Diplomarbeit:

Ein MacMahon-Losungsprogramm für Go-Turniere unter Benutzung von Maximum Weight Perfect Matching

Universität Hildesheim
Institut für Informatik
Samelsonplatz 1
D-31141 Hildesheim

Vorgelegt von: Christoph Gerlach
Alleestr. 35
30167 Hannover

Zeitraum: 11.11.1993 - 11.5.1994

Gutachter: Prof. Dr. Eike Best

Zweitgutachter: Prof. Dr. Joachim Biskup

Inhaltsverzeichnis

1 Einleitung

1.1 Ziel dieser Arbeit

Ziel dieser Arbeit ist es, eine effiziente Lösung des Problems einer MacMahon-Losung auf Go-Turnieren unter Benutzung von Maximum Weight Perfect Matching anzugeben und diese ergänzt um ein User-Interface zur Durchführung und Auswertung von Go-Turnieren bereitzustellen.

Das User-Interface beinhaltet die Verwaltung der teilnehmenden Spieler, die Eingabe von Spielergebnissen, Manipulationsmöglichkeiten für die Turnierleitung und Funktionen zur Ausgabe von Tabellen und Paarungen auf einen Drucker.

Für das MacMahon-System existieren keine offiziellen Vorgaben von nationalen oder internationalen Go-Verbänden. Die Beschreibung des MacMahon-Systems in dieser Arbeit basiert auf dem auf europäischen Turnieren benutzten Standard.

1.2. Aufbau dieser Arbeit

Diese Arbeit besteht aus einer Einleitung, die eine Motivation für die Entwicklung des MacMahon-Systems liefert, einer Spezifikation des MacMahon-Systems (Kapitel 2), der Beschreibung eines Losungsprogramms für Go-Turniere (Kapitel 3) und der Beschreibung und der Implementierung einer Variante des Maximum-Weight-Perfect-Matching-Algorithmus, der auf die Pionierarbeit von J. Edmonds Ende der 60er Jahre zurückgeht (Kapitel 4). Die Beschreibung und die Implementierung orientiert sich wesentlich an [Gib85]. Weitere Teile stammen aus [Chr75] und [PS82].

In Kapitel 5 wird ein Ausblick auf Bereiche gegeben, die in dieser Diplomarbeit nur unzureichend berücksichtigt werde konnten.

1.3 Ziel von Turnieren

Auf Turnieren für Brettspiele wie Schach und Go werden im wesentlichen zwei Ziele verfolgt:

Turnierziel 1: Der stärkste teilnehmende Spieler soll ermittelt werden.

Turnierziel 2: Für alle Spieler soll erkennbar sein, wie sich ihre Spielstärke relativ zu den Spielstärken der anderen teilnehmenden Spieler verhält. D.h. es soll eine Rangliste alle Spieler ermittelt werden.

Zu Beginn dieses Jahrhunderts wurden Turniere entweder nach dem *K.O.-System*[1] oder *„jeder gegen jeden“* gespielt. Beide Systeme stoßen bei der Umsetzung oben genannter Ziele unter realistischen Bedingungen jedoch schnell an ihre Grenzen:

„Jeder gegen jeden“-Turniere liefern für beide Ziele gute Ergebnisse, sind aber nicht mehr einsetzbar, wenn die Teilnehmerzahl größer wird, da zu viele Runden gespielt werden müßten (für $n \geq 2$ und $2n-1$ oder $2n$ Spieler braucht man $2n-1$ Runden).

K.O.-Turniere erlauben es zwar, einen Sieger aus einer großen Zahl von Teilnehmern bei begrenzter Rundenzahl zu ermitteln (Ziel 1), Ziel 2 wird aber nicht erreicht. So könnte man z.B. zwischen allen Spielern, die in der ersten Runde verlieren, nicht differenzieren. Ein weiterer Nachteil von K.O.-Turnieren ist, daß die meisten Teilnehmer nur sehr wenige Runden mitspielen, wodurch das Turnier für schwächere Spieler nicht attraktiv ist, da sie wahrscheinlich in der ersten Runde ausscheiden würden.

[1] Beim K.O.-System werden alle teilnehmenden Spieler zufällig oder nach einer Setzliste gegeneinander gepaart. Die Verlierer der Runde scheiden aus, die Gewinner werden in der nächsten Runde entsprechend gegeneinander gepaart. Dies wir fortgesetzt, bis nur noch ein Spieler ohne Spielverlust verbleibt. Dieser Spieler hat das Turnier gewonnen.

1.4 Das Schweizer System

Die beiden Nachteile des K.O.-Systems lassen sich auflösen, wenn man es verallgemeinert. Dr. J. Muller führte 1895 ein modifiziertes K.O.-System auf einem Züricher Schach-Turnier ein, das die Grundlage für das Schweizer System, wie wir es heute kennen, bildete. Die Grundidee läßt sich durch die folgenden drei Regeln beschreiben:

SSG1: Zwei Teilnehmer werden höchstens einmal gegeneinander gepaart.

SSG2: In jeder Runde wird jeder Spieler gegen einen Spieler gepaart, der bis zu diesem Zeitpunkt die gleiche Anzahl an Siegen hat (wenn kein solcher Spieler existiert, dann gegen einen Spieler, der eine möglichst gering abweichende Anzahl von Siegen hat).

SSG3: Nach einer vorher festgelegten Zahl von Runden gewinnt der führende Spieler.

Es läßt sich leicht sehen, daß das Schweizer System eine Verallgemeinerung des K.O.-Systems ist:

- Sei n die Anzahl der teilnehmenden Spieler an dem Turnier. Zur Ermittlung eines eindeutigen Siegers brauchte man bei Verwendung des K.O.-Systems $\lceil \log_2 n \rceil$ Runden.
- Bei der Verwendung des Schweizer Systems legt man dieselbe Rundenzahl fest. Da im Schweizer System immer Spieler mit der gleichen Anzahl an Siegen gegeneinander gepaart werden, entsprechen die Paarungen in der jeweils führenden Gruppe beim Schweizer System denjenigen im K.O.-System. Wie beim K.O.-System gewinnt beim Schweizer System schließlich der Spieler, der als einziger kein Spiel verloren hat.

Der Turniersieger steht bei der Verwendung des Schweizer Systems gleichermaßen fest, wie bei der Verwendung des K.O.-Systems, aber alle Spieler können $\lceil \log_2 n \rceil$ Runden mitspielen. Außerdem hat ein starker Spieler, der in einer der ersten Runden verliert, die Chance, sich durch weitere Siege wieder weiter nach oben zu schieben. Die Plazierungen im Schweizer System werden nach der Anzahl der Siege nach allen gespielten Runden vergeben.

Für die Losungen in den Runden (wie werden Spieler innerhalb einer Gruppe, d.h. Spieler mit derselben Anzahl von Siegen, gegeneinander gesetzt) und für Kriterien für die Plazierung der Spieler nach der letzten Runde existieren zahlreiche Verfahren, die auf bestimmte Gegebenheiten des Turniers und seiner Teilnehmer Rücksicht nehmen. Einige Anpassungen können sein, daß es in Schach-Turnieren ein Vorteil ist, mit Weiß zu spielen (Anzugsvorteil). Aus Gründen der Chancengleichheit sollte ein Losungsalgorithmus allen Spielern möglichst gleich oft die schwarzen und weißen Spielsteine zusprechen. Dafür muß auch der Gegner durch den Losungsalgorithmus gezielt ausgesucht werden. Die besonderen Ziele, auf die ein Losungsalgorithmus für Go-Turniere eingehen können sollte, werden in einem späteren Abschnitt erläutert.

1.5 Das MacMahon-System (Motivation)

Schach-Turniere werden fast ausschließlich nach dem Schweizer System gespielt. Dabei treffen Spieler mit unterschiedlichster Spielstärke aufeinander. Aufgrund der relativ (zum Go) breiten Remis-Spannbreite kann ein schwächerer Spieler häufig auf ein Remis gegen einen stärkeren Gegner hoffen. Ist der Spielstärkeabstand zu groß, so hat der schwächere Spieler aber fast automatisch verloren.

Beim Go wird der Anzugsvorteil durch sogenannte Komi[1] ausgeglichen. In der Regel werden 5½ Komi bei Go-Partien verwendet, wodurch ein Remis ausgeschlossen ist. Selbst bei einer ganzzahligen Anzahl der Komi ist ein Remis im Go sehr unwahrscheinlich. Beim Aufeinandertreffen unterschiedlich starker Spieler gewinnt fast immer der stärkere Spieler.

[1] Nach einer Go-Partie hat derjenige Spieler gewonnen, der mehr freie Schnittpunkte mit eigenen Steinen umschlossen hat. Die Punktzahl des anziehenden Spielers wird dabei um die Anzahl der Komi vermindert (Ausgleich des Anzugsvorteils).

Mit zunehmender Spielstärkedifferenz ist der schwächere Spieler praktisch chancenlos. In freundschaftlichen Partien wird die Spielstärkedifferenz durch die Verwendung von Vorgaben (Handicap) ausgeglichen, so daß die Partie für beide Seiten interessant bleibt. Auf Turnieren wird im allgemeinen keine Vorgabe gegeben.

Bei der Verwendung des Schweizer Systems würden in den ersten Runden viele Partien zwischen Spielern mit stark unterschiedlicher Spielstärke stattfinden, die aufgrund der Vorhersagbarkeit des Ergebnisses (und damit ihrer Redundanz) zur Ermittlung einer Rangliste der Spieler nicht herangezogen werden können. Durch gezieltes Paaren von Spielern mit annähernd gleicher Spielstärke ließe sich wesentlich effizienter (d.h. durch weniger Runden) eine Rangliste ermitteln. Dies ist die Grundidee des MacMahon-Systems.

Vor der ersten Runde eines MacMahon-Turniers werden die Spieler nach ihrer Spielstärke sortiert in Gruppen eingeteilt. Die Spieler innerhalb dieser Gruppen werden als gleich stark betrachtet. Beginnend mit dieser Gruppeneinteilung führt man das Turnier nach dem Schweizer System durch. Dabei erhält die Gruppe mit den schwächsten Spielern keinen Sieg gutgeschrieben, die nächststärkere Gruppe einen Sieg, die wiederum nächststärkere zwei Siege und so weiter. Nach dieser Initialisierung kann das Schweizer System normal angewendet werden. MacMahon ist offensichtlich eine Verallgemeinerung des Schweizer Systems.

Die Vorteile des MacMahon-Systems gegenüber dem Schweizer System sind:

- Die spielstärksten Spieler werden in allen Runden untereinander gepaart. Das erhöht die Aussagekraft der Ergebnisse in der Spitzengruppe.
- Schwächere Spieler spielen im allgemeinen nur gegen Spieler, gegen die sie eine reale Gewinnchance haben.

Die Nachteile des MacMahon-Systems gegenüber dem Schweizer System sind:

- Bereits vor Beginn der ersten Runde wird die Gruppe der möglichen Gewinner des Turniers auf eine relativ kleine Gruppe von Spielern eingeschränkt (häufig haben nur Spieler aus den beiden obersten Gruppen eine Chance, das Turnier zu gewinnen).
- Die angegebene Spielstärke der teilnehmenden Spieler beeinflußt den Turnierverlauf erheblich und sollte daher möglichst korrekt sein.

1.6 Das Einstufungssystem im Go

Alle Go-Spieler sind gemäß ihrer Spielstärke weltweit einheitlich in Klassen eingeteilt. Die Klassen sind aufgeteilt in Schülergrade (japanisch *Kyu*) und Meistergrade (japanisch *Dan*). Anfänger erhalten pauschal eine Spielstärke von 20 Kyu. Stärkere Spieler erhalten die Klassen 19 Kyu, 18 Kyu,... bis 1 Kyu. Darüber stehen die Meisterklassen 1 Dan bis 6 Dan, wobei die Klasse der 6-Dans die stärksten Amateur-Spieler enthält. In Japan, China und Korea gibt es neben dem Einstufungssystem für Amateur-Spieler noch eine Klassifizierung für Profi-Spieler von Pro-1-Dan bis Pro-9-Dan. Pro-1-Dan-Spieler sind im allgemeinen stärker als 6-Dan-Amateure.

Bei Amateur-Spielern gilt, daß eine Vorgabe exakt in der Höhe ihrer Klassendifferenz eine ausgeglichene Gewinnerwartung des stärkeren und des schwächeren Spielers ergibt. Ein 2 Dan würde einem 4 Kyu daher 5 Steine Vorgabe geben.

Ohne Vorgabe ist der schwächere Spieler bereits ab einer Differenz von ca. 3 Klassen praktisch chancenlos.

In Deutschland kontrollieren die Spieler ihre Einstufung selbst. Erzielen sie auf Turnieren gute Resultate gegen stärkere Spieler, so korrigieren sie ihre Einstufung entsprechend.

2 Das MacMahon-System

2.1 Regeln für die Durchführung von Turnieren nach dem MacMahon-System

Als Grundlage für die Herleitung einer Gewichtsfunktion als Eingabe für den Maximum-Weight-Perfect-Matching-Algorithmus, in die das MacMahon-System kodiert wird, wird eine (informelle) Regelsammlung zur Durchführung von MacMahon-Turnieren angegeben. Einige dieser Regeln konkurrieren miteinander, andere haben absoluten Vorrang. Dies wird sich auch in der Konstruktion der Gewichtsfunktion widerspiegeln. Orientiert ist diese Regelzusammenstellung an demAbschnitt „A rule set for the Swiss System“ in [Die93], in dem eine vergleichbare Regelmenge für den Spezialfall des MacMahon-Systems, das Schweizer System (allerdings auf Schachturniere ausgerichtet) angegeben wird. In einigen hier aufgestellten Regeln wird auf spezielle Bedürfnisse von Go-Turnieren eingegangen (z.B. Vorgabe).

Begründungen für die Existenz einzelner Regeln werden im nächsten Abschnitt geliefert.

0. *Vorraussetzungen/Vorbereitungen*. Allen teilnehmenden Spieler ist eine Spielstärke (üblicherweise 20 Kyu bis 6 Dan) zugeordnet, und die Zahl der Runden des Turniers wird vor Beginn der ersten Runde festgelegt.

   (a) Jedem Spieler wird entsprechend seiner Spielstärke ein Start-MacMahonScore (SMMS) zugeordnet. Alle Spieler mit der Spielstärke 20 Kyu erhalten den SMMS 0, die Spieler mit der Spielstärke 19 Kyu den SMMS 1 und so weiter. Spieler mit einer Spielstärke von 1 Dan erhalten demnach den SMMS 20, 6-Dan-Spieler den SMMS 25.

(b) Als nächstes werden sogenannte MacMahon-Bars festgelegt, die am unteren und oberen Ende des Feldes MacMahon-Gruppen zu einer neuen Bar-Gruppe zusammenfassen, die die in ihr zusammengefaßten Gruppen ersetzt. Umfasse der Bottom-Bar die Gruppen 0 bis *i*. Dann erhalten alle Spieler dieser Gruppen einen Start-MacMahonScore von 0 und der Start-MacMahonScore aller anderen Spieler wird um *i* vermindert. Umfasse der Top-Bar die Gruppen *t* bis *t-j*. Dann erhalten alle Spieler dieser Gruppen den Start-MacMahonScore *t-j*.

Der obere MacMahon-Bar sollte so gewählt werden, daß die Anzahl seiner Spieler in dem in Tabelle 2.1 in Abhängigkeit von der Zahl der Runden angegebenen Bereich bleibt. Auf einen unteren MacMahon-Bar sollte verzichtet werden.

1. *Freilose*. Wenn die Zahl der teilnehmenden Spieler in einer Runde ungerade ist, erhält ein Spieler ein Freilos. Kein Spieler sollte im Verlaufe des Turnieres zweimal ein Freilos erhalten. Für das Freilos wird derjenige Spieler ausgewählt, der den niedrigsten MacMahonScore hat. Unter mehreren Spielern wird derjenige ausgewählt, der bisher die stärksten Gegner hatte.

2. *Scoring*. Der MacMahonScore der Spieler verändert sich um einen Punkt für einen Sieg oder ein Freilos, um einen halben für ein jigo (unentschieden im Go) oder wenn der Spieler in der Runde aussetzt oder um 0, wenn er verliert. Es gibt also im Verlauf des Turniers auch MacMahon-Gruppen mit einem nicht-ganzzahligen MacMahon-Score. Spieler können beliebig Runden aussetzen.

Runden	3	4	5	6	7	8	9	10
Spieler	4-7	5-9	6-13	7-17	8-21	9-25	10-29	11-33

Tabelle 2.1: Die Wahl des oberen MacMahon-Bars

3. *Die grundlegenden Regeln des Schweizer Systems*. Die Paarung jeder Runde muß den Grundregeln des Schweizer Systems genügen:

   SSG1: Zwei Teilnehmer werden höchstens einmal gegeneinander gepaart.

   SSG2: In jeder Runde wird jeder Spieler gegen einen Spieler gepaart, der zu diesem Zeitpunkt den gleichen MacMahonScore hat (wenn kein solcher Spieler existiert, dann gegen einen Spieler, der einen möglichst gering abweichenden MacMahonScore hat).

4. *Paarung innerhalb einer MacMahon-Gruppe*. Innerhalb einer MacMahon-Gruppe werden die Spieler nach konkurrierenden Zielen gegeneinander gepaart. (a) bekommt das höchste Gewicht, (b) das zweithöchste, und die Ziele (d) bis (f) sind ungefähr gleich zu gewichten.

   (a) *Auswahl von Spielern, die hoch- oder heruntergelost*[1] *werden*. Wenn es nicht möglich ist, alle Spieler der Gruppe untereinander zu losen, ohne die Regel SSG1 zu verletzen, oder weil die Gruppe eine ungerade Zahl an Spielern enthält, so werden ggf. die Spieler hochgelost, die in der Gruppe als schwächste Spieler angesehen werden (in Runde 1 erfolgt die Auswahl anhand einer Rating-Liste, vergleichbar dem Elo-System im Schach, in späteren Runden liefern die Zweitkriterien (siehe 6.(b)) ein objektives Kriterium für die Auswahl der schwächsten Spieler der Gruppe). Ggf. werden die Spieler heruntergelost, die in der Gruppe als stärkste Spieler angesehen werden.

   (b) *Setzen innerhalb von Gruppen*. Die nach (a) verbleibenden Spieler werden gemäß einer Setzliste sortiert. Möglichst soll dann die obere Hälfte gegen die untere Hälfte der Spieler in umgekehrter Reihenfolge gepaart werden. In einer Gruppe mit 20 Spielern also Nr.1 gegen Nr.20, Nr.2 gegen Nr. 19 und so weiter. Die Spieler werden in der ersten Runde nach ihrem Rating (gemeint ist ein dem Elo-System verwandtes Rating und nicht die Spielstärke in dem *groben* Raster 20 Kyu bis 6 Dan) gesetzt. In späteren Runden werden

[1] Wenn ein Spieler gegen einen anderen Spieler gepaart wird, der einen höheren MacMahonScore hat, so spricht man davon, daß der Spieler hochgelost wurde. Hat der andere Spieler einen geringeren MacMahonScore, so wird der Spieler heruntergelost.

die Zweitkriterien (siehe 6.(b)) der Plazierung als Kriterium für die (vermutete) Spielstärke der Spieler herangezogen.

Die Regel (b) kann in allen Gruppen außer der Top-Gruppe ausgesetzt werden.

(c) *Top-Gruppe*. In der Gruppe der Spieler, die um den Turniersieg spielen, sollen die folgenden Ziele (d) bis (f) nicht berücksichtigt werden.

(d) *Farbwahl*. Es werden bevorzugt Spieler gegeneinander gepaart, deren Farbbilanzen dadurch ausgeglichener gestaltet werden können. Die Farben werden so vergeben, daß der Farbausgleich für die beiden Spieler einer Paarung maximiert wird. Würde die Partie zwischen den beiden Spielern mit Vorgabe gespielt, wird dieses Ziel ignoriert.

(e) *Berücksichtigung gleicher Länder und Klubs*. Spieler aus gleichen Ländern oder gleichen Klubs sollen bevorzugt nicht gegeneinander spielen.

(f) *Zu große Spielstärkedifferenzen*. Spieler, deren Spielstärke sich um mehr als eine Toleranzgrenze unterscheidet, sollen bevorzugt nicht gegeneinander gepaart werden. Die Toleranzgrenze wird von der Turnierleitung festgelegt, sollte aber Unterschiede bis zwei Grade in der Spielstärke tolerieren.

5. *Vorgaben*. Außerhalb der Spitzengruppe wird in allen Partien eine Vorgabe in Höhe der MacMahonScore-Differenz der beiden Spieler vermindert um zwei gegeben.

6. *Wertung der Ergebnisse*. Nachdem alle Ergebnisse der Paarungen einer Runde vorliegen, kann eine aktuelle Tabelle (Rangliste) aller Spieler erstellt werden. Nach folgenden Kriterien wird über die Rangfolge entschieden:

   (a) MacMahonScore.

   (b) SOS (Sum of Opponent's Scores, Gegnerpunkte, im Schach „Buchholzwertung“ genannt)

$$\mathrm{SOS}(\textit{Spieler}\ 1) = \sum_{\textit{Spieler}_i \text{ war Gegner von } \textit{Spieler}\ 1 \text{ in irgendeiner Runde}} \mathrm{MacMahonScore}(\textit{Spieler}_i)$$

Ein Freilos gibt dabei max{MacMahonScore(*Spieler* 1)-1, 0} Gegnerpunkte.

Der MacMahonScore der Gegner wird um die in dieser Partie gegebene Vorgabe erhöht, wenn der Spieler Weiß hatte, bzw. verringert, wenn der Spieler Schwarz hatte.

(c) SOSOS (Sum of Opponent's SOS, im Schach „verfeinerte Buchholzwertung“ genannt)

$$\mathrm{SOSOS}(\textit{Spieler}\ 1) = \sum_{\textit{Spieler}_i \text{ war Gegner von } \textit{Spieler}\ 1 \text{ in irgendeiner Runde}} \mathrm{SOS}(\textit{Spieler}_i)$$

Ein Freilos gibt dabei SOS(*Spieler* 1) Punkte.

(d) Alle Spieler mit gleichen Kriterien (a) bis (c) teilen sich ihren Rang.

2.2 Begründung, Kommentar und Modifikationen zu den Regeln

2.2.1 Festlegung der MacMahon-Bars

Regel 0.(b) gibt an, wie die MacMahon-Bars festgelegt werden sollen. Der Bottom-Bar wird in dieser Spezifikation lediglich aus Kompatibilitätsgründen zu anderen Spezifikationen des MacMahon-Systems eingeführt. Andere Spezifikationen benutzen einen Bottom-Bar, um die Verstöße gegen Regel 3.SSG2 am unteren Ende des Teilnehmerfeldes gering zu halten. Dies ist aber überflüssig, da dieselbe Regel dazu führt, daß notwendige Differenzen zwischen MacMahon-Gruppen möglichst gering gehalten werden. Das Vereinigen von MacMahon-Gruppen am unteren Rand des Teilnehmerfeldes führt häufig

dazu, daß absolute Anfänger (20 Kyu) bereits in den ersten Runden des Turniers gegen wesentlich stärkere Spieler gepaart werden, obwohl andere Spieler in ihrer Spielstärke zur Verfügung stehen. In gewisser Weise wird ein Etikettenschwindel betrieben, indem Spieler mit unterschiedlicher Spielstärke am unteren Ende des Feldes in einer Gruppe zusammengefaßt werden.

