Backtracking-Algorithmen

Felix Opatz

Juli 2005

1 Einleitung

Die Klasse der Backtracking-Algorithmen zeichnet sich unter anderem durch den großzügigen Einsatz von Rekursion aus. Die Probleme sind hierbei in einzelne Schritte zerlegbar, die alle nach der gleichen Vorschrift behandelt werden können. Das ganze hat ein bißchen Ähnlichkeit mit der Tiefensuche in Bäumen, der Weg wird solange verfolgt, bis er nichts mehr erfolgsversprechend ist; dann wird eine Alternative versucht.

Das Grundlegende Prinzip des Backtrackings lässt sich folgendermaßen verdeutlichen:

1. Bin ich am Ziel?
   - Ja: fertig!
   - Nein: weiter mit 2.
2. War ich hier schonmal? Ist das hier eine Sackgasse?
   - Ja: zurück!
   - Nein: weiter mit 3.
3. Markiere diesen Weg
4. Gehe den nächsten Weg
5. War dieser Weg erfolgreich?
   - Ja: gib "Erfolg" zurück
   - Nein: versuche weitere Wege von hier aus
6. Nimm die Markierung wieder weg
7. Es gibt keine Lösung von hier aus

In Schritt 4 taucht hierbei der Rekursion auf, um den nächsten Weg zu gehen wird die Funktion, in der dieser Algorithmus steckt, erneut aufgerufen, mit der Position nach dem jeweils gemachten Schritt. Wenn nun alle Schritte, die man von einem Punkt aus gehen kann, nicht erfolgreich – also Sackgassen – sind, so muß man einen Schritt zurückgehen, und es von dort aus mit den verbleibenden Alternativen versuchen.

Dieser Algorithmus findet sein Ende, wenn ein erfolgreicher Weg gefunden wurde. Das ist dann jedoch der *erste*, nicht ntowendigerweise der *beste* Weg. Um alle Wege zu finden, muß man an dieser Stelle weitermachen, und den als erfolgreich bewerteten Weg irgendwo merken. Um den besten Weg zu finden, merkt man sich den aktuellen erfolgreichen Weg nur dann, wenn er besser als der bisherige beste Weg ist.

2 Labyrinthe

Das klassische Beispiel für Backtracking ist das Labyrinth. Zu Beginn befindet man sich an einem Startpunkt, meistens irgendwo mittendrin. Man kann von dort aus mehrere Wege nehmen, die üblicherweise nicht komplett einsehbar sind, allenfalls aus dem Weltblick. Aufgabenstellungen könnten nun sein, entweder den Ausgang irgendwie zu finden, den kürzesten Weg dorthin zu finden, oder zu beweisen, daß es keinen Weg gibt. Je nach Zielbedingung muß die Abbruchbedingung entsprechend definiert werden, und ggf. muß eine Art Gedächtnis eingebaut werden.

Der einfachste Fall, die Suche eines beliebigen erfolgreichen Weges, zeichnet sich dadurch aus, daß man am Ausgang ankommt. Sucht man den kürzesten Weg, so kann man den aktuellen Versuch bereits dann abbrechen, wenn man mehr Schritte verbraucht hat, als dies beim bisher besten gefundenen Weg der Fall ist (dann kann er nämlich nicht mehr besser werden).

Hat man alle Alternativen ausgeschöpft, und ist dennoch zu keinem Ausgang gelangt, ist die Sache klar: es gibt keinen Weg. Auch das kann eine Auskunft sein, die man sucht.

2.1 Irgendein Weg

Um die Sache zu veranschaulichen, soll hier der einfachste Fall als Beispiel gezeigt werden. Damit der Weg sichtbar ist, werden einfach „Brotkrumen“ ausgestreut, man markiert also jeden Punkt, den man als Ausgangspunkt für einen Pfad nimmt, in geeigneter Weise, um ihn wiederzuerkennen. Ist der Weg als unbrauchbar eingestuft worden, so nimmt man die Markierungen wieder weg. Man kann dies leider schlecht als „Bilderfolge“ zeigen, da ein einfaches und für den Betrachter intuitiv lösbares Labyrinth bereits sehr viele Schritte braucht.

