TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computing Science

MASTER'S THESIS

How best to beat high scores in Yahtzee

A caching structure for evaluating large recurrent functions

by
C.J.F. Cremers

Supervisor: dr. ir. T. Verhoeff

Eindhoven, June 2002

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Yahtzee

In September 1999, Tom Verhoeff determined an optimal strategy for the game Yahtzee. This optimal strategy tries to achieve the highest possible expected total score. Determining such a strategy for Yahtzee comes down to solving a large recurrent relation. Using the resulting strategy, the expected score of the player is 254.5896, with a standard deviation of 59.6117.

This thesis builds on the work done by Verhoeff. We will determine a collection of optimal strategies for Yahtzee. These strategies have a different objective, namely maximizing the probability of achieving a given score or more. These strategies are harder to compute.

Yahtzee is a game for one or more players, but we only consider the solitaire variant[1] in this thesis. If you are not familiar with the game Yahtzee, we recommend that you read the official rules in appendix A first.

1.2 Research goals

In previous research, Verhoeff determined a Yahtzee strategy that achieves an optimal expected score.

In this research we aim to determine an alternative optimal strategy, that has an optimal probability of achieving a given score or more. This problem is significantly harder, as the size of the state space increases linearly with the score that we are trying to achieve. This gives rise to much larger computation times. Using a modified version of the the algorithm structure used by Verhoeff, would require more memory than available.

To solve these problems we develop a caching structure for tabulating a class of recurrent functions, and implement this technique for the Yahtzee problem.

1.3 Overview of thesis

Chapter two summarizes some of the research done by Verhoeff. Yahtzee is modeled as a Markov Decision Process.

[1] When more than one player is involved, the game basically remains the same. Each player plays his own solitaire game for a turn, then the next player plays his own game for a turn, and so on, until all players have ended their game. We only consider the solitaire variant here. For more than one player our results can be used as well, and we will return to this point in section 6.3.

In chapter three we analyze the main problems of determining the required strategies.

We deal with the development of the software in chapter four, in terms specific for Yahtzee.

Chapter five is the most important chapter of this thesis, and can be read independent of the other chapters. In it, we focus on the main problem, which the size of the table. We generalize the Yahtzee problem structure to a class of recurrent functions. We then develop a caching structure that performs well for evaluation of, or computing tables for, this class of functions.

Chapter six discusses the results of the computed strategies.

Chapter seven concludes the thesis and suggests areas of further research.

The two appendices deal with the game Yahtzee and the hardware that was used to compute the strategies.

Readers with limited time are advised to read chapters two and five.

Chapter 2

Previous research

This report builds on research conducted by Verhoeff ([2, 3, 4]). Before this project started, Verhoeff had already implemented a program that computed an optimal strategy for Yahtzee, and analyzed the results. In this chapter we will summarize his research and results.

2.1 Markov Decision Processes

To reason about Yahtzee, a formal model is required. Please consult Appendix A for the official rules of Yahtzee. When we are playing Yahtzee, we keep a score card, and remember if we are choosing or (re)rolling dice, which dice we kept and which values they have, and how many times we are still allowed to reroll. These elements determine the game state.

When the game Yahtzee is played, two phases alternate. In a *roll phase*, dice are thrown. The player cannot influence the outcome a roll. The player has freedom in a *choice state*, where he or she can decide to score the current throw in an open category, or reroll some of the dice. In the case of rerolling, the choice determines which dice are rerolled, and which are kept.

Yahtzee can be modeled as a Markov Decision Process ([1, 5]). The key ingredients in a Markov Decision Process are

- States
- Events leading from one state to the next state, capturing the transition function
- Probability distributions over the events in every state
- Score functions for the events in every state

We define the state space of the game as $S = R \biguplus C$, consisting of the roll states R and choice states C. The initial state of the game is referred to as I. In the initial state, which is a roll state, no categories have been scored.

For all states $s \in S$ we have an event set $E.s$. In each state, an event from $E.s$ occurs. The game ends when all categories have been scored, and at that point we have $E.s = \emptyset$.

An event $e \in E.s$ leads to the next state denoted by se, according to the game's transition function. For all states $s \in S$, we have an event probability distribution $p.s$. The player determines $p.s$ for all $s \in C$, and the dice determine $p.s$ for all $s \in R$. Event $e \in E.s$ occurs

with probability $p.s.e$, and we have

$$\sum_{e \in E.s} p.s.e = 1$$

for non-final states ($E.s \neq \emptyset$).

We have event scores $f.s$ for all $s \in S$. In particular, event $e \in E.s$ scores $f.s.e$. Note that $f.s.e$ gives the score *increment* on account of that event, and not the total score after the event. The values for $f.s.e$, the event scores, are given by the rules of Yahtzee. For example, scoring a roll $1, 1, 1, 4, 5$ in the category Aces scores three points. Rolling the dice does not yield any points, and in general we have

$$\begin{aligned} f.s.e &= 0 \qquad \text{for all } s \in R \\ f.s.e &\geq 0 \qquad \text{for all } s \in C \end{aligned}$$

In each turn, a category must be scored, resulting in exactly 13 turns. This implies that the transition function for Yahtzee is acyclic. Within a turn, there are at least one roll and one choice state, and at most three of each. R and C states alternate.

2.1.1 Lifting events to games

The state transition function can be lifted from single events to sequences of events occurring one after another. Such a sequence of events is called a game. The resulting state after the game g, starting in the state s, is denoted by sg.

$$\begin{aligned} s\langle\,\rangle &= s \\ s(eg) &= (se)g \end{aligned}$$

Where $\langle\,\rangle$ is the game with no events (empty sequence), and eg is the game that starts with event e and continues as g.

Because the transition function is acyclic, there can only exist finite sequences of events after a state s. We write $G.s$ to denote the set of complete games after s:

$$G.s = \{\, g \mid E.sg = \emptyset \,\}$$

All games start in the initial state I. Thus, the set of all possible games can be written as $G.I$.

We define the score $F.s.g$ of a game g after state s inductively by

$$\begin{aligned} F.s.\langle\,\rangle &= 0 \\ F.s.eg &= f.s.e + F.se.g \end{aligned}$$

that is, by adding all the scores for the individual events in the game.

In much the same way, we lift the probability distributions $p.s$ for the events. Event e occurs in state s with probability $p.s.e$. We define the probability $P.s.g$ that game g occurs after state s inductively by

$$\begin{aligned} P.s.\langle\,\rangle &= 1 \\ P.s.eg &= p.s.e \cdot P.se.g \end{aligned}$$

that is, by multiplying the probabilities of the individual events in the game. This works because the probability distributions are independent (this independence makes the game a Markov process).

The main observation is that the score increments and event probabilities do not depend on any previous events in the game.

2.1.2 Decision strategies

We now turn to defining strategies. A decision strategy D defines $p.s$ for all choice states, $s \in C$. If these choices depend only on the state, and not on chance or other circumstances, the strategy is deterministic. More formally, D is deterministic if for all s and $e \in E.s$ we have $p.s.e \in \{ 0, 1 \}$.

By $\mathcal{E}_D$ we denote the expected score for a strategy D. This is obtained as the weighted average of the scores for all games:

$$\mathcal{E}_D \quad = \quad \sum_{g \in G.I} P.I.g \cdot F.I.g$$

It is also useful to introduce the expected score increment $\mathcal{E}_D.s$, also known as conditional score, after an arbitrary state s:

$$\mathcal{E}_D.s \quad = \quad \sum_{g \in G.s} P.s.g \cdot F.s.g$$

We leave out the subscript D when it is clear from the context. The conditional score satisfies the following recurrence

$$\mathcal{E}.s = \sum_{e \in E.s} (p.s.e \cdot (f.s.e + \mathcal{E}.se)) \tag{2.1}$$

Note that $\mathcal{E}.s = 0$ if s is a final state ($E.s = \emptyset$).

2.2 The *OptExpected* strategy

The goal of Verhoeff's research was to compute and analyze an optimal strategy for solitaire Yahtzee.

An optimal strategy was defined as a strategy that has the highest possible expected score. We call such a strategy *OptExpected* for short.

Formally, *OptExpected* is defined as a strategy that achieves $\hat{\mathcal{E}}$:

$$\hat{\mathcal{E}} = \max_D(\mathcal{E}_D) \tag{2.2}$$

We can specialize the recurrence relation (2.1) for conditional expected scores by distinguishing roll and choice states, to obtain a recurrence for the optimum expected score:

$$\hat{\mathcal{E}}.s = \begin{cases} \sum_{e \in E.s}(p.s.e \cdot \hat{\mathcal{E}}.se) & \text{for } s \in R \\ \max_{e \in E.s}(f.s.e + \hat{\mathcal{E}}.se) & \text{for } s \in C \end{cases} \tag{2.3}$$

Based on this recurrence it is possible to determine a strategy that achieves the optimum expected score. A strategy must determine all choices for $s \in C$. A strategy that achieves an expected score of $\hat{\mathcal{E}}$ must make choices such that (2.3) holds. To be more specific, if a strategy sets $p.s.\hat{e} = 1$ for an $\hat{e} \in E.s$ for which $f.s.\hat{e} = \hat{\mathcal{E}}.s - \hat{\mathcal{E}}.s\hat{e}$ holds, the expected score for the strategy will be $\hat{\mathcal{E}}.s$.

