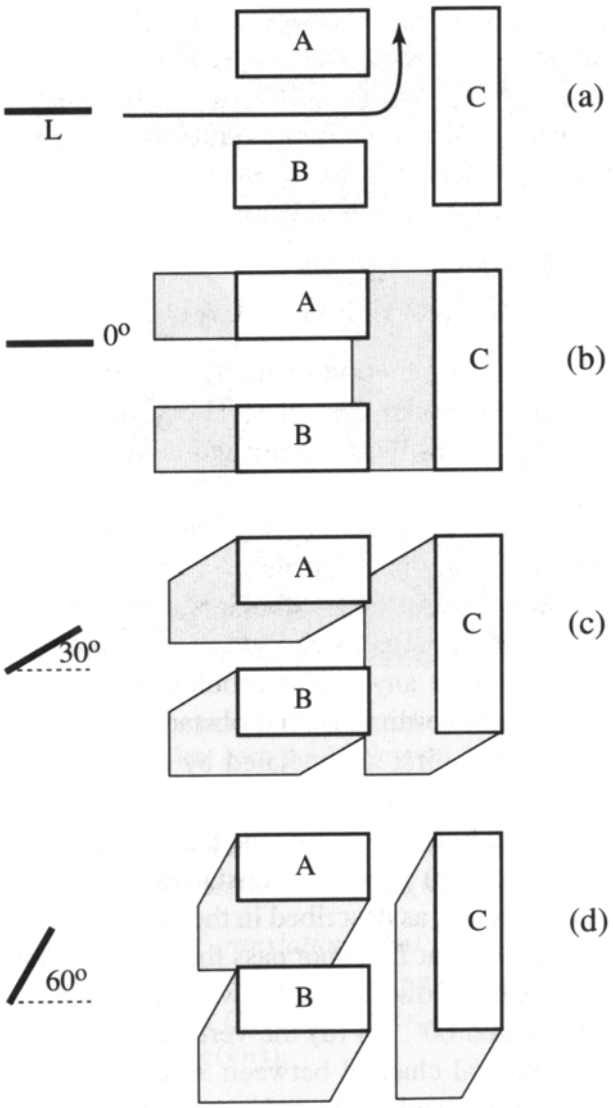


Abb. 2.1: Bewegung und Rotation einer Leiter

Man stelle sich vor, dass alle gewachsenen Hindernisse für alle Rotationen θ in allen parallelen Flächen gestapelt werden (siehe Abb. 2.2). Jeder Punkt (x, y, θ) in diesem Raum stellt eine Position des Bezugspunktes der Leiter dar. Die Fläche, auf der er liegt, stellt die Rotation dar. Somit ist bewiesen, dass das Problem des Verschiebens einer Leiter in zwei Dimensionen in ein ähnliches Problem der Bewegung eines Punktes in drei Dimensionen umgewandelt werden kann. Sei dieser 3-dimensionale Raum der *Konfigurationsraum* für den Roboter/die Leiter.

Obwohl nicht aus Abb. 2.2 ersichtlich, sind die Hindernisse kein von vielen Flächen begrenzter „Körper". In jeder θ-Ebene sind sie vieleckig. Die komplizierten Formen entstehen durch die Verdrehung entlang der θ-Richtung. Sei der Raum, in dem sich der Bezugspunkt frei bewegen kann, der „freie Raum". Stände man am Startpunkt s in diesem freien Raum, würde man einen tiefen Raum mit verdrehten Wänden sehen.
➔ Es gibt einen Weg für die Leiter, falls der Zielpunkt t im gleichen verbundenen Bestandteil des freien Raums ist wie s (vgl. Zusammenhangskomponente).

Hieran sieht man, dass diese Methode nicht auf das gegebene Problem mit einer Leiter beschränkt ist, sondern sich ebenso auf polygonale Roboter anwenden ließe. Es ist sogar möglich, diese Grundidee auf beliebige 3-dimensionale Roboter und Hindernisse zu erweitern.

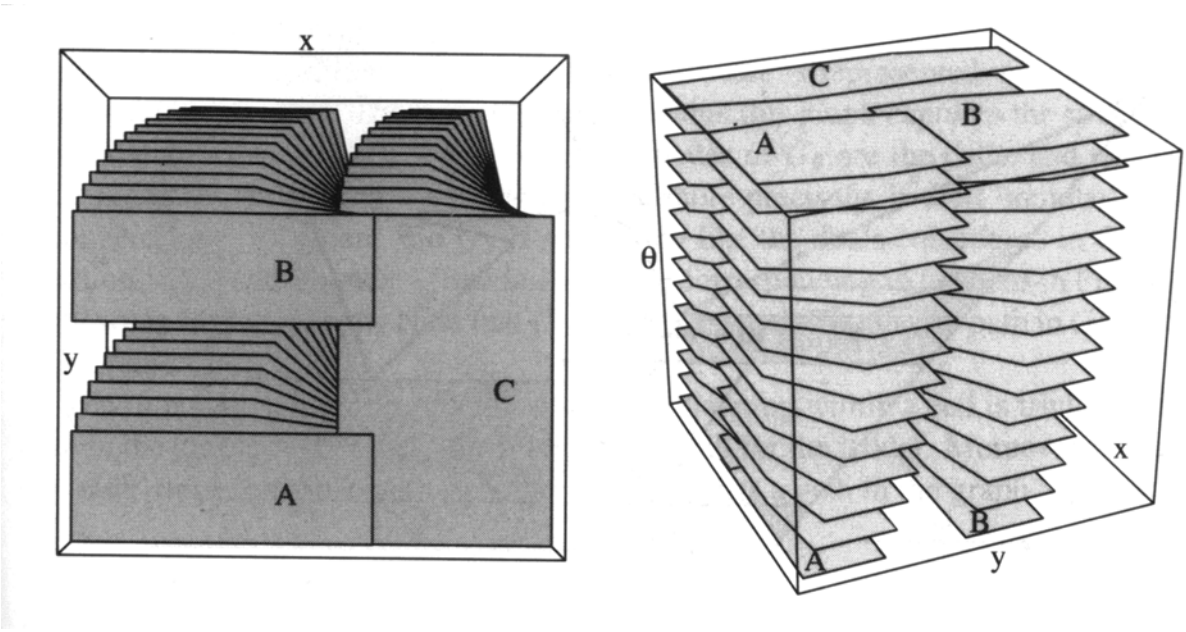


Abb. 2.2: Raum der ebenenweise gewachsenen Hindernisse

Eine Darstellung eines Konfigurationsraumes zu konstruieren und dann einen Weg im Inneren zu finden, ist eine schwierige Aufgabe. Seine Bedeutung hat eine intensive Forschung gefördert, und diese Konfigurationsräume werden teilweise bis hin zu 6-dimensionalen Räumen konstruiert, um dann Wege für entsprechende Roboter zu finden. Folgend werden zwei unterschiedliche Methoden für das Lösen des Problems der Bewegungsplanung – Finden eines freien Wegs durch den Konfigurationsraum – für die Leiterbewegung betrachtet.

2.1 Aufteilung in Zellen

Diese Methode zur Lösung von Problemen der Bewegungsplanung wurde von Schwartz und Sharir in 5 Abhandlungen – „piano movers" papers (Klavierträger) – entwickelt. Diese sichern die Lösung vieler Bewegungsplanungsprobleme durch Algorithmen mit polynomialer Zeitkomplexität, abhängig von den Problemdetails.

Definition einer Zelle
Wesentlich an der Methode der Zellaufspaltung ist: 1. die Aufteilung ungeordneter Konfigurationsräume in eine finite Anzahl „gutartiger" Zellen, und 2. einen Weg im Raum, durch Finden eines Weges durch die Zellen, zu finden (siehe Abb. 2.3). Das Beispiel besteht aus zwei Dreieckshindernissen und einer eingrenzenden (offenen) polygonalen Wand. Die Ausrichtung der Leiter L ist zunächst horizontal. Der Bezugspunkt sei links an der Pfeilspitze. Eine *Zelle* ist eine verbundene Region innerhalb eines freien Raumes des dazu passenden Konfigurationsraumes. Da die Ausrichtung der Leiter fest ist, ist der Konfigurationsraum nur die Fläche, und der freie Raum ist der Rest, der übrigbleibt, nachdem die Hindernisse durch die Minkowski-Summe mit der Leiter erweitert wurden (vgl. P^+). Um eine präzise Definition einer Zelle zu erhalten, ordnet man jedem Hindernisrand eine Signatur zu (siehe Abb. 2.3). Sei ∞ die Signatur, um einen „umgebenden" (hier rechten) Rand/Kante als unendlich weit darzustellen.