Das Programm `labyrinth.c` beinhaltet diese Funktion:

```
int search_lab(int x_pos, int y_pos)
{
  /* Wand gefunden? Sackgasse! */
  if (lab[y_pos][x_pos] == '#')
    return 0;

  /* Ende Gefunden? Erfolg! */
  if (lab[y_pos][x_pos] == 'F')
    return 1;

  /* Schleife? Zruück! */
  if (lab[y_pos][x_pos] == '.')
    return 0;

  /* Brotkrume hinlegen */
  lab[y_pos][x_pos] = '.';

  /* nach oben gehen */
  if (search_lab(x_pos, y_pos - 1))
    return 1;

  /* nach rechts gehen */
  if (search_lab(x_pos + 1, y_pos))
    return 1;
```

```
  /* nach unten gehen */
  if (search_lab(x_pos, y_pos + 1))
    return 1;

  /* nach links gehen */
  if (search_lab(x_pos - 1, y_pos))
    return 1;

  /* alles erfolglos, Brotkrume mitnehmen */
  lab[y_pos][x_pos] = ' ';

  return 0;
}
```

Zuerst wird überprüft, ob man sich bereits in einer Wand (markiert durch ein '#') befindet. In diesem Fall handelt es sich natürlich um eine Sackgasse, also einen Schritt zurück. Das Ziel wird durch ein 'F' markiert; wenn es gefunden wird, so kann die Suche mit Erfolg abgebrochen werden. Die Brotkrumen werden mit einem '.' symbolisiert – stösst man auf eine solche, so läuft man irgendwie im Kreis, also auch abbrechen. Danach werden die Möglichkeiten von diesem Standpunkt aus abgeklappert. Zuerst nach Norden, dann nach Osten, Süden und Westen. Diese Reihenfolge ist willkürlich, je nach Labyrinth hat sie aber entscheidende Konsequenzen, wie in Kürze gezeigt werden wird. Wenn nun alle Wege erfolglos waren, so wird die Brotkrume wieder mitgenommen, um später anhand der Spur den Weg erkennen zu können.

Aufgerufen wird `search_lab` mit der Startposition, die durch ein 'S' im Labyrinth markiert wird. Als Labyrinth dient zunächst `2.lab`.

```
########################################    ########################################
#              #                     #    #....        ...#                .....#
#          #   #                     #    #....       #. .#                . ...#
F          #   #                     #    F....       #. .#                . ...#
#          #   #################     #    # ...       #. .#################. ...#
#          #                         #    # ...       #. .................. ...#
#    ########                        #    # ...########. ......................#
#          #                         #    # ..........#. ......................#
#        S #                         #    # ..........#. ......................#
#                                    #    # ........... .......................#
#                                    #    #  ......  ..........................#
########################################    ########################################
```

Links ist das Labyrinth zu Beginn gezeigt, rechts nach Ablauf des Programms. Man wird sicher gerne glauben, daß dies nicht der optimale Weg ist. Durch die Maßgabe, daß wenn es nach Norden nicht weiter geht, der Osten versucht werden soll, driftet die Spur in das tote Ende ab. An der Schleife in der Mitte sieht man schön, daß zuerst solange wie möglich nach Norden gegangen wird, dann zwei Schritte nach Osten, dann solange nach Süden, bis hinter dem Winkel wieder der Weg nach Osten frei ist, um die Ecke fast vollständig aufzufüllen.

Das Programm kommt jedoch zu einem Ende, auf meinem Rechner mit Athlon XP 2400+ läuft das Programm weniger als zwei Sekunden. Allerdings ist es ebenso bedenklich, daß es deutlich verzögert erst fertig ist.

Nun soll das Labyrinth aus `3.lab` getestet werden. Das Programm kehrt nahezu sofort zurück. Wie kommt's?

```
#######################################    #######################################
#             #                       #    #           ...#                 .....#
#          #  #                       #    #          #. .#                . ...#
#          #  #                       #    #          #. .#                . ...#
#          #  #################       #    #          #. .#################. ...#
#          #                          #    #          #. ................... ...#
#    ########                         #    #    ########.     ...................#
#          #                          #    #         ..#.     ...................#
#        S #                          #    #         ..#.     ...................#
#                                     #    #          ...     ...................#
#                                     #    #                  ...................#
###################F###################    ###################F###################
```

Das einzige, das anders ist, ist die Position des Endpunktes. Was man nicht sieht, sofern man nicht nach jedem Schritt das Labyrinth auf dieselbe Position ausgeben lässt, sodaß ein „Film" entsteht, ist das Vorgehen. Der Rechner kommt relativ zügig nach rechts in den Raum, also das ungefähre Zielgebiet. Danach wird der freie Bereich nahezu ideal mit Punkten ausgefüllt, nämlich immer rauf und runter, von rechts nach links, bis das 'F' getroffen wird.