2.3 State space reduction

For the decision process, not all information of a game state is relevant.

As an example, assume we have one scorecard where the Aces category has been scored for zero points, and the Twos category has been scored for six points. For the decision process such a card is of equal consequence as a card where six has been scored for the Aces category and zero for the Twos category. It is relevant which categories have been scored, but we do not need to know how we achieved the point total in detail.

Based on these observations we can merge equal states of the game tree. In the next section we reduce the game states for the *OptExpected* strategy.

2.4 Software design

The recurrence relation for *OptExpected* can be used to compute the optimum expected value. To prevent re-evaluation of formulas, a table can be used that stores results of formulas that have been evaluated already.

A similar reason to construct a table is when we want to construct an *Advisor* program, that advises a player on the choices that have to be made for the maximum expected score.

A strategy for Yahtzee can be represented such table. We decide to store only the optimum expected score for reduced game states between turns. From this information, and advisor program can reconstruct the decisions for all game states.

A number of elements of the game state are relevant for future decisions. **free** is the set of unscored categories, and can be any subset of the thirteen main categories. **usneed**, an integer value between 0 and 63, is the score which still has to be scored in the upper section before the player receives the upper section bonus. To determine how many points an extra Yahtzee scores, the boolean value **chip** defines whether the Yahtzee category was scored with 50 points. When the Yahtzee category has not been scored, the value of **chip** has no function.[1]

The reduced game states between turns consist of

$$free \times usneed \times chip \tag{2.4}$$

The corresponding table stores the optimum expected score for all reduced game states, and consists of $3 \cdot 2^{18} = 786432$ entries of real type, taking up 6 MB storage space.

A program that computes this table was constructed. Filling the table on a low-end computer took about half an hour.

An advisor program that uses this table to advise players on choices, can be found at [4].

[1] For more details on the state reduction we refer the reader to [2, 3].

Chapter 3

Problem Analysis

We will store a strategy much in the same way as Verhoeff. A table for the strategy *MaxProb* will consist of the maximum probabilities of achieving a score or more, for all reduced game states. This will allow an advisor program to reconstruct the decisions that are made in all states.

We examine the structure of *MaxProb*. For any gamestate s we have

$$MaxProb((s, 0)) = 1 \tag{3.1}$$

In the end state of the game, where no more events are possible, we cannot score any extra points. We have for $sc > 0$

$$MaxProb((endstate, sc)) = 0 \tag{3.2}$$

In the other situations however, we still have to score more than zero points, and we have

$$MaxProb((s, sc)) = \max_{e \in E.s} (MaxProb((se, score \dot{-} f.s.e)) * p.s.e) \tag{3.3}$$

This relation can be solved with similar methods as the ones used by Verhoeff. The main difference is the size of the state space. To compute the strategy $MaxProb_{400}$, the size of the state space is 400 times as large as that of the *OptExpected* strategy.

3.1 Expectations

The original problem that was solved by Verhoeff, computing a table representing the *OptExpected* strategy, had taken about half an hour on low-end computer. At that point was not obvious that determining a *MaxProb* strategy was feasible.

Storing a strategy for an instance $MaxProb_s$, the strategy with the highest probability of scoring s or more, requires a table of $s \cdot 6$ MB. The amount of computations required increases with a factor s.

Suppose we have any amount of internal memory available on a single computer. We could then adapt the software created by Verhoeff to compute a strategy table for $MaxProb_{800}$. The table would take 4.7 GB internal memory. Computation of the table would take 400 hours, which is almost 17 days. Unfortunately, both our time and our memory are limited.

We have access to a cluster of 16 computers. These machines are about twice as fast as the original machine. They have 256 MB internal memory each. The details can be

found in appendix B. On these machines the table does not fit into internal memory, and external memory will have to be used for writing and reading. Access to external memory is significantly slower than to internal memory. In this case the external memory is a harddisk, which is approximately E times slower than internal memory.

Determining the constant E is not easy due to interaction between caches, the order of element retrieval and the physical limits of the data traffic. Theoretically, $E > 6$ because of the data busses, but it is reasonable to assume that $E > 20$.

As an example we consider determining the single strategy $MaxProb_{800}$ on the cluster of 16 computers. We expect to require about $\frac{800 \cdot E}{2 \cdot 2 \cdot 16} = 12.5 \cdot E$ hours, under the false assumptions that all parallel nodes have no idle time and are used optimally[1].

We therefore assumed we would at most be able to determine a carefully chosen group of *MaxProb* strategies.

At the end of the project, we computed all $MaxProb_{0...800}$ strategies in a single run of 40 hours.

3.2 Requirements

The main requirement is to develop a program that will compute an optimal strategy for achieving a given score or more.

3.2.1 Parallel strategy generation program

A program is constructed to compute the optimal strategy. This program will be executed on a parallel computer cluster to reduce computation time.

As its input this program will take a parameter 'ScoreToBeat', the score that the strategy needs to achieve. Upon completion, the program will output the strategy. A strategy will be defined by a large file of approximately $ScoreToBeat \cdot 6$ MB. This number corresponds to the strategy tables used by Verhoeff.

3.2.2 Specification

Given a set of rules for the Yahtzee game, a gamestate s and a score sc, the equation for which we are constructing a table is:

$$MaxProb_{ScoreToBeat}((s, sc))$$

This function determines the maximum probability of scoring sc points or more from a game state gs onwards. For a game starting in the initial state we can then compute the maximum probability of achieving a certain score or more. To formalize this, we extend the GameState with a ScoreToBeat component to BeatScoreState.

$$BeatScoreState = GameState \times Score \tag{3.4}$$

we then reformulate our problem as finding the value of

$$MaxProb_{ScoreToBeat}((I, ScoreToBeat))$$

[1] We also assume that the external retrieval of values, in order to compute an element, is significantly slower than the time it takes to compute the element from internal memory. This assumption holds because computing an element of T requires inspecting about 5000 other elements of T.

We are also interested in the strategy that achieves such a maximum probability. With some extra computation we are able to derive the strategy from the values of *MaxProb*. To this end, we would like to store all the values of *MaxProb* in a table.

Along the lines of the research by Verhoeff, after we have generated this table, we can use it to compare strategies, and possibly to construct an 'advisor' program.

The main problem of the implementation is fast storage space: the size of the BeatScoreState space is too large to fit all the desired data in fast storage at once.

3.2.3 Size requirements

The size of the GameState space is 3×2^{18}. The BeatScoreState space has $ScoreToBeat \times 3 \times 2^{18}$ elements, and if the probabilities are stored as 8-byte reals, we require $ScoreToBeat \times 6$ MB of storage space. For $ScoreToBeat = 800$ we would require approximately 4.7 GB.

When accessing the table from external memory instead of internal memory, computation time is severely increased.

In the next chapter we will develop a caching method for Yahtzee that allows us to minimize cache misses, and that at the same time allows us to parallelize the computation.

Chapter 4

Software

The software that is required to for this thesis, can be analyzed on a number of levels. The main issues will be meeting memory and parallelization requirements.

The next two chapters can be read in any order. Chapter 4 addresses the specific problems of this assignment, where we are constructing a strategy for the game Yahtzee.

Chapter 5 solves the main problems on a more abstract level. There is no mentioning of Yahtzee, or the exact function to be computed.

Readers with limited time can decide which chapter to read. Those who are more interested in the implementation of the caching structure for this particular problem are recommended to read chapter 4. Readers who are interested in the formal construction and application area of the caching method are advised to read chapter 5.

4.1 Jobs

We propose to partition the table into a number of segments. We will try to choose the segments, which we will call *jobs*, in a way that allows us to reduce external memory access as much as possible.

By "computing a $job(i)$" we mean computing the values of $MaxProb(gs)$ for all $gs \in job(i)$. We aim to find a partitioning that allows us to do some preparational work, compute all values of *MaxProb* in the job using only internal memory, and write the results to external memory.

Assuming such a partitioning exists, we can construct a skeleton program. This program is shown in figure 4.1.

For our purposes we have a number of requirements to the job definition:

Requirement A There must be at least one job we can compute without depending on any other job.

