Diplomarbeit

# Online-Matching

**Christian Schlup**

Betreuung: Ruedi Arnold
Leitung: Prof. Dr. Roger Wattenhofer

Distributed Computing Group
Departement Informatik
**ETH Zürich**

Winter 2002/03

**Zusammenfassung**

Diese Diplomarbeit beinhaltet das Entwickeln und Testen von Online-Algorithmen für das Matching von Spielern in Multiplayerspielen. Der Matchingprozess soll dabei die am besten zueinander passenden Spieler zusammenführen, die innerhalb einer bestimmten Zeitspanne zur Verfügung stehen.

Um für die Algorithmenentwicklung geeignete Werkzeuge zur Verfügung zu stellen, wird zunächst ein theoretisches Modell des Matchingprozesses erstellt und ein Güte-Mass für die Abschätzung der Qualität eines Matchings hergeleitet. Das Güte-Mass erlaubt die Berechnung der Matchingkosten.

Es werden drei Algorithmen vorgestellt, die in der Lage sind, Spieler unter Berücksichtigung von Kriterien zusammenzuführen. Anhand von Beispielen wird gezeigt, dass diese Algorithmen im Sinne der kompetitiven Analyse gegenüber einem fiktiven offline Algorithmus nicht kompetitiv sind.

Die Implementierung der Algorithmen wird anhand von Klassendiagrammen kurz erläutert.

Durch Simulationen mit realitätsnahen Daten kann gezeigt werden, dass die Matchingkosten der beiden besten Algorithmen rund 40% über den optimalen Kosten liegen.

# Inhaltsverzeichnis

# 1 Einführung

## 1.1 Online-Matching

Mit der zunehmenden Verbreitung des Internets und den immer häufiger vorhandenen permanenten Verbindungen in Privathaushalten, nimmt die Popularität von Internetspielen zu. Auch Spielkonsolenhersteller haben diesen Trend erkannt und sehen neuerdings für ihre Hardware Internetanschluss vor. Der Gedanke, sich mit einem anderen Menschen zu messen, scheint aufregender als der, mit einer Maschine zu spielen.

Die Frage ist: Wie finden sich die Spieler gegenseitig? Den Vorgang des automatischen Zusammenführens von Spielern im Internet bezeichnen wir als *Online-Matching*. Dieses besteht zum einen aus dem Sammeln der Spielwilligen und andererseits aus dem Formieren von Spielen. Für das Sammeln haben diverse kommerzielle und nicht kommerzielle Organisationen Server eingerichtet, bei denen sich Spielwillige anmelden können [1]. Der Matchingprozess kann manuell oder automatisch erfolgen. Gewisse Spiele wie *Die Siedler* [2] sehen *Spielertische* vor, die von einem Spieler eröffnet werden. Das Spiel startet, wenn sich genug Spieler an einem Spielertisch eingetragen haben. Benutzerfreundlicher ist ein automatisierter Matchingprozess, für dessen Auslösung die Äusserung des Spielwunsches genügt. Bestehende Lösungen automatischer Matchingprozesse beschränken sich oft darauf, Spieler sofort zu matchen, sobald festgestellt wurde, dass sie sich für das gleiche Spiel angemeldet haben. Diese Vorgehensweise bezeichnen wir als *Greedy*-Matching. Der Vorteil des Greedy-Matchings liegt in seiner Einfachheit. Der Nachteil liegt darin, dass man mit ungeeigneten Spielern zusammentreffen kann. Welcher Profi will schon mit einem blutigen Anfänger spielen? Besser wäre ein Matchingverfahren, das die Spielpartner vor dem Zusammenführen aufgrund ihrer Eigenschaften einordnet.

*XBox* [3], die neue Spielkonsole von Microsoft, schlägt diesen Weg ein. Die Spieler können zwischen zwei verschiedenen Matchingoptionen wählen, welche Mitspieler aufgrund von Kriterien aussuchen. Zum einen können mit *QuickMatch* geographisch nahe Spieler gefunden werden, was die Signallaufzeit verkürzt [1], zum anderen findet *OptiMatch* Mitspieler nach Vorgaben [4].

Wir wünschen uns Algorithmen, die ein kriterienbasiertes Matching leisten. Die Algorithmen sollen genau analysiert und die Ergebnisse für die Weiterverarbeitung frei verfügbar sein.

[1] Kurze Signallaufzeiten sind für Spiele interessant, bei denen der Spielverlauf selbst durch kleine Verzögerungen beeinträchtigt wird, z.B. Actionspiele. Für Denkspiele sind Signallaufzeiten unbedeutend.

## 1.2 Zu dieser Diplomarbeit

Anstoss für diese Diplomarbeit war die Idee, das in der Schweiz populäre Jass[2]-Computerspiel *Stöck Wyys Stich Platinum* [5] mit einem automatischen Matching auszustatten. Bei der aktuellen Version dieses Spiels kann entweder mit drei Computerspielern oder mit menschlichen Spielpartnern über das Internet gejasst werden. Das Zusammenführen der Spieler über das Internet geschieht über die manuelle Eingabe der IP-Adresse des jeweiligen Mitspielers.

Ziel der vorliegenden Diplomarbeit ist es, Algorithmen für das automatische Matching zu entwickeln und zu analysieren. Insbesondere sollen die Algorithmen in der Lage sein, ein kriterienbasiertes Matching durchzuführen und dadurch Spieler mit ähnlichen Voraussetzungen zusammenzuführen. Nebst der Bereitstellung von Algorithmen ist auch deren genaue Analyse für eine wissenschaftliche Arbeit unabdingbar.

### 1.2.1 Ablauf der Diplomarbeit

Der Ablauf der Diplomarbeit kann grob in folgende Abschnitte unterteilt werden:

- **Theoretisches Modell:** Nach der Einarbeitung in die Materie galt es zunächst, das Online-Matching theoretisch zu modellieren. Die dort erarbeiteten Instrumente und Definitionen standen für die weitere Arbeit zur Verfügung. Teil des theoretischen Modells ist auch ein Güte-Mass, welches Aussagen über die Qualität des Matchings erlaubt. Hierfür wurde eine Formel hergeleitet und motiviert.

- **Algorithmenentwicklung:** Der Kern dieser Diplomarbeit war die Entwicklung von geeigneten Algorithmen für das kriterienbasierte Matching. Von mehreren Ideen wurden drei weiterverfolgt.

- **Implementierung:** Nebst den drei neu entwickelten Algorithmen, dem Greedy-Verfahren und einem fiktiven, optimalen Algorithmus wurde auch eine Simulationsumgebung implementiert, die es erlaubt, das Verhalten der Algorithmen in einer realitätsnahen Situation zu untersuchen.

- **Simulation:** Die sorgfältig erstellten Simulationsdaten decken Durchschnittssituationen und Worst-Case-Szenarien ab. Die Algorithmen wurden mit diesen Daten betrieben und ihre Resultate ausgewertet.

[2] Jass ist ein in der Schweiz in vielen Ausprägungen bekanntes Kartenspiel, das in der Regel von vier Spielern bestritten wird.

- **Schreiben des Berichts:** Nach Abschluss der Forschungsarbeit wurde zuerst ein Bericht auf deutsch geschrieben, anschliessend wurden die wichtigsten Resultate in kompakter Form auf englisch abgefasst.

### 1.2.2 Aufbau des Berichts

Der Aufbau des Berichts entspricht ungefähr dem Ablauf der Diplomarbeit.

Aus der Einarbeitungszeit ist ausserdem das Kapitel 2 entsprungen, das eine kurze und prägnante Übersicht über *Online-Computing* gibt und dessen Zusammenhang mit dem Online-Matching erläutert. Kaptitel 3 und 4 liefern eine genaue Definition und das theoretische Modell des Matchingproblems. Im 5. Kapitel werden die Algorithmen vorgestellt. Genauere Erläuterungen zu deren Implementierung erhält man im Kapitel 6. Im 7. Kapitel werden die Simulationen erläutert und ausgewertet. Das 8. und letzte Kapitel enthält neben Verbesserungs- und Erweiterungsvorschlägen für die Algorithmen auch ein Fazit über die Arbeit.

# 2 Online-Computing

## 2.1 Einführung

### 2.1.1 Online-Algorithmen

Beim Zusammenführen von Spielern darf der Zeitaspekt nicht aus den Augen verloren werden. Zum Vornherein ist nicht bekannt, ob und wann sich Spieler für ein Spiel anmelden werden. In der Computerwissenschaft bezeichnet man Berechnungen, für die der Input nicht a priori sondern erst nach und nach bekannt wird, als *Online-Computing*. Trotz Unkenntnis des weiteren Inputs soll basierend auf dem aktuellen Wissensstand ein gutes Resultat errechnet werden. Algorithmen, die solche Berechnungen leisten, heissen *Online-Algorithmen* [7].

Für die Veranschaulichung von Online-Computing wird oft das Miet-Ski Problem zitiert: Stellen wir uns vor, wir gehen zum ersten Mal in unserem leben Skifahren. Im Skigeschäft erfahren wir, dass die Ski-Miete pro Tag 50€ kostet, während bei der Anschaffung eines paar Skis Kosten von 500€ anfallen. Die Frage ist, ob wir die Ski mieten oder kaufen. Wüssten wir im Voraus, dass wir in Zukunft mindestens 10 mal Skifahren werden, wäre die Entscheidung leicht zu treffen. Die Zukunft ist aber unbekannt und wir müssen dennoch eine Entscheidung treffen, wir sind also mit einem *Online-Problem* konfrontiert.

Nehmen wir an, unser Online-Algorithmus funktioniert wie folgt: Miete die Skis 9 Mal, danach kaufe sie. Gehen wir höchstens neun mal Skifahren, sind die Kosten des Algorithmus optimal, gehen wir genau 10 Mal, sind die Kosten $\frac{9*50+500}{500} = 1.9$ Mal höher als die optimalen Kosten, was für den beschriebenen Algorithmus zugleich der Worst-Case ist. Führen wir das diskrete Miet-Ski Problem in ein stetiges über, indem wir die Mietkosten gegen 0 gehen lassen, erhalten wir im Worst-Case doppelt so hohe Kosten wie im optimalen Fall.

### 2.1.2 Kompetitive Analyse

Die Bewertung von Online-Algorithmen erfolgt über den Vergleich mit einem fiktiven offline Algorithmus $OPT$, der die ganze Inputsequenz im Voraus kennt und desshalb Entscheidungen optimal treffen kann, was die Kosten minimiert. Wie im letzten Abschnitt besprochen, verursacht der Online-Algorithmus $A$ im Worst-Case 2 Mal höhere Kosten wie $OPT$, man sagt auch: $A$ ist 2-*kompetitiv*.

**Definition** Ein Online-Algorithmus $A$ ist *c-kompetitiv*, wenn es eine Konstante $b$ gibt und für jede Eingabesequenz $\sigma$ gilt:

$$cost_A(\sigma) \leq c \cdot cost_{OPT}(\sigma) + b \tag{1}$$

Mit der Kompetitivität eines Online-Algorithmus wird die Kostenobergrenze im Worst-Case gegenüber einem fiktiven offline Algorithmus $OPT$ bezeichnet. Die Berechnung der Kompetitivität eines Online- Algorithmus wird als *kompetitive Analyse* bezeichnet [6].

## 2.2 Online-Probleme in der Computerwissenschaft

In der Computerwissenschaft werden wir häufig mit Online-Problemen konfrontiert. Oft treten sie in den Bereichen Scheduling, Lastverteilung und Netzwerkrouting auf. Exemplarisch betrachten wir das *Paging-Online-Problem*, bei dem es um das Ein- und Auslagern von Speicherseiten geht. Stellen wir uns eine zweistufige Speicherhierarchie vor, mit einem langsamen und einem schnellen Speicher, dem Cache. Das Cache kann bedeutend weniger Speicherseiten als der langsame Speicher einlagern, was eine Auswahl erfordert, welche Seiten dort zu speichern sind. Befindet sich die Seite, auf die zugegriffen wird, bereits im Cache, sprechen wir von einem *Hit*, ansonsten von einem *Fault*. Ein Fault verursacht zusätzliche Kosten, da die Speicherseite erst vom langsamen Speicher geladen werden muss. Da wir nicht im Voraus wissen, auf welche Speicherseiten zugegriffen wird, handelt es sich hier um ein Online-Problem. Der Online-Paging-Algorithmus hat zu entscheiden, welche Speicherseiten im Cache gehalten und welche daraus entfernt werden. Ziel des Algorithmus ist es, durch eine geschickte Ein- und Auslagerungsstrategie die Anzahl der Page-Faults zu reduzieren. Dadurch werden die Kosten minimiert. Meist wird hier die *Least-Recently-Used (LRU)* Strategie angewendet, welche diejenigen Speicherseiten aus dem Speicher ausscheidet, deren letzte Referenzierung zeitlich am Weitesten zurückliegt. Wir verweisen auf [6] für eine kompetitive Analyse des Paging Problems.

# 3 Das $k$-Player-Matching-Problem

Als $k$-Player-Matching-Problem bezeichnen wir das Online-Problem, das schon in der Einführung besprochen wurde: Das Zusammenführen von $k$ Spielern für ein Spiel. Dabei soll darauf geachtet werden, dass die Spieler möglichst gut zueinander passen. Beispielsweise sollen sie die gleiche Spielstärke aufweisen, bei Denkspielen ähnlich lange Bedenkzeiten fordern oder geografisch nicht weit voneinander entfernt sein. Im Folgenden wollen wir diese Eigenschaften der Spieler als Kriterien und ihren Wert als Kriterienwerte bezeichnen.

Wir beschränken uns im Rahmen dieser Diplomarbeit auf symmetrisches Matching: Ein Spieler soll keine Wünsche[3] an die Kriterienwerte seiner Mitspieler stellen können, sondern wird aufgrund seiner eigenen Kriterienwerte mit geeigneten Spielpartnern (mit ähnlichen Kriterienwerten) zusammengeführt. Durch diese Einschränkung entfallen einige Folgeprobleme: Nehmen wir an, ein schwacher Spieler möchte in einem 4-Spieler Spiel mit starken Spielern gematcht werden und ein starker Spieler sucht schwache Mitspieler. Sobald die beiden zusammengeführt sind, entsteht eine *Deadlock* Situation da ein weiterer Spieler die gestellte Bedingung eines bereits gematchten Spielers nicht erfüllen kann. Ein weiteres Problem ergibt sich beim Führen von Ranglisten: Spielen ungleiche Spieler zusammen, muss abgewogen werden, wie ein Sieg zu gewichten ist. Ein Spieler, der sich nur schwache Gegner aussucht, könnte sich auf diesem Weg ungerechtfertigt profilieren.

Das Zusammenführen von Spielern bedeutet immer einen Trade-Off zwischen Übereinstimmung der Kriterienwerte und Dauer des Matchings. Soll das Matching schnell erfolgen, können weniger Spieler abgewartet werden und die Matchinggenauigkeit leidet. Wir fordern, dass das Matching innerhalb einer festgelegten Höchstwartezeit erfolgt, wird diese überschritten, übernimmt der Computer die Rolle eines fehlenden Spielers.

**Definition** Das $k$-Player-Matching-Problem besteht darin, dass $k$ durch ein Computernetz verbundene Spieler, welche (nicht notwendigerweise gleichzeitig) die Suche nach Mitspielern starten, automatisch und unter Einhaltung der symmetrischen Matchingkriterien und einer festgelegten Höchstwartezeit, zusammengeführt werden.

