EIN NETZ-EDITOR

T. Hürlimann

Working Paper No. 195

Februar 1992

INSTITUTE FOR AUTOMATION AND OPERATIONS RESEARCH

University of Fribourg

CH-1700 Fribourg / Switzerland

Bitnet: HURLIMANN@CFRUNI51

phone: (41) 37 21 95 60 fax: 37 21 96 70

Ein Netz-Editor

Tony Hürlimann, Dr. rer. pol.

Key-words: Model Building, Graphmanipulation, Networks.

Abstract: The paper presents a simple graphical editor, which allows one to edit a network (graph) together with its data on a computer in a graphical way. The network can be saved in a textfile which can be edit indepentently by any word-processor. The paper proposes a universally usable text-format for networks.
Furthermore, the program is strictly separated into three modules: the internal data structure and the manipulation operations of networks which are fully portable, graphical environment which is easy portable and user interface which is not portable at all. The reason for this architecture was to show by example, how to write portable software for a non-trivial application. The program was implemented in THINK PASCAL 4.0 on a Mac and was ported to IBM compatible PC in WINDOWS in TURBO PASCAL for WINDOWS.

Stichworte: Modellierung, Graphen-Manipulation, Netzwerk

Zusammenfassung: Dieses Paper beschreibt einen einfachen Graphikeditor, der erlaubt interaktiv Netzwerke und Graphen dem Komputer einzugeben und die Daten des Netzes unabhängig vom Editor in einer Textdatei abzuspeichern, die mit einen Texteditor bequem weiterbearbeitet werden können. Es wird ein externes Datenformat für Netzwerke und Graphen vorgeschlagen, welches universell und erweiterbar ist.
Das Programm wurde streng in drei Module aufgeteilt, und soll so auch beispielhaft zeigen, wie portable Software zu schreiben ist. Die drei Module beinhalten die interne Datenstruktur von Netzwerken, die voll portabel ist, die graphische Umgebung und die Oberfläche. Das Programm wurde auf dem Mac in THINK PASCAL 4.0 implementiert und dann auf IBM PC Kompatible unter WINDOWS in TURBO

PASCAL für WINDOWS umgeschrieben.

1. EINLEITUNG

Dieses Paper verfolgt zwei Ziele: erstens soll eine nützliche Applikation beschrieben werden: Das Programm erlaubt Netzwerke graphisch zu manipulieren; und zweitens soll beispielhaft gezeigt werden, wie eine nicht-triviale Applikation möglichst portabel erstellt werden kann. Für beide Ziele steht der zentrale Begriff der **Datenstruktur** im Vordergrund, wobei hier unter Datenstruktur die eigentlichen Daten *zusammen* mit den Operatoren, die darauf angewendet werden können, verstanden werden soll.

Um ein nicht-triviales Objekt - hier ein Graph - mit Hilfe des Komputers manipulieren und bearbeiten zu können, muss zunächst eine geeignete Datenstruktur definiert werden. Wir fordern zudem, dass das Objekt zusätzlich auf einem sekundären Speicher in einem universellen und vom Benützer leicht zugreifbarem Format abgespeichert wird. Dies erlaubt neben dem permanenten Aufbewahren des Objekts, es in verschiedenen, anderen Applikationen wiederverwenden zu können. Wichtig ist auch, dass das Objekt unabhängig vom Graphikeditor von jedem Textverarbeitungssystem manipuliert werden kann. Wir definieren daher eine *interne* Representation des Objekts, welches für den Graphikeditor bequem ist, und ein *externes* Format, welches universell und unabhängig von Editor ist. Die interne Datenstruktur soll möglichst dynamisch sein und soll erlauben, die einzelnen Manipulationsfunktionen im Editor möglichst schnell ausführen zu können. Diese werden im zweiten Abschnitt ausführlich besprochen. Die graphische Umgebung wird im dritten Abschnitt behandelt. Dort werden die graphischen Routinen zusammengestellt, welche es erlauben, die interne Datenstruktur in eine sichtbare Struktur auf einem Ausgabengerät (ein Graphikport oder Fenster auf dem Bildschirm) aufzubauen. Im vierten Teil wird die Programmumbegung näher beschrieben. Ein weiterer Abschnitt ist dem externen Format gewidmet, und seine Spezifikationen werden beschrieben.

Es mag vielleicht erstaunen, dass gerade für eine solche Applikation nicht streng das objekt-orientierte Paradigma verwendet worden ist. Dies würde sich doch scheibar geradezu anbieten: wir haben Objekte (Graph, Knoten, Kanten im Graph), die genau 'wissen', welche Eigenschaften und Attribute sie haben. Sie 'wissen' sogar wie sie sich manifestieren (zeichnen). Aber gerade hier fangen die Schwierigkeiten an: Gehört das Zeichnen eines Objekts zum Objekt selber oder gehört es zur Umgebung? Unser Ziel ist eine portable Software zu schreiben. Da das Zeichnen in jeder Umgebung anders zu handhaben ist, bin ich eher geneigt, diese Funktionen vom Objekt möglichst zu

entfernen. Dies beleuchtet ein meist unbeachteter Aspekt des objekt-orientierten Programmierens: Wenn die Objektklassen auf verschiedenen Umbegungen verschieden sind - und wo sind sie das nicht - behindert das objekt-orientierte Programmieren die Portabilität.

Portable Programme sind also Programme, die möglichst viel Kode im 'invarianten' Teil (hier die interne Datenstruktur des Graphen) platziert und der variable Teil auslagert; objekt-orientiertes Programmieren versucht hingegen, die Objekte möglichst 'intelligent' zu machen, und implementiert daher möglichst viele, zum Objekt gehörenden Methoden. Portabilität und objekt-orientiertes Programmieren schliessen sich zwar nicht aus, können aber auch konfliktuelle Ziele sein.