Ein MacMahon-Bar am oberen Ende hingegen ist notwendig. Folgende Überlegungen gehen in die Festlegung des oberen Bars ein, der alle Spieler ab einer bestimmten Spielstärke als gleich stark definiert:

a) Die Zahl der Spieler im oberen Bar darf 2^k nicht überschreiten, damit es einen eindeutigen Gewinner geben kann (k ist die Anzahl der Runden).

b) In der Regel haben nicht nur die spielstärksten anwesenden Spieler eine realistische Chance, einen der vorderen Plätze in der abschließenden Rangliste zu belegen, sondern auch solche mit einer um zwei bis drei Spielgrade schwächeren Einstufung. Würden diese Spieler zu Beginn des Turniers in niedrigeren MacMahon-Gruppen starten, wären diese dadurch benachteiligt.

c) Die Zahl der Spieler in dem MacMahon-Bar darf nicht zu groß werden, da es sonst vorkommen kann, daß die Spieler auf den ersten fünf Plätzen kaum untereinander gespielt haben und die Reihenfolge dieser Spieler mehr oder weniger durch das Losungsglück in den gespielten Runden festgelegt wird.

Aus diesen drei Überlegungen und aufgrund der Erfahrungen, die auf Turnieren gesammelt wurden, ergeben sich die Werte in Tabelle 2.1.

Überlegung b) ist besonders wichtig, da das verwendete Turniersystem möglichst alle Spieler fair behandeln sollte. Wenn Spieler eine realistische Chance haben, sich auf den vorderen Plätzen zu plazieren, sollte ihnen diese nicht durch die (willkürliche) Festlegung des MacMahon-Bars genommen werden. Turnierresultate zeigen, daß auch Spieler, die zwei Spielgrade schwächer spielen als die anwesenden Spitzenspieler, eine realistische Chance haben. Das liegt zu einem Gutteil daran, daß die Einstufungen in Europa nicht einheitlich sind. So kommt es vor, daß sich

zwei 4-Dan-Spieler in ihrer tatsächlichen Spielstärke um 3-4 Grade unterscheiden.

Wichtig sind aber auch die Überlegungen a) und c), die beide auf eine geringere Zahl von Spielern in der MacMahon-Gruppe abzielen. Gegenwärtig wird dieser Zielkonflikt durch den Turnierleiter entschieden. Die „European Go-Federation“ (EGF) bereitet aber die Einführung einer Rating-Liste vor, so daß dann ein objektives Kriterium dafür gegeben wird, welche Spieler (durch ihre absolute Spielstärke) das Recht haben, in der Spitzengruppe um den Turniersieg mitzuspielen. Der MacMahon-Bar wird dann gemäß Überlegung b) festgelegt, und darüber wird eine zusätzliche sogenannte Super-MacMahon-Bar-Gruppe geschaffen, in die eine geeignete Anzahl der (absolut) spielstärksten anwesenden Spieler aus der normalen MacMahon-Bar-Gruppe verschoben werden. Für die Teilnehmerzahl dieser Super-MacMahon-Bar-Gruppe gilt auch Tabelle 2.1. Voraussetzung für den Einsatz eines Super-MacMahon-Bars ist aber

Pl.	Name	Str	Nat	MMS	1	2	3	4	5	Pt	SOS	SOSOS
1	Dickhut, Franz-Jose	6d	BO	26	6+	13+	3+	2-	5+	4	126	622
2	Nechanicky, Radek	6d	NYM	26	8+	11+	7+	1+	4-	4	126	615½
3	Wolff, Martin	3d	HH	26	12+	5+	1-	15+	6+	4	124	620
4	Park, Sang-Nam	6d	B	26	20-	21+	11+	7+	2+	4	121	616½
5	Kretschmann, Michae	3d	GT	25	10+	3-	8+	18+	1-	3	126	604
6	Kroll, Harald	3d	BO	25	1-	16+	12+	14+	3-	3	124	614
7	Schuster, Malte	5d	B	25	14+	20+	2-	4-	15+	3	123	614
8	Ehlers, Georg	3d	B	25	2-	22+	5-	16+	12+	3	122	613
9	Nohr, Thomas	2d	HH	25	23-	19+	21+	20+	13+	4	117	599
10	Digulla, Jörg	2d	BO	25	5-	26+	41+	11+	14+	4	117	593½
11	Bergmann, Martin	3d	ER	24	17+	2-	4-	10-	29+	2	123½	588
12	Stoll, Rüdiger	3d	L	24	3-	17+	6-	24+	8-	2	123	605
13	Meyenschein, Marco	3d	HB	24	15+	1-	14-	23+	9-	2	122	604
14	Splettstößer, Peter	3d	HH	24	7-	25+	13+	6-	10-	2	122	600
15	Klenke, Achim	3d	GÖ	24	13-	23+	20+	3-	7-	2	121	611
16	Nijhuis, Caspar	2d	ASD	24	24+	6-	22+	8-	20+	3	119	606
17	Krekel, Holger	3d	H	24	11-	12-	18-	25+	22+	2	118	594½
18	Rhotert, Michael	1d	H	24	42+	30+	17+	5-	21+	4	115	590
19	Wu, Naixin	2d	H	24	22-	9-	37+	30+	24+	3	115	576
20	Nechanicky, Vitezsl	4d	NYM	23	4+	7-	15-	9-	16-	1	124	601
21	Meyer, Friedhelm	2d	F	23	30+	4-	9-	27+	18-	2	120	580
22	Jasiek, Robert	2d	B	23	19+	8-	16-	31+	17-	2	119	587
23	Steffens, Siegmar	2d	DD	23	9+	15-	24-	13-	32+	2	118	588
24	Tsche, Young-Il	2d	H	23	16-	31+	23+	12-	19-	2	117	588
25	Probst, Burghard	1d	H	23	28+	14-	46+	17-	30+	3	114	568½
26	Nguyen-Huu, Tin	1d	H	23	33+	10-	27-	51+	36+	3	112	555
27	Petzold, Jan	1k	DD	23	45+	33+	26+	21-	34+	4	111	556½
28	Tautorat, Guido	1k	J	23	25-	43+	51+	35+	31+	4	108	551
29	Stüwe, Aglef	1d	H	22½	–	–	–	39+	11-	1	106	528
30	Schmidt, Uwe	2d	ER	22	21-	18-	35+	19-	25-	1	116	573
31	Schlüter, Gilbert	1d	H	22	34+	24-	32+	22-	28-	2	113	565

Abbildung 2.2:
Ausschnitt aus der Endtabelle des Messeturniers Hannover 1994

eine Rating-Liste, die als objektives Kriterium Benachteiligungen von Spielern gering hält.

Nicht immer läßt sich der MacMahon-Bar so festlegen, daß die Ergebnisse in der Spitzengruppe am Ende des Turniers befriedigend sind. Am 30.4/ 1.5.1994 wurde das Messeturnier Hannover mit dem hier vorgestellten Programm durchgeführt. Der MacMahon-Bar wurde auf 3 Dan festgesetzt. Das führte dazu, daß bei insgesamt 15 Spielern in der Bar-Gruppe die letztendlich vier bestplazierten Spieler (darunter ein 3 Dan) alle mit einem MacMahon-Score von 26 endeten, aber nur mit einer Quote von ca. 50% untereinander gespielt hatten. Die Reihenfolge wurde ausschließlich durch die Hilfskriterien SOS und SOSOS gegeben. Ein MacMahon-Bar von 4 Dan wäre allerdings eine noch schlechtere Variante gewesen, da dann nur 5 Spieler in der Bar-Gruppe gespielt hätten.

2.2.2 Scoring

Eine andere MacMahon-Spezifikation bewertet das Aussetzen von Spielern in einer Runde leicht anders, indem für jede ausgesetzte Runde nicht ein halber MacMahon-Punkt gutgeschrieben wird, sondern in jeder zweiten ein ganzer MacMahon-Punkt und in der davor keiner.

In dieser Spezifikation wird sofort ein halber Punkt für jede ausgesetzte Runde vergeben, weil dies genau dem Erwartungswert jedes Spielers in jeder Runde entspricht. Die mittlere quadratische Fehlerabweichung zwischen dem geführten Wert und der tatsächlichen Spielstärke des Spielers wird so minimiert. Ein weiteres Argument ist, daß das Aussetzen eines Spielers nicht seinen vorherigen Gegnern einen Nachteil bringen darf, indem der Spieler mit einem halben MacMahon-Punkt unterbewertet wird, was die Gegnerpunkte seiner vorherigen Gegner beeinflussen würde.

Die MacMahon-Scores nach jeder Runde sind praktisch aktuelle (geänderte) Einstufungen der teilnehmenden Spieler. Verliert ein Spieler, so wird er für die nächste Runde um einen halben Spielstärkegrad heruntergestuft (indirekt dadurch, daß alle anderen Spieler einen ganzen

Punkt nach oben verschoben werden). Gewinnt ein Spieler, so liegt er einen halben Punkt über seinem Erwartungswert, hat sich also um einen halben Spielstärkegrad verbessert. Man könnte das MacMahon-System auch stabil, d.h. nicht inflationär, spezifizieren, indem man für eine gewonnene Partie einen halben Punkt vergibt, und für eine verlorene Partie einen halben Punkt abzieht. Das würde aber zu negativen MacMahon-Scores am unteren Ende des Feldes führen, wodurch die Implementation aufwendiger würde.

2.2.3 Auswahl von Spielern, die hoch- oder heruntergelost werden

Im Laufe eines Turniers müssen häufig Spieler hoch- oder heruntergelost werden, d.h. gegen Spieler gepaart werden, die sich nicht in derselben MacMahon-Gruppe befinden. Dies ist unvermeidbar, beispielsweise wenn die Zahl der Spieler in einer MacMahon-Gruppe ungerade ist.

Grundsätzlich werden Hoch- und Herunterlosungen als Nachteil für die Spieler angesehen. Aus Gründen der Fairness sollten die betreffenden Spieler daher sorgfältig ausgewählt werden.

Muß ein Spieler in einer MacMahon-Gruppe heruntergelost werden, wird derjenige Spieler ausgewählt, der zu diesem Zeitpunkt aufgrund seiner Zweitkriterien als stärkster Spieler seiner Gruppe eingeschätzt wird. Dies läßt sich wie folgt begründen:

Sortiert man die Spieler einer MacMahon-Gruppe nach den Zweitkriterien, so haben die Spieler an der Spitze im Mittel die schwersten Gegner gehabt, konnten aber trotzdem ihren MacMahonScore erreichen. Die Spieler am unteren Ende hatten im Mittel die leichtesten Gegner. Wird ein Spieler heruntergelost, bedeutet dies, daß er einen relativ leichten Gegner erwarten kann. Es ist fair, demjenigen Spieler einen relativ leichten Gegner zuzuordnen, der bisher die schwersten Gegner hatte. Auf keinen Fall sollte ein Spieler die Möglichkeit erhalten, sich durch einen Sieg gegen einen leichten Gegner in der Rangliste weiter zu verbessern, der zuvor besonders leichte Gegner hatte. Solche Spieler sollten aber bevorzugt

hochgelost werden, wenn es überhaupt notwendig ist, einen Spieler aus der Gruppe gegen einen Spieler aus einer höheren MacMahon-Gruppe zu losen. Sie sollen ihren MacMahonScore durch die Partie gegen einen starken Gegner rechtfertigen. Darüberhinaus gilt auch dieArgumentation des folgendenAbschnitts, der sich mit dem Setzen von Spielern innerhalb einer MacMahon-Gruppe beschäftigt.

2.2.4 Setzen innerhalb von Gruppen

Bei Spielern am unteren Ende der MacMahon-Gruppe vermutet das MacMahon-System, daß der MacMahonScore dieser Spieler nicht durch ihre Spielstärke gerechtfertigt ist, und sie sollten stärkere Gegner erhalten, wodurch sie ihren MacMahonScore rechtfertigen können (ein Sieg gegen einen stärkeren Gegner) oder vom System als tatsächlich schwächer erkannt werden (Verlust der Partie gegen einen stärkeren Gegner). Bei Spielern in der Mitte der Gruppe geht das System davon aus, daß ihr MacMahonScore gerechtfertigt ist, bei Spielern an der Spitze, daß der MacMahonScore eher mehr als gerechtfertigt ist.

Die Aussagekraft der Rangliste wird beim Setzen der stärksten Spieler einer Gruppe gegen die schwächsten Spieler einer Gruppe am deutlichsten erhöht. Man erreicht, daß sich alle starken Spieler durch einen direkten Sieg vor den schwachen Spielern plazieren können. Würde man zufällig innerhalb einer Gruppe losen, würden auch Paarungen schwache Spieler gegen schwache Spieler und starke Spieler gegen starke Spieler entstehen, wodurch mindestens ein schwacher Spieler vor einem starken Spieler (einer der beiden starken Spieler muß ja verlieren) plaziert ist, ohne daß dem starken Spieler eine Chance gegeben wurde, sich vor dem (vermutet) schwächeren Spieler zu plazieren.

In der Top-Gruppe (siehe folgenden Abschnitt) ist es sehr wichtig, die Paarungen nicht zufällig, sondern gemäß einer Setzliste aufgrund der Zweitkriterien zu bilden. Dadurch wird verhindert, daß sich schwächere Spieler durch Glückslose oder interne Turniere auf die vordersten Plätze schieben können.

Bei Spielern, die nicht in den Kampf um die vorderen Plätze eingreifen können, ist die Qualität der Rangliste nur ein Nebenziel. In diesen Bereichen bewirkt die konsequente Umsetzung von Setzlisten innerhalb der MacMahon-Gruppen Extremlosungen. Gemeint ist, daß bevorzugt Spieler gegeneinander gepaart werden, deren Spielstärke sich um einen möglichst großen Wert unterscheidet. Das liegt daran, daß nach der Setzliste bevorzugt Spieler gegeneinander gepaart werden, die voneinander möglichst stark abweichende Zweitkriteriumswerte haben. Und diese sind bei Spielern, die in derselben MacMahon-Gruppe gestartet sind - also dieselbe Spielstärke haben - insbesondere in den ersten 3-4 Runden eines Turniers sehr ähnlich, aber deutlich unterschiedlich zu den meisten Spielern, die in benachbarten MacMahon-Gruppen - also mit einer abweichenden Spielstärke - gestartet sind.

Praktische Erfahrungen auf Turnieren zeigten, daß die konsequente Umsetzung von Setzlisten dazu führt, daß fast alle Spieler nach der ersten Runde nicht mehr gegen Spieler mit der gleichen Spielstärke gepaart werden, sondern entweder gegen Spieler mit höherer oder niedrigerer Einstufung. In Deutschland haben sich die betroffenen Spieler in der Art geäußert, daß sie eine zufällige Losung innerhalb der Gruppen bevorzugen würden. Daher wird die Setzliste in dieser Spezifikation nur in der Top-Gruppe umgesetzt. Im Prinzip ist sie aber in allen MacMahon-Gruppen einsetzbar.

2.2.5 Top-Gruppe

Bestimmte Ziele bei der Losung sind bei Spielern, die um den Turniersieg mitspielen, besonders wichtig (Setzen von Spielern in den Gruppen), andere werden bewußt nicht berücksichtigt (Farben, Klubs und Spielstärkedifferenzen). Würde man z.B. Klubs berücksichtigen, wären die Losungen an der Spitze nicht mehr fair, da bestimmte Spieler bevorzugt nicht gegeneinander gelost werden würden, wodurch bestimmte Spieler benachteiligt werden könnten.

In der Implementierung wird die Top-Gruppe als Menge aller MacMahon-Gruppen ab der niedrigsten MacMahon-Gruppe definiert, die noch einen

Spieler enthält, der zu Beginn des Turniers in der Bar-Gruppe gestartet ist. Es sind auch andere Umsetzungen denkbar. Die Turnierleitung kann über die Festlegung der Grenze zur Top-Gruppe entscheiden.

2.2.6 Wertung der Ergebnisse

Mit den Ergebnissen der einzelnen Runden wird nach der letzten Runde eine Rangliste aller Spieler aufgestellt. Die Ränge können dabei in Prinzip nach vielen unterschiedlichen Kriterien vergeben werden.

In einem MacMahon-Turnier liefert der MacMahon-Score natürlich das Erstkriterium. Für die Verwendung von Zweitkriterien bei Gleichheit der MacMahonScores von Spielern gibt es zahlreiche Verfahren, die alle Vor- und Nachteile haben. Ich habe mich in dieser Spezifikation für die Kombination SOS/SOSOS entschieden, die vom Deutschen Schach Verband bereits seit über 50 Jahren auf Schachturnieren unter dem Namen Buchholzwertung/verfeinerte Buchholzwertung benutzt wird. In anderen europäischen Ländern und im Go generell wurden bislang andere Zweitkriterien verwendet. Die gängigen Zweitkriterienkombinationen werde ich in den folgenden Abschnitten vorstellen und kurz begründen, warum sie nicht verwendet werden sollten. Eine statistische Analyse der besten Kriterienkombination würde den Rahmen dieser Diplomarbeit sprengen, ich werde aber kurz angeben, wie die beste Kombination gefunden werden könnte.

2.2.6.1 Die Kombination SOS/SODOS

Das Kriterium SODOS ist wie folgt definiert:

$$\text{SODOS}(\textit{Spieler}\ 1) = \sum_{\textit{Spieler}_i \text{ war Gegner von } \textit{Spieler 1} \text{ in irgendeiner Runde und } \textit{Spieler 1} \text{ hat gewonnen}} \text{MacMahonScore}(\textit{Spieler}_i)$$

Die Hauptkritik an dieser Kombination bezieht sich auf das willkürliche Kriterium SODOS. Sinn von Zweitkriterien ist es, auch zwischen Spielern mit gleichem MacMahon-Score zu differenzieren und eine Rangliste aufzustellen. Wird ein Spieler in der Rangliste weiter oben geführt, wurde er durch die Zweitkriterien als stärker eingeschätzt. Im Fall des Kriteriums

SOS ist dies plausibel: Seien P_1 und P_2 Spieler mit demselben MacMahonScore und habe P_1 mehr SOS-Punkte als P_2. Das bedeutet, daß P_1 trotz relativ starken Programms seinen MacMahonScore erreicht hat. P_2 hatte dahingegen ein relativ leichtes Programm im Turnierverlauf und hat trotzdem nur seinen MacMahonScore erreicht. Selbstverständlich ist es besser, gegen starke Gegner in diese MacMahon-Gruppe gelangt zu sein, als gegen schwache Gegner.

Das Kriterium SODOS (Sieggegnerpunkte) wird in diesem Zusammenhang nur verwendet, wenn die beiden Spieler P_1 und P_2 den gleichen MacMahonScore und die gleiche Anzahl an SOS-Punkten haben. Hat P1 mehr SODOS-Punkte als P_2, so bedeutet dies, daß P_1 im Mittel gegen stärkere Gegner gewonnen hat als P_2. Da P_1 und P_2 aber insgesamt gleichschwere Gegner hatten (die SOS-Punkte sind identisch), muß P_1 im Mittel gegen schwächere Gegner verloren haben als P_2. Das Kriterium SODOS legt jetzt willkürlich fest, daß ein Spieler stärker ist, wenn er gegen starke Gegner gewinnt, als daß er schwach ist, wenn er gegen schwache Gegner verliert. Das ist willkürlich, und daher sollte SODOS auf Go-Turnieren nicht verwendet werden.

2.2.6.2 CUSS

Das Kriterium CUSS (Cummulative Sum of Scores) ist wie folgt definiert:

$$\text{CUSS}(\textit{Spieler}\ 1) = \sum_{i=1}^{\text{Anzahl der Runden}} \text{MacMahonScore nach Runde } i(\textit{Spieler}\ 1)$$

CUSS ist die Summe der eigenen MacMahonScores, die ein Spieler nach jeder Runde hatte. CUSS entspricht damit den erwarteten Gegnerpunkten für die Gegner, die ein Spieler in den Runden 2 bis Anzahl der Runden+1 erhalten würde. Drei Dinge fallen bei diesem Kriterium sofort auf:

1. (positiv) Dieses Kriterium ist das einzige hier vorgestellte Kriterium, das unabhängig von den Resultaten der Gegner gebildet wird. Das hat den Vorteil, daß Spieler, die einzelne Runden aussetzen, die Gegnerpunkte von anderen Spielern nicht mehr beeinflussen können. Auch ist es keinem Spieler mehr möglich, anderen Spielern - z.B. durch absichtliches Verlieren aller weiteren Partien - zu schaden, wie es bei der Verwendung von SOS grundsätzlich nicht ausgeschlossen werden kann. Das kam im Jahre 1990

tatsächlich auf europäischen Turnieren öfters vor, auf denen „Teams“ von Go-Spielern aus osteuropäischen Ländern gemeinsam spielten. Verlor ein Spieler eines solchen Teams gegen einen Konkurrenten außerhalb des eigenen Teams, so verlor er möglichst alle weiteren Partien, um diesem Konkurrenten schlechte Gegnerpunkte zu bescheren. Die EGF (European Go Federation) führte daraufhin CUSS als offizielles Zweitkriterium ein, schaffte es aber vergangenes Jahr wieder ab, da die Teams seit 1991 nicht mehr auf europäischen Turnieren aktiv gewesen sind.

2. (negativ) Runde 1 wird gestrichen und dafür erhält der Spieler für eine nie stattfindende Runde die erwarteten Gegnerpunkte. Das scheint nicht plausibel.

3. (negativ) Durch die MacMahonScores aller Spieler ist auch die tatsächliche Spielstärke aller an dem Turnier teilnehmenden Spieler relativ genau bekannt. Wenn der exakte Wert zur Verfügung steht, scheint es etwas dubios, diesen durch den Erwartungswert zu ersetzen.

Etwas verborgener sind weitere Effekte bei der Verwendung dieses Kriteriums. Dies wird am offensichtlichsten, wenn man das Hoch- und Herunterlosen von Spielern in andere MacMahon-Gruppen betrachtet. Wird ein Spieler heruntergelost, bedeutet das, daß er einen relativ leichten Gegner bekommt. Er hat eine große Gewinnwahrscheinlichkeit und wird wahrscheinlich nicht nur gewinnen, sondern erhält für diesen Sieg genausoviele CUSS-Punkte gutgeschrieben wie andere Spieler seiner Gruppe, die gegen stärkere Gegner spielen mußten. Da beim CUSS-Kriterium die Reihenfolge der Siege über die Zahl der CUSS-Punkte entscheidet (zuerst gewinnen ist besser, als später zu gewinnen), werden Spieler, die in einer Partie eine hohe Gewinnwahrscheinlichkeit haben, gegenüber anderen Spielern bevorzugt. Bei der Verwendung von SOS wird dieser Vorteil dadurch ausgeglichen, daß der Spieler durch den tatsächlich schwächeren Gegner am Ende des Turniers weniger Gegnerpunkte erhält.

Ein weiterer Nachteil von CUSS ist, daß man am Ende des Turniers relativ viele geteilte Plätze erhält. Z.B. teilen sich alle Spieler einer MacMahon-Gruppe, die in gleicher Reihenfolge gewinnen und verlieren am Ende des Turnier den Platz. Erfahrungswerte besagen, daß sich in einem fünfründigen

Turnier etwa die Hälfte aller Spieler ihren Platz mit mindestens einem anderern Spieler teilt.

2.2.6.3 Die Kombination SOS/SOSOS

Für diese Kombination sprechen einige Argumente. Das wichtigste Argument mag dabei sein, daß SOSOS eine echte Verfeinerung von SOS ist. SOS bewertet die Gegner nach ihrem MacMahonScore. SOSOS unterscheidet dann noch zwischen - für ihre MacMahon-Gruppe - relativ starken Gegnern und relativ schwachen Gegnern. Die relative Stärke der Gegner werden durch deren SOS-Punkte festgelegt.