Das $k$-Player-Matching-Problem unterscheidet sich von den im letzten Kapitel besprochenen Online-Problemen dadurch, dass Entscheidungen nicht

[3]Wünsche können dann spezifiziert werden, wenn diese zugleich die eigenen Eigenschaften widerspiegeln.

unmittelbar nach dem Eintreffen eines neuen Spielers gefällt werden müssen, sondern erst nach Ablauf einer Wartezeit. Die Wartezeit, die nach oben begrenzt ist, dient dazu, die Spieler möglichst gut untereinander zu matchen.

Wir schlagen mehrere Algorithmen für die Lösung des $k$-player-Matching-Problems vor, die im folgenden Abschnitt einzeln besprochen werden. Um festzustellen, wie gut die Algorithmen sind, führen wir ein *Güte-Mass* ein, mit dem die Kosten (oder die Qualität) des Matchings ermittelt werden können. Mit Hilfe dieses Güte-Masses werden wir zeigen, dass die Algorithmen gegenüber eines fiktiven, optimalen offline Algorithmus $OPT$ im Average-Case gute Resultate liefern, während sie im Worst-Case im Sinne der kompetitiven Analyse nicht kompetitiv sind.

# 4 Theoretisches Modell des *k*-Player-Matching-Problems

## 4.1 Definitionen

Um später eindeutige Begriffe zur Verfügung zu haben, erstellen wir ein theoretisches Modell des $k$-Player-Matching-Problems. Die darin enthaltenen Definitionen lassen sich in mehrere Kategorien ordnen:

### Spieler und Spiele

- $\mathcal{P}$ ist eine Menge von Spielern (Players)
- $P_i \in \mathcal{P}$ ist ein einzelner Spieler, $i$ ist seine eindeutige Identifikationsnummer. Spielt der gleiche Spieler nochmals, erhält er eine neue Identifiktionsnummer. $P_{COMP}$ bezeichnet einen Computerspieler, von denen $\infty$-viele zur Verfügung stehen.
- $\mathcal{G}$ ist eine Menge von Spielen (Games)
- $G_j \in \mathcal{G}$ ist ein $k$-Tupel, der die $k$ Spieler des Spiels $G_j$ enthält: $(P_1, P_2, ..., P_k)$

### Kriterien und deren Werte

- Die Funktion $\gamma_d(P_i) : \mathcal{P} \rightarrow \mathbb{R}^{[0,1]}$ ermittelt den Wert des Kriteriums $C_d$ für den Spieler $P_i$
- Die Funktion $q_d(G_j) : \mathcal{G} \rightarrow \mathbb{R}^{[0,1]}$ ermittelt die Matchingkosten des Spiels $G_j$ bezüglich des Kriteriums $C_d$, wobei gilt: $q_d(G_j) = \max \gamma_d(P \in G_j) - \min \gamma_d(P \in G_j)$. Es wird dabei die Ausdehnung des Bereichs berechnet, über den sich die Werte des Kriteriums $C_d$ innerhalb eines Spiels $G_j$ streuen. Ferner gilt: $q_d(G_j) = 1$, falls $P_{COMP} \in G_j$.

### Zeitfunktionen

- Die Funktion $\tau_1(P_i) : \mathcal{P} \rightarrow \mathbb{R}^+$ liefert die Zeit, wann ein Spieler $P_i$ die Suche nach Mitspielern startet.
- Die Funktion $\tau_2(P_i) : \mathcal{P} \rightarrow \mathbb{R}^+$ ermittelt die Zeit, wann ein Spieler $P_i$ für ein Spiel $G_j$ gematcht wird.

- $\tau_{max}$ ist ein konstanter Wert, der festlegt, wie lange ein Matching höchstens dauern darf. Ist $\tau_{max}$ erreicht, ohne dass ein Matching erzielt wurde, kommen Computerspieler zum Einsatz. Es gilt $\tau_2 - \tau_1 \leq t_{max}$.
- Die Funktion $\tau(P_i) : \mathcal{P} \rightarrow \mathbb{R}^+$ ermittelt die Zeit, die ein Spieler $P_i$ auf die Zuteilung zu einem Spiel $G_j$ wartet. $\tau(P_i) = \tau_2(P_i) - \tau_1(P_i)$. Ferner gilt: $\tau(P_{COMP}) = \tau_{max}$.

**In-/Outputsequenz und Arrival-Rate**

- Der Input des Algorithmus, der das $k$-Player-Matching-Problem löst, ist eine Spielersequenz aus $\mathcal{P} : \sigma = P_1, P_2, ..., P_N$.
- Der Output $\pi : \mathcal{P} \rightarrow \mathcal{G}$ des Algorithmus ist eine Sequenz von Spielen $(G_1, G_2, ..., G_M)$, $G_j \in \mathcal{G}$.
- Die *Arrival-Rate* $a$ einer Sequenz $\sigma$ berechnet sich aus der Anzahl Spielern pro Zeiteinheit.

## 4.2 Berechnung der Matchingqualität

Die Qualität eines Matchings aus $\pi$ über der Eingabesequenz $\sigma$ berechnet sich aus der Genauigkeit, mit der die Matchingkriterien erfüllt werden und der Zeit, die das Matching in Anspruch nimmt. Die Matchingqualität ist dann hoch, wenn das Matching aus Sicht des einzelnen Spielers schnell erfolgt und die Kriterien der Spieler innerhalb eines Spiels möglichst wenig voneinander abweichen. Wir führen für diesen Umstand eine Formel ein, mit der die *Gesamtkosten* einer Eingabesequenz $\sigma$ berechnet werden können:

$$cost(\pi, \sigma) = \sum_{j=1}^{M} \left( \sum_{d=1}^{D} f_d \cdot q_d(G_j) + \sum_{P_i \in G_j} \frac{\tau(P_i)}{\tau_{max}} \right) \tag{2}$$

Die linke Summe innerhalb der Klammer steht für die Kriterienkosten, die rechte für die Zeitkosten. $M$ sei die Anzahl Spiele (oder die Länge der Outputsequenz $\pi$) und $D$ die Anzahl Kriterien. Der Faktor $f_d$ dient zur Gewichtung der Matchinggenauigkeit der einzelnen Kriterien untereinander und gegenüber der Matchingzeit. Sollen alle Kriterien untereinander gleich gewichtet werden, ist $f_d = \frac{k}{D}$, für $d = 1...D$ eine gute Wahl, da dies die Kosten pro Spieler auf 1 normiert. Damit die Zeit gegenüber den Kriterien nicht über- oder unterbewertet wird, dividieren wir diese durch $\tau_{max}$, was ebenfalls eine Normierung pro Spieler auf 1 bewirkt. Diese Division erschwert es zwar, Matchingkosten für verschiedene $\tau_{max}$ miteinander zu vergleichen, da

die durch ein grösseres $\tau_{max}$ ermöglichten längeren Wartezeiten schliesslich kleinere Zeitkosten ergeben können. Wichtiger ist aber, dem Zeitaspekt und der Genauigkeit des Matchings ein ähnliches Gewicht zu geben, zumal $\tau_{max}$ an die Wartebereitschaft der Spieler angepasst wird und nicht ständigem Wechsel unterliegt.

Ein optimales Matching ist das, mit den kleinsten Kosten: $A(\sigma)$ ist eine Sequenz von Spielen, die von Algorithmus $A$ bei Input $\sigma$ erzeugt wird. $cost_A(\sigma)$ bezeichnet die Kosten des Algorithmus $A$ über einer Sequenz $\sigma$, welche mit $cost(A(\sigma), \sigma)$ ermittelt wird. Die Kosten eines optimalen offline Algorithmus $OPT$ für die Sequenz $\pi$ sind: $cost_{OPT}(\sigma) = \min_\pi cost(\pi, \sigma)$.

# 5 Algorithmen für das $k$-Player-Matching-Problem

Wir stellen Online-Algorithmen vor, die das $k$-Player-Matching-Problem lösen. Dabei soll das Matching möglichst geringe Kosten verursachen. Anschliessend zeigen wir anhand von Beispielen, dass diese Algorithmen im Average-Case durchaus gut, gegenüber einem optimalen offline Algorithmus $OPT$ im Sinne der kompetitiven Analyse jedoch nicht kompetitiv sind.

## 5.1 Der Greedy-Algorithmus $A_G$

### 5.1.1 Funktionsweise des Algorithmus

Der Greedy Algorithmus wartet nach Ankunft eines Spielers, bis $k-1$ weitere Spieler eingetroffen sind und matcht diese dann, ohne Rücksicht auf ihre Kriterienwerte, zu einem neuen Spiel $G$. Sollten innerhalb von $\tau_{max}$ nach Eintreffen des ersten Spielers weniger als $k$ Spieler für ein Matching zur Verfügung stehen, kommen Computerspieler $P_{COMP}$ zum Einsatz. Durch seine Vorgehensweise verursacht der Greedy-Algorithmus in der Regel wenig Zeitkosten, dafür umso mehr Kriterienkosten, da auf diese überhaupt keine Rücksicht genommen wird.

### 5.1.2 Analyse

In den im Kapitel 7 beschriebenen Tests zeigt sich, dass der Greedy Algorithmus bei kleinen Arrival-Raten im Vergleich zu komplexeren Algorithmen durchaus konkurrenzfähig ist. Die Matching-Kosten liegen bei $A_G$ im Average-Case etwa doppelt so hoch, wie das Optimum.

**Kostenabschätzung** Bevor wir zeigen, dass $A_G$ im Vergleich zu $OPT$ im Sinne der kompetitiven Analyse nicht kompetitiv ist, leiten wir eine Formel her, mit welcher sich die Kosten eines durch $A_G$ erstellten Matchings abschätzen lassen.
Zunächst stellen wir fest, dass die Kriteriendifferenz $|\gamma_d(P_x) - \gamma_d(P_y)|$ von zwei zufällig ausgewählten Spielern $P_x$ und $P_y$ im Erwartungswert $\frac{1}{3}$ ist. Für $k$ zufällig ausgewählte Spieler eines Spiels $G$ gilt: $\max \gamma_d(P_x \in G) - \min \gamma_d(P_y \in G) = \frac{k-1}{k+1}$. Für ein Kriterium $C_d$ erhalten wir somit: $q_d(G) = \frac{k-1}{k+1}$.
Um eine Formel für die Zeitkosten herzuleiten, nehmen wir vereinfachend an, dass die Spieler in regelmässigen Abständen gemäss ihrer Arrival-Rate $a$ beim Algorithmus eintreffen. Der erste Spieler muss auf $k-1$ weitere Spieler warten, der zweite auf $k-2$. Der letzte Spieler hat gar keine Wartezeit,

da das Matching unmittelbar nach dessen Ankunft erfolgt. Die Summe dieser Wartezeiten beträgt $\frac{k(k-1)}{2}$, die Anpassung an die Arrival-Rate $a$ erfolgt durch Multiplikation mit $\frac{1}{a}$. Für ein Spiel $G$ gilt somit: $\sum_{P_i \in G} \tau(P_i) = \frac{k(k-1)}{2a}$. Gehen wir von *einem* Kriterium aus ($D = 1$) und folgen dem Vorschlag, den Kriteriengewichtungsfaktor $f_d = \frac{k}{D}$ zu setzen, erhalten wir durch Anpassen von (2) folgende Näherungsformel für die Durchschnittskosten pro Spiel:

$$cost(G_j) = \frac{k(k-1)}{k+1} + \frac{k(k-1)}{2a \cdot \tau_{max}} \tag{3}$$

Diese Formel ist eine Näherung, da sie das Matching von Computerspielern $P_{COMP}$ nicht berücksichtigt. Gute Resultate werden geliefert, falls $a$ und $\tau_{max}$ so gewählt werden, dass innerhalb von $\tau_{max}$ mit grosser Wahrscheinlichkeit mindestens $k$ Spieler eintreffen. Wir wollen diesen Sachverhalt nachfolgend genauer untersuchen.

Nehmen wir für die Ankunft der Spieler eine Poisson-Verteilung und eine konstante Arrival-Rate $a$ an. Wir führen eine Zufallsvariable $X$ ein, die beschreibt, wie viele Spieler innerhalb von $\tau_{max}$ beim Algorithmus eintreffen. Für den Erwartungswert von $X$ gilt: $E(X) = \lambda = a \cdot \tau_{max}$. Die Wahrscheinlichkeit, dass innerhalb der Zeitspanne $\tau_{max}$ genau $x$ Spieler ankommen, ist $P(x, \lambda) = \lim_{n \to \infty} P_n(x) = \frac{\lambda^x \cdot e^{-\lambda}}{x!}$. Computerspieler werden immer dann gematcht, wenn innerhalb von $\tau_{max}$ mindestens einer, jedoch weniger als $k$ Spieler eintreffen:

$$Prob(P_{COMP} \in G) = \sum_{x=1}^{k-1} \frac{\lambda^x \cdot e^{-\lambda}}{x!} \tag{4}$$

Für die Abschätzung der Genauigkeit von (3) stellen wir die Spielkostenobergrenze zu $cost(P_{COMP} \in G_j) \leq 2k$ fest, welche durch Ausschöpfen der maximalen Wartezeit für jeden Spieler und der Strafkosten beim Matching eines Computerplayers entsteht. Das Kostenverhältnis zwischen Obergrenze und Durchschnittskosten beträgt $\frac{cost(G_j)}{2k}$. Um zu ermitteln, wie viele Spieler $\in P_{COMP}$ sein dürfen, wobei eine Genauigkeit von $\delta$ gewährleistet sein muss, ist $\frac{cost(G_j)}{2k}$ mit $cost(G_j) \cdot \delta$ zu multiplizieren. Da wir die Anzahl Spieler $\in P_{COMP}$ kennen, können wir die Formel zur Berechnung des maximalen Fehlers herbeiziehen. Durch umformen erhalten wir:

$$\delta_{cost(G_j)} \leq \frac{2 \cdot k \cdot Prob(P_{COMP} \in G)}{cost^2(G_j)} \tag{5}$$

**Beispiel** Wir gehen von einem Spiel mit zwei Spielern ($k = 2$) aus, welche bereit sind, höchstens 5 Sekunden auf ein Matching zu warten ($\tau_{max} =$

5). Die Arrival-Rate $a$ beträgt 3 Spieler $\cdot s^{-1}$. Wir erhalten durch Einsetzen dieser Werte in obige Formeln $cost(G_j) = 0.733$ und $Prob(P_{COMP} \in G) = 4.59 \cdot 10^{-6}$. Für den Fehler gilt: $\delta_{cost(G_j)} \leq 3.41 \cdot 10^{-5}$.

**Worst-Case-Szenario** Im Worst-Case-Szenario können die vom Greedy-Algorithmus $A_G$ verursachten Matchingkosten im Vergleich zu einem optimalen Algorithmus $OPT$ beliebig hoch sein, wie das folgende Beispiel verdeutlicht: Sei $D = 1$ und $k = 2$. Nehmen wir an, es treffen wie in Abbildung 1 dargestellt innert kurzer Zeit (Zeitdifferenz $\epsilon$ mit $\epsilon \to 0$) die vier Spieler $P_1$ bis $P_4$ ein, wobei die Kriterienwerte dieser Spieler alternierend 0 und 1 sind.