Wenn man in der Funktion `search_lab` zu Beginn die beiden Zeilen

```
system("cls");
print_lab();
```

bzw. unter UNIX `system("clear")` einfügt, so kann man recht gut beobachten, wie der Algorithmus vorgeht.

Lässt man das Programm nun über `4.lab` laufen, so ist die Ausgabe dem perfekten Weg schon sehr nahe:

```
#######################################    #######################################
#             #                       #    #           ...#                 .....#
#          #  #                       #    #          #. .#                .   .#
#          #  #                       F    #          #. .#                .   .F
#          #  #################      #    #          #. .#################.    #
#          #                          #    #          #. ...................    #
#    ########                         #    #    ########.                       #
#          #                          #    #         ..#.                       #
#        S #                          #    #         ..#.                       #
#                                     #    #          ...                       #
#                                     #    #                                    #
#######################################    #######################################
```

Das kommt daher, daß das Ziel quasi direkt auf dem Weg liegt, der gegangen wird.

Soweit so gut, doch was ist mit `1.lab`? Ausprobieren! Bei meinem Rechner, mit immerhin 2 GHz Rechenleistung, was vor einigen Jahren noch den Großrechnern vorbehalten war, habe ich das Programm nach 12 Minuten abgebrochen. Was ist passiert?

```
##################F####################    ##################F####################
#              #                     #    #            ...#                .....#
#          #   #                     #    #          #. .#                .   .#
#          #   #                     #    #          #. .#                .   .#
#          #   ##################    #    #          #. .##################.  ..#
#          #                         #    #          #. ...................  ..#
#    ########                        #    #    ########.                      ..#
#          #                         #    #         ..#.                      ..#
#         S #                        #    #         ..#.                      ..#
#                                    #    #          ...                      ..#
#                                    #    #                                   ..#
########################################    ########################################
```

Das rechte Bild ist jetzt ein Zwischenergebnis. Die Punktelinie kam eben von oben herunter, und ist im Begriff wieder raufzulaufen, um in den nächsten Minuten den gesamten linken Teil aufzufüllen, auf alle erdenklichen Arten. Dadurch, daß der direkte Weg nun durch eine Punktelinie abgeschnitten ist, die nicht durchbrochen werden kann (weil man dort ja bereits war), ist das Laufzeitverhalten wirklich übel.

In algorithmischen Maßstäben gesprochen ist dieses Programm also nutzlos, denn es braucht „ewig“. Füg kleine Werte von ewig findet es zwar noch ein Ende, aber wenn das Labyrinth nun einige hundert Zeichen breit und hoch wäre – nicht haltbar.

2.2 Der beste Weg

Als nächstes soll nun der *beste* Weg in einem Labyrinth gefunden werden. Das Programm wird ein wenig abgeändert (`labyrinth2.c`):

- Das Labyrinth besteht aus einem zusammengesetzten Datentypen
- Es werden Schritte gezählt

Der zusammengesetzte Datentyp hat zwei Felder, einmal den Typ des Feldes, das ist wieder '.', 'F', 'S' etc., und einmal einen ganzzahligen Wert, der die minimale Anzahl Schritte, die zum Erreichen dieses Punktes jemals benötigt wurden, beinhaltet.

Mittels dieser Information kann ein Weg bereits dann als erfolglos verworfen werden, wenn man zum Erreichen eines Punktes mehr Schritte als irgendwann zuvor gebraucht hat. Demnach hatte man ja vorher bereits einen besseren. Wird das Ziel gefunden, so wird das komplette Labyrinth (mit Brotkrumen) kopiert, und zwar genau dann, wenn insgesamt weniger Schritte als für den bisher besten Weg gebraucht wurden.