Requirement B We can construct a fast algorithm that can determine which (completed) jobs are required to compute all values in a given job. (Dependency)

Requirement C We can implement fast mapping and unmapping functions for the jobs data to the game state space: a large overhead on retrieving or writing data for a job in memory is unacceptable.

```
program FillMaxProbTable;
begin
  while ('there are uncomputed jobs') do
  begin
    Select an i s.t. Job(i) can be computed now
  ; Determine on which Jobs Job(i) depends
  ; Load these jobs into memory
  ; for e := all in Job(i) do
    begin
      compute MaxProb(e) using values in memory
    ; store MaxProb(e) in memory
    end
  ; Write the job from memory
  end
; combine Job data into desired MaxProb table format
end.
```

Figure 4.1: Yahtzee skeleton program using jobs

Requirement D Every job, combined with the jobs it directly depends on and the algorithm, must meet our fast-access requirements. (E.g. can fit in the available internal memory) This allows us to compute the values of the output job in a batch without accessing external memory.

Requirement E Once all jobs have been computed, we are able to construct the table for *MaxProb* from the data of the combined jobs (in a relatively short time).

If we want to compute jobs in parallel, we have one extra requirement:

Requirement F Parallelism: We want to be able to distribute the jobs over m processors, while keeping idle time at a minimum. To achieve this, the job definition should at least allow us to start more than one jobs at the same time (i.e. non-linearity).

For the Yahtzee MaxProb problem it is not immediately obvious if there exists a job definition with sufficient job sizes that meets the criteria. We initially look for the job definition that has the largest possible job sizes, to reduce overhead. Larger job sizes possibly conflict with requirement (F), and we suspect at this point that a trade-off will have to be made.

4.1.1 Observations

Single entry jobs

Note that if we define a job defined as a single entry from the table, we meet the above criteria. That is not a desired solution, as there is no caching involved. The computation time of single entry jobs is comparable to using external memory only, which we considered too slow.

Layer jobs

The GameState space has one natural ordering that results from the Yahtzee rules: A turn ends when a category is scored. For every game, we have thirteen such turns, one for each category. When we start a the first turn, no category is scored. At the start of the Nth turn, $N-1$ categories have been scored. We call a partition based on turns a *layer partition*.

We define fourteen layer jobs, with

$$job(i) = \{\, ((free, chip, usneed), score) \mid 13 - i = |free| \,\} \tag{4.1}$$

A layer corresponds to the state of the game between turns. $job(0)$ corresponds to the start of the game, where no categories have been scored. $job(13)$ corresponds to a completed game, where all categories have been scored.

We know that the values in $job(i)$ are only dependent on $job(i+1)$, and that $job(13)$ is not dependent on any other job. Unfortunately, computations show that the middle layers do not meet the memory requirement (D).

For the size of a $job(l)$ where $ScoreToBeat = 300$, we have

$$\binom{13}{l} \times \frac{3}{2} \times 300 \times ScoreToBeat = \binom{13}{l} \times 28800$$

The skeleton program caches the job it is computing, as well as the jobs the computation depends on. For a layer partitioning this means that to compute $job(i)$ the cache must be able to contain $job(i)$ and $job(i+1)$ at them same time. We compute the amount of internal memory required, in bytes:

$$\left(\binom{13}{l} + \binom{13}{l+1}\right) \times 28800 \times 8$$

We find the maximum for this function for $l = 6$, where the memory requirement is approximately 750 MB. This is too large for our purposes.

A second problemn is that the layer job definition does not meet requirement (F), because a $job(i)$ can only be started after computation of $job(i+1)$, which implies that jobs are run one after another.

4.1.2 Job proposal

We propose to refine the layer job definition. To control the coarseness of the resulting definition, we introduce a parameter $FixedCatSet$. We decide to split $CategorySet$ into two disjoint sets called $FixedCatSet$ and $VarCatSet$. The idea is to refine the layer job definition by splitting a layer into several parts according to $FixedCatSet$.

$FixedCatSet$ tells us which categories are used to determine to which job a state in the layer belongs. We only have to look at the *layer* (derived from the number of scored categories) and the scored categories to find out to which job a $BeatScoreState$ belongs.

For example, if $FixedCatSet = \{\, Aces \,\}$, a layer job is split into two jobs. One job contains the states in the layer for which *Aces* have been scored, and one job the other states of the layer.

Definition

We formally define the jobs for any $layer \in [0..13]$ and any $fixed \in \mathcal{P}(FixedCatSet)$ by

$$\begin{aligned} job(layer, fixed) &= \{((fixed \cup v, chip, usneed), sc) \\ &\quad \mid v \in \mathcal{P}(VarCatSet) \wedge 13 - layer = |fixed \cup v|\} \end{aligned}$$

For all the states belonging to a $job(layer, fixed)$, the scorecard will look the same for all categories in $FixedCatSet$.

Properties

Using this job definition, we are able to derive some properties.

If we include more categories in $FixedCatSet$, we get smaller jobs.

For job dependency, we find that $job(layer, fixed)$ depends on

$$\{job(layer + 1, fixed)\} \cup \{job(layer + 1, fixed \setminus \{c\}) \mid c \in fixed\}$$

The left half of the conjunct expresses that a category is scored which is an element of $VarCatSet$. For those states, $fixed$ remains the same.

The right half expresses that a category was scored which is not an element of $VarCatSet$, so it must be an element of $FixedCatSet$. As an added constraint we have that this category must not have been scored already.

Splitting a layer partitioning allows for parallel computation, as the parts within a layer are independent of eachother.

Recall that fitting a job computation into memory was our main problem. As we have differing job sizes, and still have to decide $FixedCatSet$, we first need to calculate how our choice affects memory requirements. We will investigate this in the generalization of the next chapter.

4.1.3 Implementation

We implemented this technique and generated a 4.7 GB table, from which the $MaxProb_{0...800}$ strategies can be derived. We used the partitioning technique to reduce internal memory requirements, and also to allow parallel computation.

In this thesis we will not consider implementation details for the constructed program. We can indicate however, that it is possible to implement fast mapping algorithms for reading from and writing to the cache. The intermediate data that was generated by job computations was stored in such a way that element retrieval was possible, and there was no need for conversion of this format to another table. An advisor program can access the table using minor modifications to existing functions.

4.2 Generalization

In the next chapter we will generalize the techniques that were introduced here, and we will give conditions under which this type of partitioning can be applied. The more general approach will allow us to derive some properties such as required cache sizes.

Yahtzee has 13 categories, and this set will be generalized to A, with $|A| = 13$, in the next chapter. The $FixedCatSet$ will return in a more general form as the set G. The other components of the state space: $chip$, $usneed$ and $ScoreToBeat$ will be combined into the set B.

Chapter 5

Evaluating certain recurrent functions on a large domain

In the previous chapter, we developed a program for a specific problem. The technique that was used, however, is more generally applicable.

In this chapter, we design a parameterized program structure that computes a table for a recurrent function. The function *MaxProb* is an example of such a function. The functions we consider have a specific dependence pattern. The program parameter controls a trade-off between speed and the amount of internal memory required.

5.1 Introduction

We would like to design a program to construct a table T for a function f. That is, for every x in the domain of f, the table T stores $f(x)$. If T is too large to fit into internal memory, we will have to resort to external memory for storage. External memory typically is significantly slower than internal memory.

For a recurrent function f, the value of $f(x)$ depends on other values $f(y)$. Instead of evaluating all those $f(y)$, we would like the program to read these values from T, as well as write $f(x)$ to T. Because T is stored in external memory, this requires additional access to external memory. A technique for reducing such access is caching. In this case, the program can use internal memory as a cache.

Caching mechanisms are based on assumptions about typical memory access patterns. We can create a more efficient cache, if we know more about the access pattern of a function. For a recurrent function, this pattern is governed by the dependencies between the values of f and the order in which we construct T.

We use the dependence relation to design a parameterized program structure for a specific class of recurrent functions. We define this class in section 5.2. The program uses buffers to cache parts of T, and the parameter controls the size of the buffers. Larger buffers result in fewer accesses to external memory, leading to faster execution. The parameter effectively controls the trade-off between speed and required internal memory size.

5.2 A class of recurrent functions

We only consider a specific kind of recurrent functions. These functions have a general dependence structure in common. In this section, we define this class of functions. We write $a \to b$ to denote that the computation of $f(a)$ depends on the value $f(b)$.

As an example in this class, consider a function h on the domain $\mathcal{P}(\{1,2,3\})$, that is, on the eight subsets of $\{1,2,3\}$, with dependence relation depicted in figure (5.1). Thus, the computation of $h(s)$ depends on the values of $h(t)$, where t is of the form $s \setminus \{a\}$ for all $a \in s$. This dependence relation is also known as a cube.

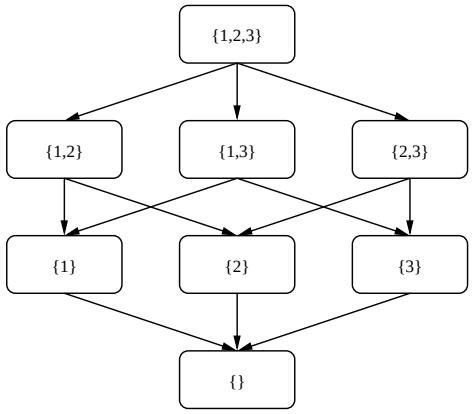

Figure 5.1: Dependence graph for a function on $\mathcal{P}\{1,2,3\}$

We generalize this structure in three ways. First, we allow arbitrary powersets $\mathcal{P}(A)$. Second, we allow an extra argument in some set B. Third, we allow the dependency to be a subrelation of the A-dimensional hypercube.