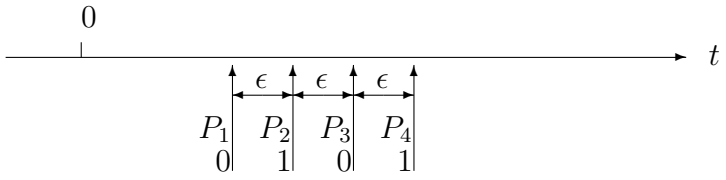


Abbildung 1: Worst-Case-Szenario für $A_G$

$A_G$ matcht die aufeinanderfolgenden Spieler $P_1$ und $P_2$, sowie $P_3$ und $P_4$ zu einem Spiel und verursacht dabei Kriterienkosten, da gilt: $q(P_1, P_2) = 1$ und $q(P_3, P_4) = 1$. Die Zeitkosten gehen gegen 0. Ein optimaler Algorithmus $OPT$ matcht $P_1$ mit $P_3$ und $P_2$ mit $P_4$. Dabei fallen keine Kosten an, denn die Kriterien dieser Spieler stimmen überein.

Da das beschriebene Szenario von $OPT$ ohne Kosten gematcht wird, während $A_G$ Kosten verursacht, gilt $A_G$ im Sinne der kompetitiven Analyse als nicht kompetitiv.

## 5.2 Der Periodic-Match-Algorithmus $A_{PM}$

### 5.2.1 Funktionsweise des Algorithmus

Der Periodic-Match-Algorithmus ist für Anwendungen geeignet, die neben der Zeit nur *ein* weiteres Matchingkriterium aufweisen ($D = 1$). Die Funktionsweise ist wie folgt: Speichere alle eintreffenden Spieler in einer FIFO-Queue $Q$. $m$ bezeichne die Anzahl Spieler, die sich in $Q$ befinden. Sortiere in periodischen Zeitabständen die ersten $n = m - m \bmod k$ Spieler aus $Q$ bezüglich ihres Kriterienwertes $\gamma(P_x)$ und matche jeweils $k$ aufeinanderfolgende Spieler als neues Spiel $G$. Diese werden aus $Q$ entfernt. Verschiebe anschliessend die in $Q$ verbleibenden $m - n$ Spieler an den Queueanfang. Die Beschriebenen Vorgänge sind in Abbildung 2 dargestellt.

Befinden sich nach Zeit $\tau_{max}$ weniger als $k$ Spieler in $Q$, werden Computerspieler $P_{COMP}$ hinzugematcht. Als Startzeitpunkt einer Periode gilt jeweils die Ankunftszeit des am frühesten eingetroffenen Spielers in $Q$.
Bei ausreichend grosser Ankunftsrate wird der Zyklus verkürzt: Anstatt bis $\tau_{max}$ zu warten, werden die Spieler bereits sortiert, wenn eine bestimmte Anzahl Spieler, z. B. $x \cdot k$, in $Q$ eingetroffen sind. Dies reduziert die Wartezeit im Einzelnen und lässt konkrete Aussagen über den Speicherbedarf des Algorithmus zu.
Der Parameter $x$ ist entweder im Voraus zu definieren oder kann bei schwankender Arrival-Rate vom Algorithmus selbst adaptiert werden (Adaptiver-Periodic-Match-Algorithmus). Wie $x$ für optimale Resultate zu wählen ist, wird im Analyse-Abschnitt besprochen.

**Beispiel** Es sei $D = 1$ und $k = 3$. Aus Übersichtsgründen werden die Spieler $P_1$ bis $P_7$ in Abbildung 2 zusammen mit dem Wert ihres Kriteriums $C$ dargestellt, der über die Funktion $\gamma(P_x)$ zugänglich ist. Zum Zeitpunkt $\tau_{max}$ nach eintreffen von $P_1$ werden die ersten $m - m \bmod k$ Spieler aus $Q$ sortiert und gematcht.

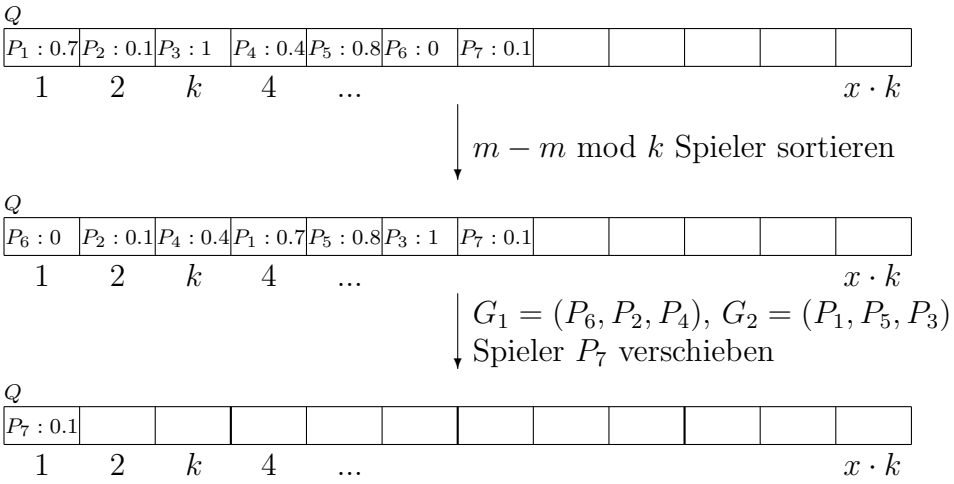


Abbildung 2: Funktionsweise des $A_{PM}$

Wir fassen jeweils $k$ aufeinanderfolgende Spieler zu einem neuen Spiel zusammen und entfernen diese aus $Q$. In unserem Beispiel ist $G_1 = (P_6, P_2, P_4)$ und $G_2 = (P_1, P_5, P_3)$. Der Spieler $P_7$ verbleibt in $Q$. Der Zeitpunkt seines

Eintreffens bestimmt den Startzeitpunkt der nächsten Periode. Ein erneutes Umkopieren, sortieren und Matchen findet demnach statt, wenn der Zeitpunkt $\tau_1(P_7) + \tau_{max}$ erreicht ist oder wenn $Q$ voll ist.

### 5.2.2 Analyse

Es lassen sich Fälle konstruieren, für welche die Kosten des Periodic-Match-Algorithmus $A_{PM}$ im Vergleich zu einem optimalen offline Algorithmus $OPT$ beliebig hoch sind. Dies ist durch die in jedem Fall geforderte Wartezeit (entweder auf $\tau_{max}$ oder auf $x \cdot k$ Spieler) bedingt. Betrachten wir dazu ein kurzes Beispiel: Sei $D = 1$ und $k = 2$. Es treffen nun gleichzeitig die beiden Spieler $P_1$ und $P_2$ ein, wobei gilt: $\gamma(P_1) = \gamma(P_2)$. $OPT$ matcht die beiden Spieler unverzüglich zu einem Spiel $G$, ohne dabei Kosten zu verursachen, da weder Zeit- noch Kriterienmatchingkosten ($q(P_1, P_2) = 0$) anfallen. $A_{PM}$ lässt $\tau_{max}$ Zeit verstreichen, bevor die beiden Spieler zu einem Spiel gematcht werden. Es entstehen Zeitkosten von $2\tau_{max}$. Da $OPT$ in diesem Beispiel gar keine Kosten verursacht, gilt $A_{PM}$ im Sinne der kompetitiven Analyse als nicht kompetitiv.

Wie wir später im Kapitel 7 noch sehen werden, sind die Matching-Kosten des $A_{PM}$ im Average-Case im Durchschnitt 64% höher als die optimalen Matching-Kosten.

**Kostenabschätzung** Wie beim Greedy-Algorithmus, lassen sich auch beim Periodic-Match-Algorithmus die erwarteten Kosten pro Spiel abschätzen. $A_{PM}$ wartet nicht auf $k$, sondern auf $n = x \cdot k (x \in \mathbb{N}^+ \backslash \{0\})$ Spieler, bevor diese zu Spielen gematcht werden. Für eine Approximation der Kriterienkosten gilt: $q_d(G) = \frac{k-1}{n+1}$. Die Zeitkosten betragen $\sum_{P_i \in G} \tau(P_i) = \frac{n \cdot (n-1)}{2 \cdot a} \cdot \frac{1}{x}$. Die Division durch $x$ ist nötig, da wir nicht Wartekosten für $n$ Spieler, sondern Wartekosten pro Spiel berechnen ($\frac{n}{x} = k$). Weitere Herleitungsschritte sind unter 5.1.2 nachzulesen. Für $f_d = \frac{k}{D}$ und $D = 1$ leiten wir aus (2) folgende Näherungsformel für die Kosten her:

$$cost(G_j) = \frac{k(k-1)}{n+1} + \frac{k(n-1)}{2a \cdot \tau_{max}} \tag{6}$$

Für die Fehlerabschätzung können die im Abschnitt 5.1.2 hergeleiteten Formeln (4) und (5) verwendet werden.

**Parameterwahl** Durch Ableitung der Kostenformel nach $n$ erhält man eine weitere Formel, mit deren Hilfe die Matchingkosten für gegebenes $k$, $a$ und $\tau_{max}$ minimiert werden können:

$$\frac{d}{dn}\left(\frac{k(k-1)}{n+1}+\frac{k(n-1)}{2a\cdot\tau_{max}}\right)=\frac{k-k^2}{(n+1)^2}+\frac{k}{2a\cdot\tau_{max}}$$

Um ein optimales, die Werte der Kostenformel (6) minimierendes $n_{opt}$ zu finden, muss die im letzten Schritt durch Ableitung gewonnene Formel zuerst gleich Null gesetzt und anschliessend nach $n$ aufgelöst werden. Von den beiden Lösungen ist nur die positive von Belang:

$$n_{opt}=\sqrt{a}\cdot\sqrt{2\cdot(k-1)}\cdot\sqrt{t_{max}}-1 \tag{7}$$

**Beispiel** Sei die Arrival-Rate $a=10$, die Anzahl Spieler pro Spiel $k=2$ und die Höchstwartezeit auf ein Matching $\tau_{max}=5$. Durch einsetzen dieser Werte in (7) erhalten wir $n=9$. Der optimale Parameter $x$ für den Algorithmus errechnet sich nun aus $\frac{n}{k}=x=4.5$. Die Matchingkosten werden für $x=4$ oder $x=5$ am Tiefsten liegen.

## 5.3 Der Multi-Queue-Algorithmus $A_{MQ}$

### 5.3.1 Funktionsweise des Algorithmus

Die Idee des Multi-Queue-Algorithmus ist, die Wertebereiche der Matchingkriterien $C$ zu diskretisieren und für jede mögliche Kombination dieser diskretisierten Bereiche untereinander eine eigene Queue zu führen. Die Eintreffenden Spieler werden gemäss den Werten ihrer Kriterien direkt in diese Queues eingefügt. Enthält eine Queue $k$ Spieler, werden diese zu einem neuen Spiel $G$ zusammengeführt und aus der Queue entfernt. Sollten innerhalb der Zeit $\tau_{max}$ nach eintreffen des ersten Spielers einer Queue nicht genügend Einträge für ein Matching zur Verfügung stehen, werden Spieler aus benachbarten Queues für das Matching herangezogen wobei das *Radius-Growth-Verfahren* zum Einsatz kommt: Zuerst wird in den direkt benachbarten Queues nach Spielern gesucht, sind diese Queues leer, werden die übernächsten durchsucht und so weiter. Dies geschieht solange, bis genügend Spieler für ein neues Spiel gefunden wurden. Die Spieler werden aus den benachbarten Queues in der Reihenfolge ihrer Ankunftszeit extrahiert - der Spieler, der am längsten wartet, wird zuerst aus der Queue entfernt. Können in allen Queues zusammen nicht genügend Spieler aufgefunden werden, werden Computerspieler $P_{COMP}$ hinzugematcht.

Per Definition erstrecken sich die Wertebereiche der Kriterien von 0 bis 1. Die Diskretisierungsschrittgrösse verschiedener Kriterien kann dabei unterschiedlich gross sein. Zum Beispiel können die Wertebereiche eines Kriteriums

$C_1$ wie in Abbildung 3 dargestellt in $n$ und die Werte eines zweiten Kriteriums $C_2$ in $m$ Bereiche aufgeteilt sein, was ein Total von $m \cdot n$ Queues ergibt. Auch innerhalb eines einzelnen Kriteriums kann je nach Verteilung der Kriterienwerte eine asymmetrische Aufteilung sinnvoll sein. Im Rahmen dieser Diplomarbeit gehen wir von einer uniformen Verteilung aus und unterteilen die Kriterienwerte dementsprechend.

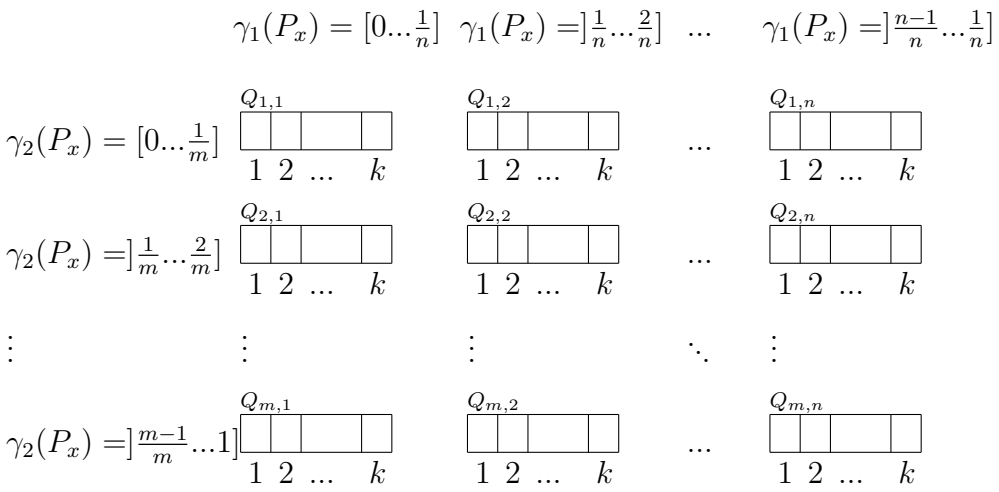


Abbildung 3: Queue-Matrix für zwei Kriterien $C_1$ und $C_2$

### 5.3.2 Analyse

Im Average-Case ist $A_{MQ}$ sehr interessant. Wie wir in Kapitel 7 sehen werden, liegen seine Matching-Kosten im Schnitt 41% über den optimalen Matching-Kosten.

Bedingt durch die Diskretisierung und die damit verbundene Wartezeit auf genügend Spieler in einer Queue, können die Kosten des Multi-Queue-Algorithmus $A_{MQ}$ verglichen mit einem optimalen Algorithmus $OPT$ in konstruierten Beispielen beliebig hoch sein: Sei $D = 1$, $k = 2$ und die Wertdifferenz zweier zum gleichen Zeitpunkt eintreffenden Spieler $P_1$ und $P_2$ vernachlässigbar gering, wie in Abbildung 4 dargestellt.

Es gilt: $|\gamma(P_1) - \gamma(P_2)| = \epsilon$, wobei $\epsilon \to 0$. In diesem Szenario können die beiden Spieler vom Algorithmus $A_{MQ}$ nicht sofort gematcht werden, da die Werte für das Kriterium $C$ jeweils knapp ober- oder unterhalb einer Diskretisierungsschrittgrenze liegen und die Spieler $P_1$ und $P_2$ in unterschiedliche Queues eingeteilt werden. Treffen nun keine weiteren Spieler ein, wird $A_{MQ}$ nach Zeit $\tau_{max}$ in benachbarten Queues nach Spielern suchen und $P_1$ und

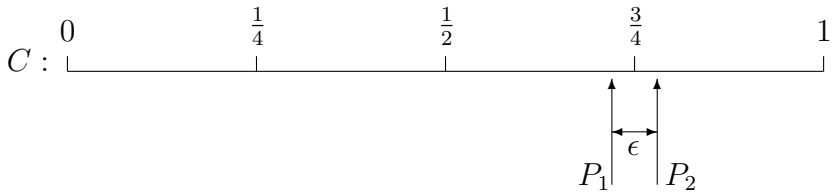


Abbildung 4: Worst-Case-Szenario für $A_{MQ}$

$P_2$ matchen. Gehen wir von $\tau_{max} = 1$ aus, erhalten wir Matchingkosten 2. Ein optimaler Algorithmus $A_{OPT}$ matcht die beiden Spieler ohne Zeitverlust und verursacht keine Kosten. Somit ist $A_{MQ}$ gegenüber $A_{OPT}$ im Sinne der kompetitiven Analyse nicht kompetitiv.