Die Funktion `search_lab` sieht nun folgendermaßen aus (enthalten in `labyrinth2.c`):

```
void search_lab(int x_pos, int y_pos, int steps)
{
  /* Wand gefunden? Sackgasse! */
  if (lab[y_pos][x_pos].cell_type == '#')
    return;

  /* Ende Gefunden? Erfolg! */
  if (lab[y_pos][x_pos].cell_type == 'F')
  {
    /* merken, wenn besser als bisher */
    if (steps < best_way)
    {
      lab[y_pos][x_pos].cell_type = '.';
      memcpy(best_lab, lab, sizeof(lab));
      best_way = steps;
    }

    /* da habe ich ewig dran gesucht... */
    lab[y_pos][x_pos].cell_type = 'F';

    return;
  }

  /* kann schon nicht mehr besser werden */
  if (steps >= lab[y_pos][x_pos].min_steps)
    return;

  /* eine neue Bestleistung wird es auch nicht */
  if (steps >= best_way)
    return;

  /* Brotkrume legen, Schritte merken */
  lab[y_pos][x_pos].min_steps = steps;
  lab[y_pos][x_pos].cell_type = '.';

  /* nach oben gehen */
  search_lab(x_pos, y_pos - 1, steps + 1);

  /* nach rechts gehen */
  search_lab(x_pos + 1, y_pos, steps + 1);

  /* nach unten gehen */
  search_lab(x_pos, y_pos + 1, steps + 1);

  /* nach links gehen */
  search_lab(x_pos - 1, y_pos, steps + 1);

  /* alles erfolglos, Brotkrume mitnehmen */
  lab[y_pos][x_pos].cell_type = ' ';
}
```

Neu ist also, daß im Erfolgsfall (Ende gefunden) das Programm nicht verlassen wird, sondern weiter gesucht wird. Vorbei ist es erst, wenn kein besserer Weg mehr gefunden wird. Die Variable `best_way` und das Feld `min_steps` das Labyrinths werden auf INT_MAX initialisiert, sodaß also im ersten Anlauf jeder Wert besser ist. Ganz wichtig ist, daß die Brotkrume, die in das Ziel gelegt wird, vor dem Weitersuchen wieder durch den Ziel-Marker ersetzt wird, denn sonst wird kein besserer als der erste Weg gefunden. Ich habe da eine ganze Weile dran gesucht, bis ich den Fehler hatte.

Die Ausgabe dieses Programms auf alle vier Labyrinthe ist:

```
##################.####################
#                # ................    #
#           #    #                .    #
#           #    #                .    #
#           #    ##################.    #
#           #.....................    #
#    ########.                       #
#           #.                       #
#          ..#.                      #
#           ...                      #
#                                    #
########################################
Der Weg hat 48 Schritte
```

```
########################################
#                #                     #
#           #    #                     #
#           #    #                     #
#           #    ##################    #
#           #                          #
#    ########                          #
#           #                          #
#          ..#                         #
#           .........                  #
#                   .                  #
###################.####################
Der Weg hat 12 Schritte
```

```
########################################
#                #                     #
#           #    #                     #
.....       #    #                     #
#   .       #    ##################    #
#   .       #                          #
#   .########                          #
#   ....... #                          #
#         . #                          #
#                                      #
#                                      #
########################################
Der Weg hat 15 Schritte
```

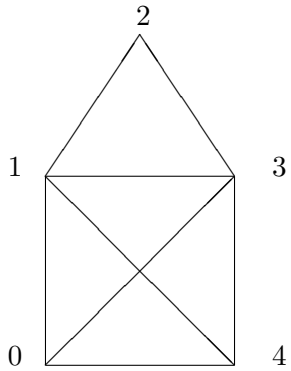
```
########################################
#                #                     #
#           #    #                     #
#           #    #                     #
#           #    ##################.    #
#           #.....................    #
#    ########.                       #
#           #.                       #
#          ..#.                      #
#           ...                      #
#                                    #
########################################
Der Weg hat 35 Schritte
```

Jeder wird einsehen wollen, daß dies tatsächlich die besten Wege sind. Das Programm läuft auch ohne merkliche Verzögerung durch. Es ist also ein befriedigender, guter Algorithmus.

Doch wenden wir uns nach all den Labyrinthen dem nächsten Thema zu.

3 Das Haus vom Nikolaus

Beim Haus vom Nikolaus ist die Aufgabenstellung den folgenden, ungerichteten Graphen zu zeichnen, ohne dabei den Stift abzusetzen:



Nun einen kleinen Schlenker zur Graphentheorie: Herr Euler sagt, daß ein Rundweg durch einen Graphen, bei dem jede Kante genau einmal besucht wird (ein sog. Eulerzyklus), dann möglich ist, wenn von jeder Ecke eine gerade Anzahl Kanten abgeht. Weiterhin sagt er, daß wenn genau zwei Ecken eine ungerade Anzahl Kanten besitzen, ein Weg möglich ist, der jede Kante genau einmal besucht. Dieser startet dann an der einen Ecke mit ungerader Kantenzahl, und endet an der anderen. Dies lässt sich leicht verstehen, wenn man sich eine weitere Kante vorstellt, die diese beiden Ecken verbindet – dann entsteht nämlich wieder ein Eulerzyklus, ein Rundweg.

