# Declarative programming: Cardguess project

Adam Juraszek

September 5, 2014

## 1 Overview of the main algorithm

Our Cardguess implementation follows the optimal strategy hinted in the project specification. The GameState stores list of all possible answers and also number of guesses which have been tried. The function initialGuess simply selects one of the precomputed first guesses and returns list of all subsets of a single deck of card of the given size. The function nextGuess first filters the list leaving only combinations which don't contradict with any previous pair of guess – feedback. Filtering is done by comparing the feedback from answerer with a feedback which would each of card combinations give with our last guess. When the list of options is shortened, we find the best option out of the remaining options.

### 1.1 Description of modules

The problem was separated into logical units which are easier to test and also easier to replace, which proved to be valuable. The division is as follows:

- Common – wraps Card and gives many types which are used in the rest of the solution. It also contains function which are useful for testing.
- Feedback – calculates answerer's feedback for given guess and answer. Although there are several functions, only one is exported.
- OptimalN – provides a Map which contains optimal second guesses based on previous feedback. These modules are mostly generated; we only fixed indentation and wrote module header.
- Optimalguess – provides three high-level functions related to lists of options and selecting the next guess. The rest of this text is mostly about development of this module.
- Cardguess – combines everyhing together in functions listed in specification.
- Main – a testing suit which can be used as a standalone interface to the game. It utilizes many 3$^{\text{rd}}$ party packages, but as it is not an essential part of the solution, it doesn't matter.

## 2 Selecting next guess

The best option for the next guess is to choose a card combination which gives us the most information. We measure the amount of information as a quadratic mean of sizes of groups of options which would give us the same feedback. The quadratic mean is

$$g_{avg} = \sqrt{\frac{\sum_i |g_i|^2}{\sum_i |g_i|}}$$

where $g_i$ is number of options which give us $i$-th feedback. Value of this mean is lowest when all $g_i$ are small and also for two sets of feedbacks groups it is smaller for set which is bigger. The smallest value is when all $g_i$ are the same and their number is biggest.

In our project it is only necessary to compare means of distribution of the same amount of feedbacks, therefore it is sufficient to just compare sums of squares: $\sum_i |g_i|^2$. There is an issue with range of integers; when a particularly bad option is chosen, its feedback distribution makes the mean overflow maximum of Int type. We solved this by computing means in Doubles rather than Integer, because it seems to be quicker and we don't mind errors in big means – they just cannot be the best choice.

In order to group the feedbacks effectively we put them all as keys into a Map with value 1 and combining function (+). That very conveniently counts sizes of groups of feedbacks. Calculation of mean is just folding the map into a single number. From list of all possible pairs of option and its mean (rating) is the best selected by another fold which is in this case much faster than library function minimum.

# 3 Precomputation

Haskell program could find the best initial guess for answers consisting of one and two cards very quickly. The computation for three cards took 3 hours but unfortunately we don't remember if the program was compiled with optimization enabled or run in ghci. Either way, it would still take too long to compute optimal first guess for four cards combinations.

We developed a program in C which is very similar to the Haskell one but runs much faster. We also cut the number of options to try to about one fifth by introduction of symmetry breaking constraints. Is is possible for the first guess; it is not interesting what the suits are as long as:

1. all used suits when numbered from zero are without gaps;
2. if a rank $r$ is used with suit $s$ then a rank $r' < r$ cannot be used for suit $s' > s$.

This way we could get the optimal first guess for four cards combinations in about 4 minutes. We also noticed that the biggest group after the initial guess contains 11844 options which means that in order to compute the rating we need to run feedback function $11844^2$ times.

Having the optimal first guess, we needed to make the second guess quicker. We tried to optimize work with Map (that gives us a log factor in time complexity) or optimize feedback function which is called more than a million times. Neither of these options could make the program run more than twice as quick; we were stuck at about 30 seconds per solution on average for four cards.

The next step involved precomputation of optimal second guesses. We could use our existing Haskell solution and just run part of it in a loop – for all feedbacks (we already know out first guess). The number of feedbacks in theory is up to $(n+1)^5$ where $n$ is number of cards to guess. But in practice the number is way lower, for 4-card answers it is only 284 of them (those are feedbacks which leave at least one card possible). We precomputed the optimal second guess for all possible feedbacks and saved it into a file which was than formatted to become a valid Haskell source file.

With all optimization and precomputationi applied, the guessing became much faster; for the main case – 2-card guessing game even breaking the limit. Guessing all 1326 possible 2-card answers took 11.9 seconds prior to optimization and 2.3 seconds after. The difference is more noticeable for 3-card guessing game: 201 minutes and 4 seconds prior to and 3 minutes and 8 seconds after the optimization. We didn't run all the games with 4 cards – just 1000 of them. They took 2 minutes and 17 seconds which is 0.13 seconds per one answer. Latter tests showed that this number is below average; it usually takes about 0.2 seconds which is still better than 10 seconds, which was the original desired goal.