**Kostenabschätzung** Auch für den Multi-Queue-Algorithmus $A_{MQ}$ lässt sich eine Formel finden, mit der sich die Erwarteten Kosten pro Spiel abschätzen lassen. Wir werden an dieser Stelle nur die Schritte besprechen, die von der Herleitung der Kostenformel für den Greedy-Algorithmus (Abschnitt 5.1.2) abweichen.

Da der Wertebereich eines Kriteriums beim $A_{MQ}$ in $r$ Bereiche aufgeteilt ist, beträgt die Kritieriendifferenz von $k$ zufällig ausgewählten Spielern eines Bereichs $q_d(G) = \frac{k-1}{(k+1)\cdot r}$. Für die Abschätzung der Zeitkosten berücksichtigen wir, dass die Arrival-Rate $a$ pro Bereich $\frac{a}{r}$ ist, was schliesslich folgende Formel liefert: $\sum_{P_i \in G} \tau(P_i) = \frac{r \cdot k(k-1)}{2a}$. Für $f_d = \frac{k}{D}$ und $D = 1$ leiten wir aus (2) folgende Näherungsformel für die Kosten von $A_{MQ}$ her:

$$cost(G_j) = \frac{k(k-1)}{r(k+1)} + \frac{r \cdot k(k-1)}{2a \cdot \tau_{max}} \tag{8}$$

Zur Fehlerabschätzung kann die Formel (4) aus dem Abschnitt 5.1.2 verwendet werden, nachdem sie an die Gegebenheiten des $A_{MQ}$ angepasst wurde: Wir führen eine neue Zufallsvariable $Y$ ein, die beschreibt, wie viele Spieler innerhalb von $\tau_{max}$ *einer* der $r$ Queues des Algorithmus eintreffen. Für den Erwartungswert von $Y$ gilt: $E(Y) = \lambda = \frac{a \cdot \tau_{max}}{r}$. Computerspieler $P_{COMP}$ oder Spieler aus benachbarten Queues $P_{NQ}$ werden genau dann gematcht, wenn innerhalb von $\tau_{max}$ mindestens einer, jedoch weniger als $k$ Spieler in einer bestimmten Queue eintreffen:

$$Prob(P_{COMP} \in G \vee P_{NQ} \in G) = \sum_{x=1}^{k-1} \frac{\lambda^x \cdot e^{-\lambda}}{x!} \tag{9}$$

Weil $q(P_{COMP} \in G) \geq q(P_{NQ} \in G)$ gilt, kann die Formel zur Berechnung des Maximalfehlers (5) direkt übernommen werden.

**Parameterwahl** Wie beim Periodic-Match-Algorithmus kann auch hier durch Ableitung der Kostenformel eine neue Formel gewonnen werden, mit welcher für gegebenes $k$, $a$ und $\tau_{max}$ ein optimaler, die Kosten minimierender Parameter $r_{opt}$ gefunden werden kann:

$$\frac{d}{dr}\left(\frac{k(k-1)}{r(k+1)} + \frac{r \cdot k(k-1)}{2a \cdot \tau_{max}}\right) = \frac{k-k^2}{(k+1)\cdot r^2} + \frac{k(k-1)}{2a \cdot \tau_{max}}$$

Die Ableitung wird gleich Null gesetzt und nach $r$ aufgelöst. Von den beiden Lösungen ist nur die positive relevant:

$$r_{opt} = \sqrt{\frac{2a \cdot \tau_{max}}{k+1}} \tag{10}$$

**Beispiel** Sei die Arrival-Rate $a = 10$ Spieler $\cdot s^{-1}$, die Anzahl Spieler pro Spiel $k = 2$ und die Höchstwartezeit auf ein Matching $\tau_{max} = 5$. Durch einsetzten dieser Werte in (10) erhalten wir $r_{opt} = 5.77$. Lassen wir den Multi-Queue-Algorithmus 6 Queues führen, liegen die Matchingkosten am Tiefsten.

## 5.4 Der Difference-Wait-Algorithmus $A_{DW}$

### 5.4.1 Funktionsweise des Algorithmus

Der Difference-Wait-Algorithmus ist für Spielerpaare ($k = 2$) und ein Kriterium ($D = 1$) geeignet. Die Idee des Algorithmus ist, die Spieler zuerst provisorisch zu matchen und erst, falls innerhalb einer gewissen Zeitspanne kein besserer Matchingpartner gefunden werden kann, das provisorische Matching für definitiv zu erklären. Diese Zeitspanne hängt dabei von der Qualität des provisorischen Matchings und der Höchstwartezeit $\tau_{max}$ für einzelne Spieler ab. Wir werden nun alle dem Algorithmus zugrunde liegenden Mechanismen der Reihe nach besprechen.

**Speicherung** Um den besten Matchingpartner zu finden, müssen die Spieler geeignet zwischengespeichert werden. Der Difference-Wait-Algorithmus kennt dafür zwei logische Speicherorte:

- Im *Single-Wait-Space* befindet sich immer dann ein Spieler, wenn der Algorithmus eine ungerade Anzahl Spieler speichert. Falls der Single-Wait-Space besetzt ist, befindet sich dort der überzählige Spieler, der nicht zur Verbesserung der Summe aller provisorischen Matchings beitragen kann.

- In der *Team-Wait-Queue* werden die provisorisch gematchten Spielerpaare solange gespeichert, bis das Matching definitiv wird.

**Matching** Wie geht der Algorithmus vor, wenn ein neuer Spieler eintrifft? Dazu muss unterschieden werden, ob die Team-Wait-Queue und/oder der Single-Wait-Space jeweils leer sind oder Spieler enthalten. Je nachdem muss der Algorithmus andere Schritte ausführen. Wir unterscheiden vier Fälle:

- Single-Wait-Space und Team-Wait-Queue enthalten keine Spieler: In diesem Fall ist kein provisorisches Matching möglich, der neu eintreffende Spieler $P_z$ wird im Single-Wait-Space gespeichert.

- Single-Wait-Space enthält einen Spieler $P_s$, die Team-Wait-Queue ist leer: Der neu eintreffende Spieler $P_z$ wird mit dem Spieler, der im Single-Wait-Space gespeichert ist, provisorisch gematcht und als Paar $(P_s, P_z)$ in die Team-Wait-Queue eingefügt. Die Team-Wait-Queue enthält nun ein Element während der Single-Wait-Space wieder leer ist.

- Der Single-Wait-Space ist leer, die Team-Wait-Queue enthält 1 bis $n$ Paare: Es muss überprüft werden, ob eines dieser provisorischen Matchings $(P_x, P_y)$ durch den neu eintreffenden Spieler $P_z$ verbessert werden kann. Trifft dies zu, wird das provisorische Matching aufgehoben und $P_z$ mit einem der beiden frei werdenden Spieler gematcht. Für den Spieler, der aus dem provisorischen Matching ausscheidet, wird wiederum in der Team-Wait-Queue nach dem besten Matchingpartner gesucht. Der Algorithmus arbeitet dabei wie folgt:

  1. Finde das provisorisch gematchte Team aus der Team-Wait-Queue $(P_x, P_y)$ für das gilt: $q(P_x, P_y) - q(P_x, P_z) = \max$ (Ohne Beschränkung der Allgemeinheit soll $P_y$ dabei den Spieler bezeichnen, der durch $P_z$ ersetzt wird).
  2. Überprüfe nun für das unter Punkt 1. gefundene Team, ob ausserdem gilt: $q(P_x, P_y) > q(P_x, P_z)$.

  Trifft der zweite Punkt zu, hebt der Algorithmus das Matching zwischen $P_x$ und $P_y$ auf und verbindet neu $P_x$ mit $P_z$ zu einem provisorischen

Matching. Für den ausgeschiedenen Spieler wird die ganze Prozedur solange wiederholt, bis kein einzelner Spieler mehr gefunden werden kann, der die Bedingung unter Punkt zwei erfüllt und somit nicht zur Verbesserung eines provisorischen Matchings beitragen kann. Dieser überzählige Spieler wird im Single-Wait-Space platziert.

- Sowohl der Single-Wait-Space als auch die Team-Wait-Queue enthalten Spieler: Die Vorgehensweise entspricht weitgehend der des letzten Punktes. Zusätzlich muss der im Single-Wait-Space gespeicherte Spieler $P_s$ miteinbezogen werden. Dazu führen wir eine weitere Bedingung ein, die nach dem Zutreffen der Bedingung unter Punkt zwei entscheidet, ob der neu eingetroffene Spieler $P_z$ mit einem provisorisch gematchten Spieler $P_x$ oder mit dem im Single-Wait-Space gespeicherten Spieler $P_s$ gematcht wird. Wenn die Bedingung

  3. $(q(P_x, P_z) + q(P_y, P_s)) < (q(P_x, P_y) + q(P_s, P_z))$

  zutrifft, wird das provisorische Matching zwischen $P_x$ und $P_y$ aufgehoben, $P_x$ neu mit $P_z$ verbunden und die ganze Prozedur für $P_y$ wiederholt. $P_s$ bleibt im Single-Wait-Space.
  Trifft die Bedingung unter Punkt drei nicht zu, was früher oder später der Fall ist, wird der aktuell bearbeitete Spieler mit $P_s$ gematcht und in die Team-Wait-Queue eingefügt.

**Zeitaspekt** Neben der Suche nach einem geeigneten Matchingpartner muss der Algorithmus auch dafür sorgen, dass das Matching für einen einzelnen Spieler nicht länger als $\tau_{max}$ dauert. Dafür gibt es zwei voneinander unabhängige Zeitmechanismen:

- Garantie einer Höchstwartezeit $\tau_{max}$ für jeden Spieler. Nach Ablauf dieser Zeit wird der Spieler $P_x$ definitiv gematcht. Wir unterscheiden dabei zwei Fälle:

  1. Befindet sich $P_x$ zum Zeitpunkt $\tau_1(P_x) + \tau_{max}$ gerade im Single-Wait-Space, wird $P_x$ mit einem Computerspieler $P_{COMP}$ gematcht.
  2. Ist ein Spieler $P_x$ zum Zeitpunkt $\tau_1(P_x) + \tau_{max}$ gerade mit dem Spieler $P_y$ provisorisch gematcht, wird dieses Matching definitiv und das Paar $(P_x, P_y)$ wird als neues Spiel $G_j$ ausgegeben und aus der Team-Wait-Queue entfernt.

- Wartezeit eines provisorisch gematchten Paars die dazu dient, auf einen besseren Matchingpartner zu warten. Das Prinzip ist dabei wie folgt:

Stimmen die Kriterien der gematchten Spieler gut überein, wird das provisorische Matching schnell definitiv. Ist das Matching aber mit grossen Kosten verbunden, da die Kriterien der beiden provisorisch gematchten Partner stark voneinander abweichen, wird noch zugewartet, in der Hoffnung, ein besseres Matching mit einem später eintreffenden Spieler zu erzielen. Die Wartezeit, die verstreicht, bevor ein provisorisches Matching eines Spielerpaars $P_x$ und $P_y$ definitiv wird, ist proportional zu $q(P_x, P_y)$. Wir definieren die Funktion $\tau_3(G_j) : \mathcal{G} \rightarrow \mathbb{R}^+$, welche bestimmt, wie viel Zeit verstreicht, bevor ein provisorisches Matching definitiv wird. Es gilt $\tau_3(P_x, P_y) = f_{wait} \cdot q(P_x, P_y)$. Der Wartefaktor $f_{wait}$ kann Werte $\geq 0$ annehmen.
Ein provisorisches Matching $(P_x, P_y)$ wird genau dann definitiv, wenn einer der beteiligten Spieler seit $\tau_{max}$ auf ein Matching wartet oder die Zeit $\tau_3(P_x, P_y)$ für dieses Matching abgelaufen ist. Der früheste dieser drei Zeitpunkte ist dabei entscheidend.

**Beispiel** Um die Funktionsweise des Algorithmus zu verdeutlichen, betrachten wir ein Beispiel, bei dem die Spieler $P_1$ bis $P_5$ gematcht werden. $f_{wait}$ sei 1. Die Spieler werden durch Pfeile dargestellt die mit dem Wert $\gamma(P_x)$ ihres Kriteriums bezeichnet sind. Zu Beginn sind Single-Wait-Space und Team-Wait-Queue leer.

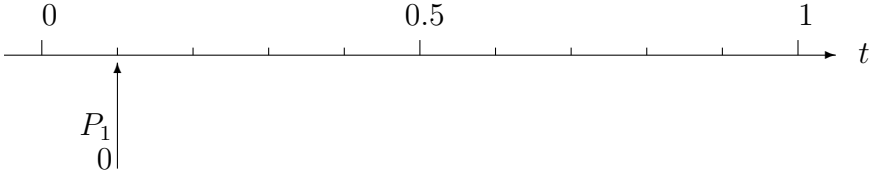


Abbildung 5: Spieler $P_1$ trifft ein

Zum Zeitpunkt $t = 0.1$ trifft der Spieler $P_1$ ein (Abbildung 5). Mangels bereits gespeicherter Spieler kann $P_1$ nicht gematcht werden und wird im Single-Wait-Space platziert. Wir erhalten: Single-Wait-Space = $P_1$, Team-Wait-Queue =$\{\ \}$.

Bei $t = 0.45$ trifft der Spieler $P_2$ ein (Abbildung 6). $P_1$ und $P_2$ werden vom Algorithmus provisorisch gematcht und in die Team-Wait-Queue eingefügt. Der Single-Wait-Space ist wieder leer. Für das provisorisch gematchte Paar $(P_1, P_2)$ erhalten wir $q(P_1, P_2) = 0.4$ was sogleich auch der Wartezeit $\tau_3$ entspricht, bis das Matching definitiv wird. Sollte also bis zum Zeitpunkt

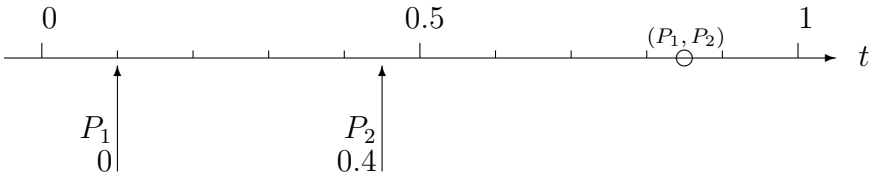


Abbildung 6: Spieler $P_2$ trifft ein

$t = 0.45 + 0.4 = 0.85$ kein besseres Matching möglich sein, werden $P_1$ und $P_2$ definitiv gematcht. Den Zeitpunkt 0.85 haben wir mit einem kleinen Kreis auf dem Zeitstrahl markiert. Wir erhalten: Single-Wait-Space = *nil*, Team-Wait-Queue =$\{(P_1, P_2)\}$.