The generalized class consists of functions on a domain of the form $\mathcal{P}(A) \times B$, for which the dependence relation satisfies

$$(s,b) \to (s',b') \;\Rightarrow\; s' = s \vee (\exists\, a : a \in s : s' = s \setminus \{a\}) \tag{5.1}$$

That is, $f(s,b)$ may only depend on $f(s,b')$ and $f(s \setminus \{a\}, b')$ for some b' and $a \in s$. Obviously, we also require that the dependence has no cycles.

The domain B could have some further structure by itself, or it could be absent. In the latter case, we take for B the singleton set $\{\cdot\}$.

From now on we assume that the function f is a member of this class of functions.

5.3 Partitioning the table

We propose a program that constructs the table T in a number of steps. In a step, the program constructs a part of T. For each of these steps, we ensure that the part of T we are constructing, as well as the parts on which it depends directly, fit together into internal memory.

We define a partitioning of T by means of an equivalence relation. An equivalence relation $\sim$ on the domain of T groups elements into equivalence classes. An element (s,b) of T is in

the same part as (s', b') iff they belong to the same equivalence class. For an equivalence relation $\sim$, we define the class $K_\sim(s, b)$ containing (s, b) by

$$(s', b') \in K_\sim(s, b) \;\Leftrightarrow\; (s, b) \sim (s', b') \tag{5.2}$$

5.3.1 Example

Suppose we have a partitioning of T according to

$$(s, b) \sim (s', b') \;\Leftrightarrow\; |s| = |s'|$$

where $|s|$ denotes the cardinality of s.

When we use this equivalence relation to partition T for the domain $\mathcal{P}(\{1, 2, 3\}) \times \{\cdot\}$, the elements are grouped as in figure (5.2). The example has 4 equivalence classes. There is a class for each possible cardinality of the elements of $\mathcal{P}(\{1, 2, 3\})$.

5.3.2 Equivalence class dependencies

To determine in which order the program should compute parts of T, we investigate how the dependence relation of f relates to the partitioning of T. For an equivalence relation $\sim$ and a dependence relation $\to$ on the domain of f, we denote the induced class dependence also by $\to$. It is defined as the least solution of

$$(s, b) \to (s', b') \;\Rightarrow\; K_\sim(s, b) \to K_\sim(s', b') \tag{5.3}$$

That is, equivalence class K depends on K' iff K contains an element that depends on an element of K'.

For the example from the previous section, the class dependence relation is depicted in figure (5.2).

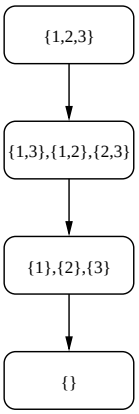

Figure 5.2: Dependence graph for the 'cardinality' equivalence on $\mathcal{P}(\{1, 2, 3\})$

Not all equivalence relations give rise to suitable class dependencies. In particular, if we want to compute the parts in a specific order, we must avoid cyclic class dependencies. Self loops on classes are not problematic. But if the class dependence relation is such that $K_1 \to K_2$ and $K_2 \to K_1$, then we cannot compute the parts in sequence.

5.3.3 A parameterized equivalence relation

If we use the 'cardinality' equivalence relation to partition the table, then we get a linear dependence graph. However, for some purposes this partitioning might be to coarse. Such was the case for optimal beating of high scores in Yahtzee.

We want to refine the 'cardinality' partitioning in a tunable way. For this, we introduce a parameter G that controls the coarseness of the partitioning. For $G \subseteq A$ we define a new equivalence relation $\sim_G$ that subdivides each 'cardinality' class based on the intersection with the set G:

$$(s, b) \sim_G (s', b') \;\Leftrightarrow\; |s| = |s'| \;\wedge\; s \cap G = s' \cap G \tag{5.4}$$

For $G = \emptyset$, this relation reduces to the 'cardinality' equivalence. This results in $|A| + 1$ equivalence classes. At the other end of the spectrum, for $G = A$, all elements of the powerset of A correspond to an equivalence class, resulting in $2^{|A|}$ equivalence classes.

For all elements of a class $K_{\sim_G}(s, b)$, the intersection of s and G is the same, but $s \cap (A \setminus G)$ may differ. Because both these components, G and $A \setminus G$, return frequently in our formulas, we define $\overline{G}$ by

$$\overline{G} = A \setminus G \tag{5.5}$$

Note that in the following analysis, only the cardinality of G is a concern. Exactly which elements of A are in G is not important here. For larger cardinalities of G we have more, but smaller, equivalence classes. In general, we find that $\sim_G$ produces $2^{|G|} \cdot (|\overline{G}| + 1)$ equivalence classes.

Note that, in equivalence class $K_{\sim_G}(s, b)$, all elements have $|s|$ and $s \cap G$ in common. The freedom is in choosing the $|s| - |G| = |s \cap \overline{G}|$ elements of s from $\overline{G}$, and the element of B. Thus, for the number of elements in the equivalence class $K_{\sim_G}(s, b)$, we find:

$$|K_{\sim_G}(s, b)| \;=\; \binom{|\overline{G}|}{|s \cap \overline{G}|} \times |B| \tag{5.6}$$

5.3.4 Analyzing the class dependence for $\sim_G$

The partitioning based on $\sim_G$ yields smaller parts, while preserving the dependence structure as much as possible. Recall that an element (s, b) in formula (5.1) depends on either an element with equal s component, or on an element with $s \setminus \{a\}$ for some $a \in s$. We discern two cases for $s \setminus \{a\}$. We have either $a \in s \cap G$, or we have $a \in s \cap \overline{G}$.

For $a \in s \cap G$, computation of $f(s, b)$ depends on a value from an equivalence class with lower cardinality. For each possible $a \in s \cap G$ this concerns a different equivalence class.

For $a \in s \cap \overline{G}$, the computation of $f(s, b)$ depends on a value from one specific class. All elements $(s \setminus \{a\}, b')$ belong to the same class, because a is an element of $\overline{G}$.

More formally, we have the following dependencies between equivalence classes:

$$\begin{array}{lcll} K_{\sim_G}(s, b) & \rightarrow & K_{\sim_G}(s, b') & \\ K_{\sim_G}(s, b) & \rightarrow & K_{\sim_G}(s \setminus \{g\}, b') & \text{for all } g \in s \cap G \\ K_{\sim_G}(s, b) & \rightarrow & K_{\sim_G}(s \setminus \{\overline{g}\}, b') & \text{for any one } \overline{g} \in s \cap \overline{G} \end{array} \tag{5.7}$$

An example of this class dependence for $A = \{1, 2, 3\}$ and $G = \{1\}$ can be found in figure (5.3). Self loops have been omitted.

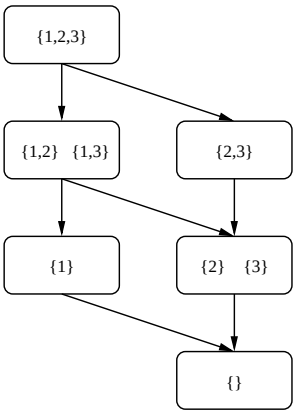

Figure 5.3: Dependency graph example for $A = \{1, 2, 3\}$ and $G = \{1\}$.

5.4 Program design

The program we propose constructs the table T in parts, based on the equivalence relation $\sim_G$, and we take G to be a parameter of the program. The values of f for one equivalence class are computed, before proceeding to another equivalence class.

From formula (5.7) we find that an equivalence class depends on a number of other classes, but that they are either the class we are computing, or classes for a lower s-cardinality. If we use the s-cardinality of a class to determine the order of class construction, then we can be sure that classes are available when needed.

The program constructs all equivalence classes in an order that preserves increasing s-cardinality. To construct a class, we must access values from classes it depends on.

5.4.1 Caching

We now introduce a program design and describe a caching mechanism. Also see figure (5.4).

To construct T for a class K, the program will preload all classes on which K depends from external memory into a cache.

The program then computes all values of T for K and writes them to the cache. All values on which these computations depend are now available in the cache. After the part of the table corresponding to K is completed, it is written from cache to external memory, and the caches are invalidated. This is repeated until T is constructed for all equivalence classes.

Depending on $|G|$ and the order in which we construct the classes, it is possible that the next class to be computed depends on classes that were in the cache already. In such a case we invalidate the cache only for classes that are no longer required.

5.4.2 Parameter choice

In general, a lower cardinality $|G|$ yields larger classes, which leads to fewer external accesses. However, this requires a larger cache.