Ein wesentlicher Vorteil ist zugleich auch ein Kritikpunkt des sehr feinen Kriteriums SOSOS. Nach dem relativ groben Kriterium SOS werden noch relativ viele Spieler als gleich angesehen. Das sehr feine Kriterium SOSOS differenziert praktisch zwischen allen Spielern. Geteilte Plätze kommen praktisch nicht mehr vor. Dies ist von Vorteil, da die ersten Plätze eines Turniers praktisch immer eindeutig vergeben werden.

Die Kritik besteht darin, daß gerade die Gegnerpunkte (SOS) und erst recht die SOSOS-Punkte durch das Losungsglück in den Runden beeinflußt werden. Je feiner man zwischen Spielern differenziert, desto mehr entscheidet das Losungsglück über die Plazierung der Spieler.

2.2.6.4 Die beste Zweitkriterienkombination

Bevor man die „beste“ Zweitkriterienkombination finden kann, muß die Relation „besser als“ zwischen zwei unterschiedlichen Zweitkriterienkombinationen definiert werden:

Definition: Sei eine Ratingliste aller Go-Spieler gegeben und seien ZK_1 und ZK_2 zwei verschiedene Zweitkriterienkombinationen. Dann gilt: ZK_1 ist besser als ZK_2 gdw. die erwartete Rangfolge nach einem Turnier unter Verwendung von ZK_1 eine höhere Übereinstimmung mit der Ratingliste hat als die erwartete Rangfolge des gleichen Turniers unter Verwendung von ZK_2. Die erwartete Rangfolge kann durch Simulation auf Grundlage der Ratingliste und der für die Ratingliste erforderlichen Gewinnwahrscheinlichkeitsfunktion ermittelt werden.

Anschließend kann man verschiedene Kombinationen von Zweitkriterien miteinander vergleichen und schließlich diejenige als „Beste“ ansehen, die bei diesen Tests am besten abgeschnitten hat. Zur Zeit existiert weder eine stabile Ratingliste noch eine Gewinnwahrscheinlichkeitsfunktion.

2.3 Eine Gewichtsfunktion für die Losung einer Runde nach dem MacMahon-System

Der Maximum-Weight-Perfect-Matching-Algorithmus erhält als Eingabe eine Matrix, die die Gewichtsfunktion für die Losung zwischen allen Spielern, die gepaart werden sollen, definiert.

Da sich die Angabe von Kosten für nicht-erwünschte Paarungen plausibler begründen läßt als der Nutzen für Paarungen, die besonders erwünscht sind, wird hier eine Kostenfunktion $c(i,j)$ definiert, die die Kosten einer Paarung von dem Spieler *i* mit dem Spieler *j* festlegt. Die Gewichtsfunktion ist dann als $w(i,j)=\text{MAXWEIGHT}-c(i,j)$ mit einer sehr großen Konstante MAXWEIGHT definiert.

2.3.1 Eine Kostenfunktion für die Paarung (i,j)

Die meisten Bestandteile der Kostenfunktion lassen sich direkt aus den Daten der Spieler selbst ermitteln. Nur für die Verwendung der Setzliste und für die Auswahl der Spieler für ein etwaiges Hoch- oder Herunterlosen sind Informationen aus der Umgebung, d.h. der MacMahon-Gruppen der Spieler *i* und *j* erforderlich. Bedingungen für die Gewichte der einzelnen Komponenten der Kostenfunktion werden im Anschluß an die Angabe der hier verwendeten Kostenfunktion aufgestellt. Erfüllen die gewählten Gewichte die Bedingungen, so genügt die durch die Kostenfunktion definierte optimale Paarung den in Abschnitt 2.1 aufgestellten Regeln für eine Losung nach dem MacMahon-System.

Liefere die Funktion MacMahonScore(*i*) den MacMahonScore des Spielers *i*.

Die Kosten einer Paarung (*i*,*j*) werden durch folgende Kostenfunktion festgelegt (o.B.d.A. sei MacMahonScore(*i*)≥MacMahonScore(*j*)):

```
c_ij=AlreadyPlayedEachOther(i,j)*PlayedBefore_Weight
   +(MMSDifference(i,j)/2)^SameGroupPower*SameGroup_Weight
   if (MacMahonScore(i)<>MacMahonScore(j))
   {
     +(100-SecondCriteriaIndex(i)+SecondCriteriaIndex(j))*OddMan_Weight
   }
   if (i∉TopGroup and j∉TopGroup)
   {
     +SameCountry(i,j)*SameCountry_Weight
     +SameClub(i,j)*SameClub_Weight
     +TooBigStrengthDifference(i,j)*StrengthDifference_Weight
     if (not Handicap(i,j)>0)
     {
       +ColorCosts(i,j)*Color_Weight
     }
   }
   else
   {
     if (MacMahonScore(i)=MacMahonScore(j))
     {
       +|SecondCriteriaIndex(i)+SecondCriteriaIndex(j)-100|*Setting_Weight
     }
   }
```

Die Funktionen `AlreadyPlayedEachOther`, `SameCountry`, `SameClub` und `TooBigStrengthDifference` sind Indikatorfunktionen, die 1 zurückliefern, wenn dieAbfrage für die beiden Spieler positiv ausfällt, und sonst 0.

Die Funktion `MMSDifference` ist als Differenz der MacMahonScores der beiden Spieler definiert. Der Bildbereich der Funktion liegt zwischen Null und dem Vierfachen der größten vorkommenden MacMahonScore-Differenz (=Top-MacMahon-Bar + Nummer der aktuellen Runde). Dem Vierfachen daher, weil bei der Berechnung von SODOS-Punkten ¼-MacMahon-Punkte vorkommen können. Damit alle Werte als Integers

verwaltet werden können, hat ein MacMahon-Punkt intern den Wert 4, ein halber MacMahon-Punkt den Wert 2.

Die Funktion `ColorCosts` testet die beiden Möglichkeiten der Farbvergabe, wenn die Partie keine Vorgabepartie ist (Handicap=0) und liefert die Kosten der besseren Variante zurück. Die Funktion bewertet die beiden Farbbilanzen inklusiv der zu losenden Runde für jeden Spieler einzeln und liefert die Summe zurück. Die Bewertung der Farbbilanz eines Spielers ist definiert als:

$$\text{Farbbilanz}(i) = 1 - \frac{\min(\#\,\text{Schwarz}(i), \#\,\text{Weiß}(i))}{\max(\#\,\text{Schwarz}(i), \#\,\text{Weiß}(i))}$$

Der Wertebereich von `ColorCosts` ist demnach [0,2].

Die Funktion `SecondCriteriaIndex` liefert die Stärke des Spielers relativ zu den anderen Spielern seiner Gruppe in %. Ist der Spieler der Stärkste seiner Gruppe, so hat er einen Index von 100. Ist er der Schwächste, so hat er einen Index von 0. Ist er in seiner Gruppe durchschnittlich stark, liefert die Funktion 50 zurück. Die relative Stärke eines Spielers entscheidet darüber, ob ein Spieler in eine andere Gruppe gelost wird und (in der Top-Gruppe) wie die Spieler innerhalb einer Gruppe gegeneinander gesetzt werden.

2.3.2 Bestimmung der Gewichte zu den Komponenten der Kostenfunktion

Sei im folgenden N die Zahl der Spieler in der Paarung und R die aktuelle Runde. In der zu optimierenden Gesamtkostenfunktion

$$C = \sum_{(i,j)\in\text{Paarung}} c_{ij} \rightarrow \min_{\text{alle perfekten Paarungen der N Spieler}}$$

werden höchstens $N/2$ Kostenwerte c_{ij} aufsummiert.

Die größte Differenz zwischen den MacMahonScores von zwei Spielern kann nicht mehr als 26 (maximaler Abstand zwischen 20 Kyu und 6 Dan,

den Grenzen der zugelassenen Spielstärken, plus einem MacMahon-Punkt im Falle der Existenz eines Super-MacMahon-Bars oberhalb von 6 Dan) + *R* betragen. Aufgrund der internen Darstellung der MacMahonScores als Integers muß dieser Wert noch mit 4 multipliziert werden. Bei höchstens 10 Runden beschränkt sich der Wert also auf 36*4=144.

Der Parameter SameGroup_Power in der Kostenfunktion steuert, wieviel schlimmer es ist, einen Spieler über zwei Gruppen hinweg zu losen als zwei Spieler um eine Gruppe. Gute Erfahrungen wurden mit dem Wert 2 gemacht. Begnügt man sich mit diesem Wert, so kann in der Kostenfunktion die SameGroup-Komponente nicht größer als 144^2*SameGroup_Weight werden.

Die Gewichtung der einzelnen Komponenten der Kostenfunktion muß jetzt so gewählt werden, daß die in Abschnitt 2.1 aufgestellten Regeln für eine Losung nach dem MacMahon-System eingehalten werden. Die Wichtigkeit der Ziele einer MacMahon-Losung lassen sich in vier Ebenen einteilen, in denen jeweils der nächsthöheren Ebene (absoluter) Vorrang zu gewähren ist.

Ebene 1:	Zwei Spieler dürfen nicht wiederholt gegeneinander spielen.
Ebene 2:	Global muß die Zahl der Hoch- und Herunterlosungen minimiert werden.
Ebene 3:	Die Auswahl der Spieler, die hoch- oder heruntergelost werden.
	Die Durchsetzung einer Paarung gemäß der Setzliste.
Ebene 4:	Die restlichen Ziele: Farbwahl, Berücksichtigung der Klubs und Länder und die Vermeidung zu großer Spielstärkeunterschiede.

Ebene 1 hat absoluten Vorrang vor Ebene 2, die wiederum absoluten Vorrang vor Zielen der Ebene 3 hat.

Auf Ebene 3 ist die Auswahl des Spielers zum Herunterlosen wichtiger, aber nicht absolut dominierend.

Der Vorrang von Ebene 3 gegenüber Ebene 4 soll stark, aber nicht absolut dominierend sein. Es kann wünschenswerter sein, nicht den objektiv stärksten Spieler einer Gruppe herunterzulosen, wenn dieser dann gegen einen Spieler aus seinem Klub spielen müßte.

Die Kostenfunktion, eingeschränkt auf die Ziele der Ebene 4, hat den Wertebereich:

E4 ={0,...,Wertebereich von `SameClub` * `Club_Weight`

+ Wertebereich von `SameCountry` * `Country_Weight`

+ Wertebereich von `TooBigStrengthDifference` * `StrengthDifference_Weight`

+ Wertebereich von `ColorCosts` * `Color_Weight`}

={0,...,`Club_Weight` + `Country_Weight` + `StrengthDifference_Weight` + 2*`Color_Weight`}

Die Kostenfunktion eingeschränkt auf die Ziele der Ebene 3 hat den Wertebereich (entweder sind beide Spieler in derselben Gruppe und sie werden gesetzt, oder sie sind in verschiedenen Gruppen und werden hoch- bzw. heruntergelost - aber nicht beides gleichzeitig):

E3 ={0,...,max(100*`Setting_Weight`, 100*`OddMan_Weight`)}

Die Kostenfunktion, eingeschränkt auf das Ziel der Ebene 2, hat den Wertebereich:

E2 ={0,...,144^2*`SameGroup_Weight`}

={0,...,20736*`SameGroup_Weight`}

Wenn Kosten für das Ziel auf Ebene 2 überhaupt auftreten, dann haben sie mindestens den Wert 4*`SameGroup_Weight`.

Die Kostenfunktion, eingeschränkt auf das Ziel der Ebene 1, hat den Wertebereich:

E1 ={0,...,`PlayedBefore_Weight`}

Die folgende Belegung der Gewichte ergab sehr gute Resultate und verletzte die Regeln aus Abschnitt 2.1 auf bisher zwei mit dem hier beschriebenen Programm durchgeführten Turnieren nicht:

`Club_Weight`	= 80
`Country_Weight`	= 20
`StrengthDifference_Weight`	= 80
`Color_Weight`	= 50
`Setting_Weight`	= 10
`OddMan_Weight`	= 100
`SameGroup_Weight`	= 1000
`PlayedBefore_Weight`	= MAXWEIGHT=268435455

Der später vorgestellte Maximum-Weight-Perfect-Matching-Algorithmus arbeitet mit Integern als Variablen. Die größte Auflösung bieten unter der verwendeten Programmiersprache C++ Integer-Variablen vom Typ long (32 Bit, davon eines für das Vorzeichen). Da in Zwischenschritten des Algorithmus Gewichte durch vier geteilt werden, gehen weitere zwei Bits des verwendeten Integer-Typs verloren. Ein weiteres Bit (zusätzlich zum Vorzeichen-Bit) kann nicht genutzt werden, da der Algorithmus zwischenzeitlich zwei Gewichte addiert und dieses Bit für einen möglichen Überlauf zur Verfügung stehen muß. Von den 32 Bit des verwendeten

Integer-Typs long stehen daher nur 28 für die Darstellung der Gewichte zur Verfügung. Aus diesen numerischen Einschränkungen heraus ist also der größte verwendbare Wert der oben als MAXWEIGHT definierte.

Die Auflösung des Bereichs 0 bis MAXWEIGHT reicht aber nicht aus, um den absoluten Vorrang von Ebene 1 gegenüber Ebene 2 und von Ebene 2 gegenüber Ebene 3 zu gewährleisten. Beispiel: Zwei Spieler über die maximal mögliche Distanz hinweg gegeneinander zu paaren, verursacht bei den oben angegebenen Werten Kosten in Höhe von (mindestens) 20736000. Dies ist nur ca. 1/13 von MAXWEIGHT. Bevor der Algorithmus also 14 Spieler über die maximale Distanz paart, würde er das absolute Ziel verletzen, zwei Spieler nicht wiederholt gegeneinander zu paaren. Praktisch ist dies allerdings weit unwahrscheinlicher. Wenn 14 Spieler über die maximale Distanz gepaart werden müßten, ließe sich dieses Problem kaum dadurch lösen, daß irgendwo zwei Spieler wiederholt gegeneinander gepaart würden.

Dennoch liefern die angegebenen Gewichte nur eine Heuristik an die in Abschnitt 2.1 aufgestellten Regeln für eine Losung nach dem MacMahon-System.

3 Ein MacMahon Losungsprogramm für Go-Turniere

3.1 Ein Anforderungsprofil aus Sicht des Anwenders

Folgende Aufgaben müssen von einem Turnierverwaltungsprogramm für MacMahon-Go-Turniere übernommen werden:

(A1) Spielerdaten (Vorname, Nachname, Klub, Land und Spielstärke) müssen eingegeben werden können und ggf. zu jedem Zeitpunkt des Turniers korrigierbar sein. Spieler sollten auch zu löschen sein.

(A2) Für jeden Spieler des Turniers sollten die folgenden Attribute modifizierbar sein (durch die Turnierleitung):

-Angabe der Runden, in denen der Spieler an dem Turnier teilnimmt. In allen anderen Runden setzt der Spieler aus.

-Der Start-MacMahonScore eines Spielers.

-Die Zugehörigkeit des Spielers zur Super-MacMahon-Bar-Gruppe (wenn diese Erweiterung des MacMahon-Systems implementiert ist).

(A3) Alle wichtigen Turnierparameter sollten vor der Durchführung der ersten Runde festlegbar und später noch modifizierbar sein:

-Anzahl der zu spielenden Runden

-MacMahon-Bars/Super-MacMahon-Bar

-Vorgabestrategien, d.h. wann wird wie zwischen welchen Spielern von dem Programm automatisch eine ggf. notwendige Vorgabe berechnet.

-Vorgaben für die Erstellung der Rangliste, d.h. die Turnierleitung kann zwischen verschiedenen Zweitkriterienkombinationen wählen, um das Turnier so den gegebenen Anforderungen oder Vorgaben anzupassen.

-Für den Ausdruck von Listen kann dem Turnier ein Name verliehen werden.

-Die Losung sollte bzgl. konkurrierender Ziele des MacMahon-Systems durch die Turnierleitung beeinflußbar sein.

(A4) Das Programm führt ggf. Losungen für einzelne Runden nach dem MacMahon-System durch. Dabei ist es sehr wichtig, daß die Losung in kurzer Zeit gefunden wird, da sich das Turnier um genau diese Zeit verzögert. Das Programm sollte die eigene Losung auf nicht erwünschte Effekte überprüfen (z.B. wenn zwei Spieler gegeneinander gepaart wurden, die einen um mehr als einen Punkt unterschiedlichen MacMahonScore haben).

(A5) Die Losung der aktuellen Runde kann „von Hand“ durch die Turnierleitung geändert, ergänzt oder verworfen werden.

(A6) Ergebnisse zu den Paarungen einer Runde können eingegeben und ggf. später noch korrigiert werden.

(A7) Der Anwender kann die aktuell zu bearbeitende Runde des Turniers frei auswählen (z.B. um nachträglich Ergebnisse vorheriger Runden zu korrigieren).

(A8) Das Programm erlaubt den Ausdruck von Paarungen, Ergebnissen und einer Rangliste auf einen angeschlossenen Drucker.

(A9) Das Programm sollte auch für einen Laien benutzbar sein. D.h., es ist nicht erforderlich mit dem MacMahon-System in

den Einzelheiten vertraut zu sein, um als Turnierleitung ein Turnier mit dem Programm nach dem MacMahon-System durchführen zu können. Das bedeutet, daß das Programm alle Eingaben des Benutzers auf Widersprüche zum MacMahon-System überprüfen muß (z.B. die Wahl der MacMahon-Bars, der Algorithmus-Parameter, doppelte Eingabe von Spielern...)

(A10) Laden und Speichern von Turnierdaten.

(A1)-(A10) beschreiben sehr wichtige Anforderungen an Turnierverwaltungsprogramme. Darüberhinaus gibt es noch eine Vielzahl an weiteren möglichen Features eines Turnierverwaltungsprogramms:

(B1) Statistiken, z.B. Gewinnwahrscheinlichkeit von Schwarz in dem Turnier, Klub-Ergebnisse, Länder-Ergebnisse,

(B2) spezielle Tabellen, z.B. sortiert nach der Reihenfolge der Siege. Auf Go-Turnieren bekommen auf einem Turnier mit 5 Runden alle Spieler mit 4 oder 5 Siegen üblicherweise einen Preis. Eine Liste mit diesen Spielern erleichtert die Siegerehrung.

(B3) Verwendung einer Spieler-Datenbank im Hintergrund. Das ermöglicht eine schnellere Eingabe der Spielerdaten vor Beginn des Turniers und erleichtert es, Eingabefehler zu finden.

(B4) Automatische Sicherung von Turnierdaten.

(B5) Import und Export zu anderen gängigen Turnierverwaltungsprogrammen und zu Ratinglisten (in beiden Punkten existiert gegenwärtig in Europa noch kein Standard, ist aber im Falle der Ratingliste in Vorbereitung).

(B6) Ausgabe der Rangliste und der Paarungs-/Ergebnisliste auf den Bildschirm/in eine Datei.

3.2 Das Turnierverwaltungsprogramm „MacMahon“

Bestandteil dieser Diplomarbeit ist das Turnierverwaltungsprogramm „MacMahon“, das von mir aufgrund des Anforderungsprofils in Abschnitt 3.1 konzipiert wurde. In das Programm ist auch der in Abschnitt 4 beschriebene Maximum-Weight-Perfect-Matching-Algorithmus integriert, der eine global optimale Losung aller Spieler in einer Runde bzgl. der in Abschnitt 2.3 hergeleiteten Gewichtsfunktion garantiert.

„MacMahon“ erfüllt die grundlegenden Anforderungen (A1)-(A8) und (A10). Das Programm ist zwar auch von einem Laien bedienbar, eine automatische Überprüfung der Semantik der meisten Eingaben wird aber nicht durchgeführt (siehe auch Abschnitt 5: Ausblick und Wertung). Von den weitergehenden Features bietet „MacMahon“ nur (B6), allerdings nur auf dem Bildschirm und nicht in eine Datei.

Die Bedienung des Programms kann der als Anlage mit dem Programm gelieferten Bedienungsanleitung entnommen werden.

Im folgenden Abschnitt werden kurz die grundlegenden Datentypen zur Verwaltung des Turniers anhand der für sie implementierten Klassen in C++ beschrieben.