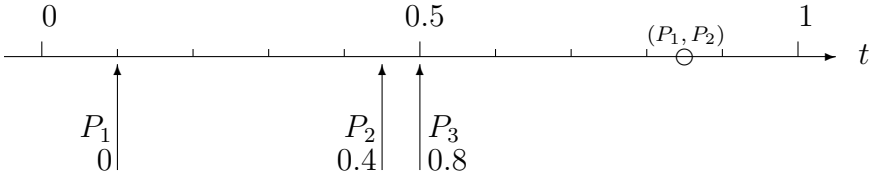


Abbildung 7: Spieler $P_3$ trifft ein

Zum Zeitpunkt $t = 0.5$ trifft der Spieler $P_3$ ein (Abbildung 7). Der Algorithmus überprüft nun, ob durch den Einsatz von $P_3$ ein bestehendes Matching verbessert werden könnte. Wir erhalten für $q(P_1, P_2) - q(P_1, P_3) = 0$ und für $q(P_1, P_2) - q(P_3, P_2) = -0.4$. Das Maximum ist 0. Da dieser Wert nicht *grösser* als 0 ist, was unsere Bedingung für das Aufheben eines bestehenden Matchings ist, wird $P_3$ auf den Single-Wait-Space verbannt. Wir erhalten: Single-Wait-Space = $P_3$, Team-Wait-Queue =$\{(P_1, P_2)\}$.

Bei $t = 0.7$ trifft Spieler $P_4$ ein (Abbildung 8). Der Algorithmus sucht zuerst nach dem bestmöglichen Matching unter Einsatz von $P_4$ und Berücksichtigung der provisorisch gematchten Spieler aus der Team-Wait-Queue und findet als Maximum $q(P_1, P_2) - q(P_4, P_2) = 0.2$. Für Werte $> 0$ ist zudem auch die Bedingung unter Punkt zwei erfüllt: $q(P_1, P_2) > q(P_4, P_2)$. Da zur Zeit der Single-Wait-Space nicht leer ist, muss zudem die Bedingung unter Punkt drei überprüft werden: $(q(P_4, P_2) + q(P_1, P_3)) < (q(P_1, P_2) + q(P_3, P_4))$. Wir erhalten für den ersten Term $0.2 + 0.8 = 1$ und für den zweiten $0.4 + 0.2 = 0.6$. Die Bedingung ist somit nicht erfüllt. Das bedeutet, dass die Kosten aller Mat-

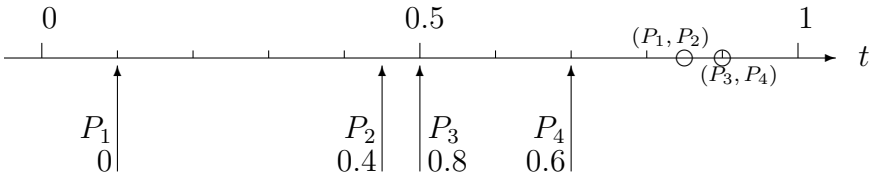


Abbildung 8: Spieler $P_4$ trifft ein

chings dann am kleinsten sind, wenn der neue Spieler $P_4$ mit dem sich auf dem Single-Wait-Space befindlichen Spieler $P_3$ gematcht und als neues Paar in die Team-Wait-Queue eingefügt wird. Wir erhalten: Single-Wait-Space $= nil$, Team-Wait-Queue $=\{(P_1, P_2), (P_3, P_4)\}$.

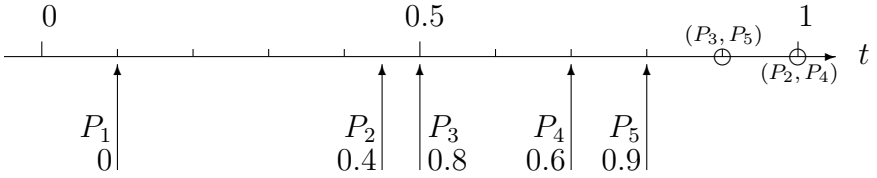


Abbildung 9: Spieler $P_5$ trifft ein

Zum Zeitpunkt $t = 0.8$ trifft der Spieler $P_5$ ein (Abbildung 9). Der Algorithmus sucht wieder nach dem besten Matching für $P_5$ und findet: $q(P_3, P_4) - q(P_3, P_5) = 0.1$. Da die zweite Bedingung auch erfüllt ist, wird das provisorische Matching $(P_3, P_4)$ aufgehoben und $P_3$ neu mit $P_5$ gematcht. Dieses neue Matching wird nach $q(P_3, P_5) = 0.1$ Zeiteinheiten, also bei $t = 0.8 + 0.1 = 0.9$, definitiv. Der frei gewordene Spieler $P_4$ wird wiederum dem Algorithmus zur Bearbeitung übergeben. Dieser sucht in der Team-Wait-Queue nach dem bestmöglichen Matchingpartner. Als Maximum wird $q(P_1, P_2) - q(P_4, P_2) = 0.2$ gefunden. Dieser Wert ist grösser als 0, was ein Auswechseln von $P_1$ durch $P_4$ bedeutet. Das neu gematchte Team $(P_4, P_2)$ wartet noch $q(P_4, P_2) = 0.2$ Zeiteinheiten auf ein besseres Matching. Es wird somit zum Zeitpunkt $t = 0.8 + 0.2 = 1$ definitiv. Für den frei gewordenen Spieler $P_1$ wird der bestmögliche Matchingpartner gesucht. Als Maximum findet der Algorithmus $q(P_2, P_4) - q(P_2, P_1) = -0.2$. Dieser Wert ist kleiner als 0 und erfüllt somit die zweite Bedingung nicht. $P_1$ kann also nicht eingesetzt werden, um das Matching zu verbessern. Er wird auf den Single-Wait-Space

verbannt und muss auf neu eintreffende Spieler warten. Wir erhalten: Single-Wait-Space = $P_1$, Team-Wait-Queue =$\{(P_3, P_5), (P_4, P_2)\}$.
Zum Zeitpunkt $t = 0.9$ wird das Matching für das Spielerpaar $(P_3, P_5)$ definitiv. Es wird aus der Team-Wait-Queue entfernt und als neues Spiel $G_j$ vermittelt.
Bei $t = 1$ wird das Matching für das Spielerpaar $(P_4, P_2)$ definitiv. Da keine weiteren Spieler mehr eintreffen, kann $P_1$ nicht mit einem anderen Spieler gematcht werden und wird zum Zeitpunkt $\tau_1(P_1) + \tau_{max}$ mit einem Computerspieler $P_{COMP}$ gematcht.

Zum Schluss berechnen wir die entstandenen Matchingkosten gemäss Kostenformel (2), die sich aus der Wartezeit auf das Matching und dessen Genauigkeit zusammensetzen. Sei $\tau_{max} = 1$, $f_d = \frac{k}{D}$ und $D = 1$. Es wurden 3 Spiele mit 6 Spielern $P_1$ bis $P_5$ und einem Computerspieler $P_{COMP}$ gematcht. Die Wartekosten für die einzelnen Spieler betrugen: $\tau(P_1) = 1, \tau(P_2) = 0.55, \tau(P_3) = 0.4, \tau(P_4) = 0.3, \tau(P_5) = 0.1$ und $\tau(P_{COMP}) = 1$ (per Definition). Die Kriterienkosten der Spiele sind: $q(G_1) = q(P_3, P_5) = 0.1, q(G_2) = q(P_2, P_4) = 0.2$ und $q(G_3) = q(P_1, P_{COMP}) = 1$ (per Definition). Wir erhalten folgende Matchingkosten: $2{\cdot}0.1{+}0.4{+}0.1{+}2{\cdot}0.2{+}0.55{+}0.3{+}2{\cdot}1{+}1{+}1 = 5.95$. Pro Spiel liegen die Matchingkosten knapp unter 2.

### 5.4.2 Analyse

In Kapitel 7 sehen wir, dass der $A_{DW}$ im Average-Case 39% höhere Matching-Kosten als ein optimaler Algorithmus verursacht. Im Sinne der kompetitiven Analyse gilt der Difference-Wait-Algorithmus jedoch als nicht kompetitiv, wie folgendes Beispiel erläutert:

**Beispiel** Sei $\tau_{max} = 1$. Nehmen wir an, dass die drei Spieler $P_1$, $P_2$ und $P_3$ zum Zeitpunkt 0 eintreffen und alle den Kriterienwert 0 haben (siehe Abbildung 10). $A_{DW}$ wird $P_1$ und $P_2$ sofort matchen, da die Kriterien übereinstimmen. Kurz bevor für $P_3$ die Höchstwartezeit $\tau_{max}$ erreicht ist, trifft ein Spieler $P_4$ ein, der ebenfalls Kriterienwert 0 hat. Dieser wird von $A_{DW}$ unverzüglich mit $P_3$ gematcht, wobei für diesen Zeitkosten $1 - \epsilon$ (mit $\epsilon \to 0$) anfallen. Das gleiche Szenario wiederholt sich ab Zeitpunkt $\tau_{max}$ nach Eintreffen von $P_1$ für die Spieler $P_5$ bis $P_8$, wobei $P_7$ wiederum Zeitkosten 1 verrechnet werden. Wiederholt sich dieses Szenario beliebig oft, verursacht $A_{DW}$ pro Spiel im Erwartungswert Kosten $\frac{1}{2}$.

Wie matcht nun ein optimaler offline Algorithmus $OPT$? Da dieser die ganze Inputsequenz im Voraus kennt, wird der überzählige Spieler $P_1$ mit einem Computerspieler $P_{COMP}$ gematcht, was zur Folge hat, dass anschliessend

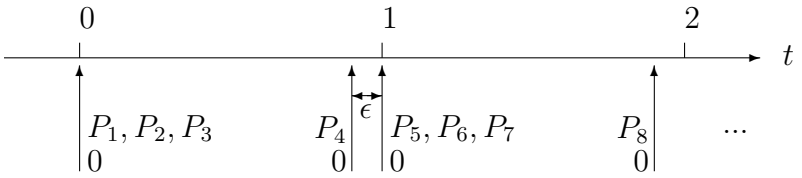


Abbildung 10: Worst-Case Szenario für $A_{DW}$

nur noch Spielerpaare mit gleichem Kriteriumswert und gleicher Ankunftszeit zu matchen sind: $P_2$ mit $P_3$, $P_4$ mit $P_5$ usw. Diese Matchings verursachen keine Kosten. Die hohen Kosten für das Matching von $P_1$ mit einem Computerspieler $P_{COMP}$ werden durch eine grosse Anzahl Spiele amortisiert und gehen im Limes gegen 0.

### 5.4.3 Erweiterung des Difference-Wait-Algorithmus für Spiele mit mehr als zwei Spielern

Die Einschränkung auf zwei Spieler pro Spiel verunmöglicht den Einsatz des $A_{DW}$ für viele Spiele und macht ihn gegenüber den anderen vorgestellten Algorithmen weniger interessant.
Wir beschreiben im Folgenden eine Erweiterung, die diese Einschränkung aufhebt. Die Idee dabei ist, jeweils die Kriterien von zwei gematchten Spielern zu mitteln und dieses Spielerpaar mit einem anderen Spielerpaar zu matchen. Für $k = 4$ sieht das konkret so aus, dass die Team-Wait-Queue höchstens *ein* Zweierteam speichert während eine *Quattro-Wait-Queue* die provisorisch gematchten Vierergruppen enthält. Das Time-Monitoring kann dabei wie im Abschnitt 5.4.1 beschrieben auf die neue Queue ausgedehnt werden.

Es ist leicht einzusehen, dass diese Erweiterung für beliebige Spielerzahlen $k = 2^i$ angewendet werden kann. Wie gehen wir aber vor, wenn wir andere Spielerzahlen benötigen? Eine einfache Lösung besteht darin, die nächsthöhere Zweierpotenz zu matchen (z.B. $2^3$ für $k = 6$) und diejenigen überzähligen Spieler, mit der spätesten Ankunftszeit, wieder aus diesem Matching auszuscheiden und neu zu matchen.

## 5.5 Algorithmen im Überblick

Alle in diesem Kapitel beschriebenen Algorithmen werden in Tabelle 1 bezüglich Spielerzahl für das Matching ($k$), Anzahl Kriterien ($D$), erwartete

Matchinkosten und Parameterwerte verglichen.

| Name | $k$ | $D$ | Erwartete Kosten | Parameter-werte | Optimaler Parameterwert |
|---|---|---|---|---|---|
| $A_G$ | $\infty$ | $\infty$ | $\frac{k(k-1)}{k+1} + \frac{k(k-1)}{2a \cdot \tau_{max}}$ | - | - |
| $A_{PM}$ | $\infty$ | 1 | $\frac{k(k-1)}{n+1} + \frac{k(n-1)}{2a \cdot \tau_{max}}$ | $\mathbb{N}^+ \backslash \{0\}$ | $\sqrt{a} \cdot \sqrt{2 \cdot (k-1)} \cdot \sqrt{t_{max}} - 1$ |
| $A_{MQ}$ | $\infty$ | $\infty$ | $\frac{k(k-1)}{r(k+1)} + \frac{r \cdot k(k-1)}{2a \cdot \tau_{max}}$ | $\mathbb{N}^+ \backslash \{0\}$ | $\sqrt{\frac{2a \cdot \tau_{max}}{k+1}}$ |
| $A_{DW}$ | 2 | 1 | k.A. | $\mathbb{R}^+$ | k.A. |

Tabelle 1: Algorithmen im Überblick

# 6 Implementierung

## 6.1 Allgemeines

Die Implementierung der Algorithmen erfolgte in Java 1.4.1., das für den Privatgebrauch frei von der Sun-Homepage [9] heruntergeladen werden kann.
Für den Algorithmus $OPT$ gibt es eine direkte Entsprechung in der Graphentheorie: Das *Weighted-Matching-Problem*. Eine Implementierung des Algorithmus war nicht nötig, da dieser in der Mathprog-Library [8] zur Verfügung steht.

## 6.2 Übersicht

Die Implementierung kann in drei Bereiche geordnet werden, welche untereinander interagieren (siehe Abbildung 11) und sich gegenseitig überschneiden:

- In der *Simulationsumgebung* werden Spieler bezüglich Ankunftszeit und Kriterienwert möglichst realitätsnah generiert und an die Algorithmen übergeben. Es muss für das Testen garantiert werden, dass die verschiedenen Algorithmen jeweils unter exakt gleichen Bedingungen arbeiten.
- Die *Algorithmen* matchen die von der Simulationsumgebung erzeugten Spieler zu Spielen und berechnen anschliessend die Qualität des Matchings.
- Den *Algorithmus* $OPT$ führen wir als eigenen Bereich auf, da dessen Implementation schon als C-Code vorliegt und spezielle Prozeduren erforderlich sind, um die von der Simulationsumgebung erzeugten Spielerlisten an dessen Anforderungen anzupassen.

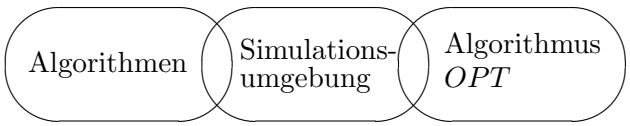


Abbildung 11: Die drei Bereiche

In den folgenden Abschnitten werden die drei Bereiche detailliert besprochen.

## 6.3 Simulationsumgebung

In der Simulationsumgebung werden Spieler möglichst realitätsgetreu mit Hilfe eines Zufallsgenerators erzeugt. Ein Spieler besteht aus einer eindeutigen Indexnummer, einer Ankunftszeit und mindestens einem Kriteriumswert. Der Kriteriumswert ist $\in [0, 1]$ und beschreibt beispielsweise den Spiellevel eines Spielers relativ zu anderen Spielern.