The requirements for the cache are the following. For all equivalence classes, the class itself, as well the classes on which it depends, must together fit into the cache. Based on

```
program ComputeTable;
begin
  for C := 0 to |A| do
  begin
    for each K(s,b) where |s| = C do
    begin
      { Read classes on which K(s,b) depends from external memory }
      ReadInputCaches;
      for each x in K(s,b) do
      begin
        { Use the caches to read and write values }
        T[x] := f(x)
      end;
      { Write class K(s,b) to external memory }
      WriteOutputCache;
      InvalidateCaches
  end
end.
```

Figure 5.4: Generic skeleton program

formulas (5.7) and (5.6), we find that the maximum number of elements in the cache at one time equals

$$\left((|G|+1)\cdot\binom{|\overline{G}|}{|\overline{G}| \text{ div } 2}+\binom{|\overline{G}|}{(|\overline{G}|-1) \text{ div } 2}\right)\cdot|B| \tag{5.8}$$

Note that for fixed n, the binomial coefficient $\binom{n}{k}$ is maximized by $k = n \text{ div } 2$.

We can calculate the cache size requirements, if we know $|A|$, $|B|$, $|G|$, and the space required to store a function value of f. Figure (5.5) shows required cache sizes as a fraction of the size of T, for the extreme cases $|G| = 0$ and $|G| = |A|$.

Formula (5.8) tells us when the suggested program structure can be of use. It implies bounds for the algorithm, and how controlling the cardinality of G can influence the required cache size.

Reducing the size of the cache comes at a cost. If we reduce the size of the cache by choosing a larger $|G|$, more parts have to be preloaded. Some parts will be preloaded multiple times

We calculate the amount of preloaded data by taking the number of times a class is preloaded, and multiplying with the size of the class, for all classes. The number of times a class is preloaded is defined by the dependence relations between the classes, resulting in

$$\begin{array}{ll} |G| - |s \cap G| & \text{iff } |s \cap \overline{G}| = |\overline{G}| \\ |G| - |s \cap G| + 1 & \text{iff } |s \cap \overline{G}| \neq |\overline{G}| \end{array} \tag{5.9}$$

Multiplying this with the class size and taking the sum over all classes, we find that the total number of elements that is preloaded is

$$((|G|+2)\cdot 2^{|A|-1} - 2^{|G|})\cdot|B| \tag{5.10}$$

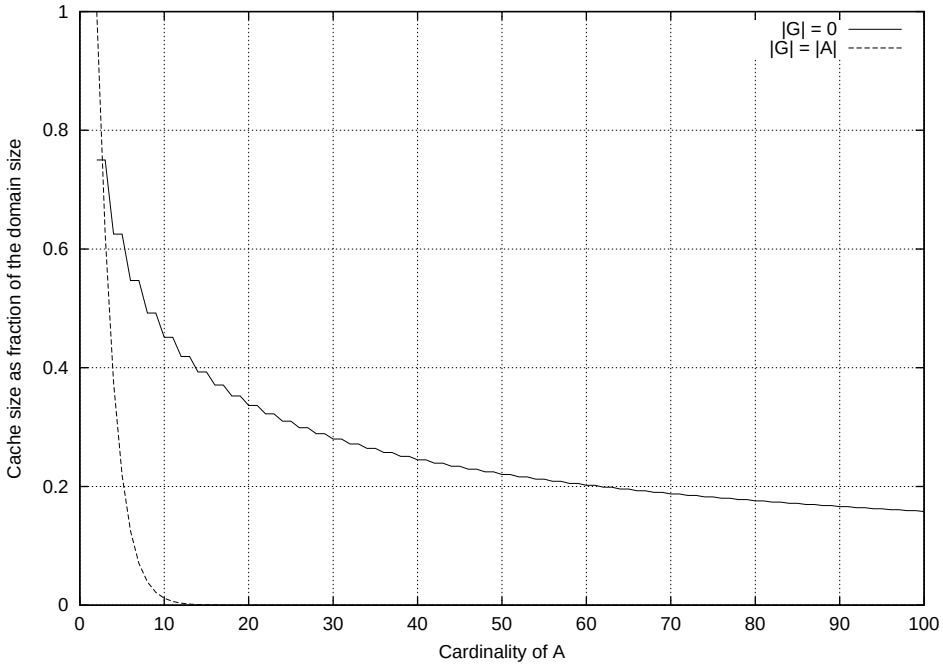

Figure 5.5: Cache size bounds. Indicated as a fraction of the size of T.

If we divide the total amount of preloaded data by the size of the table T, we get the average number of external read actions per element of T. In cache terminology, each external read corresponds to a cache miss[1]. The average number of cache misses (external reads) per element of T is given by

$$1 + \frac{|G|}{2} - \frac{1}{2^{|\overline{G}|}} \tag{5.11}$$

The bounds for the average cache misses can be seen in figure (5.6).

An efficient cache reduces cache misses as much as possible, so we will choose the cardinality of the parameter G as small as possible. However, decreasing $|G|$ results in larger cache sizes. In practice the available cache memory sets a lower bound on $|G|$.

Note that the number of cache misses is not dependent on the specific dependence relation of the function.

5.4.3 Yahtzee application

This technique was used for a real problem, where we wanted to determine an optimal strategy for playing the game Yahtzee. The goal of this strategy is to maximize the probability of scoring more than S. This score S is a parameter of the problem. The solution is based on a recurrent function that, for each game state and remaining score to achieve S' ($S' \leq S$), yields the probability to score more than S' in the moves after that game state.

We use a table to store information about the strategy. The table T represents the probabilities of achieving S or more, for all possible states of the game between turns. This

[1]Because data is preloaded, there are no actual cache misses during execution. In a conventional cache there is no preloading. It is possible to rewrite the skeleton program using a conventional cache.

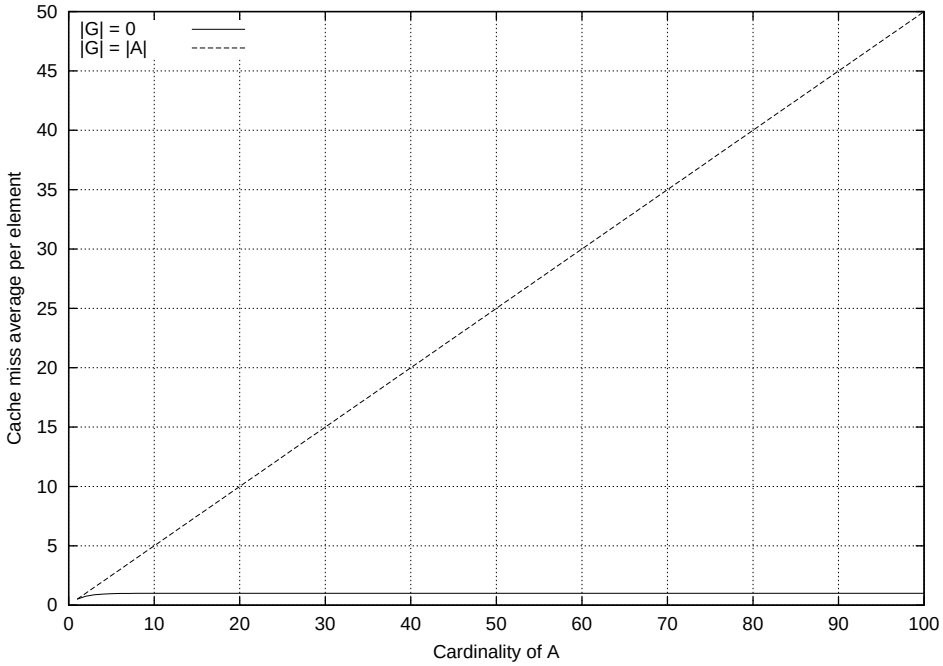

Figure 5.6: Cache efficiency bounds. Average number of cache misses per element of T.

table can be used to create an advisor program, which can help players to decide what to do in each possible state of the game.

As an example, we explain how to choose the parameter $|G|$ for constructing the table, in case of a strategy that scores more than 400 with maximum probability. To compute this table, the parameters are $|A| = 13$, and $|B| = 2 \cdot 64 \cdot 400 = 51200$. Storage of an element of the table T, of type real, requires 8 bytes. The size of the complete table is $2^{|A|} \cdot |B| \cdot 8 = 3.2$ GB. On the production machine, 256 MB internal memory is available.

In figure (5.7), a horizontal line is drawn at 256 MB. For $|G| = 5$, a cache of 186 MB is required. The corresponding equivalence relation splits the table T into $2^5 \cdot (13 - 5 + 1) = 288$ equivalence classes.

This is an optimal choice for this particular problem. Lower values of $|G|$ result in cache sizes that are too large for the given problem, and higher values of $|G|$ cause more cache misses (external reads).