3.2.1 Implementation der Spielerdaten

Die Klasse `TPlayer` ist der Headerdatei tour.h wie folgt deklariert:

```
/* Zunächst eine Struktur, in der die Höhe der in einer Runde gegebenen Vorgabe
gespeichert wird */
struct THandicap
{
   int Stones;
};
```

```
/* In der Struktur TRound werden alle Daten eines Spielers speziell für diese Runde
gespeichert */
struct TRound
{
   int Opponent;
   /* Gegner des Spielers in dieser Runde. Hat der Spieler keinen Gegner, so gilt
   Opponent = NOPLAYER */
   int Flags;
   /* In den Flags wird binär codiert gespeichert:
   -ob der Spieler in dieser Runde mitspielt oder aussetzt
   -das Ergebnis in dem Fall, daß er mitspielt
   -die Farbe, mit der der Spieler mitspielte */
   THandicap Handicap;
   /* In dieser Struktur wird die Höhe der ggf. verwendeten Vorgabe gespeichert.*/
};

/* Die Klasse TPlayer stellt alle wichtigen Datenfelder für Spielerdaten und einige
elementare Funktionen in den Spielerdaten zur Verfügung */
class TPlayer : public TStreamable
{
public:
   char Surname[MAXSURNAMELEN];
   /* String für den Nachnamen. Alle Konstanten sind in macmahon.h definiert */
   char FirstName[MAXFIRSTNAMELEN];
   /* String für den Vornamen */
   int CountryID;
   /* ID, die eindeutig einem Land zugeordnet ist. Die Länderliste wird in der Klasse
   Tournament verwaltet */
   int ClubID;
   /* ID, die eindeutig einem Klub zugeordnet ist */
   TRound* Rounds[MAXROUNDS];
   /* In diesem Array werden die Daten der Runden dieses Spielers gespeichert. */
   int Flags;
   /* In Flags werden binär bestimmte Attribute des Spielers gespeichert:
   -Der Spieler startete zu Beginn des Turniers im Top-Bar
   -Der Spieler startete zu Beginn des Turniers im Bottom-Bar
   -Der Spieler ist ein Mitglied der Super-Bar-Gruppe
   -Der Start-MacMahonScore dieses Spielers darf nicht automatisch neu berechnet
   werden, wenn sich z.B. die MacMahon-Bars ändern */
   int Calculation;
   /* In Calculation werden Flags gesetzt, die angeben, welche Werte dieses
   Spielers neu berechnet werden müssen. Z.B. die SOS- und die SOSOS-Punkte, da
   sich der MacMahonScore eines Gegners dieses Spielers geändert hat. */
```

```
   int Strength;
   /* Spielstärke in Kyu (>0) bzw. Dan (≤0). */
   int StartMacMahonScore;
   /* Enthält den Start-MacMahonScore dieses Spielers. */
   int MacMahonScore;
   /* Enhält den aktuell zu diesem Zeitpunkt gültigen MacMahonScore dieses
   Spielers. */
   int Sos;
   /* Datenfeld für die Gegnerpunte SOS */
   int Sosos;
   /* Datenfeld für SOSOS */
   int Sodos;
   /* Datenfeld für SODOS - dieses Programm erlaubt auch den Einsatz von in
   dieser Arbeit kritisierten Kriterien */
   int Cuss;
   /* Datenfeld für CUSS */
/* Die Elementfunktionen von TPlayer: */
   TPlayer(); /* Konstruktor */
   ~TPlayer(); /* Destruktor */
   void GetStrength(LPSTR AString);
   /* GetStrength liefert die Spielstärke des Spielers formatiert in einem String
   zurück */
   int SetStrength(LPSTR AString);
   /* Diese Funktion interpretiert die in dem String angegebene Spielstärke und
   setzt das Datenfeld entsprechend */
   void Clear();
   /* Setzt alle Datenfelder auf Default-Werte */
   void CheckBarMember(Tournament *tour);
   /* Setzt ggf. Flags, wenn der Spieler sich in denen in dem Objekt vom Typ
   Tournament definierten MacMahon-Bars befindet (vor der ersten Runde) */
   LPSTR GetNameString(LPSTR AString);
   LPSTR GetNameStrengthString(LPSTR AString);
   LPSTR GetShortStrengthString(LPSTR AString);
   static PTStreamable build();
   /* Liest das mit einem Spieler assoziierte Objekt aus einem Stream */
protected:
   TPlayer(StreamableInit);
   virtual Pvoid read(Ripstream is);
   virtual void write(Ropstream os);
private:
   virtual const Pchar streamableName() const { return
   "TPlayer"; };
};
```

3.2.2 Implementation der Turnierdaten

Die Klasse `Tournament` ist in der Headerdatei tour.h wie folgt deklariert (alle Konstanten sind in macmahon.h definiert):

```
/* Diese Klasse enthält alle Daten eines Turniers */
class Tournament : public TStreamable
{
public:
   char FileName[MAXPATH]; /* aktueller Dateiname */
   BOOL TournamentSaved;
   /* = TRUE, wenn alle geänderten Daten auch gespeichert sind */
   char TournamentTitle[MAXTOURNAMENTTITLELEN];
   /* String mit dem Titel des Turniers */
   int TournamentSystem;
   /* Gibt das gewählte Turniersystem an: MACMAHON oder SWISSSYSTEM */
   int NumberOfRounds;
   /* Gibt die Anzahl der Runden des durchzuführenden Turniers an */
   int CurrentRound;
   /* CurrentRound gibt an, welche Runde gerade aktuell ist, d.h. für diese Runde
   gelten augenblicklich alle Änderungen */
   int CurrentPairing[MAXPAIRS][4];
   /* Beinhaltet die aktuell gültige Paarung. Die zweite Dimension hat folgende
   Bedeutung:
   CurrentPairing[][BLACK] liefert den Spieler mit Schwarz in dieser Partie
   CurrentPairing[][WHITE] liefert den Spieler mit Weiß
   CurrentPairing[][FLAGS] enthält das Ergebnis und Angabe über die Vorgabe
   CurrentPairing[][HANDICAP] enthält die gegebene Vorgabe */
   int CurrentPairsNumber;
   /* Anzahl der Paarungen in der aktuellen Paarung. Dieses und die folgenden
   vergleichbaren Arrays werden bei jedem Wechsel zu einer anderen Runde
   automatisch aktualisiert. */
   int CurrentPairingSortStrategy;
   /* Enthält Flags für die Sortierung der Paarungen bei der Ausgabe auf den
   Bildschirm oder den Drucker */
   int CurrentFreePlayer[MAXPLAYER + 1];
   /* Dieses Array enthält Referenzen zu allen Spielern, die in der aktuellen Runde
   spielwillig sind, aber noch keinen Gegner erhalten haben. Bei der Ausführung
   der Funktion MakePairing im Programm werden alle Spieler dieses Arrays
   durch den Algorithmus (ggf. erweitert um ein Freilos) untereinander gepaart */
```

```
int CurrentFreePNumber;
```
/ Anzahl der aktuell freien Spieler */*
```
int CurrentNotPlaying[MAXPLAYER];
```
/ Dieses Array enthält die Referenz zu allen Spielern, die in der aktuellen Runde nicht mitspielen */*
```
int CurrentNotPNumber;
```
/ Anzahl der Spieler, die in der aktuellen Runde aussetzen. */*
```
TPlayer* PlayerList[MAXPLAYER];
```
/ Array mit allen Spielern. Die Position im Array dient auch als Referenz des Spielers. Bei jeder Änderung relevanter Daten (z.B. des MacMahonScores) eines Spielers wird dieses Array neu sortiert, so daß die Reihenfolge in diesem Array immer die aktuelle Rangliste widerspiegelt */*
```
BOOL PlayerSamePlace[MAXPLAYER];
```
/ Teilen sich mehrere Spieler einen Rang, so wird dies beim Ausdrucken der Rangliste angegeben. In diesem Array wird die entsprechende Information abgelegt */*
```
int DefaultPlayerRoundFlags;
```
/ In den Runden-Daten der Spieler werden die Flags auf den Inhalt dieses Datenfeldes gesetzt */*
```
int DefaultForcedPairingFlags;
```
/ Bei Paarungen, die durch die Turnierleitung „von Hand“ festgelegt wurden, werden die Flags auf den Wert in diesem Datenfeld gesetzt */*
```
int NumberOfPlayers;
```
/ Anzahl der teilnehmenden Spieler = Anzahl der belegten Felder in dem Array PlayerList */*
```
int SortCriteria[MAXSORTCRITERIA];
```
/ Dieses Array enthält die Kriterien für die Rangfolge der Spieler in der Reihenfolge, wie sie zur Differenzierung herangezogen werden sollen */*
```
int AddSortCriteria[MAXADDSORTCRITERIA];
```
/ Sind zwei Spieler nach der Anwendung der Sortierkriterien noch auf dem gleichen Rang, so legen die zusätzlichen Kriterien in diesem Array fest, in welcher Reihenfolge die Spieler auf Listen erscheinen sollen. Die Spieler haben natürlich dennoch denselben Rang. Als Default werden die zusätzlichen Sortierkriterien als 1. nominelle Spielstärke und 2. alphabethische Reihenfolge festgelegt */*
```
int LowerBar;
```
/ Alle Spieler unter der Spielstärke, die durch LowerBar angegeben ist, spielen im unteren MacMahon-Bar */*
```
int UpperBar;
```
/ Alle Spieler ab der Spielstärke, die durch UpperBar angegeben ist, spielen im oberen MacMahon-Bar */*
```
int CurrentRoundInAccount;
```

```
/* Dieses Flag gibt an, ob die (möglicherweise teilweise) schon vorliegenden
Ergebnisse der aktuellen Runde in der Generierung der Rangliste mit einbezogen
werden soll (TRUE = ja) */
int HandicapFlags[5];
/* Enthält Informationen über die Strategien, nach denen Vorgabe zwischen
Spielern ermittelt wird */
int Handicap;
/* dito, in diesem Datenfeld wird aber die grobe Strategie festgelegt: Mit
Vorgabe/ohne Vorgabe */
BOOL StartMMSChangeOnStrengthChange;
/* Ist dieses Flag auf TRUE gesetzt, so wird der Start-MacMahonScore eines
Spielers automatisch aktualisiert, wenn sich die Spielstärke (z.B. aufgrund eines
Eingabefehlers) des Spielers ändert.
int Calculation;
/* Enthält Flags, welche Datenfelder nicht mehr aktuell sind und daher neu
zusammengestellt werden müssen. Z.B. müssen nach einem Rundenwechsel alle
Arrays mit den aktuellen Paarungen, den freien Spielern und den aussetzenden
Spielern neu ermittelt werden. Außerdem ändern sich in dem Fall die aktuellen
MacMahonScores aller Spieler */
int ActiveCriteria;
/* Enthält Flags über die Ranglistenkriterien, die von der Turnierleitung
ausgewählt wurden. Nur diese müssen z.B. beim Wechsel der aktuellen Runde
neu berechnet werden */
CountryList *Countries;
/* Verkettete Liste der Länder (Name und zugeordnete ID) */
ClubList *Clubs;
/* Verkettete List der Klubs (Name und zugeordnete ID) */
TPairingOptions *PairingOptions;
/* Die Struktur vom Typ TPairingOptions enthält die eingestellten Gewichte für
die konkurrierenden Ziele des MacMahon-Turniers und Angaben darüber, wie
die Teilnehmer der Top-Gruppe bestimmt werden. */

/* Die Elementfunktionen: */
Tournament(); /* Konstruktor */
~Tournament(); /* Destruktor */
void SetParent(PTWindowsObject AParent);
/* Für Meldungsfenster und andere Windowsfunktionen brauchen bestimmte
Elementfunktionen von Tournament eine Referenz zu einem Fensterobjekt unter
Windows. Über diese werden dann Windows-Funktionen aufgerufen */
int InsertPlayer(TPlayer *APlayer);
/* Fügt einen neuen Spieler in das Array der Spieler ein */
void DeletePlayer(int APlayer);
/* Löscht einen Spieler aus dem Array der Spieler */
```

```
int GetNumberOfPlayers();
/* Liefert die aktuelle Zahl der Spieler in dem Turnier */
int ComparePlayers(TPlayer *P1, TPlayer *P2, BOOL
WithAdd);
/* Compare Players vergleicht zwei Spieler nach den gewählten Kriterien. Ist
WithAdd=TRUE, werden bei Gleichheit der Ranglistenkriterien noch die
zusätzlichen Kriterien für eine eindeutige Entscheidung herangezogen */
void SortPlayers();
/* Diese Funktion sortiert das Spieler-Array und aktualisiert die an anderen
Stellen verwendeten Referenzen aller Spieler auf die neue Position in dem Array
*/
void ClearAllData();
/* Löscht alle Daten und setzt Default-Werte */
int MacMahonScore(int APlayer, int r);
/* Diese Funktion berechnet den MacMahonScore eines Spielers in der aktuellen
Runde ausgehend von dem Start-MacMahonScore des Spielers. Ist r=1, so
werden die Ergebnisse der aktuellen Runde mit einbezogen. */
void SetStartMacMahonScore(int aplayer);
/* Setzt den StartMacMahonScore von einem Spieler neu */
int StartMacMahonScore(int aplayer);
/* Berechnet nur den StartMacMahonScore eines Spielers */
int InsertPairing(int Black, int White, int Flags, int
Stones);
/* Fügt die Partie Black gegen White mit Flags und der Vorgabe (Stones) in das
Array der aktuellen Paarung ein */
void DeletePairing(int APairing);
/* Löscht eine Partie aus der aktuellen Paarung */
void ExtractPairing();
/* Diese Funktion baut das Array mit den aktuellen Paarungen aus den
Spielerdaten auf */
void RewritePairing(int APairing);
/* Diese Funktion aktualisiert die Spieler-Daten gemäß der aktuellen Paarung
*/
void RewriteAllPairings();
/* Aktualisiert alle Spieler-Daten gemäß der aktuellen Paarung */
int InsertFreePlayer(int APlayer);
/* Fügt einen freien Spieler in das Array mit den freien Spielern ein */
void DeleteFreePlayer(int APlayer);
/* Löscht einen Spieler aus dem Array mit den freien Spielern */
void ExtractFreePlayers();
/* Diese Funktion baut das Array der freien Spieler aus den Spieler-Daten auf */
int InsertNotPlaying(int APlayer);
/* Fügt einen aussetzenden Spieler in das entsprechende Array ein */
```

```
void DeleteNotPlaying(int APlayer);
/* Löscht einen aussetzenden Spieler aus dem Array */
void ExtractNotPlaying();
/* Diese Funktion baut das Array der aussetzenden Spieler aus den Spieler-Daten
auf */
void Extract();
/* Baut die drei Arrays der aktuellen Paarung aus den Spieler-Daten auf */
void ExtractPlayer(int i);
void RewriteFreePlayer(int aplayer);
void RewriteAllFreePlayer();
void RewriteNotPlaying(int aplayer);
void RewriteAllNotPlaying();
void Rewrite();
/* Rewrite aktualisiert die Spieler-Daten über die darüber deklarierten Funktionen
gemäß den aktuellen Einträgen in den drei Arrays zur aktuellen Paarung */
int AutoHandicap(int p1, int p2);
/* Berechnet die Vorgabe, die in einer Partie zwischen zwei Spielern gemäß den
festgelegten Vorgabe-Strategien gegeben werden müßte */
BOOL Win(int aplayer, int around);
/* Liefert TRUE, wenn der Spieler in der angegebenen Runde gewonnen hat */
void UpdateAll();
/* Diese Funktion überprüft, welche Datenfelder in den Turnier-Daten und den
Spieler-Daten nicht mehr aktuell sind, berechnet diese neu */
void UpdatePlayers();
/* Dito, aber nur Spieler-Daten */
void SetUpdatePlayerFlags(int i);
/* Diese Funktion sorgt dafür, daß bei der Änderung eines Datenfeldes der
Spielerdaten diejenigen anderer Spieler aktualisiert werden, die dadurch beeinflußt
werden können */
void RewritePlayer(int i);
void DelAllCountries(); /* löscht die Liste mit den Ländern */
void DelAllClubs(); /* löscht die Liste mit den Klubs */
int AddCountry(LPSTR acountry);
/* Fügt ein Land in die Liste der Länder ein */
int AddClub(LPSTR aclub);
/* Fügt einen Klub in die Liste der Klubs ein */
void DeleteCountry(int CountryID);
/* Löscht ein Land aus der Liste */
void DeleteClub(int ClubID);
/* Löscht einen Klub aus der Liste */
int RenameCountry(int CountryID, LPSTR NewName);
int RenameClub(int ClubID, LPSTR NewName);
LPSTR GetCountryString(int CountryID, LPSTR AString);
```

```
    /* Liefert einen formatierten String des Landes */
    LPSTR GetClubString(int ClubID, LPSTR AString);
    /* Liefert einen formatierten String des Klubs */
    void CopyPlayer(TPlayer *dest, TPlayer *source, BOOL
    flag); /* kopiert einen Spieler */
/* Funktionen zur Ausgabe von Listen auf den Drucker: */
    LPSTR GetWallListString(LPSTR astring, int APlayer,
    int Dest);
    /* Liefert einen formatierten String (eine Zeile der Rangliste) eines Spielers.
    Dest=PRINTER oder Dest=DISPLAY */
    int WallListOnPrinter(TPrinter *Printer);
    /* Druckt die aktuelle Rangliste auf dem in Printer festgelegten Drucker */
    LPSTR GetPairingString(LPSTR astring, int APairing);
    /* Liefert eine Zeile der Paarungsliste */
    int PairingOnPrinter(TPrinter *Printer);
    /* Druckt die Paarungsliste (mit Ergebnissen) auf den in Printer angegebenen
    Drucker */
/* Kontrollfunktionen für die Paarung */
    void CheckPairing();
    /* CheckPairing kontrolliert die Güte der Paarung und warnt vor möglichen
    Fehlern in der Paarung (z.B. zwei Spieler wiederholt gegeneinander oder wenn
    zwei Spieler gegeneinander gelost wurden, die nicht in derselben MacMahon-
    Gruppe waren */
    void DropComputerPairings();
    /* Löscht alle mit MakePairing von dem Programm generierten Losungen */
/* Die folgenden Funktionen werden bei der Ermittlung der Kostenfunktion verwendet:
*/
    int AlreadyPlayedEachOther(int Player1, int Player2);
    /* Gleich 1, wenn die beiden Spieler in einer vorhergehenden Runde schon
    einmal gegeneinander gelost wurden */
    int SameClub(int Player1, int Player2);
    /* Gleich 1, wenn die beiden Spieler dieselbe Klub-ID haben */
    int SameCountry(int Player1, int Player2);
    /* Gleich 1, wenn die beiden Spieler dieselbe Land-ID haben */
    int TooBigStrengthDifference(int Player1, int Player2,
      int CriticalDifference);
    /* Liefert 1, wenn die Spielstärke mindestens die angegebene kritische Differenz
    erreicht */
    void GetSecondCriteriaAbsolutOfGroup(int aMacMahonGroup,
      TSecondCriteriaIndexOfGroup *SCIndexOG);
    int GetSecondCriteriaAbsolutOfPlayer(int APlayer);
    int  GetSecondCriteriaIndexOfPlayer(int  APlayer,
    TSecondCriteriaIndexOfGroup *SCIndexOG);
```

```
    /* Diese Funktion berechnet die relative Stärke eines Spielers innerhalb seiner
    MacMahon-Gruppe */
    int GetHighestGroupNumber();
    /* Liefert den höchsten im Turnier vorkommenden MacMahonScore */
    int GetLowestGroupNumber();
    /* Liefert den niedrigsten im Turnier vorkommenden MacMahonScore */
    int MMSDifference(int Player1, int Player2);
    /* Liefert die Differenz der MacMahonScores zweier Spieler */
    int GetBlack(int APlayer);
    /* Liefert die Anzahl der Gleichauf-Partien des Spielers mit Schwarz */
    int GetWhite(int APlayer);
    /* Liefert die Anzahl der Gleichauf-Partien des Spielers mit Weiß */
/* Stream-Funktionen: */
    static PTStreamable build();
protected:
    Tournament(StreamableInit);
    virtual Pvoid read(Ripstream is);
    virtual void write(Ropstream os);
private:
    virtual const Pchar streamableName() const { return
    "Tournament"; };
};
```

4 Maximum Weight Perfect Matching

4.1 Problemstellung

Definition (Matching): Gegeben sei ein Graph $G=(V,E)$. Als Matching in G bezeichnet man eine Teilmenge $M \subset E$, so daß M für alle $v \in V$ höchstens eine Kante mit v als einen Endpunkt enthält.

Definition (Perfect Matching): M heißt Perfect Matching in G, wenn M für alle $v \in V$ genau eine Kante mit v als einen Endpunkt enthält.

Anmerkung: Es kann nur dann ein Perfect Matching in G geben, wenn |V| gerade ist.

Problem: Gegeben sei ein Graph $G=(V,E)$ und eine Gewichtsfunktion $w(u,v)$ für Knoten u,v aus V. Gesucht ist eine Menge M^*, die ein Perfect Matching in G ist und die die Funktion

$$W(M) = \sum_{(u,v) \in M} w(u,v)$$

unter allen Perfect Matchings M in G maximiert.

4.2 Die Lösung von Maximum Weight Perfect Matching durch lineare Programmierung

4.2.1 Maximum Weight Perfect Matching als Integer Lineares Programm (ILP)

Zu gegebenem Graph $G=(V,E)$ und gegebener Gewichtsfunktion $w(u,v)$ für alle u,v aus V löst das folgende ILP das Problem Maximum Weight Perfect Matching, wenn es überhaupt ein Perfect Matching in G gibt:

$$\max \sum_{(u,v)\in E} w(u,v)x_{uv} \qquad (1)$$

unter den Nebenbedingungen

$$\sum_{(u,v)\in F(u)} x_{uv} = 1 \qquad \forall u \in V \qquad (2)$$

$$x_{uv} \geq 0 \qquad \forall (u,v) \in E \qquad (3)$$

$$x_{uv} \text{ integer} \qquad \forall (u,v) \in E \qquad (4)$$

Wobei $F(u)=\{(u,v)\in E | v\in V\}$ die Menge aller Kanten ausgehend von u ist.

Das ILP (1)-(4) oben beschreibt das Problem Maximum Weight Perfect Matching. Die Nebenbedingungen (2) bis (4) bewirken, daß x_{uv} ausschließlich die Werte 0 oder 1 annehmen kann. Damit ist eine Indikatorfunktion für das Matching M in G gegeben:

$x_{uv} = 1 \Leftrightarrow$ die Kante (u,v) ist in M.

Die Nebenbedingung (4) ist notwendig, da das Problem sonst fraktionale Lösungen haben kann. Die optimale Lösung des Beispiels in Abbildung 3.1 ohne Verwendung von (4) wäre $x_{12}=x_{28}=x_{83}=x_{37}=x_{71}=½$ und $x_{45}=x_{56}=x_{64}=½$ (alle anderen 0).

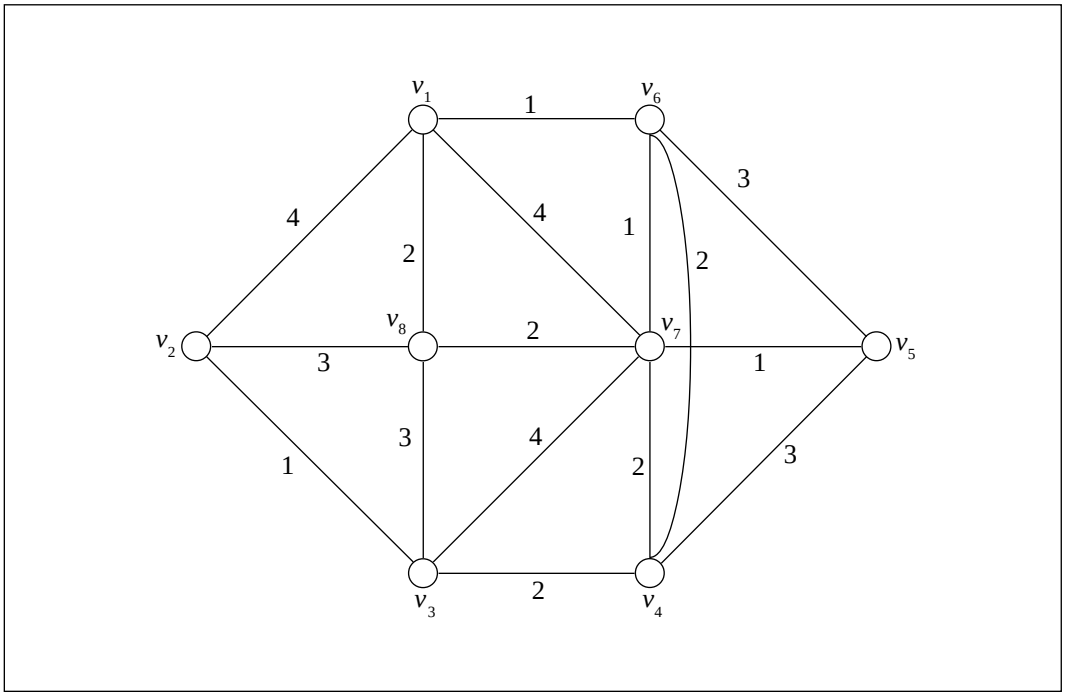

Abbildung 4.1

Die Nebenbedingung (2) bewirkt, daß jeder Knoten Endpunkt einer Kante im Matching in *G* ist, d.h. daß das gefundene Matching auch ein Perfect Matching in *G* ist.

Während lineare Programmierung durch polynomielle Algorithmen (z.B. Simplex) gelöst werden kann, ist die allgemeine Lösung von linearen Programmen mit einer Ganzzahligkeitsforderung wie in (4) np-hart, d.h. nicht effizient lösbar[1]. Edmonds wies Ende der 60er Jahre nach, daß die Nebenbedingung (4) der Ganzzahligkeit durch eine Klasse von Nebenbedingungen ersetzt werden kann, die ganzzahlige Lösungen des Problems impliziert.

4.2.2 Maximum Weight Perfect Matching als lineares Programm (LP)

Bei der Lösung des Problems durch ein lineares Programm nutzt man die Tatsache aus, daß fraktionale Lösungen innerhalb des Graphen nur in

[1] Wenn nicht P=NP gilt.

Zyklen mit einer ungeraden Anzahl von Knoten besser sein können als nicht-fraktionale Lösungen. Im folgenden LP fügt man daher für jede Teilmenge von V mit ungerader Kardinalität eine Nebenbedingung hinzu, die fraktionale Lösungen eines korrespondierenden, möglicherweise im Graphen vorhandenen Zyklus verhindert.