Das Haus vom Nikolaus ist nun ein solcher Graph, dessen einzige Ecken mit ungerader Kantenzahl die in der Zeichnung mit 0 und 4 bezeichneten sind. Aller Erfahrung nach ist es auch möglich, das Haus zu zeichnen, ohne mit dem Stift abzusetzen, wenn man denn in einer dieser Ecken beginnt (und dann keinen Mist baut).

Die interessante Frage ist nun, wie man einen solchen Graphen in einem Programm darstellen kann. Die Lösung ist die *Adjazenzmatrix*. Das ist eine quadratische Matrix, die auf beiden Achsen jede Ecke einmal verzeichnet hat. Wenn von Ecke a zu Ecke b eine Kante führt, so wird der Eintrag (a;b) auf 1 gesetzt, andernfalls auf 0. Für unseren Graphen sieht diese Matrix wie folgt aus:

	0	1	2	3	4
0	-	1	0	1	1
1	1	-	1	1	1
2	0	1	-	1	0
3	1	1	1	-	1
4	1	1	0	1	-

Man sieht, daß die Matrix spiegelsymmetrisch ist. Das ist immer dann der Fall, wenn der Graph *ungerichtet* ist, d.h. wenn man von Ecke A zu Ecke B kommt, so kann man auch von Ecke B zu Ecke A kommen. Das ist in einem *gerichteten* Graphen anders.

Die Diagonale ist nicht besetzt (zur Verdeutlichung als - markiert, wäre natürlich 0), daran erkennt man, daß der Graph keine *Schleifen* besitzt, das sind Verweise auf sich selbst.

Der Vorteil dieser Darstellung ist ganz klar: zweidimensionale Arrays gibt es in fast jeder Programmiersprache, und damit hat man eine sehr geeignete Darstellungsmöglichkeit für Graphen. Die viel interessantere Frage ist nun jedoch, wie man in so einer Matrix einen Weg – oder in unserem Fall: alle Wege – finden kann.

Das Vorgehen ist simpel: befindet man sich auf Knoten n, und möchte zu einem anderen Knoten, so nimmt man einfach die n-te Zeile, und probiert dort alle Spalten aus, die eine 1 enthalten. Man ruft dann die rekursive Funktion einfach mit diesem neuen n auf.

Um das Haus vom Nikolaus darzustellen, muß jede Kante gezeichnet sein. Das heißt wir sind dann fertig, wenn jede gegangene Kante im Array auf 0 gesetzt wurde, und die Matrix nur noch Nullen enthält. Dies zu prüfen ist jedoch nicht sehr schön, es hätte quadratische Laufzeit. Unser Vorteil ist,

daß wir die Zahl der zu zeichnenden Kanten kennen, und dies entspricht genau der Rekursionsstufe in der wir uns am Ende befinden müssen.

```
int find_way(int matrix[NODES][NODES], int node, int way[EDGES + 1], int depth)
{
  int i, ret = 0;

  assert(depth <= EDGES);

  way[depth] = node;

  if (EDGES == depth)
  {
    print_way(way, depth);
    return 1;
  }

  for (i = 0; i < NODES; i++)
  {
    if (matrix[node][i])
    {
      matrix[node][i] = 0;
      matrix[i][node] = 0;
      if (find_way(matrix, i, way, depth + 1))
        ret = 1;
      matrix[node][i] = 1;
      matrix[i][node] = 1;
    }
  }

  return ret;
}
```

Zuerst wird überprüft, daß die Tiefe wirklich nicht größer als die maximale Anzahl Knoten ist – andernfalls ist was faul, z.B. die Adjazenzmatrix fehlerhaft. In dem Array `way` wird der Weg aufgezeichnet. Wenn die Tiefe der Anzahl Kanten (EDGES) gleich ist, so ist das Haus fertig gezeichnet und die Funktion `print_way` gibt die gewählte Route aus.