5.5 Conclusions

If we want to evaluate or construct a table for a recurrent function f, as defined in section 5.2, it is possible to create an efficient caching algorithm.

A partitioning of the domain of f is used as a basis for a caching structure. The caching algorithm reduces external read actions, or cache misses, independent of the details of f. A parameter controls the trade-off between the amount of cache misses, and the size of the cache.

Formula (5.8) can be used to determine the size requirements of the cache for a given f. A smaller cache increases the amount of cache misses. The amount of cache misses can be

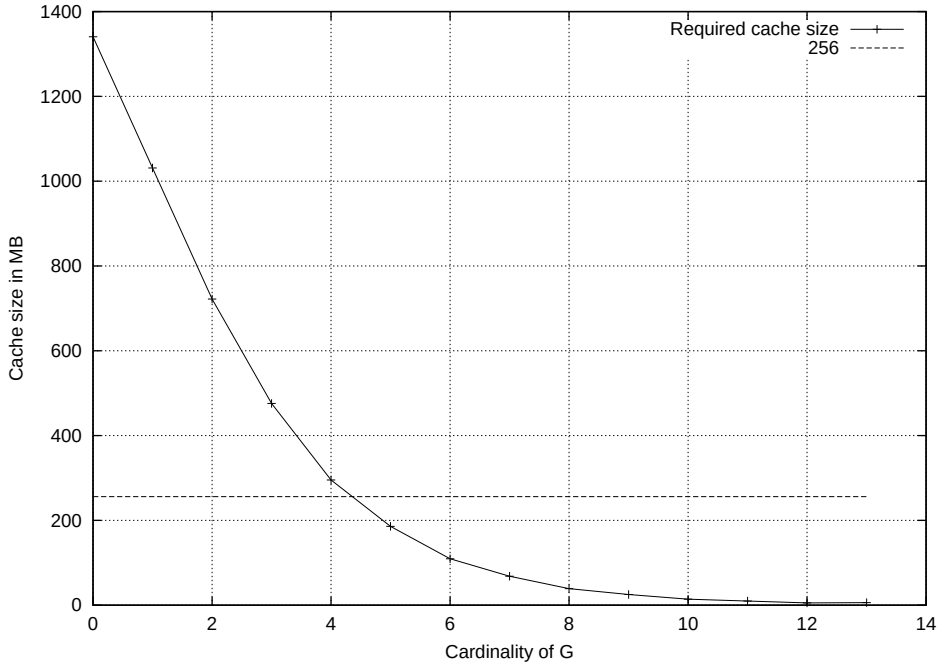

Figure 5.7: Yahtzee case study: required cache sizes

found using formula (5.11).

In practical settings, the available maximum cache sizes are often known. Our findings can also be used to choose a suitable parameter G if an upper limit for the cache size is given.

We have used this partitioning technique to compute the tables for the Yahtzee $MaxProb_{0...800}$ strategies. For the implementation we used the partitioning to parallellize the computation of the table.

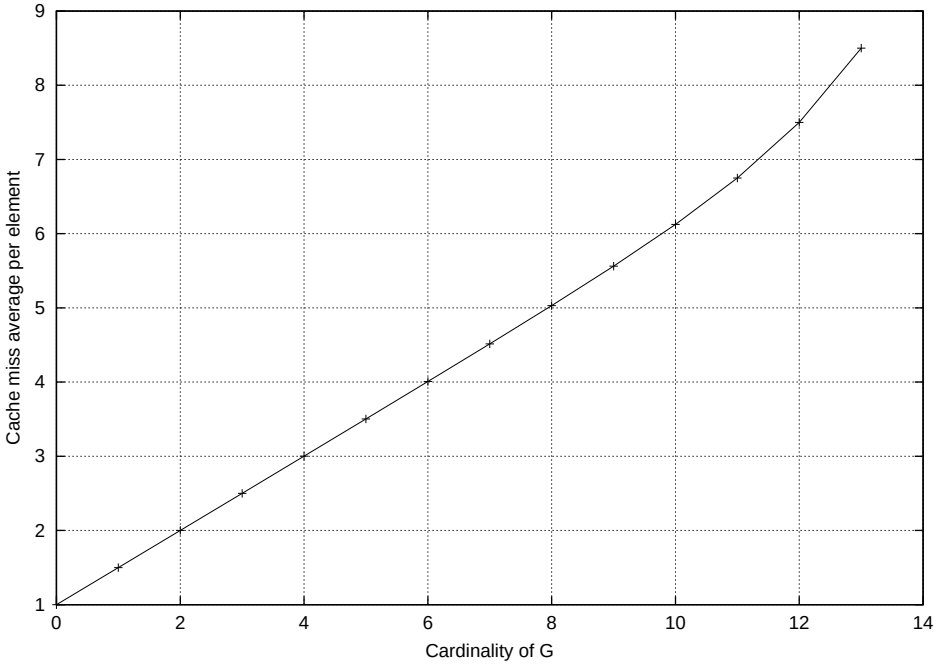

Figure 5.8: Yahtzee case study: average number of cache misses

Chapter 6

Results

At the end of the project, all $MaxProb_s$ strategies for $s \leq 800$ where determined in a single run of 40 hours on the parallel cluster.

6.1 Probabilities

In table 6.1 and figure (6.1) we find the main results of the *MaxProb* strategies. For score s, $0 \leq s \leq 800$ we have determined the strategy that has the highest probability of achieving s or more. The probability $P(score \geq s)$ is shown for each strategy $MaxProb_s$.

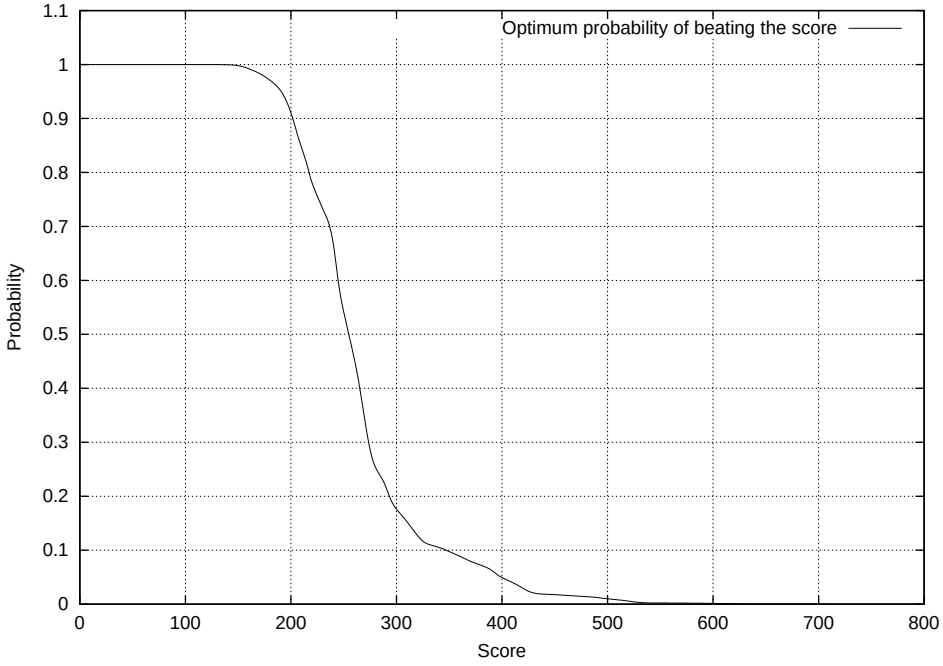

Figure 6.1: Optimum probabilities for $MaxProb_{0...800}$

During his research, Verhoeff simulated playing five million games of Yahtzee using the *OptExpected* strategy. The resulting scores were stored. We used this data to approximate $P(score \geq s)$ for the *OptExpected* strategy. These values can be found in figures 6.2 and 6.3,

Score to beat: s	$P(score \geq s)$ for $MaxProb_s$	$P(score \geq s)$ based on *OptExpected* simulations
0	100.000000	100.000000
50	100.000000	100.000000
100	99.999975	99.998800
150	99.793817	99.131700
200	91.080460	86.358500
250	54.188692	48.370300
300	17.603092	14.310200
350	9.759235	7.238200
400	4.932696	3.836780
450	1.754318	1.138200
500	0.997806	0.721620
550	0.215689	0.118140
600	0.141191	0.090720
650	0.019010	0.008800
700	0.014113	0.007380
750	0.001303	0.000000
800	0.001009	0.000000

Table 6.1: The probabilities of achieving some scores s using $MaxProb_s$ and *OptExpected*

and can be compared to the corresponding $MaxProb_s$ strategy.

The absolute differences between the simulations of *OptExpected* and the *MaxProb* strategies can be found in figure 6.4. We find that the *MaxProb* strategies perform better for beating any given score.