2. INTERNE DATENSTRUKTUR

Die interne Datenstruktur erlaubt, *unabhängig von der Darstellung* auf dem Bildschirm, einen Graphen zu manipulieren. Eine interne Darstellung muss die gesamte Struktur eines Objekts sowie die Operationen, die darauf angewendet werden sollen, zunächst *spezifizieren* und hernach *implementieren*. Die Spezifikationen eines Graphen müssen ganz unabhängig von den graphischen Manipulationen definiert sein. Die Implementation hingegen richtet sich darnach, wie das Objekt eingesetzt wird und welche Operationen darauf angewendet werden sollen. Ein Kriterium dafür ist die Effizienz und der Speicherplatzbedarf. Aber auch die Implementation der Datenstruktur sollte in dem Sinne unabhängig von der graphischen Darstellung sein, dass die interne Darstellung neu implementiert werden könnte, ohne dass die graphischen Prozeduren (Abschnitt 3) verändert werden müssen. Dies ist natürlich ein Idealfall. In der Realität muss man nicht zu weit gehen. Es genügt die Forderung, dass die graphischen Routinen nur 'unwesentlich' zu verändern sind, um sie auf die interne Darstellung neu abzustimmen.

Wir spezifizieren zunächst das Objekt Graph. Ein Graph besteht aus *Knoten* (vertices) und *Kanten* (edges) oder *Bögen* (arcs). Knoten sind die Eckpunkte eines Graphen und die Kanten sind die beid-wegige Verbindungen zwischen Knoten. Bögen sind ein-wegige Verbindungen zwischen Knoten und werden gewöhnlich als Linien mit einem Pfeil dargestellt. Die Abbildung 1.1 representiert einen Graphen als graphisches Gebilde, bestehend aus 5 Knoten (die Kreise), 5 Kanten und einem Bogen (Verbindung 5 nach 2).

Kanten sind ungerichtet und Bögen sind gerichtet. Daher bezeichnet man Graphen mit nur Kanten als *ungerichteten* Graphen, und solche mit nur Bögen als *gerichteten*

Graphen. Aber auch gemischte Graphen sind möglich, wie Abbildung 1.1 zeigt. Hingegen beschränken wir uns auf Graphen mit maximal einer Verbindung zwischen zwei Knoten. Aus der Definition kann unmittelbar die Spezifikation abgeleitet werden:

Ein Graph ist besteht aus
- einer Menge von Knoten
- einer Menge von Kanten und
- einer Menge von Bogen

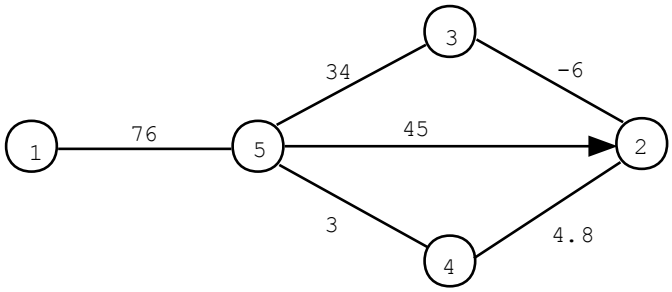

Abbildung 1.1

Jeder Knoten und jede Verbindung besitzt eine bestimmte Anzahl von *Attributen*. Z.B. besitzt jeder Knoten des Graphen in Abbildung 1.1 einen Identifikator ('1'...'5') und jede Verbindung besitzt eine Zahl: Welche Bedeutung wir den Attribute zuordnen, spielt hier keine Rolle. Fast jeder Graph besitzt jedoch einige Attribute:

Jeder Knoten besitzt
- einen Identifikator
- einen Namen (kann identisch sein mit dem Identifikator)
- eine Position (für die graphische Representation)
- eine Farbe und eine Form (Kreis, Rechteck usw.)
- je nach Applikation verschiedenen numerische Grössen (Kapazität)

Jeder Kante und jeder Bogen besitzt
- einen Identifikator (identisch mit den beiden Eckknotenidentifikatoren)
- einen Namen (kann fehlen)
- eine Farbe und eine Form (gestrichelt, ausgezogen, dicke Linie usw.)
- je nach Applikation verschiedenen numerische Grössen (Weglänge)

Um den Zugriff zu beschleunigen, definieren wir zwei Identifikatoren für einen Knoten: eine reine interne Nummerierung und ein externer Schüssel. Damit könnte die Struktur in PASCAL folgendermassen aussehen

```
const
 VMAX = 500;          { maximal number of vertices }

type
 TKey = string[3];    { key type for vertex search (Idendifikator)}
 Tname = string[30]; { name type }
 pvertex = ^vertex;  { pointer to vertex }
 pedge = ^edge;       { poiter to edge }

 vertex = record
   id: integer;           { identifier (Identifikator als Nummer)}
   key: TKey;             { key (Identifikator)}
```

```
  name: Tname;
  x, y: integer;      { .... position for drawing (COOR) }
  color: integer;
  shape: integer;
  i1, i2: integer;   { integer data }
  r1, r2: real;      { real data }
  edgelist: pedge;
 end;

edge = record
  id: integer;        { to vertex identifier }
  directed: integer;  { 0=undirected, 1=directed, -1=directed but is a in-edge }
  color: integer;
  shape: integer;    { thickness, dotted, ect. }
  i1, i2: integer;   { integer data }
  r1, r2: real;      { real data }
  next: pedge;
 end;

GRAPH = record
  size: integer;
  V: array[1..VMAX] of vertex;
 end;
```

Für die Knotenliste wird zum voraus der entsprechende Platz reserviert, und Kanten und Bogen werden als Zeigerliste in beliebiger Reihenfolge dem jeweiligen Knoten hinzugefügt. Um die Effizienz zu erhöhen, fügen wir eine Kante in die Liste beider Endpunkte (Knoten) hinzu. Das Feld 'directed' ist 0, wenn eine Kante gemeint ist, 1 oder -1 wenn es sich um einen Bogen mit der entsprechenden Richtung handelt. Jede Kante wird also zweimal mit 0 eingefügt; ein Bogen wird beim Ausgangsknoten mit 1 und beim Endknoten mit -1 eingetragen.