Eine Zelle ist eine Ansammlung von freien Punkten x, alle mit den gleichen vorwärts/rückwärts Signaturpaaren:

- Vorwärtssignatur: Name der Kante, die bei Vorwärtsbewegung (in Pfeilrichtung) vom freien Punkt x aus von L berührt wird;
- Rückwärtssignatur: Name der Kante, die bei Rückwärtsbewegung (entgegen Pfeilrichtung) vom freien Punkt x aus von L berührt wird.

Beispiel:
In Abb. 2.3 hat die Zelle A die Signaturen (3,2), die Zelle B die Signaturen (3,8), die Zelle C hat die Signaturen (1,9) und keine Zelle hat die Signaturen (3,6), da es keine freien Punkte zwischen diesen beiden Kanten gibt.

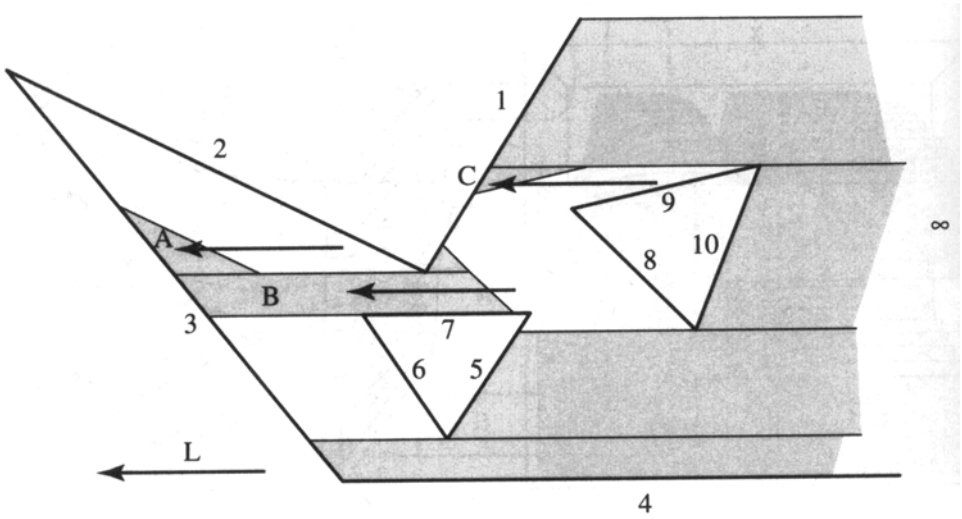


Abb. 2.3: Zellaufteilung bei horizontaler Leiter

Verbindungsgraph G_θ

Bei diesem Ansatz wird die Zellenstruktur durch einen Graphen G_θ dargestellt. Der Index θ gibt an, dass dieser Graph die Struktur für eine bestimmte Ausrichtung (θ) der Leiter repräsentiert.
Die Knoten von G_θ sind die Zellen. Zwei Knoten sind durch eine Kante verbunden, wenn sich die Zellen berühren, also wenn sich ihre Grenzen ein Segment, welches nicht leer ist, teilen. Der Graph G_θ, der den Zellen in Abb. 2.3 entspricht, wird in Abb. 2.5 dargestellt. Vorsicht, der Graph ist nicht zusammenhängend. → Es gibt keinen Weg in G_θ von A nach C.

Durch diesen Graphen wird die Bewegungsplanung stark vereinfacht. Es ist leicht möglich, die Pfade des Graphen G_α in einen Pfad für die Leiter umzuwandeln. Zudem kann man sagen, dass die um α (hier θ) geneigte Leiter nur dann von einer Zelle in eine andere bewegt werden kann, wenn ein Pfad in G_α zwischen diesen beiden Zellen existiert.

Kritischer Winkel

Wird nun die Leiter etwas rotiert, ändert sich zwar das Aussehen der Zellen ein wenig, der Graph sowie die damit ausgedrückten Zellnachbarschaften bleiben jedoch erhalten. Wenn jedoch ein kritischer Winkel θ^* erreicht wird, ergibt sich für G_{θ^*} eine gegenüber G_θ geänderte Struktur. Dreht man L gegen den Uhrzeigersinn, entstehen 2 neue Zellen: (7, 8) und (4, ∞). (Siehe Abb. 2.4). Ist die Leiter parallel zu 9, verschwindet Zelle C (vgl. Abb. 2.3 und 2.4). Somit entsteht ein neuer Graph (siehe Abb. 2.5 b)). Festzuhalten bleibt hierbei, dass der Grund für kritische Winkel immer eine entsprechende Parallelität zu Kanten ist und es somit höchstens $O(n^2)$ kritische Winkel geben kann.

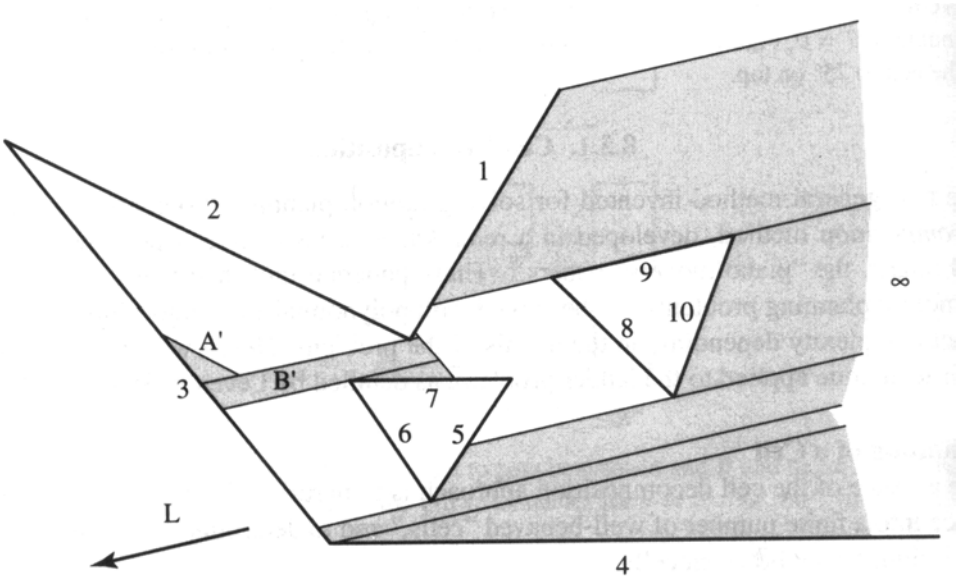
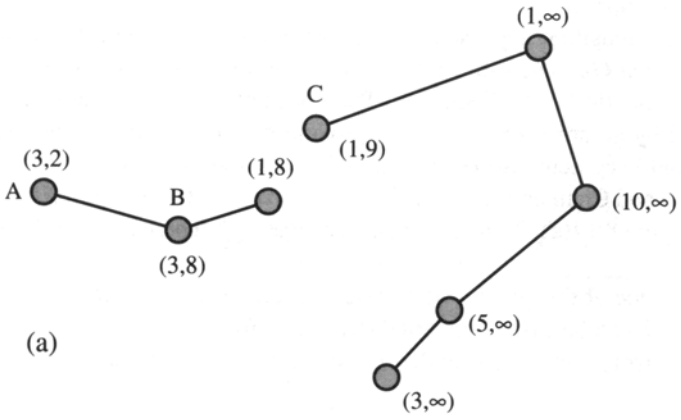


Abb. 2.4: Zellaufteilung bei gekippter Leiter



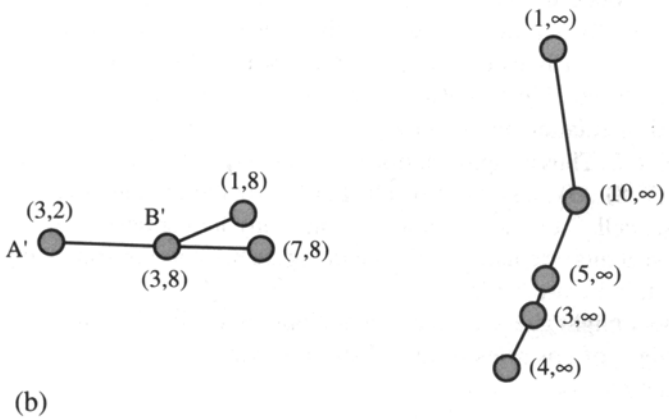


Abb. 2.5: Graph für a) horizontale und b) gekippte Leiter

Verbindungsgraph G

Nun soll ein übergreifender Verbindungsgraph G erstellt werden, der alle Informationen der Graphen G_θ enthält. Hierzu wird die Definition einer Zelle erweitert, um Regionen (gleiche vorwärts- und rückwärts-Signaturpaare) des 3-dimensionalen Konfigurationsraums darstellen zu können. Diese werden entsprechend θ übereinander gelegt. So sind die Punkte in Zelle A aus Abb. 2.3 in der gleichen 3-dimensionalen Zelle wie die Punkte in Zelle A` aus Abb. 2.3. Jede eindeutig abgegrenzte, 3-dimensionale Zelle ist ein Knoten von G, und zwischen zwei Knoten existiert wiederum eine Kante, wenn sich ihre Zellen berühren, d.h. wenn sie sich einen gemeinsamen nicht leeren Bereich (vgl. gemeinsame Kante) teilen.