In einem Graphen mit n Knoten gibt es $N=2^{n-1}-n$ verschiedene Teilmengen von V mit ungerader Kardinalität und mehr als einem Knoten. Sei $S_1, S_2,..., S_N$ eine Aufzählung dieser Mengen. Die Kardinalität von S_i ergibt sich als $|S_i|=2s_i+1$ für eine natürliche Zahl s_i. Dann gilt folgendes Theorem:

Theorem 3.1 (Edmonds): Das Problem Maximum Weight Perfect Matching bei gegebenem Graph $G=(V,E)$ und gegebener Gewichtsfunktion $w(u,v)$ für u,v aus V ist äquivalent zu dem folgenden LP:

$$\max \sum_{(u,v)\in E} w(u,v)x_{uv} \tag{5}$$

unter den Nebenbedingungen

$$\sum_{(u,v)\in F(u)} x_{uv} = 1 \qquad \forall u \in V \tag{6}$$

$$x_{uv} \geq 0 \qquad \forall (u,v) \in E \tag{7}$$

$$\sum_{u,v\in S_k} x_{uv} \leq s_k \qquad k = 1,2,...,N \tag{8}$$

Zunächst scheint es unbefriedigend, daß man unter (8) exponentiell viele Nebenbedingungen hat. Es wird sich aber herausstellen, daß dies kein Problem darstellt, wenn man das duale Programm betrachtet, das darüberhinaus zu einem effizienten Algorithmus führt.

Das duale Programm zum LP (5)-(8) oben ist:

$$\min \sum_{v\in V} \alpha_v + \sum_{k=1}^{N} s_k \gamma_k \tag{9}$$

unter den Nebenbedingungen

$$\alpha_u + \alpha_v + \sum_{k:u,v \in S_k} \gamma_k \geq w(u,v) \qquad \forall u,v \in V \qquad (10)$$

$$\gamma_k \geq 0 \qquad k = 1,2,\ldots,N \qquad (11)$$

Die α's korrespondieren mit den Nebenbedingungen (6) und die γ's mit den Nebenbedingungen (8). Aus den primal-dualen Variablenpaaren ergeben sich die folgenden ergänzenden Bedingungen:

$$x_{uv} > 0 \Rightarrow \alpha_u + \alpha_v + \sum_{k:u,v \in S_k} \gamma_k = w(u,v) \quad \forall u,v \in V \qquad (12)$$

$$a_u > 0 \Rightarrow \sum_{v \in V} x_{uv} = 1 \qquad \forall u \in V \qquad (13)$$

$$\gamma_k > 0 \Rightarrow \sum_{u,v \in S_k} x_{uv} = s_k \qquad k = 1,2,\ldots,N \qquad (14)$$

Die Bedingungen (12) werden im folgenden auch Kanten-Bedingungen, (13) Knoten-Bedingungen und (14) Blossom-Bedingungen genannt. Sind die ergänzenden Bedingungen und alle Nebenbedingungen des dualen und des primalen Programms erfüllt, maximiert die Belegung der Variablen x_{uv} die Zielfunktion des primalen Programms.

Die exponentiell vielen Bedingungen (14) brauchen nicht alle von dem Algorithmus berücksichtigt zu werden, da nur linear viele Variablen γ_k ungleich 0 sein können. Es reicht aus, sich nur diejenigen zu merken, deren korrespondierender Zyklus ungerader Länge in *G* im Verlauf des Algorithmus tatsächlich auftritt.Alle Knoten eines solchen Zyklus werden dann im Graphen in einem Pseudoknoten identifiziert, wodurch sich die Zahl der Knoten in *G* um mindestens 2 vermindert. Es können daher höchstens |*V*|/2 Zyklen gleichzeitig während des Algorithmus eine Dualvariabel ungleich 0 haben.

4.2.3 Die Grundidee zu einem effizienten Algorithmus

Aus dem primalen und dem dualen Programm, sowie den ergänzenden Bedingungen läßt sich die folgende Grundidee eines effizienten Algorithmus formulieren:

Ausgehend von dem Null-Matching (d.h. die zu bildende Menge M^* ist zunächst leer, also $x_{uv}=0$ für alle $(u,v)\in E$) und Dualvariablen

$$\alpha_u = \frac{1}{2}\max_{(u,v)\in E}(w(u,v)) \qquad \forall u \in V \qquad (15)$$

$$\gamma_k = 0 \qquad k = 1,2,\ldots,N \qquad (16)$$

wird die Menge M^* sukzessive erweitert, so daß M^* zu jedem Zeitpunkt der Iteration die Zielfunktion des primalen Programms unter allen Matchings M mit gleicher Kardinalität maximiert.

Das Null-Matching erfüllt alle Nebenbedingungen des primalen (außer (6)) und des dualen Programms sowie die Kanten-Bedingungen (12) und die Blossom-Bedingungen (14). Dies wird sich im Verlaufe des Algorithmus nicht ändern. Lediglich die Knoten-Bedingungen (13) sind nicht erfüllt. Bei jedem Hinzufügen einer Kante zu M^* werden aber zwei weitere Knoten ihre Knoten-Bedingung erfüllen, und diese bleiben bis zur Terminierung des Algorithmus erfüllt.

Wenn alle ergänzenden Bedingungen (12)-(14) erfüllt sind, terminiert der Algorithmus, und das gefundene Matching M^* maximiert die Zielfunktion des primalen Programms (die Korrektheit ist durch die Konstruktion des primal-dualen Algorithmus gegeben).

Kern des Algorithmus ist die Funktion MAPS (M-Augmenting-Path-Search), die zwei freie Knoten in $G'\subset G$ sucht, durch deren Einbeziehung das Matching erweitert werden kann. Dabei dürfen nur Kanten $(u,v)\in G$ berücksichtigt werden, für die die folgende Gleichung gilt:

$$\alpha_u + \alpha_v + \sum_{k:u,v\in S_k} \gamma_k = w(u,v) \qquad (17)$$

Würde die Kante (u,v) dem aktuellen Matching hinzugefügt, wäre entsprechend $x_{uv}>0$. Indem man nur Kanten berücksichtigt, die (17) erfüllen, stellt man sicher, daß die Kanten-Bedingungen (12) nie verletzt werden.

Der Teilgraph G' von $G=(V,E)$, in dem die Funktion MAPS einen Pfad zwischen zwei freien Knoten sucht, wird daher wie folgt definiert:

$$G'=(V,E'), \text{ wobei } E'=\{(u,v)\in E \mid \alpha_u + \alpha_v + \sum_{k:u,v\in S_k} \gamma_k = w(u,v)\} \quad (18)$$

Findet die Funktion einen Pfad, so kann entsprechend eine Kante dem aktuellen Matching hinzugefügt werden. Ist die Suche ergebnislos, so müssen die Dualvariablen so geändert werden, daß alle Nebenbedingungen und alle bereits erfüllten ergänzenden Bedingungen erfüllt bleiben, aber weitere Kanten G' hinzugefügt werden, so daß MAPS weitere Möglichkeiten für die Suche nach einem Pfad zwischen zwei freien Knoten erhält. Existiert in G kein Perfect Matching, könnten zu einem bestimmten Zeitpunkt der Iteration keine weiteren Kanten mehr zu G' hinzugefügt werden, ohne eine der ergänzenden Bedingungen zu verletzen. Man kann den hier vorgestellten Algorithmus aber dahingehend erweitern, daß er ein Maximum Weight Maximum Cardinality Matching findet (siehe [Gib85]).

4.2.4 MAPS - die Suche nach einem Pfad zwischen zwei freien Knoten

Die Funktion MAPS sucht in G' von einem zuvor ausgewählten freien Knoten in G einen sogenannten M-Augmenting-Path, also einen Pfad zwischen zwei freien Knoten, so daß anschließend die Zahl der Kanten im gesuchten Matching um eine Kante erhöht werden kann. Diese Pfade in G' können dabei auch Kanten enthalten, die bereits im aktuellen Matching enthalten sind.

Die M-Augmenting-Path-Suchprozedur MAPS.

procedure MAPS(G,T)

1. Wähle einen outer Knoten $u \in T$ und eine Kante $(u,v) \in G$, die noch nicht vorher erkundet wurde. Kennzeichne (u,v) als erkundet. Existiert kein outer Knoten in T mit einer nicht-erkundeten Kante *goto* H.
2. Wenn v ein freier Knoten ist, füge die Kante (u,v) zu T hinzu. Der Pfad von der Wurzel von T bis zu v ist ein M-Augmenting-Path in G. *goto* A.
3. Wenn v outer ist, hat MAPS einen Blossom in G gefunden. *goto* B.
4. Wenn v inner ist, *goto* 1.
5. Sei (v,w) die Kante in M mit v als einen Endpunkt. Füge (u,v) und (v,w) zu T hinzu. Kennzeichne v mit dem Label inner und w mit dem Label outer. *goto* 1.

Die Label A,B und H sind verschiedene Ausgänge aus der Prozedur.

Ausgang A kennzeichnet die erfolgreiche Suche nach einem M-Augmenting-Path. Nach dem Vertausch der Rollen der Kanten kann ein neuer freier Knoten gewählt werden und ein neuer Suchbaum aufgebaut werden.

Ausgang H bedeutet, daß kein M-Augmenting-Path in G existiert. Durch eine Veränderung der Dualvariablen werden weitere Kanten zu G hinzugefügt, bevor die Suche wiederholt werden kann.

Ausgabe B bedeutet, daß ein Blossom (Zyklus ungerader Länge) gefunden wurde. Dieser muß aus dem Graphen entfernt werden, bevor eine Suche fortgesetzt werden kann.

Definition: Ein M-Augmenting-Path in $G=(V,E)$ zu gegebenem Matching M ist ein Pfad ungerader Länge zwischen zwei freien Knoten u und v in G, dessen Kanten abwechselnd in M und nicht in M sind.

Wird ein solcher M-Augmenting-Path in G' gefunden, vertauscht man die Rollen der Kanten dieses Pfades. Alle Kanten des Pfades in M^* werden aus M^* entfernt und alle Kanten nicht in M^* werden zu M^* hinzugefügt. Da die Endpunkte des Pfades freie Knoten waren, wird dadurch die Kardinalität des gegenwärtig aktuellen Matchings M^* um eins erhöht. Diese beiden Knoten erfüllen dann ihre Knoten-Bedingungen (13). Da für alle Kanten in G' (17) gilt, werden die Kanten-Bedingungen (12) weiterhin alle erfüllt, wodurch die Optimalität unter allen Matchings in G mit derselben Kardinalität gesichert ist.

In dem Beispiel in Abbildung 4.2 sind $[v_4,v_3,v_8,v_1]$, $[v_4,v_3,v_8,v_2]$ und $[v_4,v_3,v_8,v_7,v_5,v_6]$ M-Augmenting-Paths.

MAPS baut ausgehend von einem freien Knoten einen Suchbaum auf. Stößt MAPS auf einen freien Knoten, so ist der gefundene Pfad zwischen der Wurzel und dem freien Knoten ein M-Augmenting-Path, und die Rollen der Kanten des Pfades in M können vertauscht werden, bevor die

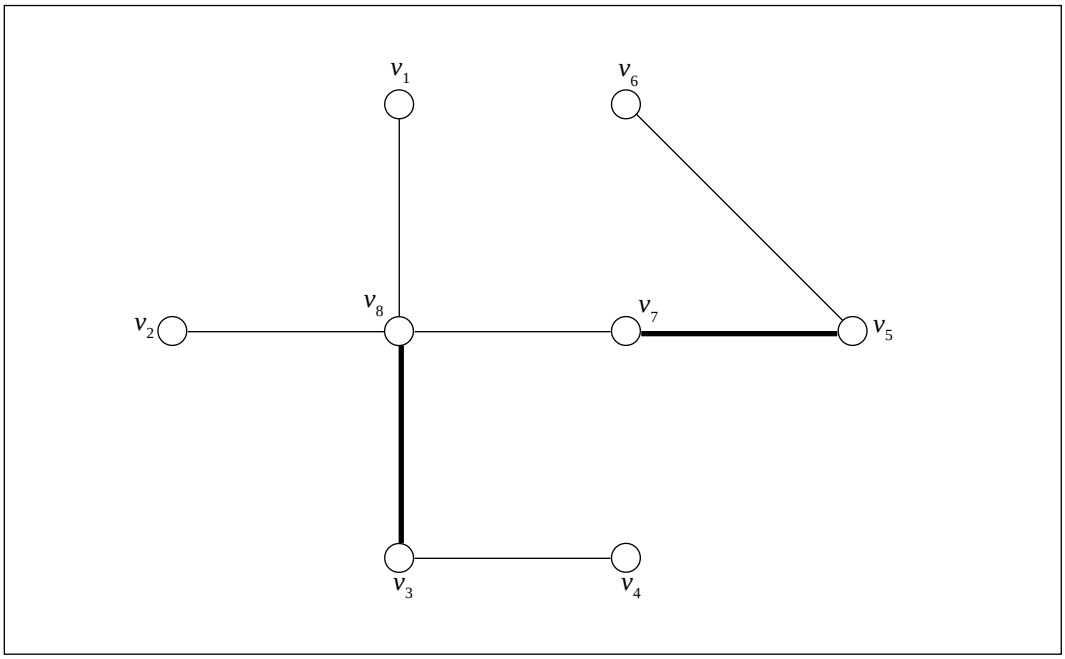

Abbildung 4.2, die Kanten in M sind fett dargestellt.

Suche von einem anderen freien Knoten ausgehend mit einem neuen Suchbaum durchgeführt werden kann.

Stößt MAPS auf einen Knoten, der bereits Endpunkt einer Kante in M ist, so wird der Suchbaum um zwei Kanten erweitert, und die Struktur eines alternierenden Pfades von der Wurzel zu den Blättern bleibt erhalten. Alle Knoten im Suchbaum, von denen aus die Suche nach freien Knoten fortgesetzt werden kann, werden mit outer bezeichnet, alle anderen Knoten mit inner. Abbildung 4.3 zeigt einen Suchbaum, den MAPS ausgehend von v_4 erzeugen würde. Allerdings würde MAPS bereits nach dem Erreichen einer der drei freien Knoten v_1, v_2 und v_6 terminieren, und das Matching könnte gemäß des gefundenen Pfades modifiziert und erweitert werden.

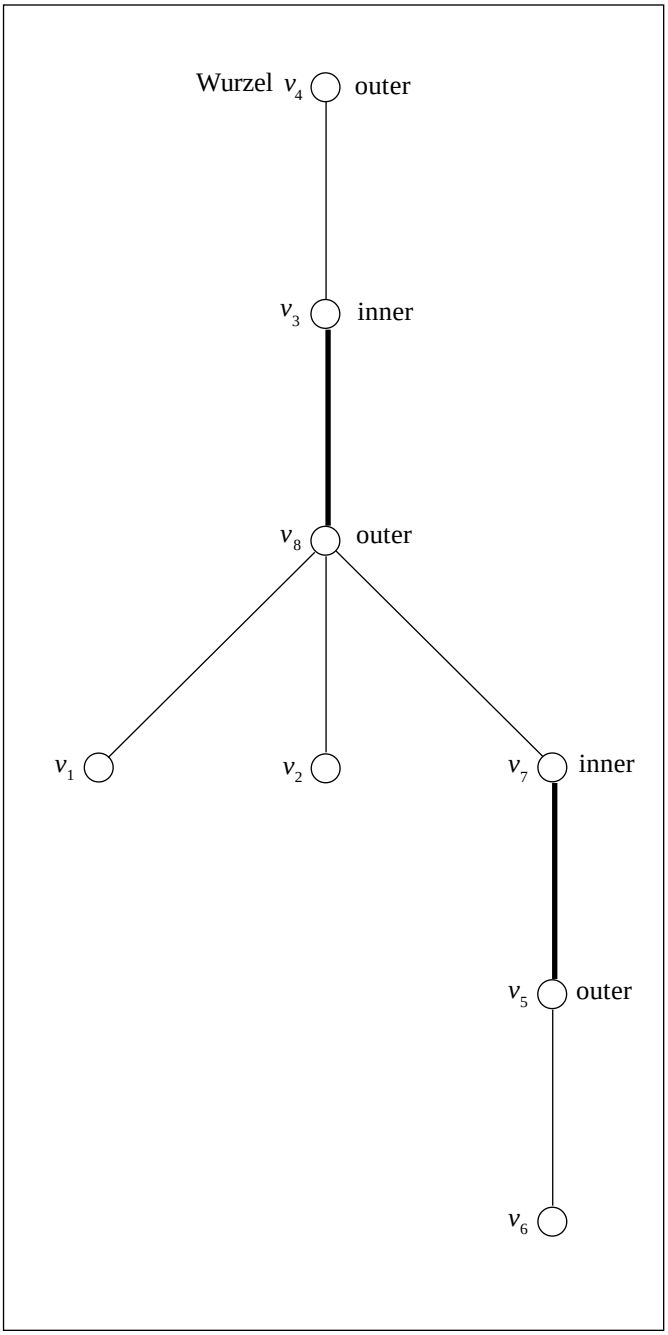

Abbildung 4.3, Kanten in M fett dargestellt.

MAPS sucht in G' nicht nur nach einem M-Augmenting-Path, sondern erkennt auch Zyklen ungerader Länge, sogenannte Blossoms, die mit einer der Mengen S_k aus Abschnitt 4.2.2 korrespondieren. Der folgende Abschnitt gibt an, wie Blossoms behandelt werden.

4.2.5 Blossoms

Blossoms sind Zyklen ungerader Länge innerhalb des Teilgraphen *G'* von *G*, in dem MAPS einen M-Augmenting-Path sucht. Nicht alle Blossoms in *G'* sind ein Problem. Solange M-Augmenting-Paths gefunden werden können, kann das Matching erweitert werden. Die Existenz von Blossoms kann aber dazu führen, daß *G'* zwar einen M-Augmenting-Path enthält, die Existenz eines Blossoms aber verhindert, daß dieser von der Prozedur MAPS gefunden wird.

Der Graph in Abbildung 4.4(a) hat den M-Augmenting-Path $[v_1,v_3,v_2,v_4]$. Der von MAPS aufgebaute Suchbaum in Abbildung 4.4 (b) findet den gesuchten Pfad aber nicht, da v_2 als inner markiert wurde und der Suchbaum daher von diesem Knoten nicht erweitert werden kann. Hätte MAPS den Suchbaum zunächst nach v_3 erweitert, wäre die Suche erfolgreich gewesen.

Würde man MAPS so implementieren, daß die Funktion grundsätzlich beide Möglichkeiten überprüfen würde, würde dies auf eine Enumeration mit exponentieller Laufzeit hinauslaufen. Dies wird in dem hier

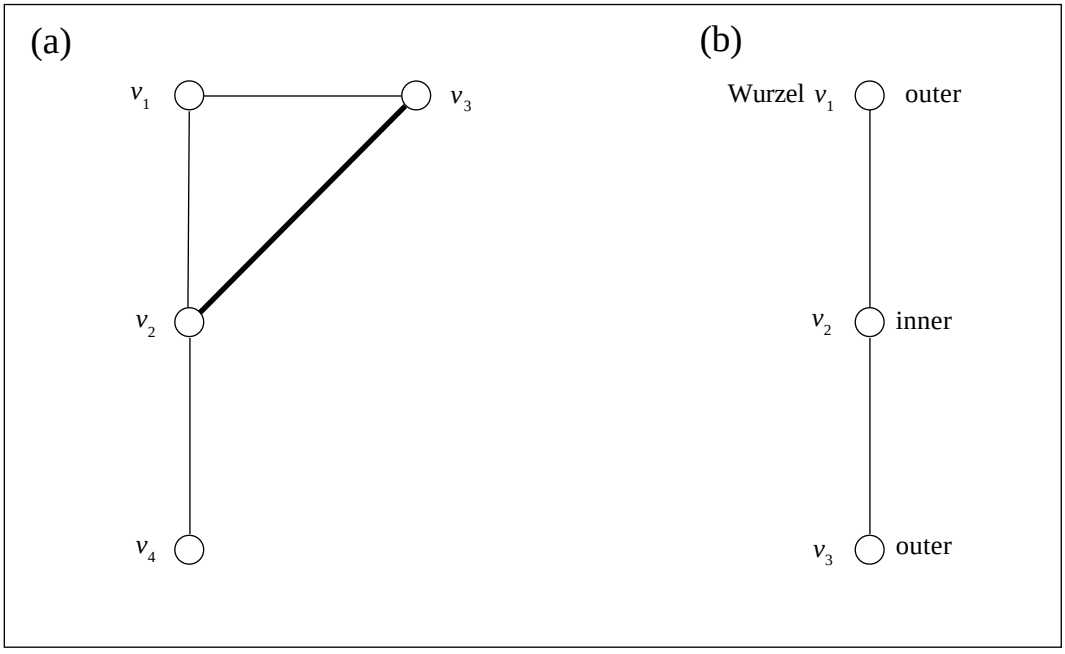

Abbildung 4.4

vorgestellten Algorithmus dadurch umgangen, daß man den gefundenen Blossom im Graphen und im Suchbaum durch einen Pseudoknoten ersetzt, der stellvertretend für alle Knoten innerhalb des Blossoms steht. Da der Suchbaum im Prinzip von jedem Knoten des Blossoms erweitert werden kann (jeder Knoten des Blossoms kann als outer markiert werden), wird der resultierende Pseudoknoten als outer markiert.

Erkennung eines Blossoms:

Die Existenz eines Blossoms erkennt MAPS daran, daß es eine Kante zwischen zwei Knoten u und v im Suchbaum gibt, die beide als outer gekennzeichnet worden sind. Sei P_u der Pfad von der Wurzel des Suchbaums bis u und P_v der Pfad von der Wurzel bis v und w der letzte gemeinsame Knoten der Pfade P_u und P_v. Dann enthält der Blossom alle Knoten der Pfade P_u und P_v ab dem Knoten w und den Knoten w selbst.

Ersetzen eines Blossoms durch einen Pseudoknoten:

Sei B die Menge aller Knoten des Blossoms und $G=(V,E)$ der zu ändernde Graph. Dann ist $G^1=(V^1,E^1)$ der Graph, in dem der Blossom B zu dem Pseudoknoten v_B „geschrumpft“ wurde, wobei

$$V^1=(V\setminus B)\cup v_B$$

$$E^1=\{(u,v)\in E|u\notin B\wedge v\notin B\}\cup\{(u,v_B)|(u,w)\in E\wedge w\in B\wedge u\notin B\}$$

D.h. alle Knoten in B werden in dem Pseudoknoten v_B identifiziert. Dabei können die Knoten in B auch Pseudoknoten sein, die in vorhergehenden Iterationen Blossoms ersetzt haben. Als Attribut des Pseudoknoten v_B merkt man sich auch die genaue zyklische Folge der Knoten in dem Blossom. Dies ist notwendig, um zu einem späteren Zeitpunkt den Pseudoknoten in dem Graphen durch den Blossom zu ersetzen.

Bemerkung: Innerhalb eines Blossoms mit n Knoten sind n-1 Knoten bereits Endpunkte einer Kante im aktuellen Matching und alle Partner dieser Knoten sind Bestandteil desselben Blossoms. Um alle Knoten-

Bedingungen der Knoten in dem Blossom erfüllen zu können, muß ein Knoten des Blossoms noch mit einem anderen freien Knoten gematched werden. Stellvertretend für diesen Knoten wird ein Partner für den Pseudoknoten gesucht, der daher als outer markiert wird.