Die Datenstruktur bietet einen guten Kompromiss zwischen Effizienz, Flexibilität und Speicherplatzbedarf vor allem für dünn besetzte Graphen, wie wir sie in der Praxis ja häufig antreffen. Vollständige Graphen (Graphen, bei denen alle Knoten miteinander verbunden sind) können zwar auch damit implementiert werden, aber die Effizienz ist sehr eingeschränkt und der Speicherplatzbedarf gross.

Die Effizienz hängt natürlich davon ab, welche Operationen auf den Graphen implementiert werden müssen. Bei einem Grapheditor spielt die Effizienz keine grosse Rolle, da unsere Operationen im Prinzip auf die folgenden 6 Operationen beschränkt werden kann

- Knoten hinzufügen
- Knoten finden
- Knoten löschen
- Kante (Bogen) hinzufügen
- Kante (Bogen) finden
- Kante (Bogen) löschen

Weitere Operationen könnten sein

- Initialisiere einen Graphen (leerer Graph);
- Testoperation, ob der Graph leer ist
- Lese einen Graphen von der externen Datenstruktur (Abschnitt 5)
- Schreibe einen Graphen in die externe Datenstruktur

Zwei zusätzliche Operationen, welche erlauben, die Struktur eines Knoten und einer

Kante aufzufüllen, erleichtern den Umgang mit Knotendaten und Kantendaten. Diese beiden Operationen scheinen trivial, sind aber äusserst hilfreich und vergrössern wesentlich die Transparenz des Kodes. Damit haben wir die folgenden Operationen.

```
function FindVertex (var G: GRAPH; key: TKey): integer;
procedure AddVertex (var G: GRAPH; var v: vertex);
procedure DeleteVertex (var G: GRAPH; key: Tkey);
function FindEdge (var G: GRAPH; fromV, toV: integer): pedge;
procedure AddEdge (var G: GRAPH; fromV, toV: integer; var e: edge);
procedure DeleteEdge (var G: GRAPH; fromV, toV: integer);
procedure ReadGraphFromFile (var G: GRAPH; FileName: string);
procedure WriteGraphToFile (var G: GRAPH; FileName: string);
procedure NewGraph (var G: GRAPH);
function IsGraphEmpty (var G: GRAPH): boolean;

procedure SetVertex (var v: vertex; key: TKey; name: Tname; x, y: integer;
                     color: integer; shape: integer; i1, i2: integer; r1, r2: real);
procedure SetEdge (var e: edge; directed: integer; color: integer; shape: integer;
                   i1, i2: integer; r1, r2: real);
```

Intern sind die Knoten beginnend bei Eins durchnumeriert. Die Funktion **FindVertex** gibt diese Nummer des Graphen G zurück, wenn der Schlüssel als Parameter übergeben wird, oder Null wenn der Knoten nicht existiert. Man sollte beachten, dass jede Prozedur einen Parameter G vom Typ GRAPH besitzt, was erlaubt, die Funktionen gleichzeitig für verschiedene Graphen zu benützen. Die Prozedur **DeleteVertex** löscht einen Knoten mit dem Schüssel *key*. Wenn der Knoten nicht existiert, macht die Prozedur nichts. **AddVertex** fügt einen Knoten am Schluss der Liste hinzu. Unabhängig davon, was der Benutzer im Feld *v.id* eingibt, dieses Feld wird von der Prozedur automatisch verteilt, der Schlüssel hingegen kann vom Benützer der Prozedur gewählt werden. Da der Schlüssel eindeutig sein muss, testet die Prozedur, ob der Schlüssel bereits verteilt wurde, wenn ja so wird von der Prozedur automatisch ein neuer Schlüssel geliefert und der alte weggeworfen. Die Function **FindEdge** gibt einen Zeiger auf die Datenstruktur der Kante zurück, oder *nil*, wenn die Kante nicht gefunden wurde. **AddEdge** fügt eine Kante zwischen den Knoten *fromV* und *toV* ein. Die Felder *e.id* wird dabei von *toV* überschrieben und *e.next* wird von der Prozedur bestimmt. Die Kante wird zweifach hinzugefügt: bei der Kantenliste des Knotens *fromV* sowie bei Knoten *toV*. Wenn die Kante schon existiert wird keine neue Kante hinzugefügt, sondern der Inhalt der Datenstruktur verändert (ausser selbstverständlich *id* und *next*). Die Prozedur **DeleteEdge** löscht eine Kante aus dem Graph, dabei werden automatisch beide Einträge gelöscht. Wenn die Kante nicht existiert, macht die Prozedur nichts. **ReadGraphFromFile** und **WriteGraphToFile** wandelt die externe Struktur (siehe Abschnitt 5) in eine interne Datenstruktur um und umgekehrt. Eine hilfreiche Spezialität der ReadGraphFromFile ist die automatische Zufallsgenerierung der Position der Knoten, falls die Position noch nicht bekannt ist. Der Benützer kann dann durch die graphische Manipulation diese Positionen bequem verändern. **SetEdge** und **SetVertex** sind trivial. Sie füllen

die Datenstruktur einer Kante und eines Knotens mit den entsprechenden Daten auf. Die Trivialität dieser Funktionen ist so frappant, dass man sich fragen muss, wozu sie eigentlich nützlich sind. Ihre einzige Funktion ist denn auch, die Übersichtlichkeit und Transparenz im Programm *wesentlich* zu erhöhen.

Damit ist die interne Datenstruktur definiert. Diese Struktur und ihre Implementation ist unabhängig von der graphischen Umgebung. Diese interne Struktur und die dazugehörigen Operationen sind auch unabhängig von der Umgebung, soweit die Funktionen in einem Standard PASCAL geschrieben werden. Dies ist aber ohne weiteres möglich, ohne grosse Konzessionen an die Effizienz zu machen. Neueren Entwicklungen des PASCALs entsprechend, kapseln wir die Spezifikation und Implementation der internen Datenstruktur des Graphen in ein UNIT ein und legen es in der Datei *GraphInt.u* ab.