Die Graph G kann konstruiert werden, indem man G_0 aufbaut, G mit G_0 initialisiert und aufsteigend alle kritischen Winkel durchwandert und dabei die notwendigen Änderungen an G_θ in G mit übernimmt.
So wurde das Problem der Bewegungsplanung wieder auf ein Graphenproblem reduziert, in welchem man einen Weg zwischen dem Knoten der Zelle, die den Startpunkt s, und dem Knoten der Zelle, die den Zielpunkt t enthält, findet. Gibt es keinen Weg in G, so ist es nicht möglich, mit der Leiter von s nach t zu kommen.

2.2 Retraktion

Die "Retraktions-Methode" ist eine alternative Methode zur Bewegungsplanung.

Voronoi-Diagramm

Punkte auf Kanten des Voronoi-Diagramms haben den gleichen Abstand zu mindestens 2 gegebenen Seiten.
Hauptsächlicher Bestandteil der Retraktions-Methode ist es, ein „Voronoi-Diagramm" der Leiter zu konstruieren, um einen Weg von s nach t innerhalb des Netzwerkes des Diagramms zu ziehen.
Für eine fest Ausrichtung der Leiter L definiert man das Voronoi-Diagramm der Hindernisse in Bezug auf L als die Menge der freien Punkte x, für die, wenn L auf den Bezugspunkt an x gesetzt wird, L von mindestens zwei Hindernispunkten den gleichen Abstand hat. Der Abstand eines Punktes p zu L ist die minimale Länge jeden Liniensegments von p zu einem Punkt auf L. So wie die Punkte im Abstand r von einem Punkt einen Kreis bilden, so bilden die Punkte im Abstand r von L die Form einer „Rennstrecke" (ein Oval) bestehend aus 2 parallelen, durch Halbkreise verbundenen Linien (siehe Abb. 2.6). Wie in Abb. 2.6 zu sehen, hat L in Position A (B, …) von e_2 (e_1, …) und e_3 (e_9, …) den gleichen Abstand. Da das Diagramm nicht verbunden ist, sieht man, dass es mit diesem Winkel z.B. keinen Weg von A nach B gibt.

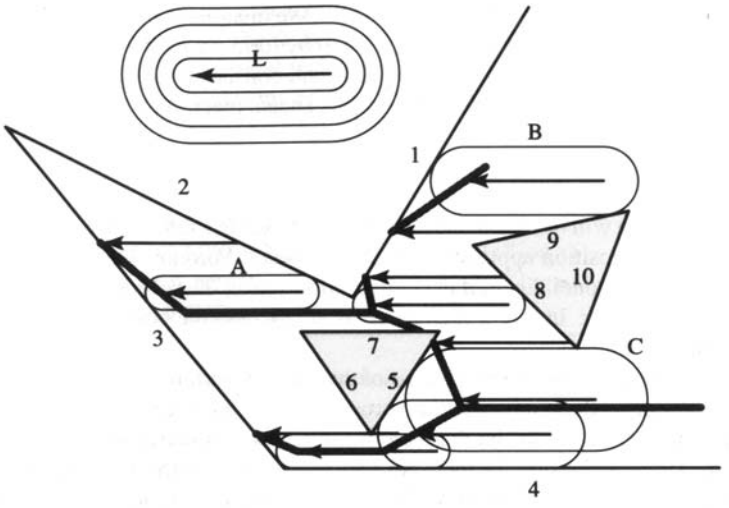


Abb. 2.6: (Teil-) Voronoi-Diagramm

Da sich L im Diagramm mit seinem Bezugspunkt immer auf den Kanten bewegt, hat L immer den für den Winkel größtmöglichen Abstand von den Hindernissen (eine sehr nützliche Eigenschaft für einen Roboter, der versucht, Zusammenstöße mit Hindernissen zu vermeiden).
Wie unter 2.1 werden nun Voronoi-Diagramme für jeden Wert von θ orthogonal zur θ-Achse übereinander gelegt. Dadurch bildet sich ein Voronoi-Diagramm für den ganzen Konfigurationsraum. Man kann sehen, dass das Diagramm aus mehreren verdrehten Schichten besteht, und aus "Graten", wo zwei der Schichten aneinanderstoßen.

Retraktion

Wiederum wird das Problem auf die Suche eines Pfades im Graphen reduziert. Hier bilden die Überschneidungen ein Netz von Kurven im Konfigurationsraum, welche den Graphen $\mathbb{N}$ bilden:

- jede Kurve ist eine Kante in $\mathbb{N}$;
- jeder Schnittpunkt von 2 oder mehreren Kurven ist ein Knoten in $\mathbb{N}$.

Abschließende Schritte:

- Abbilden der Start- und Endpunkte (s und t) auf das Voronoi-Diagramm ➔ s' und t';
- Abbilden von s' und t' auf das Netz des Konfigurationsraumes.

Dann gibt es einen Weg für L von s nach t, wenn es einen Pfad in $\mathbb{N}$ von s' nach t' gibt.

Verwendet wird der daraus resultierende Weg (mit maximalen Abständen) u. a. bei Maschinen für automatischen Zuschnitt.

2.3 Komplexität

Wie bereits gezeigt, gibt es Konfigurationen, deren Lösung (Weg des Roboters) quadratisch viele voneinander verschiedene Bewegungen benötigt. Dies stellt eine untere Schranke für jeden Algorithmus dar, der diesen Weg ausgibt. Tabelle 2.1 zeigt einige erreichte Komplexitäten.

Autoren	**Zeitkomplexität**
Schwartz & Sharir 1983a	$O(n^5)$
O`Dunlaing et al. 1987	$O(n^2 \log n \log^* n)$
Leven & Sharir 1987	$O(n^2 \log n)$
Sifrony & Sharir 1987	$O(n^2 \log n)$
Vegter 1990	$O(n^2)$
O`Rourke 1985b	$\Omega(n^2)$

Tabelle 2.1: Komplexitäten in der Ebene

Bisher wurde hier noch nicht auf ein schwierigeres Problem eingegangen: Bewegen von L im 3-dimensionalen Raum mit komplizierten Hindernissen. Solch ein Problem führt zu einen 5-dimensionalen Konfigurationsraum. Tabelle 2.2 zeigt wiederum einige hierbei erreichte Komplexitäten.

Autoren	**Zeitkomplexität**
Schwartz & Sharir 1984	$O(n^{11})$
Ke & O'Rourke 1987	$O(n^6 \log n)$
Canny 1987	$O(n^5 \log n)$
Ke & O'Rourke 1988	$\Omega(n^4)$

Tabelle 2.2: Komplexitäten im Raum

Dies lässt sich allgemein in folgendem Theorem 2.1 zusammenfassen.

Theorem 2.1:
Jedes Bewegungsplanungsproblem für einen Roboter R mit d Freiheitsgraden kann in $O(n^d \log n)$ gelöst werden.

Dies gilt jedoch nur für den allgemeinen Fall, Spezialfälle können selbstverständlich je nach entsprechenden Randbedingungen schneller gelöst werden.

3. Roboterarm-Bewegung

Problemdefinition:
Planung der Bewegung eines einseitig befestigten, ebenen Roboterarmes mit mehreren Gelenken. Man betrachte eine Kette von jeweils längenkonstanten Segmenten. Die Segmente seien L_i (i=0, …, n) und J_i (i=0,…, n) die entsprechenden Gelenke. Der Ursprung J_0 stellt die „Schulter" des Armes dar. $\forall$ J_i(0<i<n): J_i ist Gelenk zwischen L_i und L_{i+1}.