# 4 Other versions

During the development several ambitious attempts were made to make the result better.

## 4.1 Deep optimal first guess

The idea was (in C) to find the best first guess not based on the distribution of feedbacks after the initial guess but rather on number of steps needed to solve all possible games. This approach worked for 1-card games and implemented and optimized found the optimal solution in 10 seconds. The solution surprisingly was not the same as the one based on distributions; the optimal should have been 7C or 9C in comparison to 8C which is in the middle. Because the number of steps to solve all instances of the 1-card game differed just by ones, we didn't use this solution.

The run time for 2-card game would be enormous if we let the computer work; we stopped computaton after 10 minutes. In that time not a single two card combination out of 1326 of them was rated.

## 4.2 Look ahead a few rounds

When we could finish any game in less than one second and we were left (after two precomputed guesses) with list of options which contained only hundreds of them, we could utilize the remaining time to find a better guess for the third and following rounds. We planned to introduce strategy which would switch the look-up based on number of remaining options, number of cards and current round. This ran in finite time but it was much harder to rate particular guesses in several levels; it was based on an ordinary MIN-MAX searching algorithm without cutting branches. It worked nice – it preferred a guess which would with several optimal next guesses distribute the feedbacks best. Unfortunately, in third guess even with a small look ahead (1 or 2) we got very often a prediction of a perfect distribution – all groups containing only one combination – the answers. It was a correct guess but not the optimal one because it tended to waste – it chose non-optimal guess which would still give optimal distribution in several rounds. The reason why we dropped this idea was, that we would need much better rating function than a distribution after several rounds; it would have to depend on intermediate distributions and count guesses for all options.

## 4.3 Testing suit

Because the response module given was compiled with a different version of GHC and ours didn't like it we had to develop our own testing program which is a wrapper around the Cardguess module. As a result we have a program which works in a read-eval-print loop. In the loop it reacts to several commands:

- i(nteractive) – followed by 0-4 cards in format RS where R is a rank and S is a suit; for example 9H is 9 of heart. This way the user can specify an answer. A result is a list of Guesser's responses – pairs of guess and GameState. The GameState part is printed as a number of remaining options to choose from.
- o(ptimal) – followed by a single number which should be between 0 and 4. This option calculates pair of feedback to initial guess and the best second guess formatted nearly as a Haskell source code. We used this to precompute the optimal second guesses to make the program run faster.
- r(andom) – followed by two numbers – the first one is a number of cards in an answer, and the other one is number of tries to test. Output is the same as for the option interactive. If you specify number of tries greater than number of all possibilities, only those will be tried.
- s(ample) – same input as for option random, but this just prints the card combinations which would be tried as answers.
- h(elp) – prints help with description of all options.
- q(uit) or nothing at all – quits the program.

# 5 Development tools

To develop this project we used Eclipse IDE with a plugin EclipseFP. This plugin needs several Haskell packages and it took us 3 days to install everything without conflicts.

The Haskell packaging manager is Cabal and comes with the latest GHC in version 1.18.3. The main package which is used to operate the manager is cabal-install which comes in version 1.20.0.1 with the Linux distribution. Building programs requires buildwrapper which needs to by built by the same version of Cabal and use the same version of Cabal library. The only way how to make it work without complaining on missing Cabal-1.20.0.0 nor Segmentation faults was to install older version of Cabal and cabal-install.

After that we had to install several other packages with a switch `--avoid-reinstalls` to not break the previous ones. One of the packages missed a dependency in cabal file and another was dependent on an imported type which doesn't exist. Some other packages couldn't be built, but using a different version worked. We supposed that all the packages are rebuilt periodically and any build problems are reported to maintainers and the package itself is flagged as broken. It doesn't seem to be this case in Cabal world.

During development of the project we also used several packages; most of them worked but some couldn't be built at all. We wanted to use a package serving as a C binding for readline, but this proved to be impossible. There is a bug in GHC which prevents building the package without changing the library file `/usr/lib/libncurses.so`. Link to the issue: `https://ghc.haskell.org/trac/ghc/ticket/9237`.

# 6 Conclusion

Our solution is not only nearly optimal in terms of number of guesses but also in running speed which is way below the specified time limit. The project is available online at `https://github.com/juriad/Cardguess`.

We have to admit that our solution is not optimal as we found a better solution for 1-card game which minimize number of guesses across all possible games. However, usage of this approach shown to be at least impractical and too slow for bigger instances of the problem. It could have been optimized by heuristics and cutting unpromissing search branches, for example by implementing A* algorithm.

We also proved to be untrue the hint claiming that for the first guess it is best to pick cards with ranks which divide the sequence 2-A into equal chunks. Much better strategy is to pick ranks which divide the sequence into three equal chunks with the remainig guessed cards being in the middle one. On the other hand, we couldn't prove this is the best one. For 2-card game we even found out that choosing the middle rank is suboptimal for strategy rating guesses by distribution of potencial feedbacks.