3. GRAPHISCHE UMGEBUNG

Die graphische Umgebung fasst alle Operationen zusammen, welche den Graphen oder Teile davon auf ein Ausgabegerät zeichnen. Diese Operationen können auch weitgehend unabhängig von der Betriebssystem-Umgebung geschrieben sein. Die graphische Umgebung benötigt einzig die interne Datenstruktur und der Zugang zu einigen primitiven Zeichnungsfunktionen wie LineTo(), MoveTo(), SetRect(), FrameOval(), ForeColor(), um einige zu erwähnen, welche in der Toolbox des Macintosh zu finden sind. Selbstverständlich sich die meisten dieser Funktionen in verschiedenen Betriebsumgebungen verschieden, haben aber meist ihre Entsprechung, andernfalls muss eine entsprechende Funktion geschrieben werden. Diese Funktionen sind meist sehr einfach und bestehen aus wenigen, oft nur aus einer Anweisung. Falls die Effizienz nicht kritisch ist, sollte man sich nicht scheuen, solche Funktionen zu schreiben: sie erlauben, den Kode in unserer graphischen Umgebung meist eins zu eins zu übernehmen. Folgende Funktionen gehören zur graphischen Umgebung:

```
procedure DrawIt (var G: GRAPH);

procedure DrawEdge (var G: GRAPH; fromV, toV: integer; e: pedge; hidden: boolean);
procedure DrawNode (v: vertex; hidden: boolean);
procedure DrawNodewithEdges (var G: GRAPH; v: vertex; hidden: boolean);
procedure MoveNode (var G: GRAPH; id: integer);
procedure Redraw (var G: GRAPH; dZoom, Dx, Dy: integer; hidden: boolean);
function IsInSomeNode (var G: GRAPH; var which: integer; p: point): boolean;
function IsInShapeMenu (var G: GRAPH; var which: integer; p: point): boolean;
procedure InitDraw (vColor, eColor: integer);

procedure AddAndDrawEdge (var G: GRAPH; fromV, toV, d: integer; hidden: boolean);
procedure AddandDrawNode (var G: GRAPH; shape: integer; p: point);
procedure DialogNode (var G: GRAPH; var v: vertex);
procedure DialogEdge (var G: GRAPH; var e: edge);

procedure EditIt (var G: GRAPH; p: point; ModKeys: integer;
                                  var id, id1, which: integer);
```

Die meisten Funktion haben als Parameter *G* und *hidden*. *G* ist der Graph und *hidden* bestimmt, ob die Zeichenfunktionen in der Hintergrundfarbe oder der in der Datenstruktur definierten Fabe auszuführen sind. Dadurch kann dieselbe Funktion zum Zeichnen und zum Löschen verwendet werden.
Nur die letzten vier Prozeduren verändern die interne Struktur des Graphen, alle andern lassen sie ungerührt (ausser *MoveNode* und *Redraw*, welche die Koordinaten der Knoten ändert). Wenn also eine Kante mit der Funktion **DrawEdge** gelöscht wird (hidden:=TRUE), so wird die interne Struktur nicht tangiert. Dasselbe gilt für die Prozeduren **DrawNode** und **DrawNodeWithEdges**, welche die Knoten bloss zeichnen aber nicht verändern.
Die Prozedur **MoveNode** löscht und zeichnet den Knoten mitsamt den Kanten in schneller Folge, solange die Maustaste bedrückt gehalten wird. Diese Prozedur muss natürlich fähig sein, die Maustaste abzufragen. Die Prozedur **ReDraw** zeichnet den ganzen Graphen neu. Die Parameter dZoom, Dx und Dy führen eine lineare Transformation (Dehnung oder Stauchung (dZoom) und Translation (Dx, Dy)) durch, falls diese verschieden von Null sind. Auch dadurch werden die Koordinaten der Knoten verändert. Wenn also der gesamte Graph gezoomt werden soll, so muss er zuerst mit *ReDraw(G,0,0,0,TRUE)* gelöscht werden, und anschliessend mit *ReDraw(G,dZoom,0,0,FALSE)* (mit dZoom>0) wieder gezeichnet werden. **IsInSomeNode** testet, ob der Punkt *p* sich in einem Knoten befindet, wenn ja, wird TRUE zurückgegeben und *which* gibt den Knoten an. Typischerweise wird in *p* die Position des Mauszeigers der Funktion übergegeben, sodass getestet werden kann, ob der Mauszeiger sich in einen Knoten befindet. Die Funktion **InitDraw** löscht die Zeichenoberfläche und zeichnt das Zeichenmenu (siehe Abbildung 3.1) auf der linken Zeichenoberfläche. **IsInShapeMenu** testet, ob sich der Punkt *p* im Zeichenmenu befindet und in welcher Zeichenform. Die ersten sechs Zeichenformen stehen für Knoten, die siebente steht für Kanten und die achte für Bogen. Eine der Formen wird auf 'aktiv' gesetzt. Die aktive Form wird einer globalen Variable *which* übergeben.

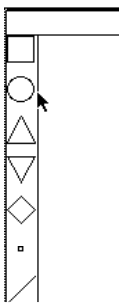

Abbildung 3.1

AddAndDrawNode und **AddAndDrawEdge** fügen einen Knoten bzw. eine Kante der internen Datenstuktur hinzu und zeichnen sie anschliessend. **DialogNode** und