As the probabilities decrease rapidly for higher scores, a relative difference graph as in figure 6.5 may be more useful. This graph shows the probability differences between the simulations of *OptExpected* and the probability of $MaxProb_{0...800}$, divided by the probability of the simulations for each score. This indicates how much better $MaxProb_{0...800}$ strategies perform, relative to *OptExpected* results.

The *MaxProb* strategies for high scores will attempt to score a Yahtzee every turn. After the first roll, the *OptExpected* strategy will reroll in a way that maximizes the expected score, possibly missing the chance of rolling a Yahtzee. A *MaxProb* strategy therefore has a higher probability of scoring Yahtzees.

In figure 6.5 we can see these characteristics: For a game where a Yahtzee is thrown every turn, we score 50 points in the first turn, and 100 points in every turn after that. The graph shows that *MaxProb* performs better in general, but is also significantly better in scoring Yahtzees. This difference becomes increasingly clear for $score > 400$.

6.2 Comparing strategies

For this thesis we only compared strategies based on their expected results, and not on the way they progress through a game.

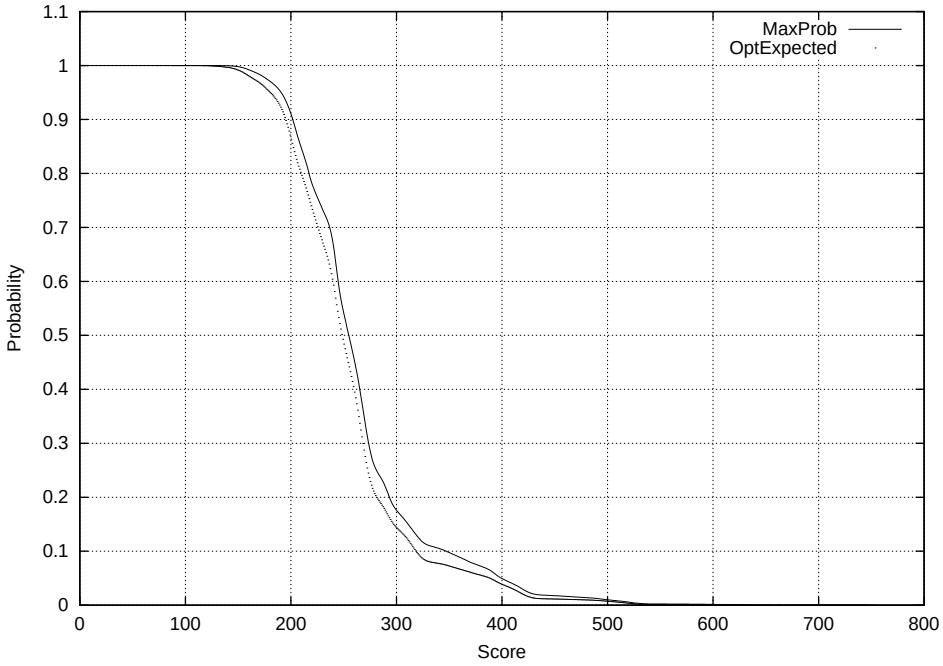

Figure 6.2: $MaxProb_{0...800}$ and *OptExpected*

6.2.1 OptExpected

In general, *OptExpected* is the best solitaire strategy, because the aim of the game is to achieve the highest possible score. When it comes to achieving the highest probability of achieving a score s or more, the $MaxProb_{0...800}$ strategies are always at least as good as *OptExpected*. Moreover, $MaxProb_{0...800}$ strategies are significantly better at achieving very high scores.

6.2.2 MaxProbApprox

During the writing of this thesis, Verhoeff designed an approximation strategy, that aims for the maximum probability of achieving a score s or more. This approximation technique assumes that the the probabilities for scoring are normally distributed for all gamestates. The *OptExpected* scores and variances were known for all gamestates, and this was used to construct a *MaxProb* approximation strategy.

It turns out that this approximation does not perform well at all. This was expected, as the probabilities are not normally distributed. To achieve a score s or more, it is even better to use *OptExpected* than to use *MaxProbApprox*.

6.3 Extending for multi player games

Although the *OptExpected* and *MaxProb* strategies are intended for solitaire games, they could be used in multi player games.

When we use *OptExpected* to play, we play the solitaire game as usual. Our strategy does not change, whatever an opponent does, because we have no options.

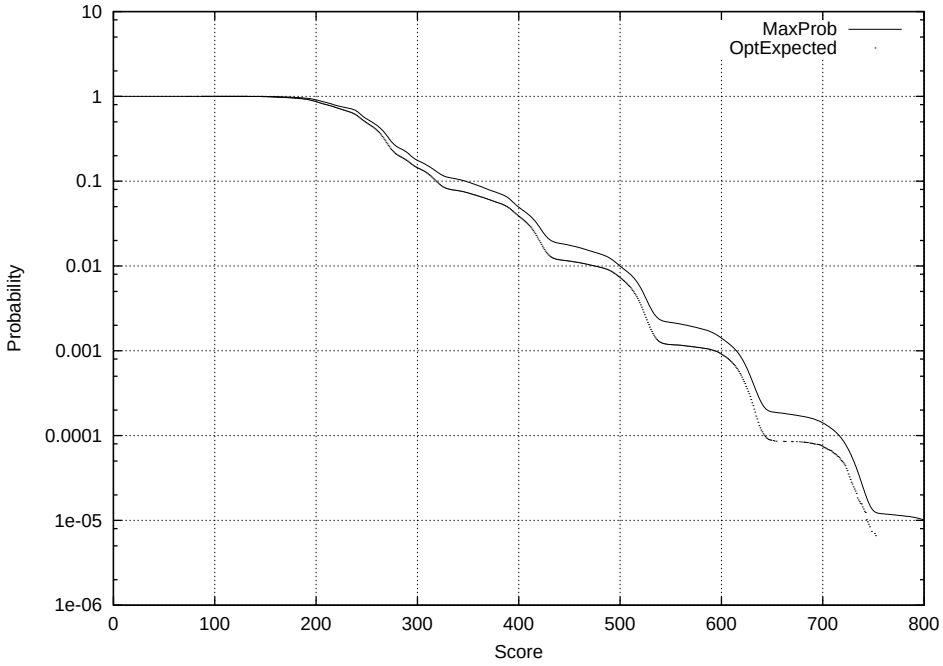

Figure 6.3: $MaxProb_{0...800}$ and *OptExpected* (logarithmic scale)

Common sense suggests that if an opponent is lucky in his first few turns, we might consider taking more risks. We know how high our expected score is, and if that is not good enough, a different strategy might be better.

Conversely, if an opponent is very unlucky in the first few turns, we might prefer a strategy that takes less risks. We would like to play it safe.

The *MaxProb* family of strategies can be used for such changes in our game strategy. If we can estimate the expected scores of our opponents, we can use a *MaxProb* strategy to maximize the probability of beating our best opponent. This gives us a better chance of winning than just playing for the highest expected score.[1]

[1] Estimating the expected score of opponents assumes that we have knowledge of their strategies. We could assume that good players play according to *OptExpected*.