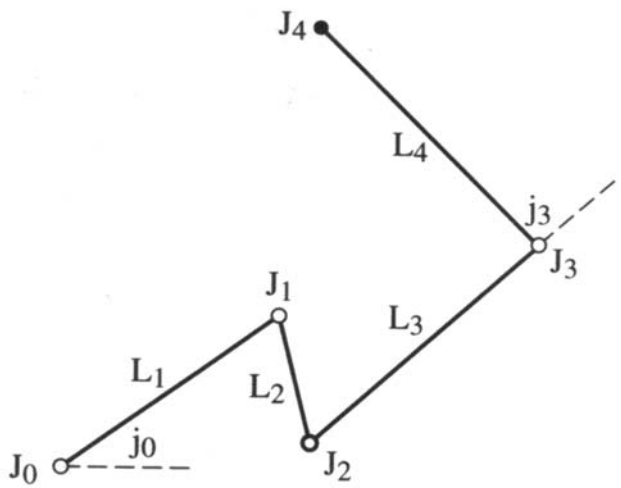


Abb. 3.1: Beispielarm

Weitere Konventionen:
- l_i sei die Länge von L_i;
- j_i sei der Winkel am Gelenk J_i, gegen den Uhrzeigersinn von L_i nach L_{i+1} gemessen, mit L_i (J_{i-1}, J_i) und L_{i+1} (J_i, J_{i+1}) als Vektoren;
- j_0 sei der Winkel zwischen der positiven x-Achse und L_1;
- j_n ist nicht definiert;
- der Arm A ist gegeben durch (s)eine Liste von Segmenten (l_0, …, l_n).

Für die weiteren Betrachtungen gelten folgende Voraussetzungen:
1. $\forall$ j_i(0<=i<n): j_i sei beliebig;
2. der Arm kann sich selbst überschneiden;
3. die Bewegungsebene des Arms sei frei von Hindernissen.

Aufgabe:
Gegeben sei der Arm A, definiert durch n Segmente l_i , und ein Zielpunkt p in der Ebene. Man bestimme, ob A p erreichen kann, und falls ja, welche Winkel j_i notwendig sind, damit gilt: J_n=p. Ist dies durch einfache Winkelberechnung möglich?

Anmerkung
Laut Hopcroft gilt die Lösung dieses Problems als „einfach", mit Hindernissen ist die Lösung „NP-hart", und beschränkt man den Bereich auf einen Kreis, so ist die Aufgabe polynomial lösbar.

3.1 Aufteilung in Zellen

Alle von A erreichbaren Punkte liegen auf einem Kreisring K (siehe Lemma 3.1).

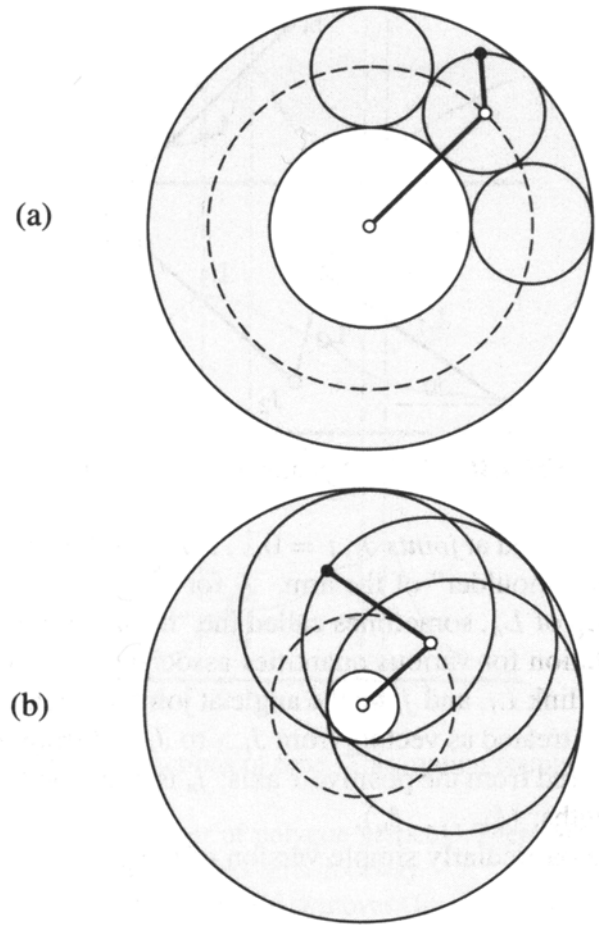


Abb. 3.2: 2-Segment-Arm. a) $l_1 > l_2$, b) $l_1 < l_2$

Erreichbare Regionen:
Ein 1-Segmentarm kann lediglich einen auf den Ursprung zentrierten Kreisrand (innerer Radius r_i = äußerer Radius r_o des Kreisrings K) erreichen.
Ein 2-Segmentarm A= (l_1, l_2)

- mit $l_1 \geq l_2$ erreicht einen Kreisring mit dem äußeren Radius $r_o = l_1 + l_2$ und dem inneren Radius $r_i = l_1 - l_2$ (siehe Abb. 3.2 a));
- falls $l_1=l_2$, so ist $r_i = 0$ und der Kreisring eine Scheibe vom Radius $r_o = l_1 + l_2$.
- Wenn $l_1<l_2$, ist wiederum $r_o = l_1 + l_2$, aber $r_i = l_2 - l_1$, bzw. $r_i = |l_1 - l_2|$.

So könnte man K eines 2-Segmentarms auch als Minkowski-Summe von 2 Kreisen C_1 mit Radius l_1 und C_2 mit Radius l_2 darstellen. Diese Summe von zentrierten Kreisen ist, wie bereits angedeutet, ein zentrierter Kreisring. Ebenso ist die Minkowski-Summe eines ursprungszentrierten

Kreisrings und eines zentrierten Kreises wiederum ein ursprungs-zentrierter Kreisring.

Lemma 3.1:
Die erreichbare Region eines n-Segmentarmes ist ein (an der „Schulter") ursprungszentrierter Kreisring.

Radien des Kreisrings
Der äußere Radius ergibt sich durch „Ausstrecken" aller Verbindungen ➔ $r_o = \sum_{i=1}^{n} l_i$. Ob $r_i > 0$ oder nicht, hängt von der Relation zwischen der Länge des längsten Segments und der Summe der Länge der anderen Segmente ab.

- $r_i > 0$, falls das längste Segment länger ist, als alle anderen Verbindungen kombiniert.

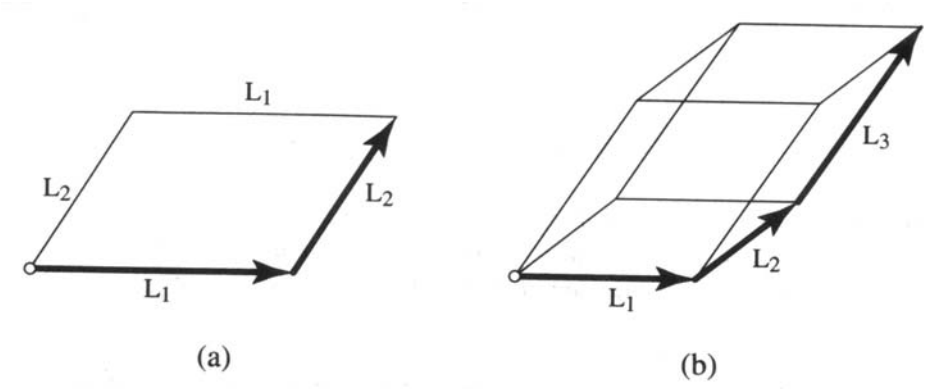


Abb. 3.3: Parallelogramme für a) 2- bzw. b) 3-Segmentarme, die zeigen, dass die Reihenfolge der Segmente keine Rolle für die Erreichbarkeit spielt

Lemma 3.2:
Die erreichbare Region eines Armes hängt nicht von der Reihenfolge, in welche die Verbindungen angeordnet sind, ab.

Beweis:
Dieses folgt dem Kommutativgesetz der Vektoraddition (siehe Abb. 3.3). Somit kann man sich o.B.d.A. auf Arme beschränken, deren Segment l_1 das längste ist. Für solche Arme gilt: $r_i = l_1 - \sum_{i=2}^{n} l_i$. Solange diese Summe positiv ist, ist auch $r_i > 0$. Andernfalls ist $r_i = 0$. Siehe Abb. 3.4.

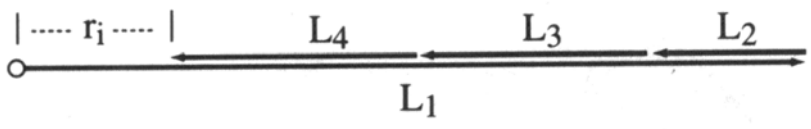


Abb. 3.4: $r_i = l_1 - (l_2 + l_3 + l_4)$

Theorem 3.1:
Die erreichbare Region für einen Arm mit n Verbindungen ist ein ursprungszentrierter Kreisring mit dem äußeren Radius $r_o = \sum_{i=1}^{n} l_i$ und dem inneren Radius $r_i = 0$, wenn einerseits das längste Segment l_M kleiner oder gleich der Hälfte der Gesamtlänge von A ist und andererseits $r_i = l_M - \sum_{i \neq M} l_i$ ist.