DialogEdge erlauben, interaktiv die Daten eines Knotens oder einer Kante zu modifizieren. Diese beiden letzten Funtionen hängen am meisten von der Betriebsumgebung ab. Auf dem Macintosh wird dies über ein Dialogfenster getan. Die Prozedur **EditIt** verwaltet die ganze Editierabeit auf der Zeichenfläche. Der Prozedur werden zwei Parameter, ein Punkt *p* - typischerweise die Mauszeigerposition - und eine *ModKeys* übergeben. Der *ModKeys* Parameter übernimmt die gesamte Information, die angibt, welche Umschalttasten (Kontrol, Alt usw.) gedrückt worden sind. Die Referenzparameter *id* und *id1* geben an, ob und in welchem Knoten zuletzt die Maustaste betätigt wurde. Sie sind Null, wenn die Maus nicht in einem Knoten gedrückt wurde. Die beiden Variablen werden ausser der Initialisierung nur in dieser Prozedur EditIt verändert, müssen aber in PASCAL leider als globale Variablen definiert werden, da PASCAL keine statischen Variablen (wie C) kennt. *Which* gibt den aktiven Menueintrag an. Wenn also *which* auf 1 (=Rechteck) eingestellt ist, so werden alle neuen Knoten als Rechtecke definiert. *Which* wird auf 1 initialisiert und nur in IsInShapeMenu verändert.
Die Prozedur EditIt unterscheidet drei Fälle:
1. *p* ist im Zeichenmenu: dann wird *which* verändert und *id* und *id1* auf Null initialisiert.
2. *p* befindet sich in einem Knoten: dann wird der Knoten manipuliert, falls which<=6 (Knoten aktiv) ist, sonst wird der Knoten in *id1* registriert. Falls jedoch *id1* schon registiert war und which>6 (Kante aktiv) ist, wird eine Kante zwischen *id1* und *id* gezeichnet. Falls - wie gesagt - which<=6 ist, dann kann der Knoten selber manipuliert werden und zwar, wird keine Umschalttaste gedrückt, so wird der Knoten verschoben, bis die Maustaste losgelassen wird, wird die OPTIONS-Taste gedrückt, so können die Daten der Knoten interaktiv über ein Dialogfenster modifiziert werden, wird die □-Taste gedrückt, so wird der Knoten gelöscht, wird die SHIFT-Taste gedrückt und gedrückt gehalten, währenddem die Maus bis zu einem andern Knoten verschoben wird, so wird eine Kante gezogen.
3. *p* ist woanders auf der Zeichenfläche: dann wird eine neuer Knoten kreiert, falls which <=6 ist.

Die Funktionsweise von EditIt ist natürlich etwas willkürlich. *Aber sie erlaubt uns, alle nötigen Funktionen zur Manipulation der Graphen auszuführen, ohne dass die Betriebsumbegung wesentlich zu berücksichtigen ist*. Abgesehen davon, dass die Umschalttasten selbstverständlich in jeder Umgebung anders abgefragt werden müssen, ruft diese Prozedur nur andere Funktionen aus der graphischen Umgebung und der internen Datenstruktur auf, und ist daher in hohem Masse portabel. Ihre Implementation könnte folgendermassen skizziert werden.

```
procedure EditIt (var G: GRAPH; p: point; ModKeys: integer;
                                var id, id1, which: integer);
 var
  i: integer;
begin
 if IsInShapeMenu(G, which, p) then begin    {-----1.mouse is in DrawMenu----}
   id := 0; id1 := 0;
  end
 else if IsInSomeNode(G, id, p) then begin   {-----2.mouse is in a node----}
   if which > 6 then begin              {--2.1 edge drawing is active--}
     if id1 = 0 then
      id1 := id
     else begin
        i := which - 7;
        AddAndDrawEdge(G, id1, id, i, FALSE);
        id := 0; id1 := 0;
      end
    end {which>6}
   else if cmdKeyWasPressed then begin   {--2.2 node drawing is active--}
     DrawNodeWithEdges(G, G.V[id], TRUE);  {-2.2.1 so delete it-}
     DeleteVertex(G, G.V[id].key);
    end
   else if optionKeyWasPressed then begin {-2.2.2 modify it-}
     DialogNode(G, G.V[id]);
    end
   else if shiftKeyWasPressed then begin  {-2.2.3 draw edge from here-}
     while button do
      GetMouse(p);
     if IsInSomeNode(G, id1, p) then             {-to here-}
      AddAndDrawEdge(G, id, id1, 0, FALSE);
    end
   else
    MoveNode(G, id);                      {-2.2.4 move it-}
  end
 else begin                                  {----3. mouse is elsewhere----}
   AddandDrawNode(G, which, p);
  end;
end;
```

Abgesehen von den drei Funktionen *cmdKeyWasPressed*, *optionKeyWasPressed*, *shiftKeyWasPressed*, welche einen boolschen Wert zurückgeben, und der Mausabfrage, ist die Funktion EditIt vollständig transparent und portabel. Die Prozedur könnte natürlich noch weitere Funtionalitäten übernehmen: Das Kantenlöschen ist z.B. nicht vorhanden. Es sollte einfach sein, diese Option einzufügen.

4. INTERFACE

Eine Bedingung wird allerdings von der Prozedur EditIt vorausgesetzt: ein ereignisgesteuerter Ablauf des Hauptprogramms. EditIt setzt voraus, dass ihr als Parameter die Angabe der Mausposition sowie die Information der Umschalttasten des letzten Ereignisses (Event) in *p* und *ModKeys* übergeben wird. Diese Annahme ist nicht sehr einschränkend, da ein graphischer Editor ohnehin nur in einer ereignisgesteuerten Umgebung Sinn macht.

Der ereignisgesteuerte Ablauf ist ganz von der Betriebsumbegung abhängig. Die Grundstruktur eines ereignisgesteuerten Programm ist zwar in verschiedenen Umgebungen in etwa gleich:

```
program EventProgramm;
 var
  Finished: boolean;
  myEvent: EventRecord;

 procedure Initialize;
 begin
  Finished := FALSE;
 end;

 procedure MainLoop;
 begin
  SystemTask;  { let the system get a chance to do something }
  DoEvent;     { process all Events }
 end;

 procedure Finalize;
 begin
 end;

begin      {---- main program ----}
 Initialize;
 repeat
  MainLoop;
 until Finished;
 Finalize;
end.
```

Initialize muss alle nötigen Initialisierungen vornehmen, jedoch mindestens die globale Variable *Finished* auf FALSE setzen, damit der MainLoop wiederholt ausgeführt wird. **Finalize** muss alle Dateien schiessen, den Speicher freigeben, Fenster entfernen und jede Art Aufräumarbeit ausführen, bevor das Programm beendet werden kann. **MainLoop** stellt zunächst dem System eine Zeitspanne zur Verfügung und behandelt dann all Ereignisse (Events). Die Ereignisse werden in eine Schlange eingetragen.