Der aktuelle Suchbaum muß nicht verworfen werden, wenn ein Blossom geschrumpft wird. In dem Baum wird der Teilbaum, der den Blossom repräsentiert, in gleicher Weise durch einen Pseudoknoten ersetzt, wie dies im Falle des Graphen praktiziert wird. Abbildung 4.5(a) zeigt einen Suchbaum, in dem durch die Kante (v_4,v_5) in G gerade ein Blossom aufgetreten ist, und Abbildung 4.5(b) zeigt den Suchbaum nach dem Ersetzen des Teilbaumes durch den entsprechenden Pseudoknoten. Bei der Erweiterung des Suchbaums können jetzt auch Kanten ausgehend von den ursprünglich als inner markierten Knoten v_2 und v_3 berücksichtigt werden. Auf diese Weise wird sichergestellt, daß MAPS immer einen M-Augmenting-Path findet, wenn in G ein solcher existiert.

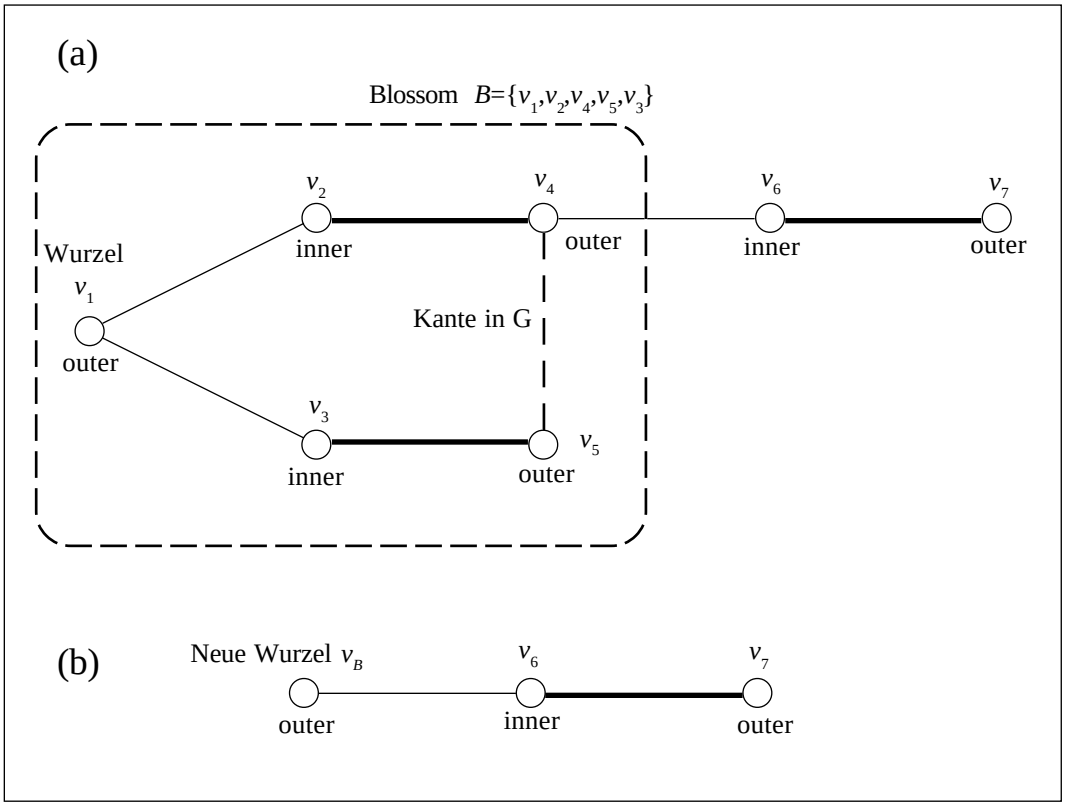

Abbildung 4.5

Expandieren von Blossoms:

Es gibt zwei Situationen, in denen Pseudoknoten wieder zu Zyklen ungerader Länge expandiert werden:

1. Wenn es in dem Graphen keine freien Knoten mehr gibt, erfüllen alle Knoten ihre ergänzende Bedingung (13). Dann terminiert der Algorithmus und expandiert alle verbleibenden Blossoms von außen nach innen, d.h. zuerst diejenigen, die in keinem anderen Blossom enthalten sind.

2. Wenn ein Blossom B_1 irgendwann in dem aktuellen Suchbaum als inner markiert wurde und MAPS auf keinem anderen Weg einen M-Augmenting-Path finden kann, wird der Blossom expandiert. Da alle Knoten des Blossom B_1 in dem Suchbaum sowohl als outer als auch als inner markiert sein könnten (bei expandiertem Blossom), ist es möglich, daß eine Teilmenge des Blossoms mit anderen Knoten des Suchbaumes wieder einen neuen Blossom B_2 bilden würde. Dieser Blossom kann von MAPS nur gefunden werden, wenn B_1 zuvor expandiert wird. Würde man dies nicht tun, würde MAPS möglicherweise einen existierenden M-Augmenting-Path in G übersehen.

Verfahren: Sei M das aktuelle Matching in $G=(V,E)$, $B=\{v_1,v_2,...,v_n\}$ der zu expandierende Blossom zu dem Pseudoknoten v_B und $u \in V$, so daß $(u,v_B) \in M$. Der Knoten u existiert immer, da der Blossom nur expandiert wird, wenn es keinen freien Knoten mehr gibt, also auch v_B nicht frei ist, oder wenn er in einem Suchbaum als inner markiert wird, was nur geschehen kann, wenn er mit einem Knoten gematched ist.

Beim Expandieren des Blossoms wird der entsprechende Pseudoknoten mit allen Kanten aus dem Graphen entfernt, und alle Knoten in dem Blossom werden mit den entsprechenden Kanten zu den anderen Knoten in G wieder hinzugefügt.

Anschließend muß das aktuelle Matching M ggf. noch modifiziert werden. In dem Blossom B sind n-1 Knoten bereits mit anderen Knoten aus B gematched. O.B.d.A. sei v_1 der Knoten aus B, der noch keinen Partner hat. Es müssen zwei Fälle unterschieden werden:

Fall 1: $(u,v_1) \in E$. Dann wird die Kante (u,v_1) in M aufgenommen, und alle Knoten des Blossoms und der Knoten u erfüllen ihre Knoten-Bedingung (13).

Fall 2: $(u,v_1) \notin E$. Da die Kante (u,v_B) in G enthalten war, muß es im ursprünglichen Graphen (d.h. bevor der Blossom B geschrumpft wurde) eine Kante (u,v_i) geben $(v_i \neq v_1)$. Innerhalb des Blossoms gibt es genau zwei Pfade zwischen v_i und v_1, einen gerader und einen ungerader Länge. Sei $[v_i,v_{j1},...,v_{jm},v_1]$ der Pfad gerader Länge zwischen v_i und v_1 in B. Dann ist der Pfad $[u,v_i,v_{j1},...,v_{jm},v_1]$ ein M-Augmenting-Path in G. Die Rollen der

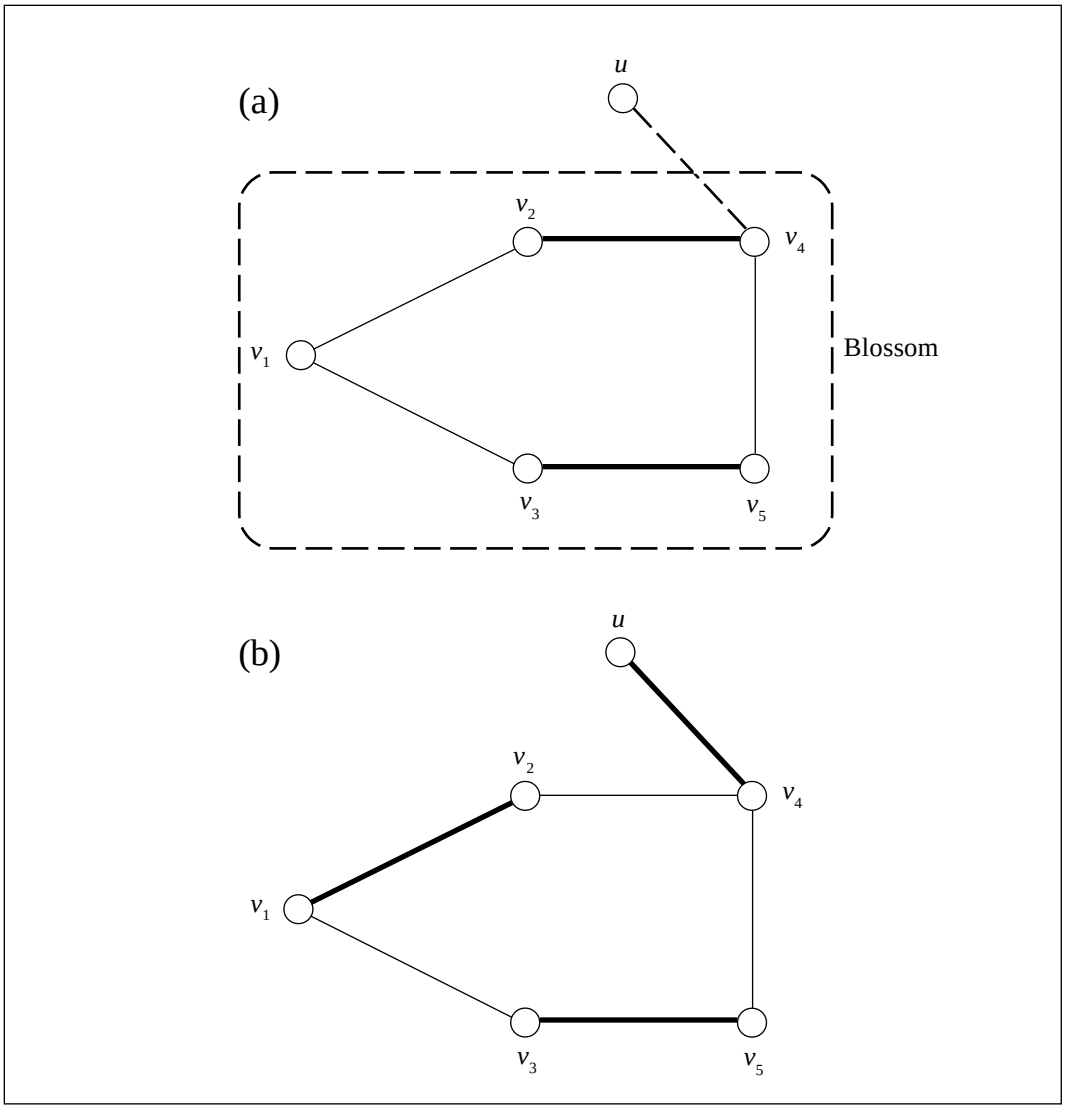

Abbildung 4.6

Kanten auf diesem Pfad in *M* werden ausgetauscht, und man erhält das gültige Matching.

Beispiel: In Abbildung 4.6(a) hat der abgebildete Blossom eine Kante zu dem Knoten *u*, die auf die Kante (u,v_4) in dem ursprünglichen Graphen zurückgeht. Die Kante (u,v_4) kann nicht einfach in das aktuelle Matching aufgenommen werden, da gegenwärtig noch die Kante (v_4,v_2) in *M* ist. Gemäß Fall 2 oben bildet $[u,v_4,v_2,v_1]$ einen M-Augmenting-Path, und (u,v_4) und (v_2,v_1) werden in *M* aufgenommen und (v_4,v_2) aus *M* entfernt. Abbildung 4.6(b) zeigt den resultierenden Teilgraphen. Die fett dargestellten Kanten sind in *M*.

Wenn MAPS trotz der Elimination von Blossoms, d.h. Zyklen ungerader Länge, keinen M-Augmenting-Path in *G* finden kann, müssen die Dualvariablen derart modifiziert werden, daß anschließend noch alle Nebenbedingungen des primalen (außer (6)) und des dualen Programms und alle vor der Änderung gültigen ergänzenden Bedingungen (12)-(14) weiterhin gültig bleiben, der Graph aber so modifiziert wird, daß MAPS weitere Möglichkeiten erhält, den Suchbaum zu erweitern. Dies wird im folgenden Abschnitt erläutert.

4.2.6 Änderung der Dualvariablen

Die Ausgangssituation ist folgende:

Der aktuelle Graph sei $G'=(V,E')$ und der aktuelle Suchbaum *T*. Alle Knoten aus *G*', die auch in *T* sind, sind entweder als inner oder als outer markiert. Der Suchbaum kann nur auf zwei Arten modifiziert werden, so daß MAPS als Folge entweder einen Blossom finden kann oder Knoten zu dem Suchbaum hinzufügen kann, die vorher noch nicht berücksichtigt werden konnten, da sie nicht in dem aktuellen Graphen *G*' enthalten waren.

Notation: In diesemAbschnitt werden alle Knoten des zugrundeliegenden Graphen *G*, die in *G*' als outer markiert sind oder in einem Pseudoknoten enthalten sind, der als outer markiert ist, als outer bezeichnet und entsprechend werden alle Knoten, die in *G*' als inner markiert sind oder in einem Pseudoknoten enthalten sind, der als inner markiert ist, als inner bezeichnet. Pseudoknoten werden gesondert behandelt.

4.2.6.1 Hinzufügen weiterer Kanten aus dem zugrundeliegenden Graphen *G*=(*V*,*E*) in *G*'

Es hat nur Sinn, Kanten zu *G*' hinzuzufügen, die als einen Endpunkt einen Knoten haben, der als outer markiert ist oder in einem Blossom enthalten ist, der als outer markiert ist. Nur von solchen Knoten aus kann der Suchbaum durch die Funktion MAPS erweitert werden. Knoten, die in einem Blossom enthalten sind, müssen auch berücksichtigt werden, da eine zusätzliche Kante des Knotens auch eine zusätzliche Kante des Blossoms zu anderen Knoten des Graphen impliziert. Kanten zwischen zwei Knoten in demselben Blossom führen allerdings nicht zu weiteren Möglichkeiten für die Funktion MAPS und werden daher nicht in Betracht gezogen.

Für alle Kanten (*u*,*v*) in *G* gilt zu jedem Zeitpunkt:

$$\alpha_u + \alpha_v + \sum_{k:u,v\in S_k} \gamma_k = w(u,v) + \delta_{uv} \qquad \delta_{uv} \geq 0 \tag{19}$$

Würde α_u und α_v zusammen um δ_{uv} verringert, würde die Gleichung (17) gelten und die Kante (*u*,*v*) könnte dem aktuellen Graphen *G*' hinzugefügt werden. Es muß dabei sichergestellt werden, daß für alle Kanten des Suchbaums die Gleichung (17) auch weiterhin gilt.

Es müssen zwei Fälle unterschieden werden. O.B.d.A. sei *u* als outer markiert.

Fall 1: *v* ist unmarkiert. In diesem Fall wird nur α_u geändert, aber um den vollen Betrag von δ_{uv}.

Knoten, die als outer markiert sind, können in *G*' ausschießlich Kanten zu Knoten haben, die als inner markiert sind oder zu Knoten, die in

demselben Pseudoknoten enthalten sind. Wenn es Kanten zu unmarkierten Knoten gäbe, wäre der Suchbaum von MAPS erweitert worden, und bei Kanten zu Knoten, die als outer markiert sind und nicht in demselben Pseudoknoten sind, hätte MAPS einen Blossom entdeckt, und dieser wäre durch einen Pseudoknoten ersetzt worden.

Damit alle Kanten zwischen als inner und outer markierten Knoten des Suchbaumes erhalten bleiben, müssen die Dualvariablen aller als outer markierten Knoten um δ_{uv} verringert werden und die Dualvariablen aller als inner markierten Knoten um δ_{uv} vergrößert werden. So ist gewährleistet, daß die Gleichung (17) für alle Knoten des Suchbaumes erhalten bleibt.

Jetzt muß noch sichergestellt werden, daß alle Kanten innerhalb von Blossoms durch die Änderung der Dualvariablen unverändert bleiben. Die Dualvariablen aller Knoten eines Pseudoknotens, der als outer markiert ist, werden um δ_{uv} vermindert. Damit die Gleichung (17) dennoch erhalten bleibt, muß die Dualvariable γ_k des Pseudoknotens um $2\delta_{uv}$ erhöht werden. Eine entsprechende Überlegung führt dazu, daß die Dualvariable von allen Pseudoknoten, die als inner markiert sind und nicht in anderen Pseudoknoten enthalten sind, um $2\delta_{uv}$ vermindert werden muß.

Jetzt muß nur noch darauf geachtet werden, daß die Nicht-Negativitätsbedingungen des dualen Programms nicht verletzt werden und die Gleichungen (19) für alle Kanten gültig bleiben.

Wird die Dualvariable eines normalen Knotens kleiner Null, so bedeutet dies, daß es kein Perfect Matching in G gibt. Der Algorithmus bricht mit einer entsprechenden Fehlermeldung ab.

Es muß verhindert werden, daß die Dualvariable eines Pseudoknotens kleiner Null wird. Dies kann nur bei Pseudoknoten auftreten, die als inner markiert sind. Ist δ_{uv} so groß, daß dieser Fall eintritt, wird gemäß Abschnitt 4.2.6.2 vorgegangen und der entsprechende Pseudoknoten expandiert.

Um zu verhindern, daß die Änderung der Dualvariablen für ein Knotenpaar die Gleichung (19) verletzt, muß unter allen in Frage kommenden Kanten (u,v) diejenige ausgewählt werden, die den kleinsten Wert δ_{uv} hat.

Fall 2: *v* ist als outer markiert. Völlig analog zu Fall 1 werden die Dualvariablen geändert. Jetzt reicht es aber aus, die Dualvariablen aller als outer markierten Knoten um $\frac{1}{2}\delta_{uv}$ zu ändern, da dadurch bereits die Kante (*u*,*v*) in *G*' eingefügt wird. Alles Weitere analog zu Fall 1.

4.2.6.2 Expandieren eines als inner markierten Pseudoknotens

Würde das Hinzufügen einer Kante nach Abschnitt 3.2.6.1 dazu führen, daß die Dualvariable eines (als inner markierten) Pseudoknotens kleiner als Null würde, wird die Änderung der Dualvariablen gemäß der Vorschrift aus Abschnitt 4.2.6.1 durchgeführt, aber mit dem folgenden Wert:

$\delta = \frac{1}{2}\min\{\gamma_k | S_k$ entspricht einem als inner markierten Pseudoknoten, der nicht in einem anderen Pseudoknoten enthalten ist$\}$

Der entsprechende Pseudoknoten wird expandiert, und MAPS setzt seine Suche nach einem M-Augmenting-Path fort.

4.2.6.3 Die δ-Berechnungsfunktion

Die folgende Funktion berechnet den Wert δ, um den die Dualvariablen geändert werden:

$\delta_1 = \frac{1}{2}\min\{\gamma_k | S_k$ entspricht einem als inner markierten Pseudoknoten, der nicht in einem anderen Pseudoknoten enthalten ist$\}$

$\delta_2 = \min\{\alpha_u - \alpha_v - w(u,v) | (u,v) \in E$, *u* ist als outer markiert, *v* ist nicht markiert$\}$

$\delta_3 = \frac{1}{2}\min\{\alpha_u - \alpha_v - w(u,v) | (u,v) \in E$, *u* und *v* sind als outer markiert, aber nicht in demselben Pseudoknoten enthalten$\}$

$\delta = \min\{\delta_1, \delta_2, \delta_3\}$

4.2.6.4 Die Dualvariablen-Modifikationsfunktion

Nachdem δ gemäß Abschnitt 4.6.2.3 berechnet wurde, werden die Dualvariablen gemäß folgender Vorschrift modifiziert:

1. Die Dualvariablen aller normalen Knoten, die als outer markiert sind, oder in einem Pseudoknoten enthalten sind, der als outer markiert ist, werden um δ vermindert.

2. Die Dualvariablen aller normalen Knoten, die als inner markiert sind, oder in einem Pseudoknoten enthalten sind, der als inner markiert ist, werden um δ vergrößert.

3. Die Dualvariablen aller als outer markierten Pseudoknoten, die nicht in anderen Pseudoknoten enthalten sind, werden um 2δ vergrößert.

4. Die Dualvariablen aller als inner markierten Pseudoknoten, die nicht in anderen Pseudoknoten enthalten sind, werden um 2δ vermindert.

4.2.7 Der Maximum-Weight-Perfect-Matching-Algorithmus

Der angebene Algorithmus geht von einem vollständigen Graphen als Grundlage aus. Gegeben ist ferner eine Gewichtsfunktion $w(u,v)$, die zwischen allen Knoten des Graphen definiert sein muß. Kanten können durch das Gewicht Null verboten werden. Der Algorithmus ist auf Seite 63 angegeben.

4.3 Die Implementation des Maximum-Weight-Perfect-Matching-Algorithmus

Der Algorithmus wurde in Borland C++/Windows implementiert. Der eigentliche Maximum-Weight-Perfect-Matching-Algorithmus nutzt aber keine Windows-Funktionen aus, so daß er relativ einfach auf andere Plattformen portiert werden kann. Die Module enthalten auch Funktionen zur Anzeige von Zwischenzuständen des Algorithmus unter Windows,

Der Maximum-Weight-Perfect-Matching-Algorithmus:

```
   G'=(V,∅);
   M=∅;
   for alle Knoten u in G'
     α_u=½max{w(u,v)|(u,v)∈E};
   for alle Kanten (u,v) in G
     if α_u+α_v=w(u,v)
       Füge die Kante (u,v) in G' ein;
C: for alle Knoten in G'
     lösche alle Markierungen;
   for alle Kanten in G'
     markiere die Kanten als nicht erkundet;
   wähle einen freien Knoten v aus G'. Gibt es keinen
   solchen Knoten, goto L;
   MakeEmpty(T);
   Root(T)=v;
   markiere v als outer;
M: MAPS(G',T)
A: identifiziere den M-Augmenting-Path in T, vertausche
   die Rollen der Kanten in M;
   goto C;
B: identifiziere den Blossom und ersetzte ihn durch einen
   Pseudoknoten;
   weise der Dualvariable des Pseudoknotens den Wert 0 zu;
   goto M;
H: berechne δ gemäß Abschnitt 3.2.6.3;
   modifiziere die Dualvariablen gemäß Abschnitt 3.2.6.4;
   for alle als inner markierte oder in einem als inner
   markierten Pseudoknoten enthaltenen Knoten
     Lösche alle Kanten von diesem Knoten zu als inner
     markierten oder unmarkierten Knoten in G';
   if (δ=δ_1)
     begin
       expandiere alle als inner markierten Pseudoknoten,
       deren Dualvariable Null ist;
       goto M;
     end
   if (δ=δ_2 or δ=δ_3)
     Füge die entsprechende Kante zu G' hinzu;
   goto M;
L: Expandiere solange denjenigen Pseudoknoten, der in
   keinem anderen Pseudoknoten enthalten ist, bis alle
   Pseudoknoten expandiert sind gemäß Abschnitt 3.2.5;
```

die zur Kontrolle eingebaut wurden (in den hier abgedruckten Header-Dateien nicht mit aufgeführt).

Der Algorithmus setzt sich aus drei Modulen zusammen.

Das Modul **tree** stellt den abstrakten Datentyp Baum zur Verfügung, enthält aber auch einige Attribute, die speziell von anderen Funktionen desAlgorithmus genutzt werden. So werden die beiden Knoten, die einen Blossom in dem Baum entstehen lassen als Attribut des Baumes zwischengespeichert.

Das Modul **graph** stellt den abstrakten Datentyp Graph in einer stark an die Anforderungen des Algorithmus angepaßten Form zur Verfügung. Dabei werden die wichtigen Funktionen des Schrumpfens und Expandierens von Blossoms bereits mit angeboten. Diese Funktionen greifen auch auf den aktuellen Suchbaum zu und modifizieren ihn so, daß der eigentliche Algorithmus den Suchbaum nicht verwerfen muß.