➔ Die Erreichbarkeit kann in O(n) entschieden werden:
1. Finde l_M;
2. berechne r_o und r_i;
3. p ist erreichbar, falls $r_i \leq |p| \leq r_o$ (mit |p| = Abstand Schulter – p).

3.2 Konstruktion der Erreichbarkeit

Es gibt Methoden, die versuchen, alle möglichen Lösungen zum Erreichen von p zu finden, was leicht auf exponentiell anwachsende Komplexität führen kann. Glücklicherweise können viel leistungsfähigere Algorithmen erzielt werden, indem man die einfache Anforderung ausnutzt, dass nur eine Lösung gewünscht wird.

Erreichbarkeit mit einem Segment
Einen „Schulterwinkel" j_0 für einen Arm mit einem Segment zu erfassen, um einen Punkt auf seinem Kreis zu erreichen, ist trivial.

Erreichbarkeit mit 2 Segmenten
Sei p der zu erreichende Punkt. Man schneide den Kreis C_1 des Radius l_1, welcher auf den Ursprung zentriert ist, mit dem Kreis C_2 des Radius l_2, zentriert auf p. Hier kann es null, eins, zwei oder eine endlose Menge von Lösungen geben, abhängig davon, wie viele Kreise sich schneiden (siehe Abb. 3.5).

Erreichbarkeit mit 3 Verbindungen
Ziel ist die Reduktion eines „Vielsegmentsproblems" auf ein „Zweisegmentproblem". Seien $A_3 = (l_1, l_2, l_3)$ und die erreichbare Region für $A_2 = (l_1, l_2)$ ein Kreisring K. Alle Punkte auf ∂K stellen extreme Konfigurationen von A_2 dar:

- $j_i = 0$, i = 1, …, n ➔ entspricht 1-Segment der Länge $l = \sum_{i=1}^{n} l_i$;
- $j_1 = \pi$, $j_i = 0$, i = 2, …, n ➔ entspricht 1-Segment der Länge $l = l_1 - \sum_{i=2}^{n} l_i$.

Man überprüfe, wie der Kreis C des Radius l_3, zentriert auf p = J_3, K schneidet. Man unterscheide zwei Fälle, abhängig davon, ob ∂K ∩ C = ∅ oder nicht.

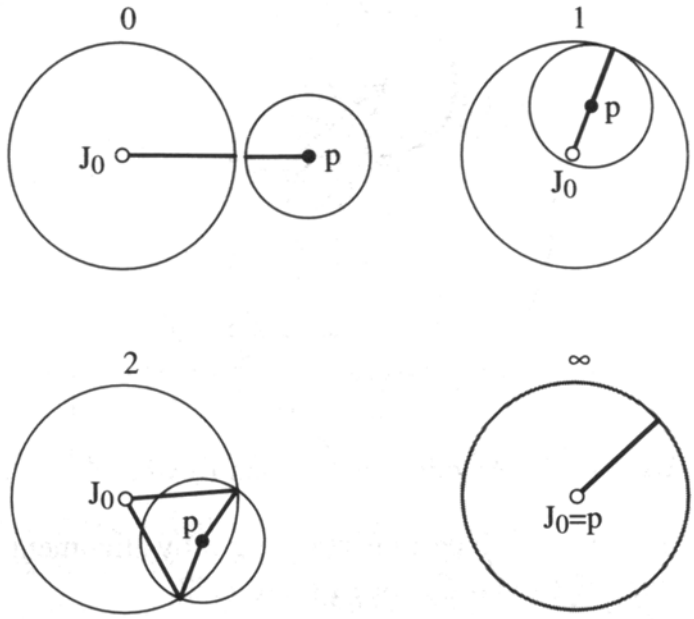


Abb. 3.5: 2-Segment-Erreichbarkeit

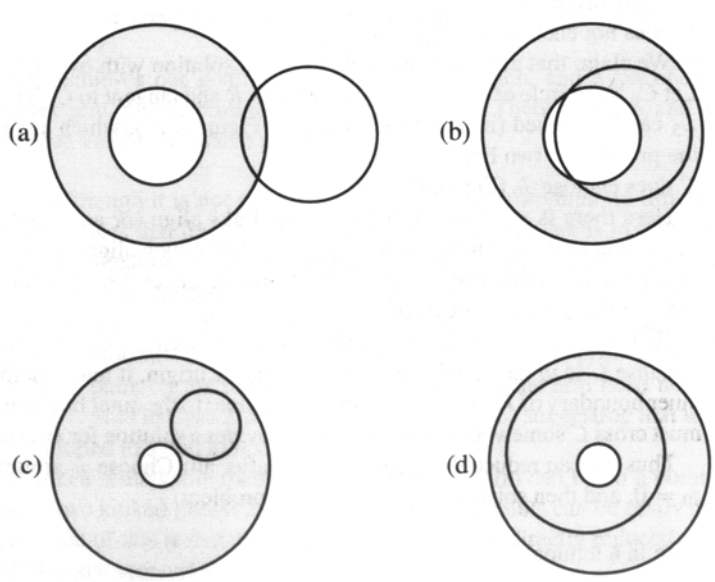


Abb. 3.6: 3-Segment-Erreichbarkeit

Fall 1: $\partial K \cap C \neq \varnothing$ (siehe Abb. 3.6 a), b))
In diesem Fall kann das Problem auf ein "Zweisegmentproblem" reduziert werden:

- durch Ausrichten (a)
- oder Entgegenrichten (b)

von L_1 und L_2.

Sei $\partial K = l \cup O$, wobei l der innere und O der äußere Rand des Kreisrings ist. Wenn $O \cap C = \varnothing$ und $C \neq \varnothing$, kann ein Kreis C_2 des Radius l_2 tangential zu C ausgewählt werden. Das ermöglicht es, p durch Ausrichtung von L_2 und L_3 anstatt des Entgegenrichtens von L_1 und L_2 zu erreichen. So erspart man sich Entgegenrichten von L_1 und L_2.

Fall 2: $\partial K \cap C = \varnothing$ (siehe Abb. 3.6 c), d))
Unterscheidung:

- C schließt den Ursprung J_0 ein.
  Sei C_2 ein Kreis des Radius l_2 im Kreisring K und tangential zu C. Dann können L_2 und L_3 ausgerichtet werden (siehe Abb. 3.7).
- C schließt den Ursprung J_0 nicht ein.
  Hier gibt es keine entsprechende Lösung mit zwei Segmenten. ➔Problem aber: Es gibt eine Lösung für jeden Wert von j_0. Sei j_0 willkürlich ausgesucht und ein Kreis C_2 zentriert auf J_1. Weil C im Kreisring K ist und den Ursprung umgibt, muss es l beinhalten, den inneren Rand von K. Da C_2 den inneren und äußeren Rand von K verbindet, muss es C irgendwo schneiden. Dieser Schnittpunkt bietet eine Lösung für ein willkürliches j_0. Somit kann man auch diesen Fall auf 2 Segmente reduzieren:
  - j_0 willkürlich wählen z.B. $j_0 = 0$
  - resultierendes Problem mit 2 Segmenten lösen.

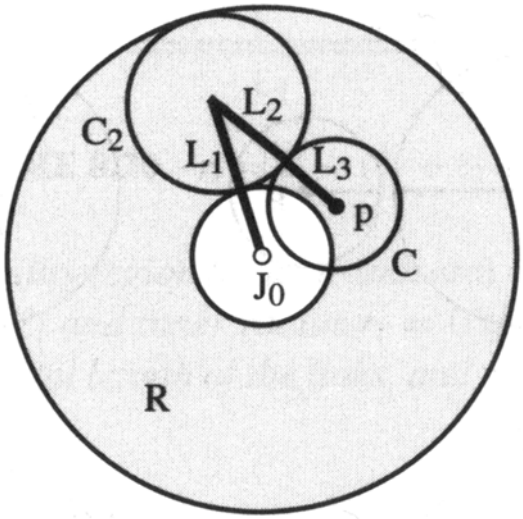


Abb. 3.7: 3-Segment-Erreichbarkeit (2)

Lemma 3.3:
Jedes Problem mit 3 Segmenten kann durch Rückführung auf eines der folgenden Probleme mit 2 Segmenten gelöst werden:

(1) $(l_1 + l_2, l_3)$;
(2) $(l_1, l_2 + l_3)$;
(3) $j_0 = 0$ und (l_2, l_3).