In der Macintosh Umgebung gibt die Toolboxfunktion *GetNextEvent* das nächste Ereignis als Datenstruktur (EventRecord) zurück, welche folgende Felder besitzt:

```
type
  EventRecord = record
    what      : integer;  { event type }
    message   : longint;  { type dependent info }
    when      : longint;  { time of event }
    where     : point;    { mouse position }
    modifiers : integer;  { state of modifier keys }
  end;
```

Das what-Feld kann unter anderem sein:

NullEvt=0	kein Ereignis
MouseDown=1	Maus gedrückt
KeyDown=3	Taste gedrückt
AutoKey=5	repeat Taste
.....	
UpdateEvt=6	Fenster verschoben
ActivateEvt=8	neues Fenster

In der DoEvent Prozedur wird je nach Ereignis in die entsprechende Prozedur

verzweigt. Diese Prozedur sieht dann folgendermassen aus:

```
procedure DoEvent;
begin
 if GetNextEvent(everyEvent, myEvent) then
  case myEvent.what of
    MouseDown:         DoMouseDown;
    KeyDown, AutoKey:  DoKeystroke;
    updateEvt:         DoUpdate;
    activateEvt:       DoActivate;
    otherwise ;        { do nothing}
  end;
end;
```

Wenn die Maustaste gedrückt wurde, stellen wir über das Feld *where* fest, wo die Maus gedrückt wurde. Die Prozedur DoMouseDown sieht dann folgendermassen aus:

```
procedure DoMouseDown;
 var
  myWindow: WindowPtr;
begin
 case FindWindow(myEvent.where, myWindow) of
  inDesk :     { to nothing };
  inMenuBar :  DoMenuClick;
  inDrag :     DoDrag(myWindow);
  inGrow :     DoGrow(myWindow);
  inContent :  DoContent(myWindow);
  InGoAway:    DoGoAway(myWindow);
 end;
end;
```

Die DoKeystroke Prozedure ist auf dem Macintosh folgendermassen zu implementieren:

```
procedure DoKeystroke;
 var
  theChar: char;
begin
 theChar := CHR(BitAnd(myEvent.message, charCodeMask));
 if BitAnd(myEvent.modifiers, cmdKey) = cmdKey then begin
    DoMenu(MenuKey(theChar))
  end;
end;
```

Im Netzeditor werden nur Tasten zusammen mit der Kontrolltaste verwendet, die auf dem Macintosh standartmässig die Bedeutung einer Menuauswahl haben.

Mit den wenigen Basisprozeduren haben wir hier gleichzeitig den Aufbau einer jeden Macintosh Applikation gezeigt. Die einzelnen Prozeduren müssen jetzt noch fertig aus gestaltet werden, die Prozedur EditIt wird in die Prozedur DoContent hineingehängt. Über Menüpunkte LOAD und SAVE werden die Prozeduren ReadGraphFromFile und WriteGraphToFile aufgerufen, und schon haben wir einen funktionierenden Netzeditor!

Eine Bemerkung muss allerdings noch zu der internen Datenstruktur in diesem Zusammenhang gemacht werden. In der Prozedur AddEdge wird eine NEW Prozedur aufgerufen, welche im PASCAL standardmässig einen nicht relozierbaren

Speicherbereich auf dem Heap anfordert. Dies ist natürlich für den Memory-Manager des Macintosh völlig intolerabel, da dieser u.U. das Garbage Collection nur sehr unvollkommen durchführen kann. Eine Lösung wäre anstelle der NEW Prozedur die Prozedur NewHandle einzusetzen. Ganz abgesehen vom Effizienzverlust, der sich für den Editor allerdings in Grenzen hält, müsste so eine grosse Zahl von Handles zur Verfügung gestellt werden. Für kleine Graphen mag diese Lösung zweckmässig sein, da sie nur eine geringe Änderung in der internen Datenstruktur erfordert. Ein bessere Alternative wäre, die Prozedur NEW ganz durch eine neue Prozedur zu ersetzen, welche die Kantendatenstrukturen in Paketen von, sagen wir, 100 zur Verfügung zu stellen, und einen Handle auf je ein Paket global zu definieren. Mit 10 solchen Handles könnten wir 1000 Kanten eingeben. Natürlich müssen die Funktionen der Kantenmanipulationen umgeschrieben werden (Sie werden einfacher!).

5. EXTERNE DATENSTRUKTUR

Es wäre selbst für kleine Graphen (sprich Datenstrukturen) relativ mühsam, wenn ein Graph dem Programm jeweils interaktiv neu eingegeben werden müsste, selbst wenn die Eingabe graphisch ist. Bereits ein Graph mit 5 Knoten verlangt 35 Eingaben, wenn alle Kanten definiert werden müssen. Wird eine falsche Angabe gemacht oder stürtzt das Programm in der Entwicklungsphase ab, muss die gesamte Eingabe wiederholt werden. Die Datenstruktur muss daher extern in eine Datei abgespeichert werden können. *Wir fordern zudem, dass die Struktur dieser Datei auch unabhängig vom Programm (dem Graphikeditor) bearbeitet und gesichtet werden kann.* Dies ist vor allem von grossem Nutzen, wenn die Problemstellung (hier ein Netzwerk erstellen) unabhängig vom Programm entsteht, beispielweise die Daten aller Kanten in einer langen Liste bereits als editierfähige Datei verfügbar sind.
Wir definieren daher ein Dateiformat für Graphen, das für alle Graphen als Datenstruktur brauchbar ist und zudem von jedem Textverabeitungsprogramm modifiziert werden kann. Dies gibt uns eine grosse Flexibilität. Die Struktur der Datei hat folgendes Format:

1. Daten werden zeilenweise als Text eingegeben.

2. Vier verschiedene Zeilentypen werden unterschieden:
- Zeilen, die mit einen '*' beginnt oder leer ist. Sie werden ignoriert und werden vom Programm als blosser Kommentar interpretiert und übersprungen. Für die Dokumentation eines Graphen sind solche Zeilen wichtig und nützlich. Auch Leerzeilen werden ignoriert.
- Zeilen bestehend aus reservierten Wörtern (siehe unten).