Der rekursive Teil befindet sich in einer Schleife, die über alle Ecken läuft (NODES ist die Anzahl dieser). Die Ecke, auf der sich das Programm gerade befindet, steht in `node`. Deshalb ist die Bedingungen `matrix[node][i]` genau dann wahr, wenn eine Verbindung von Ecke `node` zu Ecke `i` besteht (also an dieser Stelle der Adjazenzmatrix eine 1 steht). Ist dies der Fall, so wird diese Kante gelöscht – und natürlich ihr Spiegelbild! Die Funktion rekursiert an dieser Stelle, und wird mit `i` als neues `node` aufgerufen; die Tiefe `depth` wird um eins erhöht. Nach Rückkehr der Funktion wird die Kante (und ihr Spiegelbild) wiederhergestellt.

Wenn auf diese Weise das Ziel erreicht wurde (der Graph also vervollständigt wurde), so springt die Funktion mit dem Rückgabewert 1 raus, andernfalls 0. Dies ist notwendig, damit der Aufrufer der Funktion weiß, ob es überhaupt einen Weg gibt. Wenn diese Information nicht interessiert, so kann man die Funktion auch als void deklarieren, einzig und alleine das return nach dem Ausgeben des Graphen ist sinnvoll (aber nicht unbedingt notwendig, denn in der Schleife wäre die Bedingung eh immer unwahr, weil alle Kanten beseitigt wurden). Zu beachten ist, daß der Rückgabewert `ret` nur auf

1 gesetzt, und später, wenn die ganze Matrix durchkämmt ist, zurückgegeben wird. Wäre das `return 1` bereits in der if-Abfrage der rekursiven Funktion zu finden, so würde nur genau ein Weg gefunden werden.

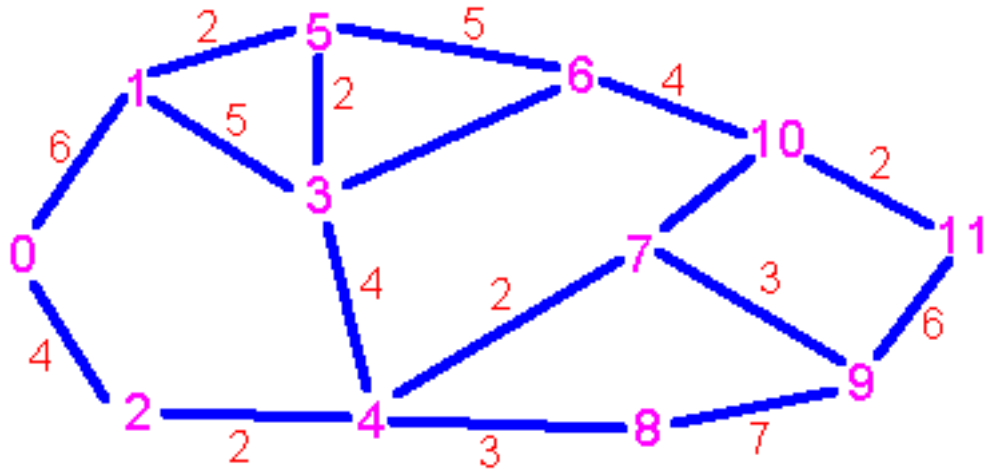
Dadurch, daß der Schleifenzähler `i` nach Rückkehr der rekursiven Aufrufe an seiner Stelle bleibt, werden gleiche Wege nicht erneut versucht. Auf der obersten Stufe wird die Funktion mit `node` = 0 aufgerufen, das ist die Ecke unten links. Man kommt damit auf 44 mögliche Wege – von Ecke 4 unten rechts aus gibt es genau so viele, nämlich die Wege zurück.

4 Der kürzeste Weg zum Flughafen

Die Graphentheorie hält noch weitere interessante Algorithmen und Probleme bereit. Eins ist beispielsweise das Finden des besten Weges in einem gewichteten Graphen. Der Algorithmus ist als *Dijkstras Algorithmus* bekannt, und wurde 1959 von Edsger Wybe Dijkstra erstmals vorgestellt.