Die Spielerdaten werden durch eine Instanz der Klasse *CreatePlayerList* erzeugt und in ein File mit dem Suffix *.pli* (=playerlist) geschrieben. Dabei kann spezifiziert werden, wie viele Spieler in welcher Zeit eintreffen. Um eine Arrival-Rate von $a = 10$ zu erhalten, lassen wir ein File erzeugen, dass innerhalb von 60 Sekunden 600 Spieler spezifiziert. Die Speicherung der Spieler in einem File dient dazu, sicherzustellen, dass eine Simulation für jeden Algorithmus unter den exakt gleichen Bedingungen wiederholt werden kann. Dies ist eine Voraussetzung für objektive Testergebnisse.

Eine Instanz der *ReadPlayerList*-Klasse liest das *.pli*-File mit den Spielerdaten ein, und hält diese in einem Vector zur weiteren Verarbeitung bereit.

Nach dem Einlesen des Files erzeugt *Main* eine *PlayerFactory*, welche gemäss den Spielerdaten Instanzen der *Player*-Klasse erzeugt und diese an den *PlayerBuffer* übergibt. Der *PlayerBuffer* ist zugleich auch die Schnittstelle zwischen Simulationsumgebung und Algorithmen. Die *PlayerFactory* erweitert die Klasse *Thread*, um die dort zur Verfügung stehenden Methoden für das Zeitmanagement zu nutzen. Dabei befindet sich die *PlayerFactory* meist im Schlafzustand, um für die Erzeugung des nächsten Spielers zum richtigen Zeitpunkt kurz aufzuwachen. Der Aufbau der *PlayerFactory* als Thread erlaubt zudem, das Erzeugen durch die Factory und das Abarbeiten durch die Algorithmen quasi-parallel zu betreiben.

Die *Player*-Klasse speichert alle für den einzelnen Spieler während des Matchingprozesses relevante Daten, unter anderem die Ankunfts- und Wartezeiten für das Time-Monitoring.

In der Abbildung 12 sind alle Klassen mit den wichtigsten Referenzen übersichtlich dargestellt.

## 6.4 Algorithmen

Die vier Algorithmen (Greedy, Periodic-Match, Multi-Queue und Difference-Wait) wurden in separaten Klassen mit entsprechendem Namen implementiert. Die Klassen erweitern die gemeinsame *MatchingAlgorithm*-Oberklasse, welche ihrerseits die *Thread*-Klasse erweitert.
Die für Thread zu implementierende *run*-Methode ist in den Algorithmen für zwei Sachen zuständig:

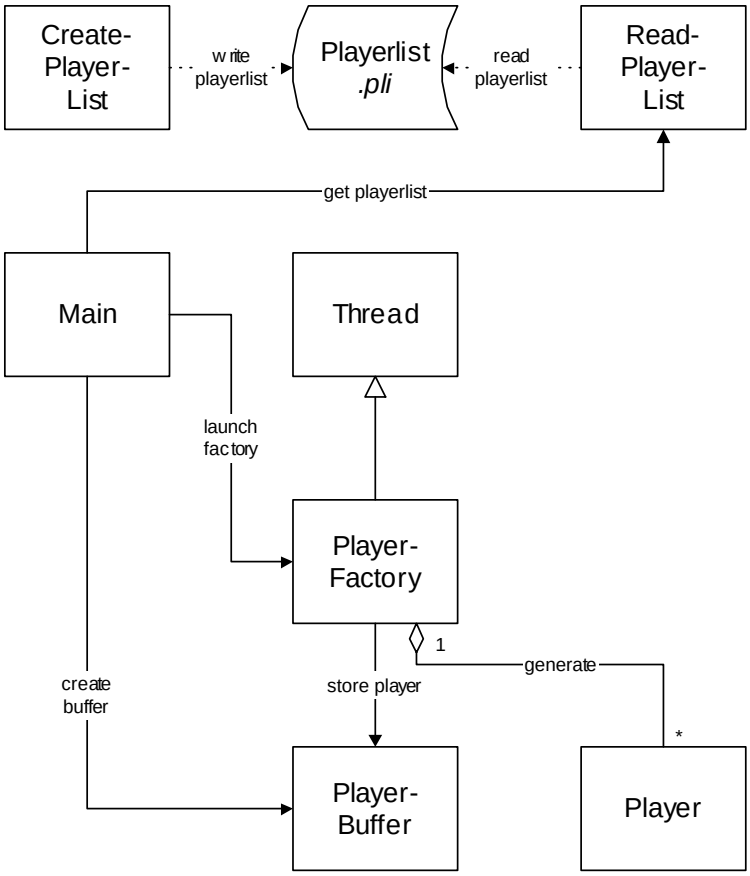


Abbildung 12: Diagramm der Simulationsumgebung

- Time Monitoring: Je nach Algorithmus werden einzelne Spieler, einzelne Queues oder provisorisch gematchte Spiele überwacht. Bei Zeitüberschreitung wird die Kontrolle an die *timeOut*-Methode übergeben.

- Polling des Player Buffers: Befinden sich Spieler im Player Buffer, werden diese ausgelesen und an die *newPlayer*-Methode weitergereicht.

Die *newPlayer*-Methode ist für die Behandlung des neuen Spielers verantwortlich und matcht neue Spiele, sofern die Bedingungen dazu erfüllt sind. Neue Spiele werden der Instanz der *GameList*-Klasse überreicht.

Für die Testumgebung ist eine Referenz der *PlayerFactory* auf die Algorithmen nötig, um diesen das Ende des Tests nach der Ankunft des letzten Spielers im *PlayerBuffer* mitteilen zu können und die Auswertung des Matchings durch die *GameList* auszulösen.

**Time Monitoring** Durch den Zufallsgenerator in *CreatePlayerList* wird zu Beginn jedem Spieler eine Ankunftszeit zugeteilt, die grösser als 0 und kleiner als die spezifizierte Endzeit des Tests (z.B. 60 Sekunden) ist. Die *PlayerFactory* berechnet die Zeitdifferenz zwischen zwei Spielern und geht für diese Zeitspanne in den Sleep-Mode. Beim Aufwachen wird unmittelbar der nächste Spieler im *PlayerBuffer* gespeichert. Jeder Spieler speichert seine Ankunfts- und seine Timeout-Zeit, wobei gilt: Timeout-Zeit $= \tau_1 + \tau_{max}$. Der Time-Monitor überwacht die Timeout-Zeiten und startet bei deren Überschreiten ein neues Spiel.

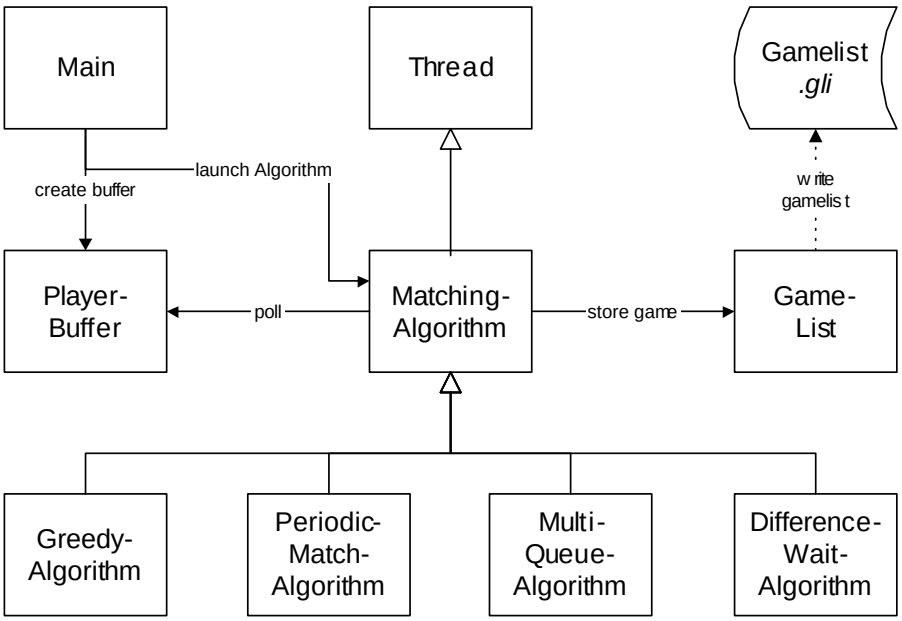


Abbildung 13: Diagramm der Algorithmen

## 6.5 OPT

Wie bereits erwähnt, kann $OPT$ für $k = 2$ auf das *Weighted-Matching-Problem* reduziert werden. Dieses Problem ist gut erforscht, da es auch für die Wirtschaftswissenschaften im Bereich *Operations-Research* interessant ist. Für die Lösung des Weighted-Matching-Problems wurden mehrere Algorithmen vorgeschlagen. Eine effiziente Implementierung eines solchen Algorithmus (Edmonds-Algorithmus [10] [11]) ist in der Mathprog-Library [8] als C-Programm frei verfügbar und konnte für diese Diplomarbeit verwendet werden. Der Algorithmus $OPT$ ist auf Spielerzahlen $k = 2$ beschränkt.

### 6.5.1 Bereitstellung des Input

Für den Algorithmus $OPT$, der unter dem Namen *Wmatch* in der Mathprog-Library aufgeführt ist, muss ein Graph als Input bereitgestellt werden (die Namen $OPT$ und *Wmatch* werden fortan synonym verwendet). Es stellt sich zunächst die Aufgabe, aus der im Abschnitt 6.3 erläuterten Spielerliste einen Graph zu konstruieren. Das dazu nötige Vorgehen zeigen wir an einem Beispiel.

**Beispiel** Nehmen wir an, die Spieler $P_1$ bis $P_5$ treffen beim Algorithmus analog der Werte in Tabelle 2 ein. $\tau_{max}$ sei 5.

In einem Graphen entsprechen die Knoten den Spielern. Zwischen zwei Knoten existiert genau dann eine Kante, wenn die beiden Spieler gematcht werden können. Dies ist genau dann möglich, wenn die Ankunftszeiten der beiden Spieler höchstens $\tau_{max}$ auseinanderliegen. In Tabelle 2 ist aufgelistet, welcher Knoten zu welchem adjazent ist.

| Spielerparameter | | | Kantenverbindungen | | | | |
|---|---|---|---|---|---|---|---|
| $P_i$ | $\tau_1(P_i)$ | $\gamma(P_i)$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
| 1 | 0 | 0 | | • | • | | |
| 2 | 1 | 0 | • | | • | • | |
| 3 | 4 | 1 | • | • | | • | • |
| 4 | 6 | $\frac{1}{2}$ | | • | • | | • |
| 5 | 8 | $\frac{1}{2}$ | | | • | • | |

Tabelle 2: Kantenverbindungen zwischen den einzelnen Spielern

Im nächsten Schritt werden die Kantengewichte berechnet. Das Kantengewicht entspricht den Kosten, die anfallen, wenn die beiden durch die Knoten repräsentierten Spieler gematcht werden. Im vorliegenden, einfachen Fall von $k = 2$ (zwei Spieler pro Spiel) und $D = 1$ (ein Kriterium), errechnen sich die Kosten aus der durch $\tau_{max}$ dividierten Zeitdifferenz und der doppelten Kriteriendifferenz. Für das Matching von $P_2$ und $P_3$ fallen gemäss Formel (2) beispielsweise Kosten $2 \cdot |0 - 1| + \frac{|1-4|}{5} = 2.6$ an. Der Graph ist in Abbildung 14 dargestellt.

Wie wird nun der Umstand einer Höchstwartezeit im Graph berücksichtigt? Wir erinnern uns, dass ein Spieler nach höchstens $\tau_{max}$ Wartezeit gematcht werden muss. Ist bis dann kein geeigneter Spielpartner gefunden, wird der Spieler mit einem Computerspieler $P_{COMP}$ gematcht. Dieser Umstand lässt sich im Graph modellieren, indem an jeden Knoten ein Dummy-

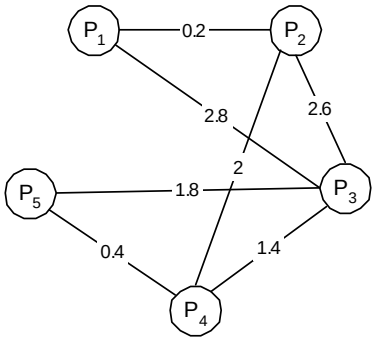


Abbildung 14: Spielergraph mit Kantengewichten

knoten angehängt wird, der für einen Computerspieler steht. Die Nummerierung der Dummyknoten beginnt bei (Anzahl Spieler +1). Das Kantengewicht entspricht den Strafkosten für das Matching eines Computerspielers: $2 \cdot 1 + \frac{5+5}{5} = 4$ (siehe Abschnitt 4.1, bzw. Formel (2)). In Abbildung 15 ist der modifizierte Graph dargestellt.

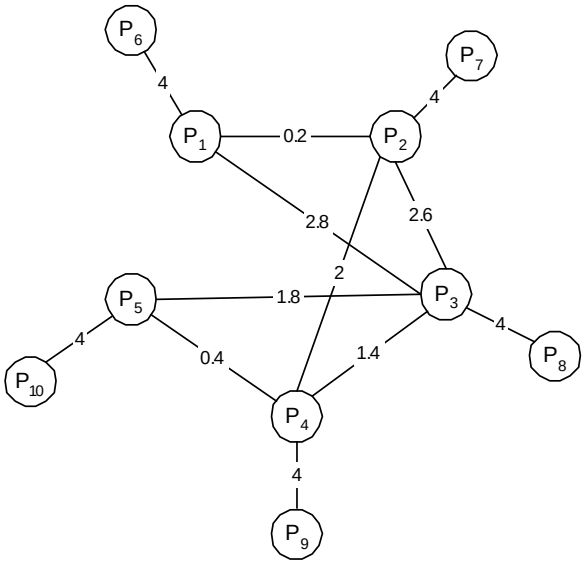


Abbildung 15: Spielergraph mit Dummyknoten für Computerspieler

Vor dem Speichern in einem File muss der Graph serialisiert werden. Dazu werden Knoten- und Kantenrecords erzeugt. Auf einen Knotenrecord folgen die Kantenrecords der zu diesem Knoten inzidenten Kanten. Der Knotenre-

cord beschreibt implizit[4] den Knotenindex und explizit die Anzahl inzidenter Kanten. Die Kantenrecords enthalten den Index des anderen, nicht vorausgehenden Knotens und das Kantengewicht. Jede Kante ist somit zweimal im File.

Alle in diesem Beispiel besprochenen Arbeitsschritte werden von der Klasse *InFileAdaptor* geleistet. Als Input wird die Spielerliste durch *ReadPlayerList* eingelesen. Siehe Abbildung 16.

### 6.5.2 Verarbeitung des Output

Der Output des Wmatch-Algorithmus ist eine Liste mit den gepaarten Spielern. Steht als Spielpartner eine 0, wurde der Knoten nicht gematcht. Dies trifft in der Regel für fast alle Dummyknoten zu. Tabelle 3 beschreibt den Output des Wmatch-Algorithmus für den Graphen des letzen Abschnitts.

| Spielerpaare |
|---|
| 1 2 |
| 2 1 |
| 3 8 |
| 4 5 |
| 5 4 |
| 6 0 |
| 7 0 |
| 8 3 |
| 9 0 |
| 10 0 |

Tabelle 3: Output des Wmatch-Algorithmus

Um wieder alle Spielerdaten wie Kriterienwerte und Ankunftszeiten für die Matchinganalyse verfügbar zu haben, muss der Output des Wmatch-Algorithmus mit der Spielerliste *.pli* zusammengeführt werden. Wie in Abbildung 16 dargestellt, wird dieser Arbeitsschritt durch *OutFileProcessor* ausgeführt.