- Knotenzeilen: Sie beinhalten die Daten der Knoten (Knotenattribute)
- Kantenzeilen: Sie beinhalten die Daten der Kanten (Kantenattribute).

3. Folgende Wörter sind reserviert:

NODES VERTICES EDGES ARCS
NAME COOR REAL INT COLOR SHAPE

Sie müssen grossgeschrieben werden und haben ein spezielle Bedeutung.

NODES, VERTICES, EDGES, ARCS müssen eine Zeile einleiten, und bedeuten, dass von der nächsten Zeile an alle folgenden Zeilen Daten zu Knoten (vertices oder nodes), ungerichtete Verbindungen zwischen Knoten (edges) oder gerichtete Verbindungen (arcs) enthalten. NODES und VERTICES bedeuten dasselbe und werden identisch interpretiert.

Alle andern reservierten Wörter (NAME, COOR, REAL, INT, COLOR, SHAPE) können in beliebiger Reihenfolge auf derselben Zeile wie die vier reservierten Wörter beliebig oft wiederholt werden und müssen durch einen Zwischenraum oder einen Tabulator voneinander getrennt werden. COOR darf nur auf der Zeile beginnend mit VERTICES (oder NODES) vorkommen.
Beispiele von zulässigen Zeilen:

```
VERTICES COOR REAL INT NAME
EDGES INT COLOR
ARCS INT SHAPE COLOR
```

Die reservierten Wörter NAME, COOR, REAL, INT, COLOR, SHAPE geben an, welche Attribute ein Knoten oder eine Kante (Verbindung) besitzt. Diese Attribute bedeuten das folgende:

NAME steht für den Namen eines Knotens (oder einer Kante) (höchstens 20 Zeichen).

COOR steht für die Weltkoordinaten eines Knoten. Die Daten bestehen aus zwei ganzen, positiven Zahlen, welche die Position (x,y) des Knotens in der zwei-dimensionalen Fläche representieren. x - die erste Zahl - bestimmt die horizontale und y die vertikale Position. Die Weltkoordinate (0,0) ist die linke, obere Ecke der Fläche. Die Daten COOR dienen ausschliesslich für die graphische Representation der Graphen. Die Koordinaten sind aber nicht unbedingt identisch mit den Koordinaten auf dem Bildschirm. Falls diese Angabe in der Datei fehlt, werden beim Lesen der Datei zufallsgenerierte Koordinatenpunkte für alle Knoten generiert.

REAL fügt dem Knoten oder der Kante einen reelen Wert hinzu (z. B. Kosten, oder maximarer Fluss einer Kante).

INT fügt dem Knoten oder der Kante einen ganzzahligen Wert hinzu (z. B. Kapazität an einem Knoten).

COLOR gibt die Farbe des Knotens oder der Kante an. Ist die Farbe identisch mit der Hintergrundfarbe, so ist das Objekt nicht sichtbar.

SHAPE gibt für den Knoten die Form des Knotens an (Rechteck=1, Kreis=2, Dreieck=3, Dreieck 'auf dem Kopf'=4, Diamond=5, kleines Rechteck=6); für die Kante bestimmt es die Dicke der Kante.

Jede Datei muss mindestens eine Zeile beginnend mit VERTICES (NODES) besitzen. Die Zeilen beginnend mit ARCS oder EDGES können fehlen. Wir haben es dann mit einem kantenlosen Graphen zu tun.

4. Knotenzeilen werden durch eine Zeile beginnend mit VERTICES (NODES) eingeleitet. Sie bestehen aus Zeichenketten (Token), die voneinander durch einen oder mehrere Zwischenräume oder Tabulatoren getrennt sind. Ein Token definiert ein Datum des Knotens, entsprechend der durch VERTICES (NODES) eingeleiteten Zeile. Jede Knotenzeile muss durch einen Kurznamen - das erste Token auf der Zeile - des Knotens (höchstens 3 Zeichen) eingeleitet werden. Die Kurznamen für verschiedene Knoten müssen alle unterschiedlich sein, da sie den Knoten identifizieren. Weitere Tokens können folgen entsprechend der in der vorausgehenden VERTICES (NODES) Zeile definierten Einträgen.
Besteht die VERTICES (NODES) Zeile aus

```
VERTICES REAL INT
```

so sollte jede nachfolgende Knotenzeile aus drei Tokens bestehen: aus dem Kurznamen, einer reelen Zahl und einer ganzen Zahl. (z.B.)

```
ZU   34.56   67
TZ   45.67   100
```

Fehlende oder fehlerhafte Einträge werden als 0 (für INT oder REAL) oder als leere Zeichenkette (für NAME) interpretiert. Zuviele Einträge werden ignoriert. Dies erlaubt auf jeder Zeile zusätzlich Kommentar anzubringen.

5. Kantenzeilen werden durch eine Zeile beginnend mit EDGES oder ARCS eingeleitet. Jede Kantenzeile muss mit zwei Kurznamen beginnen und entsprechend der Definition in der vorhergehenden EDGES oder ARCS Zeile mit Tokens gefüllt werden. Die beiden Kurznamen identifizieren die benachbarten Knoten einer Kante.