Ein gewichteter Graph hat an jeder Kante einen Wert stehen (das *Gewicht*, oder auch die *Kosten*). Für solche Graphen kann man in der echten Welt jede Menge Anwendungen finden, eine steht schon in der Überschrift: das Ermitteln einer Route zum Flughafen. Wir nehmen an, wir hätten eine Straßenkarte. Es gibt nun jede Menge Möglichkeiten zum Flughafen zu fahren. Damit es etwas spannender wird, suchen wir nicht den kürzesten bezogen auf die Entfernung (der ist meistens leicht zu erkennen), sondern den kürzesten bezogen auf die Fahrzeit. Hier kommen Werte zustande, die auch die kleineren Nebenstraßen bevorzugen können. Nehmen wir z.B. an, daß eine kurze Straße eine 30-Zone ist, und man auf einer etwas längeren dafür 50 fahren darf. Oder ist es klug sich durch einen Ort zu quälen, oder eine Umgehungsstraße zu verwenden, auf der jedoch eine Baustelle ist, an der eine Ampel für Verzögerungen sorgt?

Ein solcher Graph könnte wie dieser aussehen:



Ziel soll es sein, von Ecke 0 zu Ecke 11 zu kommen, und dabei möglichst wenig zu „zahlen“. Der Algorithmus von Dijkstra besagt nun: Fang am Anfang an. Geh eine neue Kante entlang, und merk dir die Kosten dieser. Von dieser Kante aus geh zur nächsten, und addiere die Kosten auf die vorhandenen drauf. Das treibe solange, bis du entweder am Ziel bist, oder zu einem Knoten kommst, zu dem du schonmal billiger gekommen bist. In diesem Fall gehe einen Schritt zurück (backtracking!) und versuche einen anderen Weg. Wenn du am Ziel angekommen bist, dann treibe das Spiel dennoch solange weiter, bis es keine billigeren Lösungen mehr gibt.

Ein Beispiel: wir gehen von der 0 zur 1 (Kosten $0 + 6 = 6$), weiter zur 3 ($6 + 5 = 11$), zur 4 ($11 + 4 = 15$), zur 8 ($15 + 3 = 18$), zur 9 ($18 + 7 = 25$) und zur 11 ($25 + 6 = 31$). Wir haben das Ziel also mit den Kosten 31 erreicht. Jetzt sieht man relativ einfach schon, daß es besser geht – aber der Graph ist ja auch überschaubar. Wir fangen wie für Backtracking üblich damit an, einen Schritt zurück zu gehen und die Alternative auszuprobieren. Das wäre von der 9 zur 7. Das kostet nur 3, wir sind also bei 28. Von dort aus gehen wir zur 4 und stellen fest, daß wir da schonmal waren. Also gehen wir stattdessen zur 10, haben bisher Kosten von 33, das ist mehr als der bisher beste Weg (31), und damit untauglich. also gehen wir einen weiteren Schritt zurück, bevor wir nach Alternativen suchen, und stecken in der 8. Da gibt es nur die Möglichkeit zur 9, von der wir uns gerade zurückgezogen haben. Also noch ein Schritt zurück, und von der 4 zur 7. Kosten sind dabei $15 + 2 = 17$. Das ist gut, weiter zur 9 und wir bezahlen 20. Von dort aus kommen wir mit 6 weiteren zur 11 und haben damit Gesamtkosten von 26. Wir vergessen also den bisher besten Weg und nehmen den jetzigen als solchen. Man sieht wieder auf einen Blick, daß es günstiger ist über die 10 zu laufen ($5 + 2 < 3 + 6$). Der Algorithmus wird systematisch alle durchprobieren, wenn man ihn denn entsprechend programmiert (also wie beim Labyrinth und beim Haus vom Nikolaus alle Möglichkeiten nach einer festen Reihenfolge abklappern).

Ein gewichteter Graph wird ebenfalls in einer Adjazenzmatrix dargestellt, jedoch wird nicht eine 1 für jede Kante eingetragen, sondern direkt das Gewicht dieser. Um die nichtverbundenen Kanten zu erkennen, kann man eine -1 eintragen. Die Matrix für den Graphen sieht dann wie folgt aus:

	0	1	2	3	4	5	6	7	8	9	10	11
0	+	6	4	-1	-1	-1	-1	-1	-1	-1	-1	-1
1		+	-1	5	-1	2	-1	-1	-1	-1	-1	-1
2			+	-1	2	-1	-1	-1	-1	-1	-1	-1
3				+	4	2	3	-1	-1	-1	-1	-1
4					+	-1	-1	2	3	-1	-1	-1
5						+	5	-1	-1	-1	-1	-1
6							+	-1	-1	-1	4	-1
7								+	-1	3	5	-1
8									+	7	-1	-1
9										+	-1	6
10											+	2
11												+

Die untere Hälfte habe ich ausgelassen, denn sie ist sowieso ein Spiegelbild der oberen, schließlich handelt es sich um einen *ungerichteten* Graphen. Statt der -1 kann man auch einen beliebigen anderen Wert verwenden, der nicht mit einem gültigen Wert für die Kosten verwechselt werden kann.