[4]Durch seine Position im File relativ zu den anderen Knotenrecords.

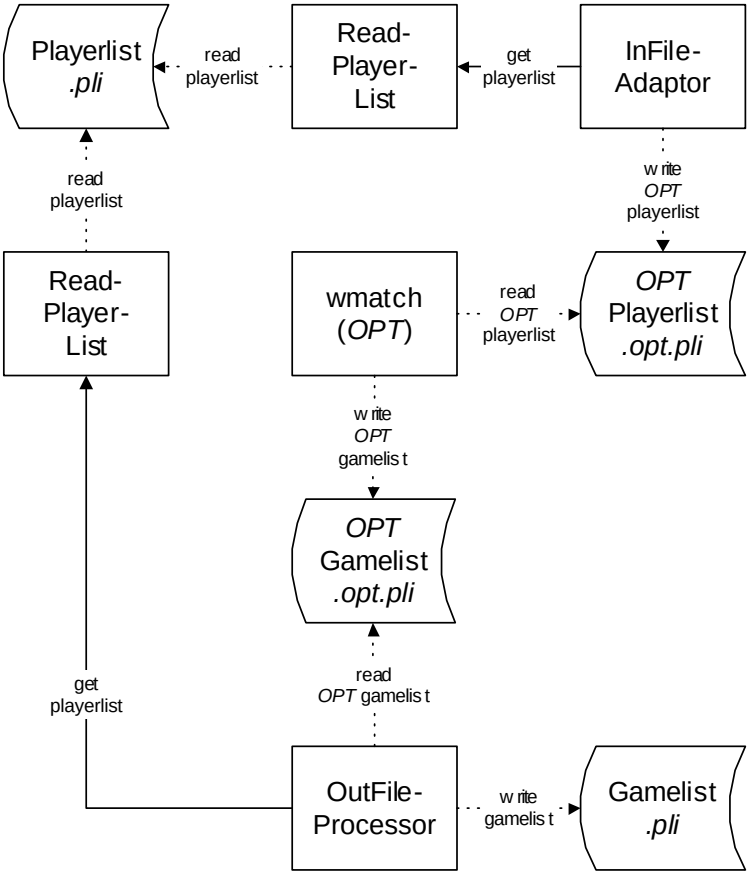


Abbildung 16: Diagramm der Umgebung um *OPT*

# 7 Simulation

## 7.1 Einleitung

Die Simulation soll die Eignung und die Grenzen der Algorithmen für unterschiedliche Situationen aufzeigen. Wir werden am Ende des Kapitels in der Lage sein, Empfehlungen über die Benutzung einzelner Algorithmen abzugeben. Die Simulationen sollen zudem die Richtigkeit der Formeln für die Kostenvorhersage und Parameterwahl bestätigen.

## 7.2 Auswahl der Simulationsdaten

Die Simulationsdaten sollen eine möglichst realitätsnahe Umgebung vortäuschen. Die Frage ist, wann melden sich wie viele Spieler für ein Spiel an? In diesem Zusammenhang ist die Arrival-Rate $a$ der bedeutende Parameter. $a$ kann Schwankungen unterliegen, die sich über einen kurzen oder langen Zeitraum erstrecken. Einfach und realitätsnah kann die Ankunft der Spieler als *Poisson-Verteilt* angenommen werden. Dies bedeutet eine konstante Arrival-Rate $a$ für die Dauer einer Simulation. Tatsächlich gibt es auch Zeiträume mit steigender oder fallender Arrival-Rate. Es wurden jeweils drei Simulationen mit konstantem $a$ und drei mit linear steigender Arrival-Rate durchgeführt. Siehe dazu Tabelle 4.

| | Arrival-Rate | | |
|---|---|---|---|
| konstant | 1 | 3 | 10 |
| steigend | 0...2 | 0...6 | 0...20 |

Tabelle 4: Arrival-Raten für die Simulationen

Ein einzelne Simulation dauert eine Minute. Bei einer Arrival-Rate von $a = 10$ enthält das mit *CreatePlayerList* erstellte Datenfile 600 Spieler. Der Timeoutparameter $\tau_{max}$ sei für alle Simulationen 5, da dies einer angemessenen Höchstwartezeit für ein Matching entspricht. Für sehr tiefe Arrival-Raten, z.B. unter 1, wäre $\tau_{max}$ nach oben anzupassen, um nicht zu viele Computerspieler zu matchen.

Nebst Average-Case-Daten wurden für jeden Algorithmus auch Worst-Case-Daten erzeugt.

Um alle Algorithmen untereinander vergleichen zu können, ist eine Beschränkung auf $k = 2$ Spieler notwendig da sowohl der Difference-Wait-Algorithmus als auch der fiktive offline Algorithmus $OPT$ auf diese Spie-

lerzahlen beschränkt sind. Ähnlich sieht es mit der Anzahl Kriterien aus, die für alle Simulationen 1 ist.

## 7.3 Simulationsverfahren

Für die in Tabelle 4 aufgeführten Arrival-Raten wurden je 3 Files erstellt. Jeder Algorithmus wurde mit allen sinnvollen Parameterwerten über diesen total 18 Files getestet. Die Resultate der 3 Files mit gleicher Arrival-Rate wurden gemittelt.

Anschliessend wurden die Algorithmen für ihre Worst-Case-Szenarien getestet.

## 7.4 Simulationen und deren Resultate

Zuerst besprechen wir Ausführlich die Simulationen mit Arrival-Rate $a = 10$ für alle sinnvollen Parameterwerte und für jeden Algorithmus.

**Periodic-Match-Algorithmus, $a$=10** Der Periodic-Match-Algorithmus geht für den Parameterwert 1 in den Greedy-Algorithmus über. Beim Greedy-Algorithmus fallen bedingt durch die grosse Arrival-Rate kaum Zeitkosten, jedoch hohe Kriterienkosten an. Wird durch vergrössern des Parameters auf mehr Spieler gewartet bevor ein Matching ausgelöst wird, verschieben sich die Kosten zu Lasten der Zeit (siehe Abbildung 17). Je mehr Spieler pro Matching zur Verfügung stehen, desto näher liegen ihre Kriterien beieinander. Ab einer bestimmten Parametergrösse verändert sich das Verhältnis zwischen Zeit- und Kriterienkosten nicht mehr, denn $\tau_{max}$ wird für $p > \frac{a \cdot \tau_{max}}{k}$ entscheidender Faktor für Auslösen des Matchings. Für die in dieser Simulation gegebenen Grössen gilt dies ab $p > 25$.

Aus den Simulationen geht hervor, dass die Gesamtkosten pro Matching für die Parameterwerte 4 und 5 am tiefsten sind, wie im Abschnitt 5.2.2 postuliert wurde (siehe Beispiel für Parameterwahl). Durch einsetzen von $k = 2, a = 10, n = k \cdot 4 = 8$ und $\tau_{max} = 5$ in (6) erhalten wir 0.362 als Durchschnittsspielkosten für die Parameterwahl 4, was die Testresultate bestätigen.

**Multi-Queue-Algorithmus, $a$=10** Wird der Multi-Queue-Algorithmus auf eine Queue beschränkt, liegt ein Greedy-Algorithmus vor. Wie schon beim $A_{PM}$ kann auch beim $A_{MQ}$ für wachsenden Parameterwert eine Verschiebung von Kriterienkosten nach Zeitkosten beobachtet werden (Abbildung 18). Der optimale Parameterwert kann gemäss Formel (9) berechnet werden (siehe Beispiel in Abschnitt 5.3.2) und beträgt für die gegebenen Rahmenbedingungen 5.77. Wir beobachten, dass die Kosten tatsächlich für Parameterwerte

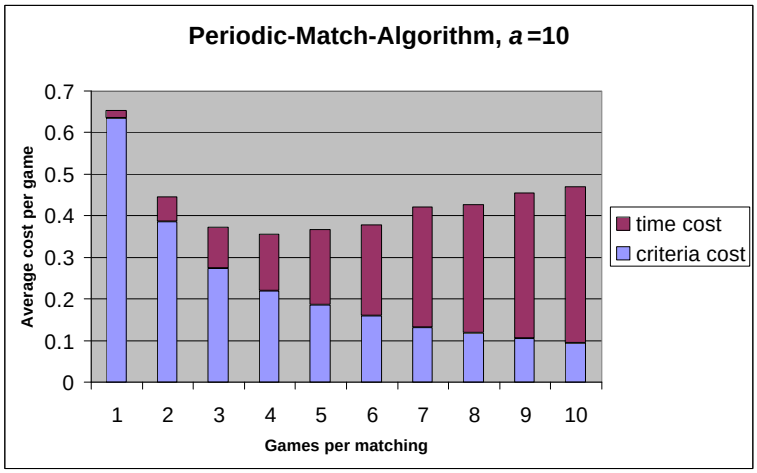


Abbildung 17: Resultate für den Periodic-Match-Algorithmus, $a = 10$

zwischen 4 und 8 am tiefsten sind. Die erwarteten Spielkosten für Parameterwert 6 betragen gemäss (8) 0.231, was mit den Testresultaten übereinstimmt.

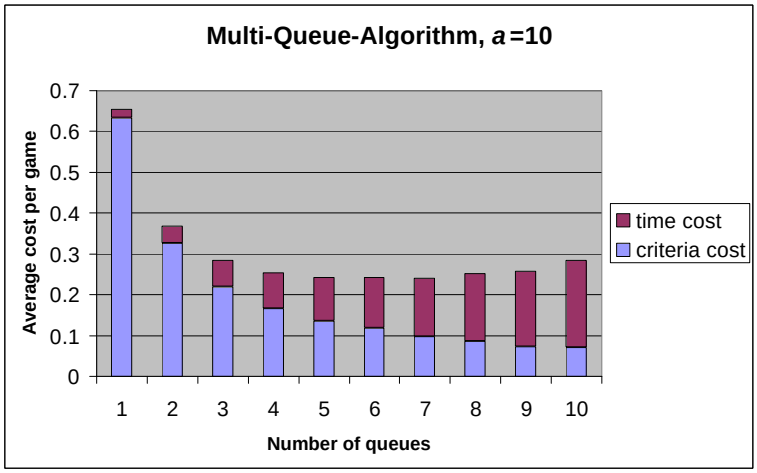


Abbildung 18: Resultate für den Multi-Queue-Algorithmus, $a = 10$

**Difference-Wait-Algorithmus, $a$=10** Der Difference-Wait-Algorithmus geht für Parameterwert 0 in den Greedy-Algorithmus über. Auch beim $A_{DW}$ kann mit dem Parameter das Kostenverhältnis zwischen Kriterien- und Zeit-

kosten gesteuert werden, wobei die Gesamtkosten stagnieren wie aus Abbildung 19 hervorgeht.

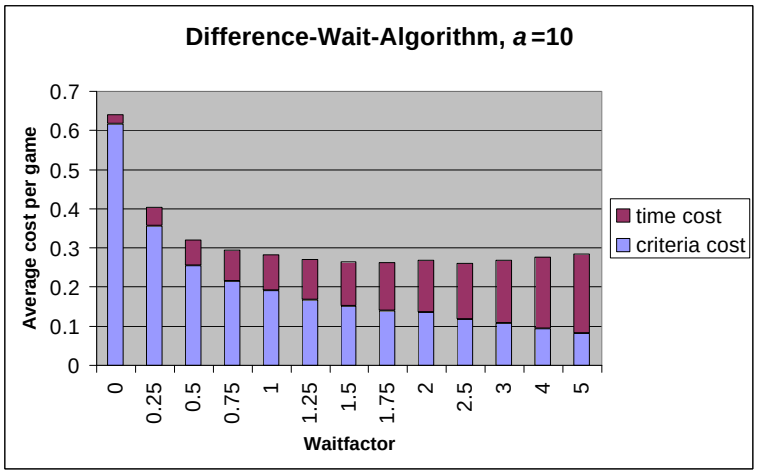


Abbildung 19: Resultate für den Difference-Wait-Algorithmus, $a = 10$

**Algorithmenvergleich für $a$=10** Im direkten Vergleich schneidet $A_G$ am schwächsten ab, da er die Kriterienwerte nicht berücksichtigt und beim Matching hohe Kriterienkosten verursacht. Die Kosten von $A_{PM}$ sind gegenüber $A_G$ beinahe halbiert, liegen aber etwa einen Drittel über den Kosten von $A_{MQ}$ und $A_{DW}$. Problematisch am $A_{PM}$ ist, dass viele Spieler gleichzeitig warten müssen. Dieses Problem spitzt sich für kleinere Arrival-Raten weiter zu. Als Vergleich ist das optimale Matching durch einen fiktiven offline Algorithmus $OPT$ aufgeführt, der alle Daten schon im Voraus kennt. Die optimalen Matchingkosten liegen etwa einen Viertel unter den von $A_{MQ}$ und $A_{DW}$ verursachten Kosten. Vergleiche hierzu auch Abbildung 20.

**Algorithmenvergleich für $a$=3 und $a$=1** In Abbildung 21 und 22 sind alle wichtigen Resultate für $a = 3$ und $a = 1$ zusammengefasst. Mit der kleiner werdenden Arrival-Rate schnellen beim $A_{PM}$ die Zeitkosten in die Höhe. Das beste Resultat wird erzielt, wenn der Parameterwert auf 1 gesetzt wird und der $A_{PM}$ somit in den Greedy-Algorithmus übergeht. $A_{MQ}$ und $A_{DW}$ stehen besser da. Sie liefern sich für $a = 3$ ein Kopf-an-Kopf-Rennen, das $A_{DW}$ bei $a = 1$ knapp für sich entscheidet.

Für sehr kleine Arrival-Raten scheint es am besten zu sein, die Algorithmen durch Parameterwahl 1 ($A_{PM}$ und $A_{MQ}$) oder 0 ($A_{DW}$) in den Greedy-Algorithmus übergehen zu lassen.

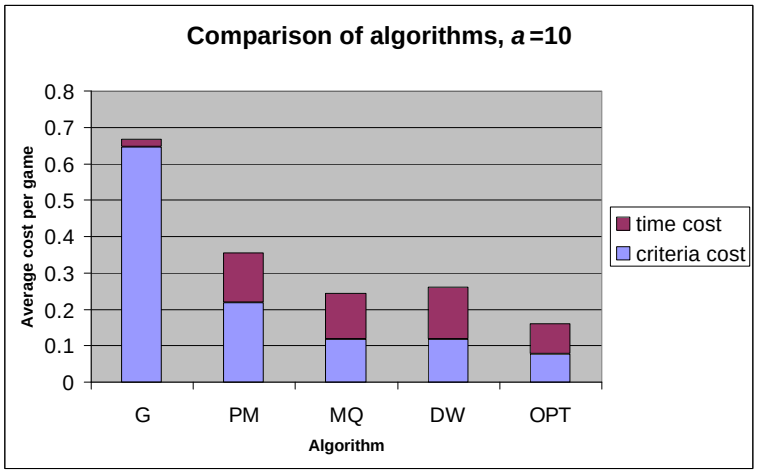


Abbildung 20: Vergleich aller Algorithmen für $a = 10$

**Algorithmenvergleich für steigende Arrival-Raten** Die Resultate für nicht konstante Arrival-Raten stimmen bis auf wenige Prozente mit den Resultaten der konstanten Arrival-Raten überein. Der Grund dafür liegt darin, dass sich die Kostenunterschiede der tiefen Arrival-Rate zu Beginn und der hohen Arrival-Rate am Ende der Simulation gegenseitig aufheben. Errechnet man die Durchschnittskosten pro Spiel, unterscheiden sich diese kaum von den durchschnittlichen Spielkosten einer Simulation mit konstanter Arrival-Rate. Betrachtet man die einzelnen Spielkosten im *.gli*-File (Gamelist), ist eine stetige Abnahme der Matchingkosten pro Spiel erkennbar.