Beweis: vgl. (1) – (3) mit Abb. 3.6 + 3.7.

Erreichbarkeit mit n Segmenten
Angenommen, der Kreisring K stellt n–1 Segmente eines Armes dar, wobei der Kreis C des Radius l_n auf p zentriert ist. Angenommen, A kann den Zielpunkt erreichen, d.h. $K \cap C \neq \varnothing$ (vgl. Abb. 3.6).
➔ es ergibt sich folgendes rekursive Verfahren für die Bestimmung einer Konfiguration, die von einem gegebenen Punkt den Punkt p erreicht:

Fall 1: $\partial K \cap C \neq \varnothing$
Wählen eines Punktes (t) von den normalerweise zwei Punkten der Schnittmenge.

Fall 2: $K \supseteq C$ (Figur 8.22 (c,d))
Wählen irgendeines Punktes t auf C, z.B. der von J_0 am weitesten entfernte.
In jedem Fall findet man rekursiv eine Konfiguration für $A_{n-1} = (l_1, \ldots, l_{n-1})$, um t zu erreichen. Das letzte Segment L_n wird dieser Lösung angeschlossen, um t mit p zu verbinden (C ist auf p zentriert). Da die Fälle in Abb. 3.6 vollständig sind, findet man durch dieses Verfahren garantiert eine Lösung in O(n) Schritten, da das Verringern von n um 1 konstante Zeit benötigt, indem man C mit O und I schneidet, wobei $\partial K = I \cup O$ ist.
➔ Um einen gegebenen Punkt p mit einer Liste von Segmenten eines Armes zu erreichen, stellt man zuerst fest, ob p erreichbar ist (mittels Theorem 3.1), und wenn dies der Fall ist, findet man eine Konfiguration über das gegebene rekursive Verfahren.

Es ist nicht möglich, die Zeitkomplexität von O(n), die es dauert, die Segmente zu summieren, zu verbessern.

Eine Möglichkeit der Vereinfachung bietet die Lösung des Falls 1:
Die ersten n–1 Gelenke werden geradeaus gerichtet, wenn p ϵ O, und werden nur an den Enden des längsten Segments „geknickt", wenn p ϵ I. Dies ergibt sich aus der Formel für r_i. Somit braucht der Arm in Fall 1 nicht viele Schleifen. Im zweiten Fall könnte p überall auf C liegen. Diese Freiheit könnte genutzt werden, um keine Knicke zu verwenden. Außerdem kann leicht festgestellt werden, welche zwei Gelenke es sind, die geknickt werden müssen. D.h. jedes Problem mit n Segmenten kann auf eines mit 3 Segmenten reduziert werden.

Theorem 3.2:
Wenn ein Arm mit n Segmenten einen Punkt erreichen kann, so kann er dies auch mit höchstens zwei geknickten Gelenken:
nur zwei Gelenke aus $J_1, \ldots, J_{n-1}$ haben einen Winkel ≠ 0. Man wähle L_m wie folgt: $\sum_i^{m=-1} l_i \leq ½ \sum_i^{n=1} l_i$ und $\sum_i^{m=1} l_i > ½ \sum_i^{n=1} l_i$. Die beiden Gelenke sind die an L_m anschließenden.

Beweis:
Zu zeigen ist, dass der Arm A' aus Theorem 3.2 mit 2 geknickten und ansonsten eingefrorenen (Winkel = 0) Gelenken die gleiche erreichbare Region wie der Arm A hat. Da r_0 nur von der Summe der Segmentlängen abhängt (siehe Theorem 3.1), kann r_0 fixiert werden. Der Beweis zeigt, dass r_i unverändert ist. Sei L die Gesamtlänge der Segmente.

Fall 1: $r_i > 0$ (siehe Abb. 3.8 a))
Laut Theorem 3.1 ist r_i nur ≠ 0, wenn das längste Segment L_M die Länge der restlichen Segmente übertrifft.
Somit muss $l_M > l/2$ sein. ➔ $L_M = L_m$ unabhängig von der Reihenfolge der Segmente: Da L_M so lang ist, umfasst es den Mittelpunkt der Längen unter jeder möglichen Verschiebung der Reihenfolge. Nun ist $L_m = L_M$ und $L_M = \sum_{i \neq M} l_i$. Wenn man alle Gelenke außer den Endpunkten von L_M einfriert, um einen neuen Arm A´ zu erhalten, bleibt L_M das längste Segment. In Abb. 3.8 a) ist 6 das längste Segment in A und A'. Da r_i nur von l und l_M abhängt, hat A´ die gleiche erreichbare Region wie A.

Fall 2: $r_i = 0$ (siehe Abb. 3.8 b))
Das längste Segment ist L_M mit $l_M \leq l/2$ und $l_M \leq \sum_{i \neq M} l_i$. Sei L_M das mittlere Segment und seien alle Gelenke, die nicht an L_M grenzen, eingefroren, um einen neuen Arm A´ zu formen. Eventuell entsteht daraus ein neues längstes Segment (Segment 8 im Beispiel aus Abb. 3.8 b)). Für die Länge des neuen längsten Segments L'_M gilt: $l'_M \leq l/2$. Da L_M den Mittelpunkt der Längen spreizt, muss das vorangehende und das nachfolgende Segment gelten: $l'_{M-1}, l'_{M+1} \leq l/2$. Da r_i nur ungleich 0 ist, wenn das längste Segment l/2 übersteigt, ist sicher, dass $r_i = 0$.

➔ Die erreichbare Region von A´ ist die gleiche wie die von A.

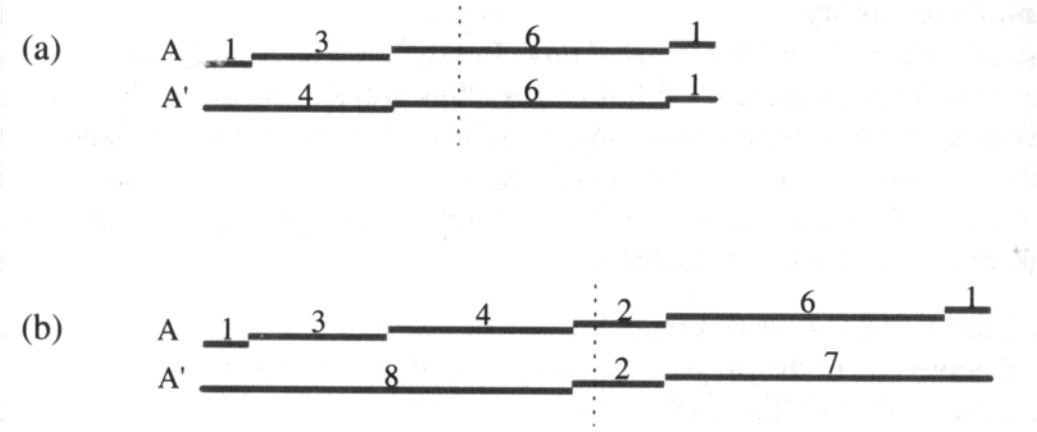


Abb. 3.8: Beispiel zu Theorem 3.2

Algorithmus

Der einzige Teil, der nach Theorem 3.2 von n abhängt, ist die Summierung der Längen von n Segmenten, O(n). Sonst ist der Algorithmus in O(1), wohingegen der rekursive Algorithmus zusätzlich O(n) für die Schnittmengenbildung benötigt. Nachdem L_m gefunden wurde, reduziert man also das n- auf ein 3-Segment-Problem und dieses wiederum auf 3 2-Segment-Probleme (nach Lemma 3.3).