Die Matrix zu füllen ist eine lästige und fehlerträchtige Angelegenheit; im Programm sieht es in etwa so aus:

```
void fill_matrix(int matrix[NODES][NODES])
{
  int i, j;

  for (i = 0; i < NODES; i++)
    for (j = 0; j < NODES; j++)
      matrix[i][j] = ILLEGAL;

  matrix[0][1] = 6;
  matrix[0][2] = 4;

  matrix[1][3] = 5;
  matrix[1][5] = 2;
```

```
  matrix[2][4] = 2;

  matrix[3][4] = 4;
  matrix[3][5] = 2;
  matrix[3][6] = 3;

  matrix[4][7] = 2;
  matrix[4][8] = 3;

  matrix[5][6] = 5;

  matrix[6][10] = 4;

  matrix[7][9] = 3;
  matrix[7][10] = 5;

  matrix[8][9] = 7;

  matrix[9][11] = 6;

  matrix[10][11] = 2;
}
```

Danach wird die Matrix noch gespiegelt, und schon kann's losgehen. Der Weg wird auf der Suche nach einem noch besseren immer geändert, weshalb es nötig ist, die Matrix nach jedem Erreichen des Zielknotens zu kopieren. Hier ist interessante Funktion `find_way`:

```
int find_way(int matrix[NODES][NODES], int node, int way[EDGES + 1],
  int depth, int sum, int best_way[EDGES + 1])
{
  int i, ret = 0, old;

  assert(depth <= EDGES);

  way[depth] = node;

  if (sum > best_sum)
    return 1;

  if (node == FINISH)
  {
    best_sum = sum;
    last_depth = depth;
    for (i = 0; i <= depth; i++)
      best_way[i] = way[i];
    return 1;
  }

  for (i = 0; i < NODES; i++)
  {
    if (matrix[node][i] != ILLEGAL)
```

```
    {
      old = matrix[node][i];
      matrix[node][i] = ILLEGAL;
      matrix[i][node] = ILLEGAL;
      if (find_way(matrix, i, way, depth + 1, sum + old, best_way))
        ret = 1;
      matrix[node][i] = old;
      matrix[i][node] = old;
    }
  }

  return ret;
}
```

Wieder wird in `way` der Weg gemerkt, `node` ist die aktuelle Ecke. Wenn die Summe größer als die bisher beste Summe ist, so wird eine 1 zurückgegeben (wieder ist es nur entscheidend, daß der Funktionsablauf unterbrochen wird, der Wert ist nur für den allerersten Aufrufer interessant). Wenn die Ecke der Zielpunkt ist, so wird die Summe als neue beste Summe gemerkt (sie muß besser als die alte sein, denn sonst wäre die Funktion nicht so weit ausgeführt worden), die dann aktuelle Tiefe wird ebenfals zur Auswertung gemerkt. Der aktuelle Weg wird in `best_way` kopiert, denn er wird ja andauernd verändert. In der Schleife ist nichts neues, außer daß sich das Programm die Kosten merken muß, bevor es durch `ILLEGAL` überschrieben wird, um es später wiederherstellen zu können (beim Nikolaus war dies ja nur 0 und 1, hier jedoch gibt es alle möglichen Werte).

Die Rekursion nimmt neben der Tiefe noch die aktuelle Summe mit. Der beste Weg ist nur mit dabei, damit er nicht als globale Variable erscheint (glaube ich, ist schon ein bißchen her, und natürlich habe ich das Programm *nicht* kommentiert...).

Dijkstras Algorithmus kennt noch ein paar andere nette Anwendungsmöglichkeiten. Wenn man mit offenen Augen durch die Welt geht, so findet man nicht selten Probleme, die sich durch (ggf. gewichtete und/oder gerichtete) Graphen darstellen lassen. Ein Beispiel ist das Finden der besten Route in einem Netzwerk. Die Kosten hierbei können neben tatsächlichen Leitungskosten, bei denen über Volumen abgerechnet wird, auch Parameter wie Geschwindigkeit, MTU oder Latency sein.