Das Modul **algorithm** letztlich stellt die Schnittstelle nach außen her und führt den Algorithmus inklusive der Funktion MAPS und der Dualvariablenberechnung durch. Die Eingabe ist die Gewichtsmatrix des (vollständigen) Graphen, für den das Matching durchgeführt werden soll. Als Ausgabe wird jedem Knoten des Graphen die Referenz seines Partners in dem gefundenen Maximum Weight Perfect Matching zugewiesen.

4.3.1 Das Modul TREE

Der abstrakte Datentyp Baum ist vollständig dynamisch implementiert worden. Zugriff auf die Kinder erfolgt über das erste Kind, das auf seinen nächsten Nachbarn zeigt usw.

```
/* Tree.h */
/* Headerdatei für den abstrakten Datentyp Baum */

struct TKnoten
{
   int Label;
   /* Referenz auf den entsprechenden Knoten im aktuellen Graphen */
   TKnoten *Parent;
   TKnoten *FirstChild;
   TKnoten *PreviousSibling;
   TKnoten *NextSibling;
};

class TTree
{
public:
   /* Datenelemente */
   TKnoten *Root;
   /* Zeiger auf die Wurzel des aktuellen Suchbaums */
   TKnoten *EndOfPath;
   /* Wird ein M-Augmenting-Path gefunden, so ist es der eindeutige Pfad
   [Root,...,EndOfPath] */
   int Blossomed;
   /* Wurde ein Blossom entdeckt, wird der eine Knoten in EndOfPath, der andere
   Knoten in Blossomed gespeichert */
   TKnoten *ContinueHere;
   /* Nachdem die HungarianTreeProcedur eine Kante zu dem Suchbaum hinzugefügt
   hat, kann MAPS mit diesem Knoten fortsetzen und findet sofort die neue Kante,
   ohne zunächst den gesamten Suchbaum durchsuchen zu müssen */

   /* Elementfunktionen */
   TTree();
   /* Konstruktor */
   ~TTree();
   /* Destruktor, gibt alle noch in dem Baum befindlichen Objekte frei */
   void MakeEmpty();
   /* Gibt alle in dem Baum befindlichen Objekt frei */
   TKnoten *FindKnoten(int AKnotenLabel);
   /* Sucht einen Knoten mit der Nummer (Referenz), die er im Graphen trägt */
   void AddChild(TKnoten *Parent, TKnoten *Child);
   /* Fügt dem Knoten Parent das Kind Child hinzu */
   TKnoten *NextKnoten(TKnoten *AKnoten);
```

```
    /* Liefert den nächsten Knoten in preorder von AKnoten aus gesehen. Wird von
    MAPS genutzt, um den Baum nach allen als outer markierten Knoten zu
    durchsuchen */
    void DelKnoten(TKnoten *aKnoten);
    /* Löscht einen Knoten aus dem Baum. Wird verwendet, wenn ein Blossom durch
    einen Pseudoknoten ersetzt wird */
    void InsertKnoten(TKnoten *before, TKnoten *aKnoten,
    TKnoten *after);
    /* Fügt zwischen before und after den Knoten aKnoten ein. Wird benutzt, wenn
    ein Blossom expandiert wird und der passende Teil des Blossoms in den
    Suchbaum integriert wird */
};
```

Die Laufzeit aller Funktionen von TTree is O(#Knoten in dem Baum), wobei die Zahl der Knoten in dem Suchbaum durch die Zahl der Knoten in dem zugrundeliegenden Graph beschränkt wird. Dennoch sind die Funktionen sehr schnell, da der Suchbaum selten groß wird. Enthält er viele Knoten, so wird mit hoher Wahrscheinlichkeit ein Blossom auftreten, wodurch die Zahl der Knoten in dem Suchbaum wieder reduziert wird.

4.3.2 Das Modul GRAPH

Der abstrakte Datentyp Graph wurde hier statisch implementiert. Kernobjekt ist das Array VertexList vom Typ TVertex, der hier als eigene Klasse implementiert ist. Die Konstanten MAXPLAYER und MAXVERTICES (=1½*MAXPLAYER) sind in macmahon.h definiert und können praktisch beliebig (eingegrenzt durch die beschränkte Größe von Ojekten unter Windows von 64 KByte) festgelegt werden. Das Array VertexList enthält auf den Plätzen 0 bis MAXPLAYER-1 die Knoten des zugrundeliegenden Graphen. Alle auftretenden Pseudoknoten werden auf den Plätzen MAXPLAYER bis MAXVERTICES-1 gespeichert und sofort wieder freigegeben, wenn der Pseudoknoten expandiert wird.

```
/* Graph.h */
/* Headerdatei für den abstrakten Datentyp Graph */

struct TBlossom
/* Diese Struktur enthält Informationen über den Zyklus, der durch einen Pseudoknoten
ersetzt wurde */
{
   int b[MAXPLAYER];
   /* Dieses Array enthält die Knoten, die in dem Suchbaum den Blossom gebildet
   haben, in der korrekten zyklischen Reihenfolge. Das ist wichtig für das Expandieren
   des Blossoms */
   int v[MAXPLAYER];
   /* Dieses Array enthält (unsortiert) alle Knoten des zugrundeliegenden Graphen,
   die in diesem Blossom enthalten sind - also keine Pseudoknoten. Dieses Array
   wird benötigt, um die Kanten ausgehend von dem Pseudoknoten ggf. neu
   berechnen zu können und wird auch bei der Umbenennung der Knoten des
   Blossoms verwendet */
};

class TVertex
{
public:
   int Vertex;
   /* Dieses Attribut enthält die Nummer des Knotens (normalerweise gleich der
   Nummer im Array von TGraph). Ist dieser Knoten in einem Pseudovertex
   enthalten, enthält Vertex die Nummer des äußersten Pseudoknotens, in dem
   dieser Knoten enthalten ist. Auf diese Weise werden alle Knoten in einem
   Blossom mit dem Pseudoknoten identifiziert */
   BOOL PseudoVertex;
   /* Ist TRUE, wenn dieser Knoten ein Pseudoknoten ist */
   BOOL Outermost;
   /* Wenn dieser Knoten ein Pseudoknoten ist, gibt dieses Flag an, ob er noch in
   einem anderen Pseudoknoten enthalten ist (FALSE) oder nicht (TRUE) */
   int Matched;
   /* Nummer (in VertexList) des Knotens mit dem dieser Knoten aktuell gepaart ist.
   Die Menge M wird durch dieses Attribut aller Knoten implizit definiert */
   int Flags;
   /* In diesem Attribut werden die Markierungen INNER und OUTER (Konstanten
   in macmahon.h) gespeichert */
   int Edges[MAXPLAYER];
   /* In diesem Array werden alle Kanten zu anderen Knoten im aktuellen Graphen
   gespeichert. Die Felder des Array können die Werte EDGE oder NOEDGE
   (Konstanten in macmahon.h) annehmen. Kanten zu Pseudoknoten sind indirekt
```

```
   über die Nummer des Zielknotens gespeichert: Edges[G->VertexList[i]->Vertex]
   = ? */
   TBlossom Blossom;
   /* Wenn es sich bei diesem Knoten um einen Pseudoknoten handelt, dann
   befinden sich in dieser Struktur die Informationen über den Blossom */
   ALG_USED_INTTYPE alpha;
   /* Dualvariable dieses Knotens. Wird sowohl bei normalen als auch bei
   Pseudoknoten benutzt. ALG_USED_INTTYPE wird in macmahon.h festgelegt
   und kann die Typen INT (16-bit-Integer) und LONG (32-bit-Integer) annehmen
   */
   int ContainedInBlossom;
   /* Ist dieser Knoten in einem Blossom enthalten, so steht in diesem Attribut die
   Nummer des direkt übergeordneten Pseudoknotens. Sonst ist dieser Wert
   NOVERTEX */

   TVertex();
   /* Konstruktor. Setzt die Attribute auf Default-Werte */
   ~TVertex();
   /* Destruktor */
};

class TGraph
{
public:
   TVertex* VertexList[MAXVERTICES];
   /* Die Liste mit allen im aktuellen Graphen enthaltenen Knoten */

   TGraph();
   /* Konstruktor */
   ~TGraph();
   /* Destruktor. Gibt alle noch vorhandenen Objekte frei */
   void MakeEmpty();
   /* Gibt alle vorhandenen Objekte (die Knoten) frei */
   void AddVertex(TVertex *TheVertex);
   /* Fügt einen Knoten in das Array ein. Wird ausschließlich bei der Initialisierung
   zu Beginn des Algorithmus genutzt. Pseudoknoten werden von den entsprechenden
   Funktionen direkt in das Array geschrieben */
   void AddEdge(int Vertex1, int Vertex2);
   /* Fügt eine Kante zwischen zwei normalen Knoten ein. Sind diese auch in
   Pseudoknoten enthalten, werden deren Kanten gleichzeitig aktualisiert */
   void DeleteEdge(int Vertex1, int Vertex2);
```

```
    /* Löscht eine Kante auf unterster Ebene, d.h. zwischen zwei normalen Knoten.
    Dieser Löschvorgang wird nicht an übergeordnete Pseudoknoten weitergegeben!
    */
    int ExistEdge(int Vertex1, int Vertex2);
    /* Prüft, ob eine Kante zwischen zwei beliebigen (auch Pseudoknoten) Knoten
    des Graphen existiert */
    void ShrinkBlossom(TTree *T);
    /* Diese Funktion schrumpft den in T definierten Blossom zu einem Pseudoknoten
    und modifiziert T entsprechend, so daß die Suche nach einem M-Augmenting-
    Path mit demselben Suchbaum fortgesetzt werden kann */
    void ExpandBlossom(int theblossom);
    /* Diese Funktion expandiert einen Blossom, ohne den Suchbaum entsprechend
    zu modifizieren. Diese Funktion wird ausschließlich am Ende des Algorithmus
    eingesetzt, um alle verbliebenen Pseudoknoten zu expandieren. theblossom gibt
    die Nummer des Blossoms in VertexList an */
    void ExpandBlossomwithTree(TTree *T, int theblossom);
    /* Wie ExpandBlossom, nur wird jetzt der Suchbaum entsprechend modifiziert.
    Diese Funktion wird benutzt, wenn die Dualvariable eines als inner markierten
    Pseudoknotens 0 wird */
    void AugmentPath(TTree *T);
    /* Diese Funktion vertauscht die Rollen der Kanten des durch T definierten M-
    Augmenting-Paths in M */
    void RemoveAllLabels();
    /* Löscht die inner und outer Markierungen aller Knoten */
};
```

Interessant sind lediglich die Funktionen ShrinkBlossom und ExpandBlossomwithTree (stellvertretend auch für ExpandBlossom). Anhand dieser Funktionen werde ich erläutern, wie die Blossoms auf relativ einfache Weise verwaltet werden können und welche Änderungen in dem jeweils aktuellen Suchbaum vorgenommen werden müssen.

4.3.2.1 Die Funktion ShrinkBlossom

Die Funktion ShrinkBlossom hat für den Algorithmus eine zentrale Bedeutung, löst sie schließlich das Problem mit den Zyklen ungerader Länge.

Es erweist sich dabei als nicht sinnvoll, den Graphen als rein abstrakten Datentyp zu behandeln und alle Knoten in dem zu schrumpfenden Blossom zu löschen und einen neuen Knoten mit allen Kanten der zuvor

gelöschten Knoten in den Graphen einzufügen. Dies wäre sehr aufwendig. Da der Graph von der Funktion MAPS (M-Augmenting-Path-Search) nur nach seinen Kanten durchsucht wird und der Pseudoknoten alle Kanten der in ihm enthaltenen Knoten erbt, liegt es nahe, die Identifizierung der Knoten mit dem Pseudoknoten durch die Benennung der Knoten zu erreichen. Leider müssen für den Fall, daß MAPS von einem Pseudoknoten ausgehend die Kanten überprüft, alle Kanten auch in den Pseudoknoten kopiert werden, wodurch die Funktion insgesamt eine hohe Komplexität erhält.

Die Funktion hat drei Teile. Im ersten Teil wird der durch T definierte Blossom erkannt und ein Array mit der korrekten zyklischen Folge der Knoten des Blossoms aufgebaut. Im zweiten Teil werden dann die Knotenbeschriftungen aller Knoten des Blossoms in die Nummer (Referenz zum Eintrag in dem Array VertexList) des neuen Pseudoknotens geändert. Das bedeutet konkret: Immer wenn auf einen Knoten des Blossoms zugegriffen werden soll, wird dieser Zugriff automatisch an den Pseudoknoten weitergeleitet. So werden z.B. alle Knoten des Blossoms automatisch als outer markiert, wenn der Pseudoknoten als outer markiert wird. Außerdem wird im zweiten Teil die Zyklusinformation in den Pseudoknoten kopiert und eine Liste aller in dem Pseudoknoten enthaltenen normalen Knoten aufgebaut. Anschließend werden alle Kanten ausgehend von dem Pseudoknoten berechnet. Im dritten Teil schließlich wird der Suchbaum modifiziert. Alle Knoten des Blossoms bis auf einen werden gelöscht, und der letzte wird in den Pseudoknoten umbenannt.

```
void TGraph::ShrinkBlossom(TTree *T)
{
   TKnoten *x1, *x2, *pos1, *pos2, *pos;
   TKnoten *halfcicle1[MAXPLAYER], *halfcicle2[MAXPLAYER];
   int hc1pos, hc2pos;
   TVertex *Blossom;
   int blossompos;
   int i,j;
   int theblossom;
   int vertexpos;

   Blossom = new TVertex();
```

```
/* Der Knoten für den Pseudoknoten wird angelegt */
theblossom = MAXPLAYER;
while (VertexList[theblossom] != NULL)
{
  theblossom++;
}
/* theblossom ist jetzt die Referenz zu einem freien Platz in der Liste der
Pseudoknoten in VertexList */
VertexList[theblossom] = Blossom;
Blossom->Vertex = theblossom;
Blossom->PseudoVertex = TRUE;
Blossom->Flags = OUTER;
/* Jeder Pseudoknoten wird zunächst als outer markiert */
Blossom->alpha = 0;
/* Die Dualvariable ist zunächst 0 */
Blossom->Outermost = TRUE;
Blossom->ContainedInBlossom = NOVERTEX;
/* In dem folgenden Abschnitt werden zwei Pfade von der Wurzel zu den beiden
Knoten aufgebaut, die den Blossom verursachten. Aus ihnen läßt sich dann
ermitteln, welche Knoten in welcher zyklischen Reihenfolge in dem Blossom
enthalten sind */
x1 = T->EndOfPath;
/* erster Knoten */
x2 = T->FindKnoten(T->Blossomed);
/* zweiter Knoten*/
pos1 = x1;
pos2 = x2;
halfcicle1[0] = pos1;
halfcicle2[0] = pos2;
hc1pos = 1;
hc2pos = 1;
while (pos1 != T->Root)
{
  pos1 = pos1->Parent;
  halfcicle1[hc1pos] = pos1;
  hc1pos++;
}
/* Der Pfad von Knoten 1 zu Wurzel ist jetzt aufgebaut */
while (pos2 != T->Root)
{
  pos2 = pos2->Parent;
  halfcicle2[hc2pos] = pos2;
  hc2pos++;
```

```
}
/* Der Pfad von Knoten 2 zu Wurzel ist jetzt aufgebaut */
hc1pos--;
hc2pos--;
/* Als nächstes wird der letzte gemeinsame Knoten gesucht: */
while (!((hc1pos < 0) || (hc2pos < 0)) &&
(halfcicle1[hc1pos] == halfcicle2[hc2pos]))
{
  hc1pos--;
  hc2pos--;
}
blossompos = 0;
vertexpos = 0;
hc1pos++;
hc2pos++;
Blossom->Blossom.b[0] = halfcicle1[hc1pos]->Label;
/* Oben: Blossom.b[0] enthält den Knoten, den beide Pfade gerade noch
gemeinsam haben. Dies wird der einzige Knoten in dem Blossom sein, der
zunächst keinen Partner in M hat (daher wird der Blossom als outer markiert) */
for (i = hc1pos - 1; i >= 0; i--)
{
  blossompos++;
  Blossom->Blossom.b[blossompos] = halfcicle1[i]->Label;
}
for (i = 0; i < hc2pos; i++)
{
  blossompos++;
  Blossom->Blossom.b[blossompos] = halfcicle2[i]->Label;
}
/* Jetzt enthält das Array Blosssom.b[] die Knoten des Blossoms in der korrekten
zyklischen Reihenfolge. Als nächstes werden die entsprechenden Attribute dieser
Knoten auf die neue Situation eingestellt */
for (i = 0; i <= blossompos; i++)
{
  VertexList[Blossom->Blossom.b[i]]->Outermost = FALSE;
  VertexList[Blossom->Blossom.b[i]]->ContainedInBlossom
    = theblossom;
  if (VertexList[Blossom->Blossom.b[i]]->PseudoVertex)
  {
    /* Wenn ein Pseudoknoten in dem Blossom enthalten ist, müssen alle in ihm
    enthaltenen Knoten mit der Referenz zu dem neuen Pseudoknoten beschriftet
    werden: /*
    for (j = 0; j < MAXPLAYER; j++)
```

```
    {
      if ((VertexList[j]) && (VertexList[j]->Vertex ==
        Blossom->Blossom.b[i]))
      {
        VertexList[j]->Vertex = theblossom;
        Blossom->Blossom.v[vertexpos] = j;
        vertexpos++;
      }
    }
  }
  else
  {
    VertexList[Blossom->Blossom.b[i]]->Vertex  =
      theblossom; /* Beschriftung des Knotens */
    Blossom->Blossom.v[vertexpos]  =  Blossom-
      >Blossom.b[i];
    vertexpos++;
  }
}
/* Als nächstes müssen alle (bis auf einen) Knoten des Blossoms aus dem
aktuellen Suchbaum entfernt werden: */
for (i = 0; i <= blossompos; i++)
{
  if (i > 0)
  {
    T->DelKnoten(T->FindKnoten(Blossom->Blossom.b[i]));
  }
  else
  { /* Ein Knoten des Suchbaums wird in den Blossom umbenannt */
    pos = T->FindKnoten(Blossom->Blossom.b[i]);
    pos->Label = theblossom;
  }
}
/* Als nächstes müssen die Kanten ausgehend von dem neuen Pseudoknoten in
dem Graphen G' berechnet werden: */
i = 0;
while (VertexList[theblossom]->Blossom.v[i] != NOVERTEX)
{
  for (j = 0; j < MAXPLAYER; j++)
  {
    if ((VertexList[j]) && (VertexList[j]->Vertex != theblossom))
    {
```

```
          Blossom->Edges[j]  |=  VertexList[Blossom-
            >Blossom.v[i]]->Edges[j];
        }
      }
      i++;
    }
    /* War der Ursprung (der letzte gemeinsame Knoten der beiden Pfade) des
    Blossoms mit einem anderen Knoten gematched, so übernimmt der neue
    Pseudoknoten diesen Partner: */
    Blossom->Matched = VertexList[Blossom->Blossom.b[0]]->Matched;
    if (Blossom->Matched != NOVERTEX)
    {
      VertexList[Blossom->Matched]->Matched = Blossom->Vertex;
    }
    VertexList[Blossom->Blossom.b[0]]->Matched = NOVERTEX;
};
```

Die Laufzeit der Funktion ShrinkBlossom ist $O(n^2)$. Am Aufwendigsten ist die Berechnung aller Kanten, die in dem Graphen von dem neuen Pseudoknoten ausgehen werden. Dafür kann MAPS auf diese Kanten später in O(1) zugreifen.

4.3.2.2 Die Funktion ExpandBlossomWithTree

Zunächst expandiert die Funktion den Pseudoknoten im Graphen gemäß der Beschreibung in Abschnitt 4.2.5. Anschließend wird der Suchbaum, der bei Aufruf der Funktion ExpandBlossomWithTree die allgemeine Form aus Abbildung 4.7 hat, so modifiziert, daß der Suchbaum seine alternierende Struktur (d.h. auf allen Pfaden abwechselnd als outer und inner markierte Knoten) behält. Da der Pseudoknoten als inner markiert ist, hat er genau zwei Nachbarn in dem Suchbaum, die beide als outer markiert sind.

Sei x_1 ein Knoten in dem Blossom mit einer Kante zu $x_1 neighbour$ und x_2 ein Knoten in dem Blossom mit einer Kante zu $x_2 neighbour$. Diese Knoten existieren (und sind möglicherweise identisch), da der Pseudoknoten Kanten zu $x_1 neighbour$ und $x_2 neighbour$ hatte. Beim Expandieren des Pseudoknotens wird x_2 mit $x_2 neighbour$ gematched und

ggf. die Rollen in *M* von anderen Kanten in dem Blossom so modifiziert, daß jetzt alle Knoten in dem Blossom einen Partner in *M* haben.