3.3 Implementierung des Roboterarm-Algorithmus

Seien die Segmentlängen in einem `Array` abgespeichert. Aus Genauigkeitsgründen wird so lange als möglich an `Integer`-Werten festgehalten.
Nachdem in Code 3.1 mit `ReadLinks` die Segmentlängen ausgelesen wurden, durchläuft das Hauptprogramm eine Schleife, in der die Erreichbarkeit eines Ziels mittels `Solven` gelöst wird. Die Lösung wird hierbei schrittweise, wie zuvor gefordert, auf die Lösung einfacherer Probleme reduziert: `Solven` → `Solve3` → `Solve2` → `TwoCircles` → `TwoCircles0a` → `TwoCircles0b` → `TwoCircles00`.
Die Solve-Funktionen geben bei Erreichbarkeit jeweils `TRUE` zurück. Die vier TwoCircles-Funktionen errechnen eine Anzahl von Kreisschnittpunkten und den Schnittpunkt, Punkt p. Dieser Punkt wird als J mitgeführt und als Endpunkt zum Zeichnen des jeweiligen Segments genutzt.

```
/* Global variables. */
int       linklen [NLINKS];            /* link lengths */
int       nlinks;                      /* number of links */
tPointi   target;                      /* target point */
main()
{
     tPointi origin = { 0, 0 };
     nlinks = ReadLinks();
     /* loop broken by EOF in ReadTarget */
     while (TRUE){
          ReadTarget( target );
          MoveTo_i( origin );
          if( !Solven( nlinks ) )
               printf( "Solven: no solutions!\n" );
          LineTo_i( target );
     }
}
```

Code 3.1: main

Solven (Code 3.2) findet das mittlere Segment und ruft Solve3 mit den neuen Segmenten mit eingefrorenen Gelenken davor und dahinter auf (mit `L1, L2, ...` als Längen).

```
bool Solven( int nlinks )
{
     int  i;
     int  m;                  /* index of median link */
     /* length if links between kinks */
     int L1, L2, L3;
     int totlength;           /* total length of all links */
     int halflength;          /* floor of half of total */

     /* Compute total and half length. */
```

```
     totlength = 0;
     for( i = 0; i < nlinks; i++ )
          totlength  += linklen[i];
     halflength = totlength / 2;

     /* Find median link. */
     L1 = 0;
     for( m = 0; m < nlinks; m++ ){
          if( ( L1 + linklen[m] > halflength )
               break;
          L1 += linklen[m];
     }

     L2 = linklen[m];
     L3 = totlength - L1 - L2;
     if  ( Solve3( L1, L2, L3, target ) )
          return TRUE;
     else return FALSE;
}
```

Code 3.2: Solven

Solve3 (Code 3.3) folgt den gegebenen Ausführungen und ruft Solve2 bis zu dreimal auf. (Für SubVec siehe VL Algorithmische Geometrie)

```
bool Solve3( int L1, int L2, int L3, tPointi target )
{
     /* cords of kinked joint returned by Solve2 */
     tPointd Jk;
     tPointi J1;              /* Joint1 on x axis */
     tPointi Ttarget;         /* translated target */

     if( Solve2( L1 + L2, L3, target, Jk ) ){
          LineTo_d( Jk );
          return TRUE;
     }
     /* pin J0 to 0. */
     else{
          /* Shift so J1 is origin. */
          J1[X] = L1;
          J1[Y] = 0;
          SubVec( target, J1, Ttarget );
          if(  Solve2( L2, L3, Ttarget, Jk ) ){
               /* Shift solution back to origin. */
               Jk[X] += L1;
               LineTo_i( J1 );
               LineTo_d( Jk );
               return TRUE;
          }
          else
               return FALSE;
```

```
    }
}
```

Code 3.3: Solve3

Solve2 (Code 3.4) ordnet die Argumente für TwoCircles, welches dementsprechend zwei Kreise schneidet.

```
bool Solve2( int L1, int L2, tPointd J )
{
    tPointi c1 = { 0, 0 };        /* center of circle 1 */
    int nsoln;     /* # of solns : 0, 1, 2, 3 (infinite) */

    nsoln = TwoCircles( c1, L1, target, L2, J );
    return nsoln != 0 ;
}
```

Code 3.4: Solve2

Schnittpunkt von zwei Kreisen

Der Schnittpunkt von zwei Kreisen kann in O(1) gefunden werden. Seien die Mittelpunkte der zwei Kreise C_1 und C_2 $c_i = (a_i, b_i)$ mit den Radien r_i, i=1,2. Da die Gleichung eines Kreises eine quadratische Gleichung auf der Basis allgemeiner algebraischer Prinzipien ist, kann es nicht mehr als vier Schnittpunkte geben. Aufgrund der Gleichungsform kann es aber tatsächlich nicht mehr als zwei „echte" Schnittpunkte geben. 0, 1 und bel. viele Schnittpunkte sind, wie bereits angemerkt, ebenfalls möglich. Zunächst werden diese Fälle unterschieden, gefolgt von der Schnittpunktberechnung.

Sei o.B.d.A. $c_1 = (0, 0)$ und $c_2 = (a_2, 0)$. TwoCircles (Code 3.5) führt die Translation von c_1 durch und ruft TwoCircles0a auf.

```
/* TwoCircles finds an intersection point between two circles.
  General routine : no assumptions. Returns # of intersection;
  point in p. */
int TwoCircles( tPointi c1, int r1, tPointi c2, int r2,
                tPointd p )
{
    tPointi c;
    tPointd q;
    int nsoln = -1;

    /* Translate so that c1 = { 0,0 }. */
    SubVec( c2, c1, c );
    nsoln = TwoCircles0a( r1, c, r2, q );
    /* Translate back. */
    p[X] = q[X] + c1[X];
    p[Y] = q[Y] + c1[Y];
    return nsoln ;
}
```

Code 3.5: TwoCircles

TwoCircles0a (Code 3.6) behandelt die Spezialfälle. Die Berechnung von $(r_1+r_2)^2$ und $(r_1-r_2)^2$, sowie der Vergleich mit dem Abstandquadrat zu c_2 ermöglicht es, die gegebenen Fälle zu unterscheiden. Im Fall mit einem Schnittpunkt weiß man, dass dieser auf der Strecke von Ursprung zu c_2 im Abstand von r_1 liegt. Falls kein Spezialfall auftritt, wird TwoCircles0b (Code 3.7) aufgerufen.

```
/* TwoCircles0a assumes that the first circle is centered on
  the origin. Returns # of intersections: 0, 1, 2, 3 (inf) ;
  point in p. */
int TwoCircles0a( int r1, tPointi c2, int r2, tPointd p )
{
     double  dc2;             /* dist to center 2 squared*/
     double rplus2, rminus2;  /*(r1 +/- r2)²*/
     double f ;          /*fraction along c2 for nsoln = 1*/

     /*Handle special cases.*/
     dc2       = Length2( c2 );
     rplus2    = ( r1 + r2 ) * ( r1 + r2 );
     rminus2   = ( r1 – r2 ) * ( r1 – r2 );

     /*No solution if c2 out of reach + or -.*/
     if( ( dc2 > rplus2 ) ‖ ( dc2 < rminus2 ) )
          return 0;

     /*One solution if c2 just reached.*/
     /*Then solution is r1-of-the-way (f) to c2.*/
     if( dc2 == rplus2 ) {
          f = r1 / (double)(r1 + r2);
          p[X] = f * c2[X];
          p[Y] = f * c2[Y];
          return 1 ;
     }
     if( dc2 == rminus2 ) {
          /*Circles coincide.*/
          if( rminus2 == 0 ) {
               p[X] = r1;
               p[Y] = 0;
               return 3;
          }
          f = r1 / (double)(r1 – r2);
          p[X] = f * c2[X];
          p[Y] = f * c2[Y];
          return 1;
     }
     /*Two intersections.*/
     return TwoCircles0b( r1, c2, r2, p );
}
```

Code 3.6: TwoCircles0a

TwoCircles0b (Code 3.7) führt die Rotation von c_2 auf die x-Achse des Koordinatensystems aus. In diesem gedrehten Koordinatensystem löst TwoCircles00 das Problem. Anschließend wird c_2 wieder zurückgedreht.
Die Rotation erfolgt durch folgende Formel:

$$\begin{vmatrix} p_0 \\ p_1 \end{vmatrix} = \begin{vmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{vmatrix} * \begin{vmatrix} q_0 \\ q_1 \end{vmatrix}$$

```
/*TwoCircles0b also assumes that the 1st circle is origin-
  centered.*/
int TwoCircles0b( int r1, tPointi c2, int r2, tPointd p )
{
     double a2;/*center of 2nd circle when rotated to x axis*/
     tPointd q;/*one solution when c2 on x axis*/
     double cost, sint;  /*sine and cosine of angle of c2*/

     /*Rotate c2 to a2 on x axis.*/
     a2 = sqrt( Length2 ( c2 ) );
     cost = c2[X] / a2;
     sint = c2[Y] / a2;

     TwoCircles00 (r1, a2, r2, q );

     /*Rotate back*/
     p[X] = cost * q[X] + -sint * q[Y] ;
     p[Y] = sint * q[X] +  cost * q[Y] ;

     return 2 ;
}
```

Code 3.7: TwoCircles0b

Letztlich berechnet TwoCircles00 (Code 3.8) die eigentlichen 2 Schnittpunkte als Lösung folgender Gleichungen:

- I) $x^2 + y^2 = r_1^2$;
- II) $(x-a_2)^2 + y^2 = r_2^2$.