**Worst-Case-Szenarien** Die hier vorgestellten Worst-Cases sollten besser als Bad-Cases bezeichnet werden, denn als Worst-Case bezeichnen wir ein Beispiel, das vom fiktiven, optimalen offline Algorithmus ohne Kosten gelöst werden kann, während der Online-Algorithmus Kosten verursacht. Beispiele dazu wurden im Kapitel 5 bei der Analyse des jeweiligen Algorithmus besprochen. Für die Simulationen konstruieren wir ein Beispiel für jeden Algorithmus, das für diesen aufgrund seiner Funktionsweise möglichst ungünstig ausfällt. Anschliessend Testen wir dieses konstruierte Beispiel für alle Algorithmen. Die Beispiele umfassen jeweils 60 Spieler, $\tau_{max}$ ist 5. Die im folgenden erläuterten Datenfiles sind in Tabelle 5 beschrieben. Das Datenfile für den entsprechenden Algorithmus wird mit Kleinbuchstaben bezeichnet:

- **g**: Alle Spieler treffen gleichzeitig ein. Die Spieler haben abwechselnd Kriterienwert 0 und Kriterienwert 1.

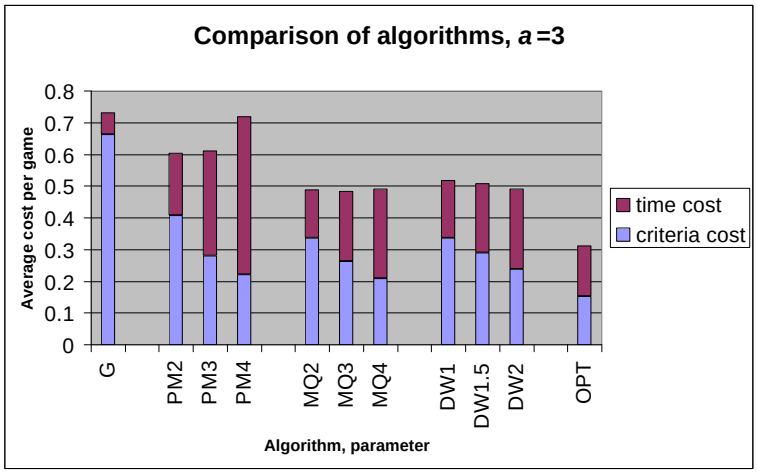


Abbildung 21: Vergleich aller Algorithmen für $a = 3$

- **pm**: $A_{PM}$ wartet vor jedem Matching auf 4 Spieler. Alle Spieler haben den gleichen Kriterienwert. Jeweils 3 Spieler treffen gleichzeitig ein, der vierte $\tau_{max} - \epsilon$ später usw. Siehe auch Beispiel in Abschnitt 5.4.2.

- **mq**: $A_{MQ}$ führt zwei Queues. Es treffen jeweils zwei Spieler gleichzeitig ein, deren Kriterienwerte nahe beieinander jedoch gerade ober- und unterhalb des Werts liegen, der die Zuteilungsgrenze zu den beiden Queues ist, also $\frac{1}{2} - \epsilon$ und $\frac{1}{2} + \epsilon$. Nach Ablauf von $\tau_{max}$ treffen die nächsten beiden Spieler ein usw.

- **dw**: Der Wartefaktor $f_{wait}$ sei auf 1 gesetzt. Jeweils zwei Spieler treffen gleichzeitig ein, der eine mit Kriterienwert 0, der andere mit $1 - \epsilon$. Die nächsten zwei Spieler treffen kurz nach Ablauf der Wartezeit für dieses provisorische Matching ein, also eine Sekunde später usw.

Die Simulationsresultate können in Abbildung 23 eingesehen werden.

Da bei **g** alle Spieler gleichzeitig eintreffen und die Kriterienwerte 0 und 1 gleich häufig auftreten, kann von allen Algorithmen mit Ausnahme des $A_G$ kostenlos gematcht werden.

Für das File **pm** verursacht $A_{PM}$ drei mal höhere Kosten wie die anderen Algorithmen, da drei mal mehr Spieler auf das Matching warten müssen. Diesen Bad-Case könnte man beliebig verschlechtern, indem man $A_{PM}$ auf mehr als 4 Spieler warten lässt, bevor das Matching veranlasst wird. Die Kosten für $OPT$ liegen tiefer, da dieser den ersten und den letzten Spieler mit einem Computerspieler $P_{COMP}$ paart und somit alle anderen Spieler optimal

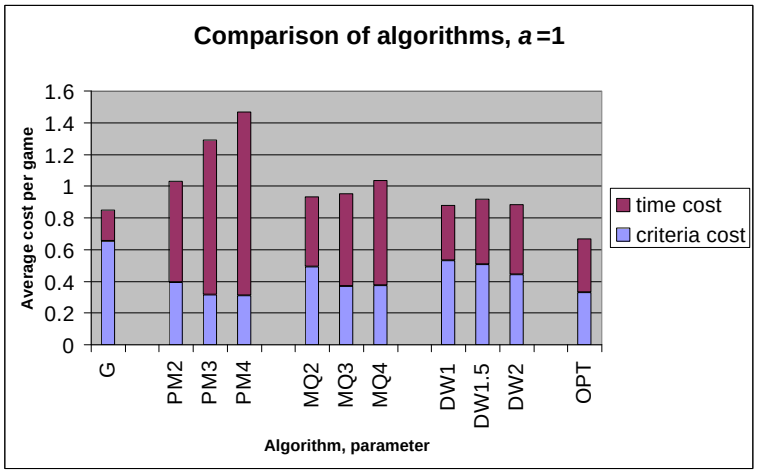


Abbildung 22: Vergleich aller Algorithmen für $a = 1$

| | g | | pm | | mq | | dw | |
|---|---|---|---|---|---|---|---|---|
| $P_i$ | $\tau_1(P_i)$ | $\gamma(P_i)$ | $\tau_1(P_i)$ | $\gamma(P_i)$ | $\tau_1(P_i)$ | $\gamma(P_i)$ | $\tau_1(P_i)$ | $\gamma(P_i)$ |
| 1 | 0 | 0 | 0 | 0 | 0 | $\frac{1}{2} - \epsilon$ | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 | 0 | $\frac{1}{2} + \epsilon$ | 0 | $1 - \epsilon$ |
| 3 | 0 | 0 | 0 | 0 | $\tau_{max} + \epsilon$ | $\frac{1}{2} - \epsilon$ | 1 | 0 |
| 4 | 0 | 1 | $\tau_{max} - \epsilon$ | 0 | $\tau_{max} + \epsilon$ | $\frac{1}{2} + \epsilon$ | 1 | $1 - \epsilon$ |
| 5 | 0 | 0 | $\tau_{max}$ | 0 | $2 \cdot (\tau_{max} + \epsilon)$ | $\frac{1}{2} - \epsilon$ | 2 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... |

Tabelle 5: Simulationsdaten für Bad-Case-Szenarien

matchen kann. Die hohen Kosten für den Computerspieler amortisieren sich durch die Spielerzahl. Siehe hierzu auch Abschnitt 5.4.2.

Der **mq**-Bad-Case wird vom $A_{MQ}$ und vom $A_{PM}$ gleich gehandhabt: Erst nach Ablauf von $\tau_{max}$ werden jeweils zwei Spieler gematcht. $A_{MQ}$ wartet, weil die Spieler in unterschiedlichen Queues liegen, $A_{PM}$, weil vor der Auslösung des Matchingprozesses 4 Spieler eingetroffen sein müssen und erst zwei da sind. Die anderen Algorithmen verursachen keine Kosten, da keine Wartezeiten anfallen und die Kriterienwerte sehr nahe zusammenliegen.

Bei der Simulation mit **dw** schneiden $A_{DW}$ und $A_G$ schlecht ab. Im Gegensatz zu $A_{DW}$ verursacht $A_G$ keine Zeitkosten, da die beiden Spieler sofort und nicht erst nach Ablauf eines Wartefaktors gematcht werden. Die übrigen Algorithmen warten jeweils eine Sekunde auf das zweite Spielerpaar und können so ohne Kriterienkosten matchen.

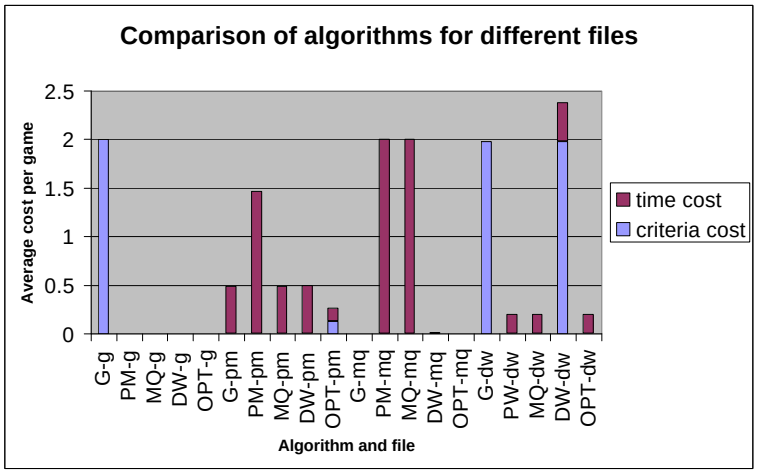


Abbildung 23: Worst-Case-Szenarien für die Algorithmen $A_G$, $A_{PM}$, $A_{MQ}$ und $A_{DW}$

## 7.5 Fazit der Simulationen

Mittelt man die Average-Case Resultate aller durchgeführten Simulationen und normiert die Kosten für $OPT$ auf 1, erhalten wir aussagekräftige Werte über die Kosten pro Algorithmus. Siehe Tabelle 6.

| Algorithmus | normierte Kosten |
|---|---|
| $A_G$ | 2.04 |
| $A_{PM}$ | 1.64 |
| $A_{MQ}$ | 1.41 |
| $A_{DW}$ | 1.39 |
| $OPT$ | 1 |

Tabelle 6: Gemittelte und normierte Kosten über alle Simulationen

Bei der Auswertung der Resultate darf nicht ausser Acht gelassen werden, dass diese von der in Kapitel 4 vorgestellten Formel (2) zur Berechnung der Matchingqualität abhängen. Eine andere Zeitgewichtung würde die Ergebnisse verändern.

Es erstaunt nicht, dass die durchschnittlichen Matching-Kosten des $A_G$ mehr als doppelt so hoch wie die von $OPT$ sind, sorgt doch der $A_G$ in der Regel für tiefe Zeitkosten während $OPT$ Kriterien- und Zeitkosten optimiert.

Die Matchingqualität des Periodic-Match-Algorithmus liegt im Schnitt zwischen der des einfachen Greedy-Algorithmus und der von $A_{MQ}$ und $A_{DW}$.

Der Multi-Queue- und Difference-Wait-Algorithmus liefern eine gute Matchingqualität. Die Matching-Kosten liegen rund 40% über dem Optimum. Die beiden Algorithmen eigenen sich sowohl für hohe als auch für tiefe Arrival-Raten. Wir empfehlen, den $A_{MQ}$ oder den $A_{DW}$ für 2-Spieler-Spiele und den $A_{MQ}$ für Spiele mit mehr als 2 Spielern einzusetzen.

# 8 Erweiterungen und Fazit

## 8.1 Erweiterungen der Algorithmen

Die gefundenen Algorithmen können in mancher Hinsicht ergänzt oder verbessert werden:

- Da für die beiden Algorithmen $A_{PM}$ und $A_{MQ}$ Formeln für optimale Parameterwerte zur Verfügung stehen, liesse sich das Adaptieren des Paramters an die aktuelle Arrival-Rate automatisieren. So entstehen adaptive Algorithmen.
- Für den Mehrkriterienbetrieb des $A_{MQ}$, der im Rahmen dieser Diplomarbeit nicht implementiert wurde, könnten Verfahren entwickelt werden, die Festlegen, in welcher Reihenfolge die Queues verschiedener Kriterien nach Spielern abgesucht werden (falls sich nach Ablauf von $\tau_{max}$ weniger als $k$ Spieler in einer Queue befinden). So liessen sich Prioritäten über die Einhaltung der unterschiedlichen Kriterien umsetzen.
- Der $A_{DW}$ könnte, wie in Abschnitt 5.4.3 vorgeschlagen, für das Matching von mehr als zwei Spielern erweitert werden.
- Alle in dieser Arbeit vorgestellten Algorithmen sind für ein zentrales Matching ausgelegt: Die Spieler werden an einem Ort gesammelt und gematcht. Es wäre interessant, diesen Prozess zu dezentralisieren und auf unterschiedliche Peers aufzuteilen. Dies bietet Vorteile bezüglich Verfügbarkeit und Skalierbarkeit.

## 8.2 Fazit

Abschliessend kann ein positives Fazit gezogen werden. Die Diplomarbeit war ein dynamischer Vorgang, mit dessen Fortschreiten neue Ziele gesteckt und andere unwichtig wurden. In den ersten Wochen galt es, theoretische Aufgaben wie das Entwickeln des Modells und der Algorithmen auszuführen, anschliessend folgte ein praktischer Teil, der das Implementieren und Testen der Algorithmen unter realitätsnahen Bedingungen enthielt.

Als Resultat stehen ein theoretisches Modell des $k$-Player-Matching-Problems und Algorithmen zur Verfügung, deren Matchingqualität im Average-Case gut ist, wie der Vergleich mit einem optimalen offline Algorithmus zeigt. Als Wermutstropfen bleibt, dass keiner der gefundenen Algorithmen im Sinne der kompetitiven Analyse kompetitiv ist. Diese Einschränkung ist jedoch für den praktischen Einsatz der Algorithmen von untergeordneter Bedeutung.

Schade ist auch, dass am Schluss dieser vier Monate dauernden Diplomarbeit keine Zeit mehr blieb, eine Dezentralisierung der Algorithmen ins Auge zu fassen.

Nichtsdestotrotz stellt diese Arbeit Resultate zur Verfügung, die hoffentlich weiterverwendet werden können.

# Literatur


[1] Gaming Networks im Google Web Directory:
http://directory.google.com/Top/Games/Video_Games/
Computer_Platforms/Multiplayer/Gaming_Networks/

[2] Blue Byte Game Channel, Die Siedler IV,
http://www.bluebyte.net/siedler4/

[3] XBox Live, http://www.xbox.com

[4] Live-Bericht, Microsofts Xbox-Online-Dienst im Betatest, c't 2002, Heft 24, S. 20-22

[5] Optobyte Software, Stöck Wyys Stich Platinum,
http://www.optobyte.ch/swsplatinum/

[6] S. Irani, A. R. Karlin: On Online Computation, 1997

[7] S. Albers, S. Leonardi: Online Algorithms, 1999

[8] MATHPROG: A Collection of Codes for Solving Various Mathematical Programming Problems, http://elib.zib.de/pub/Packages/mathprog/

[9] Sun Microsystems, Java[tm] 2 Platform Standard Edition v 1.4.1.,
http://java.sun.com/j2se/1.4.1/

[10] H. N. Gabow: Implementation of Algorithms for Maximum Matching on Nonbipartite Graphs, Ph.D. thesis, Stanford University, 1973

[11] H. N. Gabow: An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs, Journal of the Association for Computing Machinery, Vol. 23, No 2, April 1976, pp 221-234