In dem Blossom gibt es zwei Pfade zwischen x_1 und x_2. Genau einer beginnt mit einer Kante in *M*. Dann wird der Pseudoknoten in dem Suchbaum durch diesen Pfad ersetzt.

```
void TGraph::ExpandBlossomwithTree(TTree *T, int theblossom)
/* T ist der zu aktualisierende Suchbaum und theblossom ist eine Referenz zu dem zu
expandierenden Blossom im Array VertexList in den Turnierdaten */
{
   int i,j,k;
   TKnoten *pos, *x1neighbour, *x2neighbour, *aKnoten;
   int path1[MAXPLAYER];
   int p1;
   BOOL augment;
   int dir, count;
   int MatchInBlossom;
   int ablossom;

   /* Als erstes werden die Attribute aller Knoten in dem Blossom so geändert, daß
   sie wieder im Graphen aktiviert sind. */
   i = 0;
   MatchInBlossom = NOVERTEX;
   while (VertexList[theblossom]->Blossom.b[i] != NOVERTEX)
```

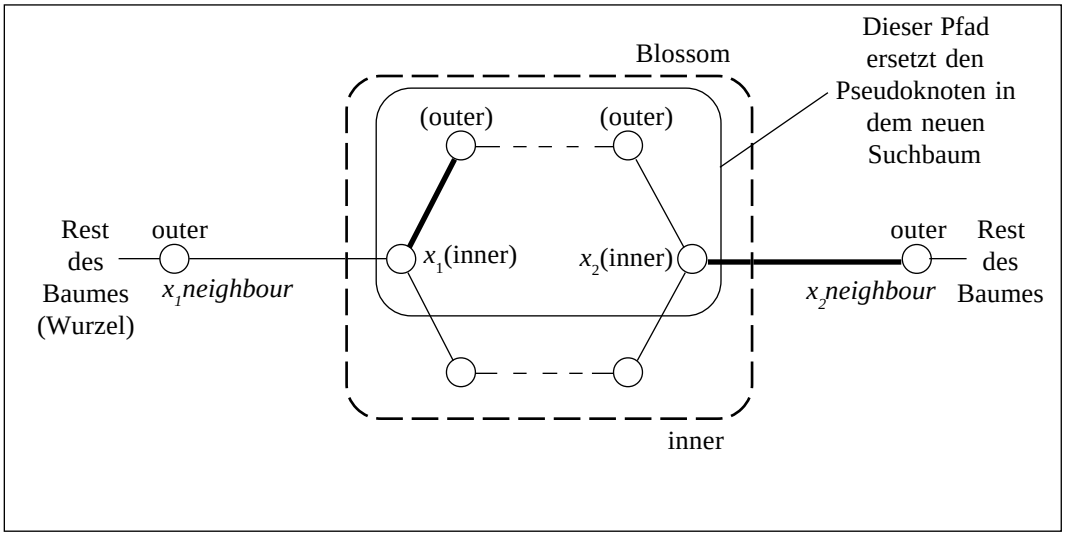

Abbildung 4.7, die entsprechenden Kanten in *M* sind fett dargestellt.

```
{
  VertexList[VertexList[theblossom]->Blossom.b[i]]-
    >Vertex = VertexList[theblossom]->Blossom.b[i];
  VertexList[VertexList[theblossom]->Blossom.b[i]]-
    >Outermost = TRUE;
  VertexList[VertexList[theblossom]->Blossom.b[i]]-
    >ContainedInBlossom = NOVERTEX;
  VertexList[VertexList[theblossom]->Blossom.b[i]]-
    >Flags &= ~(INNER | OUTER);
  if (VertexList[VertexList[theblossom]->Blossom.b[i]]-
    >PseudoVertex)
  {
/* Wird ein Pseudoknoten freigesetzt, so müssen dessen Kanten neu berechnet
werden */
    ablossom = VertexList[theblossom]->Blossom.b[i];
    j = 0;
    while (VertexList[ablossom]->Blossom.v[j] != NOVERTEX)
    {
      VertexList[VertexList[ablossom]->Blossom.v[j]]-
        >Vertex = VertexList[theblossom]->Blossom.b[i];
      j++;
    }
    for (j = 0; j < MAXPLAYER; j++)
    {
      VertexList[ablossom]->Edges[j] = NOEDGE;
    }
    j = 0;
    while (VertexList[ablossom]->Blossom.v[j] != NOVERTEX)
    {
      for (k = 0; k < MAXPLAYER; k++)
      {
        if ((VertexList[k]) && (VertexList[k]->Vertex
          != ablossom) && (ExistEdge(k, ablossom)))
        {
         VertexList[ablossom]->Edges[k] = EDGE;
        }
      }
      j++;
    }
  }
  i++;
}
```

```
/* Da der Pseudoknoten als inner markiert ist, hat er einen Partner im aktuellen
Matching M. Jetzt muß zunächst ein Knoten in dem Blossom gefunden werden,
der mit dem Partner des Blossoms gematched werden kann. Dafür wird zunächst
untersucht, ob der Knoten benutzt werden kann, der keinen Partner innerhalb des
Blossoms hat. Hat dieser keine Kante zu dem Partner des Pseudoknotens, wird
ein anderer Knoten mit dieser Eigenschaft ausgewählt und entsprechend ein M-
Augmenting-Path in dem Blossom identifiziert */
i = VertexList[theblossom]->Matched;
if (i != NOVERTEX)
{
  if (ExistEdge(i, VertexList[theblossom]->Blossom.b[0]))
  {
    VertexList[VertexList[theblossom]->Blossom.b[0]]->Matched = i;
    VertexList[i]->Matched = VertexList[theblossom]->Blossom.b[0];
    MatchInBlossom = 0;
  }
  else
  {
    j = 1;
    while (VertexList[theblossom]->Blossom.b[j] != NOVERTEX)
    {
      if (ExistEdge(VertexList[theblossom]->Blossom.b[j], i))
      {
        break;
      }
      j++;
    }
    k = VertexList[VertexList[theblossom]->Blossom.b[j]]->Matched;
    if (k == VertexList[theblossom]->Blossom.b[j-1])
    {
      dir = -1;
    }
    else
    {
      dir = 1;
    }
    VertexList[i]->Matched = VertexList[theblossom]->Blossom.b[j];
    VertexList[VertexList[theblossom]->Blossom.b[j]]->Matched = i;
    MatchInBlossom = j;
    p1 = 0;
    path1[p1] = VertexList[theblossom]->Blossom.b[j];
    count = 0;
    while (VertexList[theblossom]->Blossom.b[count] != NOVERTEX)
```

```
      {
        count++;
      }
      count--;
      k = j + dir;
      while (TRUE)
      {
        p1++;
        path1[p1] = VertexList[theblossom]->Blossom.b[k];
        i = k + dir;
        if (i < 0)
        {
          i = count;
        }
        if (i > count)
        {
          i = 0;
        }
        p1++;
        path1[p1] = VertexList[theblossom]->Blossom.b[i];
        VertexList[VertexList[theblossom]->Blossom.b[k]]-
          >Matched = VertexList[theblossom]->Blossom.b[i];
        VertexList[VertexList[theblossom]->Blossom.b[i]]-
          >Matched = VertexList[theblossom]->Blossom.b[k];
        if (i == 0)
        {
          break;
        }
        k = i + dir;
      }
    }
  }
```

/ Jetzt hat der Suchbaum die allgemeine Form aus Abbildung 3.7. Der richtige Pfad in dem Blossom wird ausgewählt und entsprechend in den Suchbaum integriert */*

```
pos = T->FindKnoten(theblossom);
x1neighbour = pos->Parent;
x2neighbour = pos->FirstChild;
count = 0;
while  (VertexList[theblossom]->Blossom.b[count]  !=
NOVERTEX)
{
  count++;
```

```
}
count--;
if (ExistEdge(x2neighbour->Label, VertexList[theblossom]-
>Blossom.b[MatchInBlossom]))
{ /* always true */
  k = 0;
  while (!ExistEdge(VertexList[theblossom]->Blossom.b[k], x1neighbour->Label))
  {
    k++;
  }
  augment = TRUE;
  i = k + 1;
  if (i > count)
  {
    i = 0;
  }
  if (VertexList[VertexList[theblossom]->Blossom.b[k]]-
    >Matched == VertexList[theblossom]->Blossom.b[i])
  {
    dir = 1;
  }
  else
  {
    dir = -1;
  }
  while (TRUE)
  {
    aKnoten = new TKnoten;
    aKnoten->Label = VertexList[theblossom]->Blossom.b[k];
    if (augment)
    {
      augment = FALSE;
      /* Die Knoten werden entsprechend markiert: */
      VertexList[VertexList[theblossom]->Blossom.b[k]]->Flags |= INNER;
    }
    else
    {
      augment = TRUE;
      VertexList[VertexList[theblossom]->Blossom.b[k]]->Flags |= OUTER;
    }
    T->InsertKnoten(pos, aKnoten, x2neighbour);
    pos = aKnoten;
    if (k == MatchInBlossom)
```

```
      {
        break;
      }
      k = k + dir;
      if (k > count)
      {
        k = 0;
      }
      if (k < 0)
      {
        k = count;
      }
    }
    /* Der Pseudoknoten wird im Suchbaum gelöscht: */
    T->DelKnoten(T->FindKnoten(theblossom));
  }
  /* Als letztes wird der Pseudoknoten im Graphen gelöscht: */
  delete VertexList[theblossom];
  VertexList[theblossom] = NULL;
};
```

Die Funktion ExpandBlossomWithTree hat eine Laufzeit von $O(n^3)$ und ist damit die aufwendigste Prozedur des Algorithmus.

4.3.3 Das Modul ALGORITHM

Die Funktionen der Klasse TAlgorithm greifen auf die Module Tree und Graph zurück. Die Klasse TAlgorithm enthält als Datenelement die Gewichtsmatrix, nach der das Maximum Weight Perfect Matching gefunden werden soll. Als Elementfunktionen der Klasse TAlgorithm sind auch der Maximum-Weight-Perfect-Matching-Algorithmus aus Abschnitt 4.2.7 und die Funktion MAPS implementiert.

Zur Durchführung des Algorithmus wird ein Objekt der Klasse TAlgorithm angelegt, die Gewichtsmatrix mit entsprechenden Werten belegt und das Datenelement N mit der Zahl der zu paarenden Knoten initialisiert. Anschließend wird MakePairing() aufgerufen. Diese Funktion terminiert, wenn sie das Maximum Weight Perfect Matching in dem durch N und die

Gewichtsmatrix definierten zugrundeliegenden Graphen gefunden hat. In dem Datenelement G sind zu allen Knoten die Partner in dem Matching enthalten.

Die Klasse TAlgorithm ist in alg_mwpm.h wie folgt deklariert (die Konstanten sind in macmahon.h definiert):

```
class TAlgorithm
{
public:
   TMatrixColumn *WeightMatrix[MAXPLAYER];
   /* Die Gewichtsfunktion als Matrix abgelegt */
   TExploredColumn *Explored[MAXVERTICES];
   /* In dieser Matrix wird markiert, welche Kanten durch MAPS bereits erkundet
   wurden. */
   int N;
   /* Zahl der Knoten in dem zugrundeliegenden Graphen */
   TGraph *G;  /* Der aktuelle Graph */

/* Die Elementfunktionen: */
   TAlgorithm(PTWindowsObject aParent); /* Konstruktor */
   ~TAlgorithm(); /* Destruktor */
   void Initialize(int Number);
   /* Diese Funktion setzt N auf Number */
   void MakePairing();
   /* Der Maximum-Weight-Perfect-Matching-Algorithmus */
   int MAugmentingPathSearch(TTree *T);
   /* Die Funktion MAPS, die auf der Suche nach einem M-Augmenting-Path
   versucht, den Suchbaum T auszuweiten */
   int HungarianTreeProc(TTree *T);
   /* Diese Funktion wird aufgerufen, wenn MAPS den Suchbaum nicht mehr
   erweitern kann. Sie berechnet δ und führt die Änderungen der Dualvariablen
   durch. Anschließend wird ggf. eine Kante zu G hinzugefügt, oder ein Blossom
   expandiert. */
   void ClearExploredMatrix();
   /* Löscht alle Markierungen in der Matrix der erkundeten Kanten */
   void AugmentGraph();
   /* Baut den aktuellen Graphen G' aus dem zugrundeliegenden (vollständigen)
   Graphen auf */
   void UpdateExploredMatrix(TTree *T);
```

```
    /* Wird ein Blossom expandiert, kommt es vor, daß Kanten in dem Suchbaum
    nicht als erkundet markiert sind. Diese Funktion markiert alle nicht mehr
    erkundbaren Kanten als erkundet */
};
```

4.3.4 Komplexität der Implementation

Die in „MacMahon“ durchgeführte Implementation des Maximum-Weight-Perfect-Matching-Algorithmus hat eine Komplexität von $O(n^5)$. Der Nachteil gegenüber der theoretisch möglichen Komplexität von $O(n^3)$ um zwei Größenordnungen resultiert aus der Verwaltung der Blossoms.

Eine Verbesserung auf $O(n^4)$ gegenüber der gegenwärtigen Implementation ist relativ leicht möglich: Werden Kanten aus dem aktuellen Graphen entfernt, geschieht dies auf unterster Ebene, d.h. zwischen normalen Knoten. Ist einer dieser normalen Knoten in einem Pseudoknoten enthalten, ist nicht sicher, ob der Pseudoknoten seine Kante zu dem anderen Knoten nur durch den einen normalen Knoten, oder auch durch einen anderen Knoten in dem Blossom erhalten hat. Daher kann diese Kante nicht in $O(1)$ gelöscht werden.

Eine Lösung dieses Problems wäre die Verallgemeinerung des Begriffes „Kante“. Intern wird eine nicht vorhandene Kante mit der Konstanten NOEDGE=0 und eine vorhandene Kante mit der Konstanten EDGE=1 identifiziert. Die Menge der Kanten eines Pseudoknotens ist als Summe aller Kanten der in ihm enthaltenen Knoten definiert. Setzt man den Wert einer Kante eines Pseudoknotens zu einem anderen Knoten auf die Häufigkeit, mit der eine solche Kante bei den im Pseudoknoten enthaltenen Knoten vorkommt und ändert die Bedingung für die Existenz einer Kante von dem Gleichheitstest mit EDGE auf die Ungleichheit mit NOEDGE, kann die Kante aus dem Blossom in $O(1)$ gelöscht werden, indem der entsprechende Zähler einfach um 1 reduziert wird. Dies würde dem Algorithmus eine Komplexität von $O(n^3)$ ermöglichen. Die Funktionen ShrinkBlossom und ExpandBlossomWithTree wären aber immer noch von der Komplexität $O(n^2)$, wodurch die Komplexität des gesamten Algorithmus noch bei $O(n^4)$ läge.

Theorem: Die Implementation des Maximum-Weight-Perfect-Matching-Algorithmus hat eine Komplexität von $O(n^5)$, wobei n die Zahl der teilnehmenden Spieler ist.

Beweis:

Der Schritt zwischen zwei erfolgreichen Suchen nach einem M-Augmenting-Path wird eine „Stage“ genannt. Da durch jedes Auffinden eines M-Augmenting-Paths die Zahl der freien Knoten um 2 vermindert wird, kann es höchstens n/2 Stages im Verlauf des Algorithmus geben.

Zwischen zwei Stages versucht MAPS, einen Suchbaum aufzubauen, der schließlich zumAuffinden eines M-Augmenting-Paths führt. JederAufruf von MAPS hat eine Komplexität von $O(n^2)$, da MAPS von O(n) als outer markierten Knoten eine Kante suchen muß, was auch O(n) Zeit benötigt. Jeder Aufruf von MAPS wird ein „Schritt“ genannt.

War MAPS nicht erfolgreich, können zwei Fälle eintreten:

1. Ein Blossom wurde erkannt und muß durch einen Knoten ersetzt werden. Dies kann höchstens O(n) mal passieren, da das Auffinden jedes Blossoms die Zahl der Knoten in dem Graphen um mindestens 2 senkt.

2. MAPS hat keine unerkundete Kante mehr gefunden. Dann müssen die Dualvariablen geändert werden, und entweder wird ein als inner markierter Blossom expandiert oder eine Kante hinzugefügt. Da es zu keinem Zeitpunkt mehr als n/2 Blossoms in dem Graphen geben kann, wird höchstens O(n) mal ein Blossom expandiert.

   Da nur Kanten hinzugefügt werden, die den Suchbaum erweitern oder zur Erkennung eines Blossoms (höchstens O(n) mal) führen, werden im Laufe einer Stage höchstens O(n) Kanten hinzugefügt.

Insgesamt wird die Funktion MAPS also höchstens O(n) mal aufgerufen. Die Komplexität einer Stage ergibt sich daher als $O(n^4)$, da das Expandieren von Blossoms innerhalb einer Stage O(n) mal auftreten kann und jeweils

$O(n^3)$ Zeit verbraucht. Die anderen Funktionen innerhalb einer Stage - MAPS, die Berechnung und Änderung der Dualvariablen und das Schrumpfen von Blossoms zu Pseudoknoten brauchen höchstens O(n) mal $O(n^2)$.

Insgesamt garantiert diese Implementierung eine Laufzeit von $O(n^5)$.

Der Speicherbedarf für die Durchführung des Algorithmus ist $O(n^2)$.

4.3.5 Laufzeiten

Trotz der hohen theoretischen Komplexität von $O(n^5)$ erweist sich diese Implementation in der Praxis als sehr effizient. Am 30.4./1.5.1994 in Hannover und am 7./8.5.1994 in Rostock wurde das Programm bereits auf zwei Turnieren eingesetzt.

Während das Rostocker Turnier mit 44 Teilnehmern nicht als Maßstab herangezogen werden kann, hat das Turnier in Hannover mit 132 Teilnehmern schon gezeigt, daß das Programm auch bei größeren Teilnehmerzahlen schnell eine Losung generieren kann. Die Berechnung der Kostenfunktion ($O(n^2)$) brauchte auf einem 486er DX2/66 MHz in keiner der fünf in Hannover gespielten Runden mehr als eine Sekunde. Die Optimierung mit dem Maximum-Weight-Perfect-Matching-Algorithmus dauerte zwischen 15 und 25 Sekunden.

Auch Losungen mit Testdaten von bis zu 190 Spielern lassen keinen wesentlichen Verlust in der Effizienz erkennen (Rechenzeit ca. 1 Minute). Das läßt vermuten, daß auch Turniere mit bis zu 300 Spielern ohne weitere Optimierungsmaßnahmen problemlos mit dem Programm durchgeführt werden können.

5 Ausblick und Wertung

Obwohl sich das MacMahon-System formal beschreiben läßt, wird von einem Losungsprogramm für Go-Turniere weit mehr erwartet, als eine Losung nach einem klar definierten MacMahon-System. Alle Wünsche von Turnierleitungen bezüglich der Bedienoberfläche und der Flexibilität des Algorithmus kann man unmöglich erfüllen, aber auch die Befriedigung einfacher Bedürfnisse (Spielereingabe, Aufstellung und Ausgabe einer Rangliste) ist bereits mit erheblichem Programmieraufwand verbunden.

Da die Umsetzung des Inhaltes dieser Diplomarbeit in ein zuverlässig arbeitendes Losungsprogramm eine hohe Priorität hatte, blieben einige Fragen eher theoretischer Natur unbeantwortet. Das gilt sowohl für das MacMahon-System, wo ich den wissenschaftlichen Beweis für die Güte der Zweitkriterienkombination SOS/SOSOS schuldig geblieben bin, als auch für die Implementierung des Algorithmus, bei dem das Optimierungspotential noch nicht ausgeschöpft wurde.

Dennoch liefert diese Diplomarbeit und das damit bereitgestellte Losungsprogramm eine gute Grundlage für spätere Erweiterungen und für eine weitere Flexibilisierung des Algorithmus (z.B. zum Einsatz auf Schachturnieren). Wichtige Erweiterungen der Bedienoberfläche sind:

- Eine Datenbank aller Spieler im Hintergrund zur schnelleren und fehlerfreieren Eingabe.
- Eine Benutzerführung, die auf den Kenntnisstand der Turnierleitung Rücksicht nimmt. Absolute Laien sollten nur wenige Wahlmöglichkeiten haben, Experten sollten auch auf komplexe Parameter einen Einfluß nehmen können.
- Integration einer Online-Hilfe.
- Semantische Kontrolle der Eingaben des Anwenders.

Ich werde in Zukunft das Programm weiterentwickeln, um es auch sich ändernden Anforderungen im Go in Europa anzupassen. Bei dieser Diplomarbeit war mir persönlich besonders wichtig, nachzuweisen, daß global optimale Losungen nach dem MacMahon-System durchaus effizient zu ermitteln sind. Dieses Ziel hat die Implementierung (für mich) überraschend gut erreicht.

Anhang

Literaturverzeichnis

[Chr75] Nicos Christofides: *Graph theory - an algorithmic approach*. Academic Press Inc, ISBN 0-12-174350-0, 1975

[Die93] N. W. P. van Diepen: *From formal specification towards derivation: the MacMahon (Swiss) System*. Technical Report CSI-R9329, Katholieke Universiteit Nijmegen, Dezember 1993

[E67] J. Edmonds: *An introduction to matching*. Lecture notes, Engineering Summer Conference, University of Michigan, Ann Arbor, USA, 1967

[EJ70] J. Edmonds & E. Johnson: *Matching: a well-solved class of ILP*. *Combinatoral Structures and their Applications*. Gordon & Breach, NY, 89-92, 1970

[Gar72] H. Garbow: *An efficient implementation of Edmonds' maxiumum matching algorithm*. Technical Report 31, Stanford University, Computer Science Dept, June 1972

[Gib85] Alan Gibbons: *Algorithmic graph theory*. Cambridge University Press, ISBN 0-521-28881-9, 1985

[Kn79] Bernd Knauer: *A SimpleAlgorithm For Maximum Matching Of Graphs*. In *Numerische Methoden bei graphentheoretischen und kombinatorischen Problemen (Band 2)*, BirkhäuserVerlag, ISBN 3-7643-1078-2, 1979

[PR88] R. Gary Parker&Ronald L. Rardin: *Discrete Optimization*. Academic Press Inc., ISBN 0-12-545075-3, 1988

[PS82] C. H. Papadimitriou & K. Steiglitz: *Combinatorial optimization - algorithms and complexity*. Prentice-Hall Inc., ISBN 0-13-152462-3, 1982

Die Ursprünge des MacMahon-Systems

(von Francis Roads, London, GB)

Angesichts der Diskussion, ob wir schon den besten Weg gefunden haben, die Losung auf einem Go-Kongreß zu machen, fragte ich mich, ob die Spieler vielleicht gerne wüßten, wie das MacMahon-System erfunden wurde.

Die ersten drei britischen Go-Kongresse (jeweils 6ründige Turniere) fanden in den Jahren 1968 bis 1970 statt und wurden mitVorgabe gespielt. Die Losungen für alle Runden wurden vor der ersten Runde ausgearbeitet. Das bedeutete, daß die Resultate von Spielern nicht die ihnen nachfolgenden Paarungen beeinflußten. Dieses System mag auch seine Vorteile haben.

Da dies praktisch das einzige Turnier in Britanien war, wurde die britische Meisterschaft auch auf dem Kongreß ausgespielt. Die Dan-Spieler (es gab sechs) spielten jeder gegen jeden.

Abgesehen von der regelmäßigen Notwendigkeit von Entscheidungsspielen nach dem Kongreß hatte dieses System einen ernsthafteren Nachteil, indem die starken Kyu-Spieler nicht ein einzige Partie mit einem Dan-Spieler spielte. Die Folge war, daß einige 1 Kyu's den britischen Kongreß 1970 boykottierten.

Ich erinnere mich, wie ich auf der nächsten Sitzung der BGA (british go association) vehement für ein System gestritten habe, bei dem Spieler aus der britischen Meisterschaft ausschieden und in das Hauptturnier wechselten, sobald sie so viele Partien verloren hatten, daß sie nicht mehr britischer Meister werden konnten.

Mein System war kompliziert und wurde sofort zu Gunsten eines Systems abgelehnt, von dem wir dachten, daß es in den USA bereits benutzt würde. Später stellte sich heraus, daß das MacMahon-System vom New Yorker Go-Klub mehr als Einstufungssystem innerhalb ihres Klubs eingesetzt wurde. Und so waren die Briten die Ersten, die es benutzten, um ein Turnier zu organisieren.

Was wir dann auch zum ersten Mal auf dem britischen Go-Kongreß 1971 in Leeds taten. Und wir haben es falsch gemacht! Wir addierten nicht nur einen MacMahon-Punkt für eine gewonnene Partie, sondern zogen auch einen für eine verlorene Partie ab. Das gab den Spielern eine Parität und wir spielten praktisch zwei Turniere! Nur beim Auftreten eines jigo's, oder wenn ein Spieler hoch- oder runtergelost werden mußte, konnte ein Spieler die Parität wechseln.

Ganz nebenbei sei erwähnt, daß wir jigo's auf britischen Turnieren mögen, um dem Turnierleiter eine höhere Flexibilität beim Losen zu geben und um die Notwendigkeit weiterer tie-breaker zu reduzieren. Das ist ein Grund, warum die Briten augenblicklich mit 6 Komi spielen.

Vom folgenden Jahr an wurde das MacMahon-System mehr oder weniger so verwendet, wie wir es auch heute tun. Es wird auf praktisch allen britischen Turnieren (wir haben ungefär ein Dutzend) verwendet und inzwischen auch auf dem Kontinent. Ich frage mich, wann wir unsere Freunde von Fernost überreden können, mal das MacMahon-System zu probieren.

Der Nachteil praktisch aller anderer Systeme ist es, daß sie normalerweise erfordern, irgendwo Grenzen zwischen Gruppen von Spielern zu ziehen. Aber wann immer man Spieler in Gruppen aufteilt, wird es Spieler geben, die sich beschweren, sie seien in der falschen Gruppe.

Schließlich realisierten wir, daß auf einem Turnier, wo Spieler aller Stärken zusammenkommen, um Spaß am Go-Spiel zu haben, nicht auch der britische Meister ausgespielt werden kann. Wir nennen den Gewinner des britischen Go-Kongresses jetzt den offenen britischen Meister, ob er nun Brite oder Ausländer ist.

Der britische Meister wird jetzt in einem play-off nach Qualifikations-turnieren ausgespielt. Diese Turniere haben nichts mit mehr mit dem britischen Go-Kongreß zu tun.