Man löse I) nach y^2 auf und setze es in II) ein:

- III) $y^2 = r_1^2 - x^2$;
- IV) $(x-a_2)^2 + r_1^2 - x^2 = r_2^2$.

Auflösung nach x:

- $x = ½ [a_2 + (r_1^2 - r_2^2) / a_2]$;
- Einsetzen von x in III).

Der Vorteil des Arbeitens in diesem Koordinatensystems ist, dass beide Lösungen die gleiche x-Koordinate und die gleiche y-Koordinate (einmal plus, einmal minus) haben.

```
/*TwoCircles00 assumes circle centers are (0,0) and (a2,0).*/
void TwoCircles00( int r1, double a2, int r2, tPointd p)
{
     double r1sq, r2sq;
     r1sq = r1 * r1;
     r2sq = r2 * r2;

     TwoCircles00( r1, a2, r2, q );

     /*Return only positive-y soln in p.*/
     p[X] = ( a2 + ( r1sq – r2sq ) / a2 ) / 2;
     p[Y] = sqrt( r1sq - p[X] * p[X] );
}
```

Code 3.8: TwoCircles00

Beispiel

Gegeben sei ein Arm mit 4 Segmenten, mit den Segmentlängen 100, 10, 40 und 90. Sei die Schulter (0, 0) das erste Ziel. `Solven` errechnet die Gesamtlänge 240 und kennzeichnet die dritte als die mittlere Verbindung. Anschließend wird Solve3(110, 40, 90) aufgerufen. Dort wird wiederum Solve2(150,90) und Solve2(110,130) – nicht erfolgreich – aufgerufen, da die Hand (0, 0) nicht erreichen kann.

Der erste Winkel sei auf $j_0 = 0$ festgelegt und Solve2(40,90) wird aufgerufen und versucht, (–110,0) zu erreichen (da die ersten zwei Verbindungen auf 0° eingefroren sind, ist die Länge $l_1 = 100 + 10$). Man muss nun den Schnittpunkt p = (25,45; 30,86) mittels TwoCircles00 finden, welcher nach der Umwandlung durch Solve2 als p = (–25,45, –30,86) und abschließend von Solve3 als p = (84,55, –30,86) zurückgegeben wird. Dieser Punkt wird als Koordinate von J_3 ausgegeben (siehe Abb. 3.9).

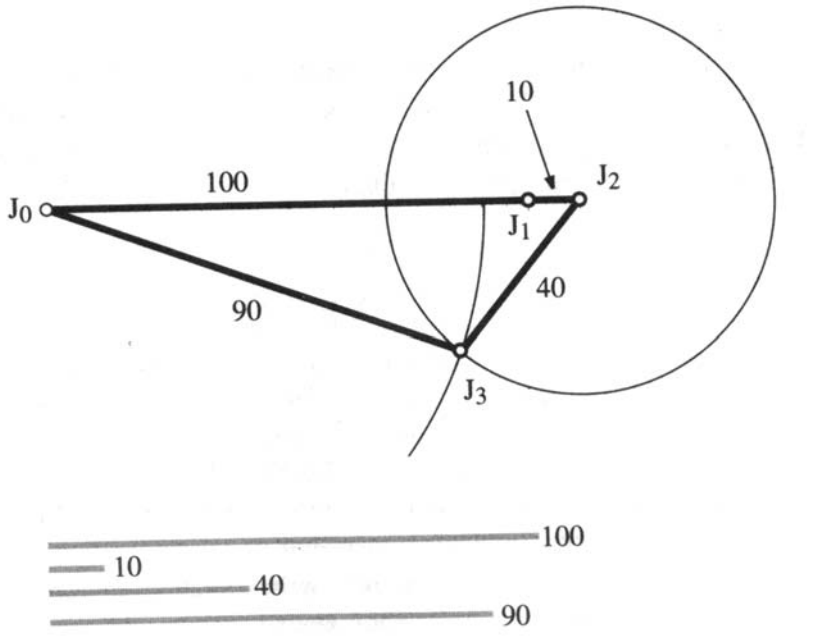


Abb. 3.9: Beispielkonfiguration

Dieser Ansatz kann mit einer Reihe von weiteren ähnlichen Beispielen veranschaulicht werden.

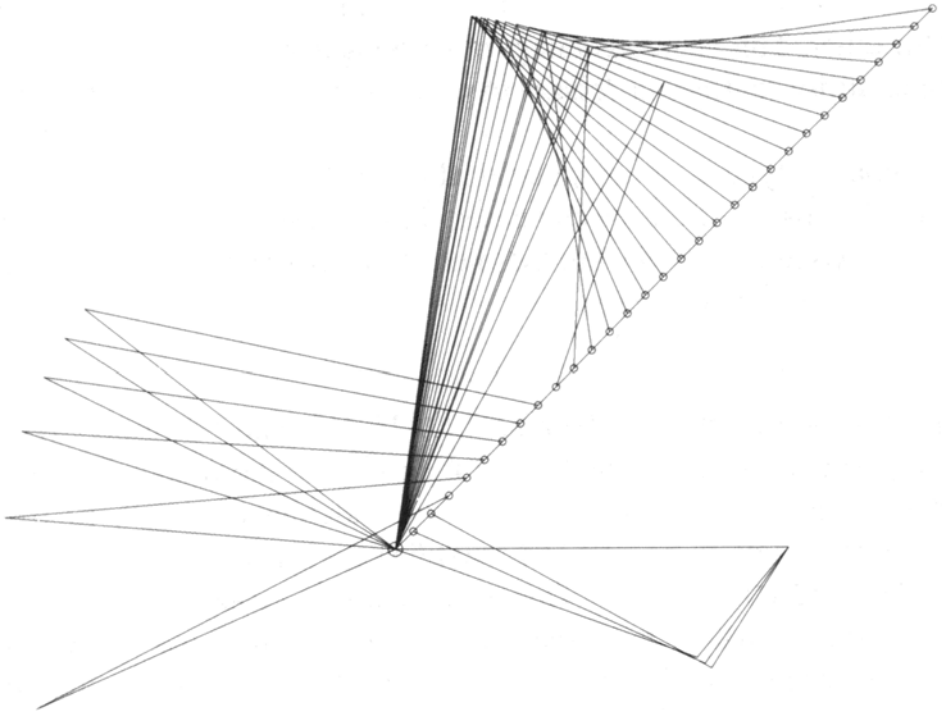


Abb. 3.10: Beispielkonfigurationen entlang einer Linie

An diesen Beispielen lässt sich ersehen, dass das Finden einer Reihenfolge von sich allmählich ändernden Konfigurationen, die einem bewegten Ziel folgen, ebenfalls ein interessantes Problem wäre.

4 Verzeichnis

4.1 Abbildungen

4.2 Tabellen

4.3 Code

4.4 Abkürzungen und Formelzeichen

∃	es gibt
∀	für alle
∞	Kante im Unendlichen
K	Kreisring
∅	Leere Menge
L	Leiter
⊕	Minkowski-Summe
o.B.d.A.	ohne Beschränkung der Allgemeinheit
τ	Pfad/Polygonzug
P	(Hindernis-)Polygon
∂	Rand von … (Polygon oder Kreis)
R	Roboter
θ	Rotationswinkel
∩	Schnittmenge
s	Startpunkt
Σ	Summe
⊇	(evtl. unechte) Teilmenge
U	Vereinigungsmenge
θ	Winkel
t	Zielpunkt

5 Literaturliste

[01] Joseph O'Rourke: ”Computational Geometry in C”, second edition, Cambridge University Press, Cambridge 1998.
[02] Winfried Kurth: ”Algorithmische Geometrie”, Winfried Kurth, Vorlesungsskript, BTU Cottbus 2003. (http://www-gs.informatik.tu-cottbus.de/~wwwgs/alg_vorles.htm)
[03] Johannes Steinmüller: ”Robotik". Vorlesung an der TU Chemnitz-Zwickau 1997. (www-sst.informatik.tu-cottbus.de/~db/doc/Reactive_Systems/Rug_Warrior/docs_soft/robo.ps)
[04] Achim Schweikard (Tutor): ”Geometrisches Schliessen”. Universität Lübeck 2003. (www.rob.uni-luebeck.de/downloads/KI/KI_7.doc)
[05] John E. Hopcroft / Jeffrey Ullman: “Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie“, Pearson Studium bzw. Oldenbourg, 2002.

Weitere Angaben: siehe Literaturverzeichnis aus [01].

Alle Abbildungen aus [01].