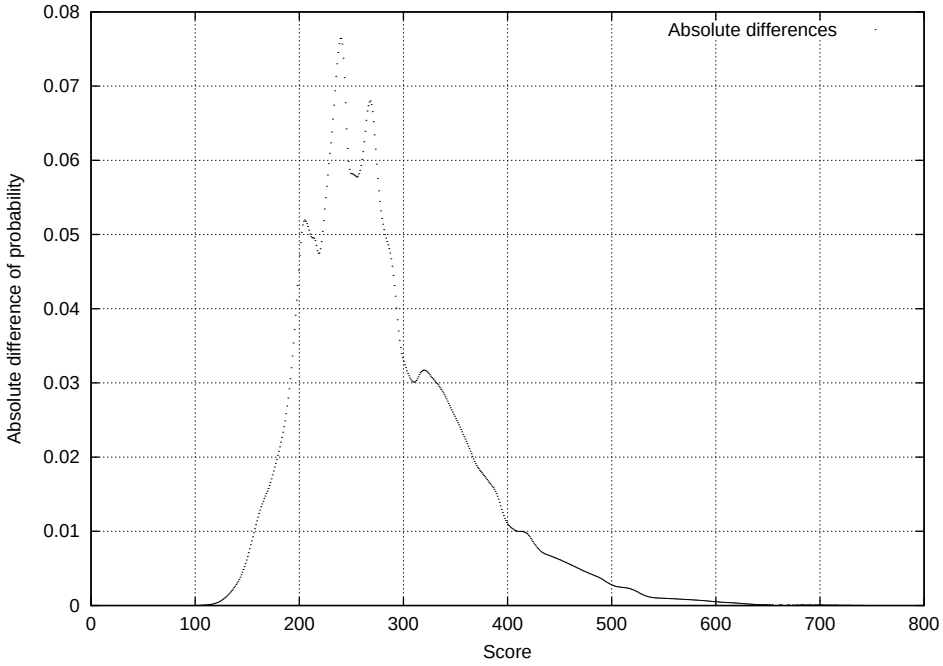

Figure 6.4: Absolute differences: $MaxProb_{0...800}$ and *OptExpected*

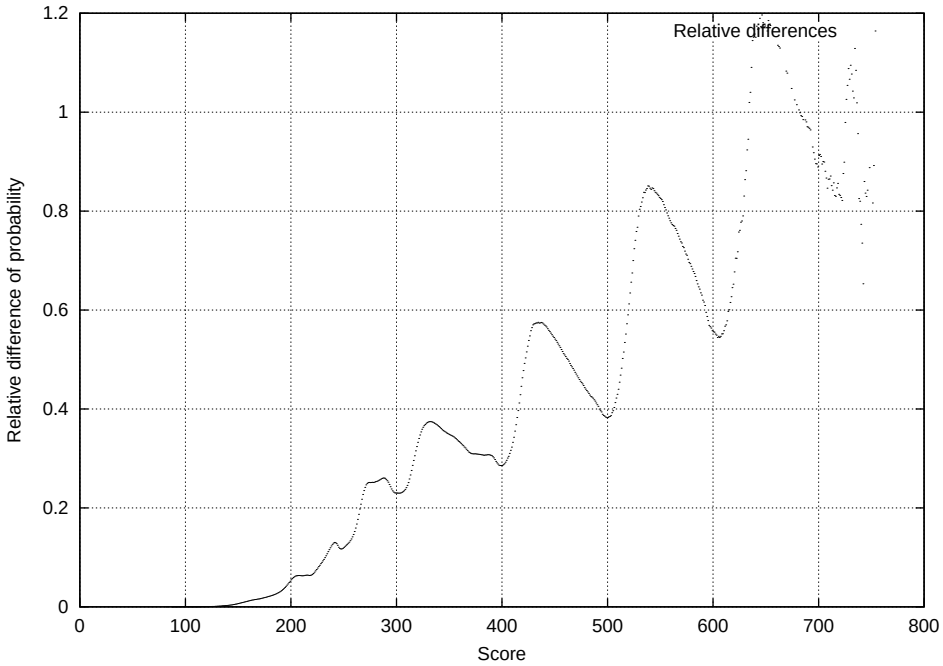

Figure 6.5: Relative differences: $MaxProb_{0...800}$ and *OptExpected*

Chapter 7

Conclusions and suggestions for further research

7.1 Conclusions

The writing of this thesis has resulted in two kinds of results. On one hand we have computed the $MaxProb_{0...800}$ strategies, and compared them to the *OptExpected* strategy.

On the other hand, to be able to compute these strategies, we developed a caching algorithm for a class of recurrent functions. This is a result that has scope beyond the Yahtzee problem, and raises some further research questions.

7.2 Further research concerning Yahtzee

7.2.1 *MaxProb* advisor program

Using the generated table, it is possible to write an advisor program for the $MaxProb_{0...800}$ strategies, just as there is one now for *OptExpected*.

7.2.2 Expected scores for *MaxProb*

There is still a lot of analysis for Yahtzee to be done, for example concerning the *MaxProb* strategies. It would be of interest to determine the expected scores for these strategies, and to compare them to the results of the *OptExpected* strategy.

Using the software and data that is available now, it should not be difficult to compute the expected scores for the *MaxProb* strategies.

7.2.3 Comparing strategies

Comparing two strategies can be based on some special features, such as the expected score, but also on the structure of the strategy graph. Comparison requires criteria that define when two graphs are more or less alike. As a result, analysis of these graphs can only be conducted if there are clear criteria for *strategy graph likeness*. For any nontrivial definition of likeness, we expect comparison will require a considerable amount of computation[1].

[1] If *likeness* is defined recurrently, it is likely that the structure of the corresponding function will allow for the use of fixed subset partitioning.

7.3 Further research concerning fixed subset partitioning

The technique used to partition a domain for this class of recurrent functions leaves a number of questions for further research.

7.3.1 Applications

The partitioning technique might be of use in a number of areas. We could actively investigate which other problems have this kind of dependence structures, and apply this technique.

7.3.2 Generic function library

The Pascal programs that were created here, implements fixed subset partitioning for Yahtzee. The names that we used for constants, variables and procedures consistently refer to such things as Yahtzee categories.

Based on chapter 5 it should be possible to write a generic set of functions that implements this type of partitioning.

7.3.3 Load balancing

When partitioning is used to create a large number of classes, it is possible to generate a table using parallel computation. The class dependencies are well suited for this end. The only problem is caused by the varying sizes of the classes, making it more difficult to balance the load of the parallel nodes.

A possible solution to this problem lies in the formula for the class sizes. We saw that at most $2^{|G|}$ classes could be computed in parallel. If we have less than $2^{|G|}$ nodes, we try to distribute the classes such that each node has an equal amount of work to do. We believe this can be analytically solved on the basis of the class size formula, because of the properties of the binomium.

Appendix A

The game Yahtzee

Yahtzee is a registered trademark of the Milton Bradly Company.

Official rules for solitaire Yahtzee

The game Yahtzee comes is played with five dice and a score card.

Objective

The player rolls dice for scoring combinations, and tries to earn the highest total score.

Game summary

A game consists of 13 turns. In each turn, the player rolls the dice up to 3 times to get the highest scoring combination for one of the 13 categories.

After the player has finished rolling, a score or zero must be placed in one of the 13 category boxes on the score card.

The game ends when all category boxes have been filled in. All scores are totalled, including any bonus points.

Taking a turn

In a turn, the player may roll the dice up to 3 times, and may decide to stop and score after the first or second roll. After the first or second roll, the player may decide to "keep" any of the dice, rerolling only the other dice.

Scoring

When the player has finished rolling, he or she decides which box to fill in on the score card. If the player can't or doesn't want to enter a score, a zero is entered. Each box can be filled in only once.

The score card can be found in table A.1 and is divided into an upper and lower section. Scoring combinations can be found in table A.2.

Category		Score
Upper section	Aces*	...
	Twos*	...
	Threes*	...
	Fours*	...
	Fives*	...
	Sixes*	...
Upper section bonus		...
Lower section	Three of a Kind*	...
	Four of a Kind*	...
	Full House*	...
	Small Straight*	...
	Large Straight*	...
	Yahtzee*	...
	Chance*	...
Extra Yahtzee Bonus		...
Grand Total		...

*: Primary categories.

Table A.1: Yahtzee score card

Category	**Condition**	**Score**
Aces	–	sum 1s
Twos	–	sum 2s
Threes	–	sum 3s
Fours	–	sum 4s
Fives	–	sum 5s
Sixes	–	sum 6s
Upper section bonus	Upper section total ≥ 63	35 once
Three of a Kind	≥ 3 equals	sum values
Four of a Kind	≥ 4 equals	sum values
Full House	$2 + 3$ equals*	25
Small Straight	≥ 4 in sequence*	30
Large Straight	5 in sequence*	40
Yahtzee	5 equals	50
Chance	–	sum values
Extra Yahtzee bonus	5 equals and 50 at Yahtzee	100 each
Grand Total		sum above

*: 5 *y*s act here as *joker*, provided that the categories *y*s and Yahtzee have been scored already.

Table A.2: Yahtzee scoring rules

Appendix B

Beowulf cluster

In general, a Beowulf cluster is a kind of high-performance massively parallel computer built primarily out of commodity hardware components, running a free-software operating system like Linux or FreeBSD, interconnected by a private high-speed network. It consists of a cluster of PCs or workstations dedicated to running high-performance computing tasks. The nodes in the cluster don't sit on people's desks; they are dedicated to running cluster jobs. It is usually connected to the outside world through only a single node.

`pacluster.win.tue.nl` is a Beowulf-class Linux cluster, located at the Mathematics and Computing Science faculty of the Eindhoven University of Technology. It is used for research and education in parallel programming. It currently consists of 17 PC's connected through a 100 Mb/s fully switched ethernet network.

One of these machines, called the master or host machine, is connected to the Internet. The other 16 machines are the cluster nodes. All these machines are identical:

- Pentium III 533 MHz, EB version:
  - 133 MHz FSB
  - full speed on-die 256 kB cache
- 256 MB RAM
- 10 GB harddisk

The switch is a Cisco Catalyst 3524 XL, 100 Mbit 24-ports switch.

Bibliography

[1] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[2] T. Verhoeff. How to maximize your score in solitaire yahtzee.

[3] T. Verhoeff. Optimal solitaire yahtzee strategies, 2000. Sheets for presentation. A PDF file can be found at http://wwwpa.win.tue.nl/misc/yahtzee/slides-2up.pdf.

[4] T. Verhoeff and E.T.J. Scheffers. Solitaire yahtzee: Optimal player and proficiency test, 1999. Website at http://wwwpa.win.tue.nl/misc/yahtzee/.

[5] D. J. White. *Markov Decision Processes*. Wiley, 1993.