Die Kurznamen müssen also legale Knotenidendifikatoren sein. Ansonsten gelten hier dieselben Regeln wie für Knotenzeilen.

```
ARCS      INT  NAME         SHAPE
  ZU TZ    34  KanteZU-TZ      1
  TZ IK   234  KanteTZ-IK      1
```

6. Es ist nicht notwendig, dass alle Knotennamen *vor* allen Kanten eingeben werden. Jedoch muss jeder Knotenkurzname in einer Kantenzeile bereits vorher als Knoten defineirt worden sein. Das bedeutet, dass die Zeilen mit beginnend mit VERTICES (NODES), EDGES oder ARCS mehrmals wiederholt werden dürfen.

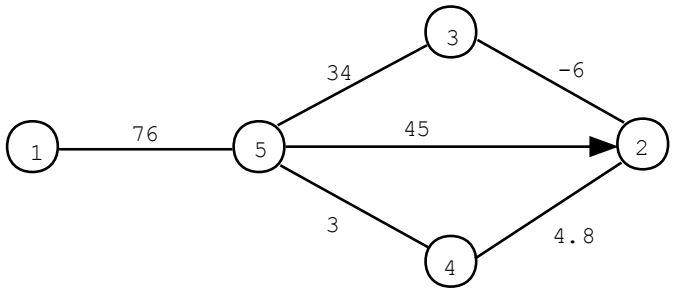

Abbildung 5.1: Der Graph

```
*** Graph der Abbildung 1

VERTICES INTEGER
  1  20
  2  23
  3  45
  4  12
  5  54
EDGES  REAL
  1 5  76
  2 3  -6
  2 4  4.8
  5 3  34
  4 5  3
ARCS   REAL
  5 2  45
** Ende der Datei
```

Abbildung 5.2: Dateiformat

Beispiel: Der Graph in Abbildung 5.1 widerspigelt die Datenstruktur der Datei in Abbildung 5.2. Er besitzt fünf Knoten mit den Kurznamen '1', '2', '3', '4' und '5'. Alle Kanten sind ungerichtet (edges) ausser die Kante zwischen dem Knoten '5' und '2'. Die Datei besteht aus zwei Kommentarzeilen und einer Leerzeile, welche für die interne Datenstruktur des Graphen ohne Belang sind. Die Knotendaten werden durch die Zeile VERTICES INT eingeleitet. Jeder Knoten besteht aus seinem Kurznamen und einer ganzen Zahl. Dies sind die Knotenattribute. Die Definition der ungerichteten Kanten werden durch die Zeile EDGES REAL eingeleitet. Jede Kante hat somit drei Attribute: die beiden Endpunkte der Kante und eine reel-wertige Zahl.

Abgeschlossen wird die Definition des Graphen mit einem Bogen (arc) und einer weiteren Kommentarzeile.

Das vorgeschlagene Format hat verschiedene Vorteile.

- Es ist universell und erweiterbar.
- Es ist in seiner Länge nicht beschränkt (beliebig grosse Netze und Graphen).
- Es ist flexibel: Nichts muss sortiert sein und die Reihenfolge ist beliebig.
- Es ist erweiterbar: Jede Knoten- und Kantenzeile kann aus einer variablen Anzahl Attributen bestehen.
- Es ist einfach: Es kann durch jedes Textprogramm manipuliert werden.

Dem stehten zwei relativ leicht wiegende Nachteile gegenüber:

- Es braucht relativ viel Speicherplatz. Dies sollte aber auf sekundären Speichermedien nie ein ausschlaggebendes Argument sein.
- Der Zugriff ist streng sequentiell und relativ langsam, da die Daten in das entsprechende interne Format umgewandelt werden müssen. Da die Lese- und Schreibarbeit jedoch einmalige Prozesse sind, sollte dies nicht schwer wiegen.

Dass das Format universell ist, kann man leicht einsehen: Jeder beliebige Graph kann damit gebildet werden. Das Format kann zudem in verschiedener Hinsicht erweitert werden. Erstens könnten weitere vordefinierte Datentypen, wie BOOLEAN, VISIBLE, PATTERN, STRING, eingeführt werden. Dies würde für einen Knoten oder eine Kante definieren, ob eine Eigenschaft zutrifft oder nicht, ob diese(r) sichtbar oder unsichtbar ist oder in welchem Pattern er (sie) gezeichnet werden soll. Mit STRING könnte weitere Zeichenzettenattribute definiert werden.
Damit ist es auch denkbar, dass wir verschiedene Knoten- oder Kantentypen haben, d.h. Knoten oder Kanten mit verschiedenen Attributen. Die Knoten und Kanten können dann in verschiedene Untermengen aufgeteilt werden. Damit könnten verschiedene Untergraphen definiert werden. Dies wäre vor allem auch interessant in Problemstellungen, in denen wir nicht nur *einen* abstrakten Graphen untersuchen, sondern einen Graphen in eine Umgebung einbetten wollen (z.B. Strassennetz, Eisenbahnnetz und politische Grenzen als drei Graphen). Die Anzahl der Attribute und deren Definitionen könnte also für verschiedene Teilgraphen unterschiedlich sein. Ganz allgemein könnte so Graphen in Teilgraphen verfallen und als separate 'Graphen-Module' behandelt werden.

Aus dem beschriebenen Format ist es zudem sehr einfach das sogenannte SHARE

Format (Klingman 1973) oder das MPS-Format für Transportmodelle zu generieren, welche standartisierte Inputformate für die meisten Transport-Lösungsalgorithmen bilden. Auch die Modelliersprache LPL (Hürlimann 1992) kann diese Formate über eine einfache READ und PRINT Anweisung lesen und schreiben. Damit ist der Bezug zum Modellierer und seiner Umgebung hergestellt.

REFERENCES

HÜRLIMANN T. [1992], Reference Manual for the LPL Modeling Language, Version 3.8, Institute for Automation and Operations Research, Working Paper No. 191, September 1991, revised February 1992, Fribourg.

KLINGMAN D., STUTZ J., BARR R., GLOVER F., [1973], NETGEN, A Program for Generating Large Scale (Un)capacitated Assignment, Transportation, and Minimum Cost Flow Network Problems, Technical Documentation NETGEN, February 1973.