Project Steamed Bacon

Program Design and Data Structures (1DL201), 2014/2015

Group 30: Simon Lövgren, Erik Melander, Fredrik Svensson

March 5, 2015

Contents

1 Introduction

Steamed Bacon provides means to search for the shortest distance between two Steam users. It will search the Steam community for the user information and present it in an informative manner. In essence the program calculates the so called Kevin Bacon Index for Steam Users.

1.1 The Steam platform

Steam is an internet-based digital distribution-, multiplayer- and social networking platform created by a company called Valve. Steam provides the user with installation and updates of games and community features such as friends lists and groups as well as in-game voice and chat functions.
Steam was initially released for Windows in 2003, but support for both Mac and Linux was later added as well as support for mobile devices. Steam is one of the biggest online marketplaces for games and software with more than 4500 games in its library. It is also become a convenient way for more than 125 million active users to connect and play with friends [1].

1.2 Degree of separation game

The degrees of separation game has the objective to find the link (usually in six or less intermediary steps) between two persons or other occurrences[2]. A famous implementation of the game comes in the form of the Kevin Bacon Index, named after the actor Kevin Bacon, in which the objective of the game is to find actors that are connected to Kevin Bacon by way of other actors they have appeared in movies with.
We have decided to try to implement a version of this game using the friend lists in the Steam platform. The game takes two steamIDs and tries to find the shortest connection between them by searching the friends network of IDs of the first ID for the second. A limit was set for a maximum of three intermediaries in order to limit the runtime and to avoid the risk of running into the Steam API limit of 100 000 request per 24 hours.

2 User Cases

This section presents the required third party modules and explain how they can be installed, a description on how to run the program and what inputs can be used are presented.

2.1 Required Inputs

To operate the program the user needs two Steam IDs that he or she wishes to calculate distance between. The two IDs must be in the steamID64 format, sometimes referred to as community ID or friendID. To obtain this 17 digit ID number, the user needs to look it up on the steam community website by going to www.steamcommunity.com and there search for the desired username. When the correct user has been found and the web browser is on that profiles unique profile page, the steamID64 string is found as part of the profile page's URL.

```
Example: http:\\steamcommunity.com/profiles/76561197999847293
```

Here the steamID64 is the last 17 digits in the URL, 76561197999847293. Any format of IDs other than this 17 digit ID will not be accepted by the program and usage of any other format will return an error message.

Once the two IDs have been obtain the user is ready to use the program. The program will ask for both IDs, one by one.

There are several other web sites that offer alternative ways of finding SteamIDs. If some other type of steamID is known it is possible to convert it into the steamID64 format.

```
Example: http:\\ www.steamidconverter.com
```

2.2 Running the program

In order to run the program, type ghci main.hs in the console and run the main function. To run the main function, type main. User will now be asked to input the first Steam ID and the second Steam ID.

Example IDs: 76561198028357851, 76561197999847293 .

This will return the following response:

```
76561198028357851 has depth 1
76561197979971024 has depth 2
Id found, 1 id between them.
From Erikun to Maustronaut to Avari
```

A simplified function for running all test cases in the program has been created and can be used by calling `runAllTestsProject` from the `main.hs` file. Tests from each module will be run and in between each test result, a heading will be printed describing what set of test cases will be run.

2.3 Interpretation of the output

Once the program has been given the the IDs needed for the calculations, it will start processing them. The progress will be shown in real time as output in the terminal. The text printed to the console shows which Steam ID the program is currently evaluating and the current depth. Once the processing is done, the program will display the link from start to finish with the profiles' nicknames, not the steamID64, if said link has been found. However if a link cannot be found between the two IDs inputted by the user, the program will tell the user this by printing the message: *"Didn't find any match before reaching the end of the network or too many degrees of separation"*.
The program has been limited to a depth of 5 steps due to performance. Because of this it is possible that a link will not be found even though it might exist, only that it would require further steps of evaluation in order to be confirmed.

2.4 Requirements

This section describes requirements needed to run the program.

2.4.1 Internet connection

The program uses resources located on the internet and thus requires an internet connection.

2.4.2 Third party JSON library

The program uses a third party haskell library called `Text.json` for parsing data retrieved from Steam, so if using a non-compiled version of the program it needs to be installed. This library is easily installed via terminal, or the command prompt if on Windows, using the package installer that comes with the haskell platform, called `cabal`. For all listed commands below, use terminal for linux/mac and the command prompt for windows.

To install for the current user only, use the following command:

```
cabal install json
```

To install Text.json as a global package (and thereby accessible by all users on the computer), use:

```
cabal install json --global
```

If an error occurs during installation, you may need to rebuild the local package cache using the following command:

```
ghc-pkg recache
```

then try running `cabal install json` again.

3 Technical description

This part of the report provides a technical explanation of the project, how the different parts of the program work both individually and with each other. This section will also try to explain why parts of the code has been implemented the way it has, where an explanation may be needed.

3.1 Overview

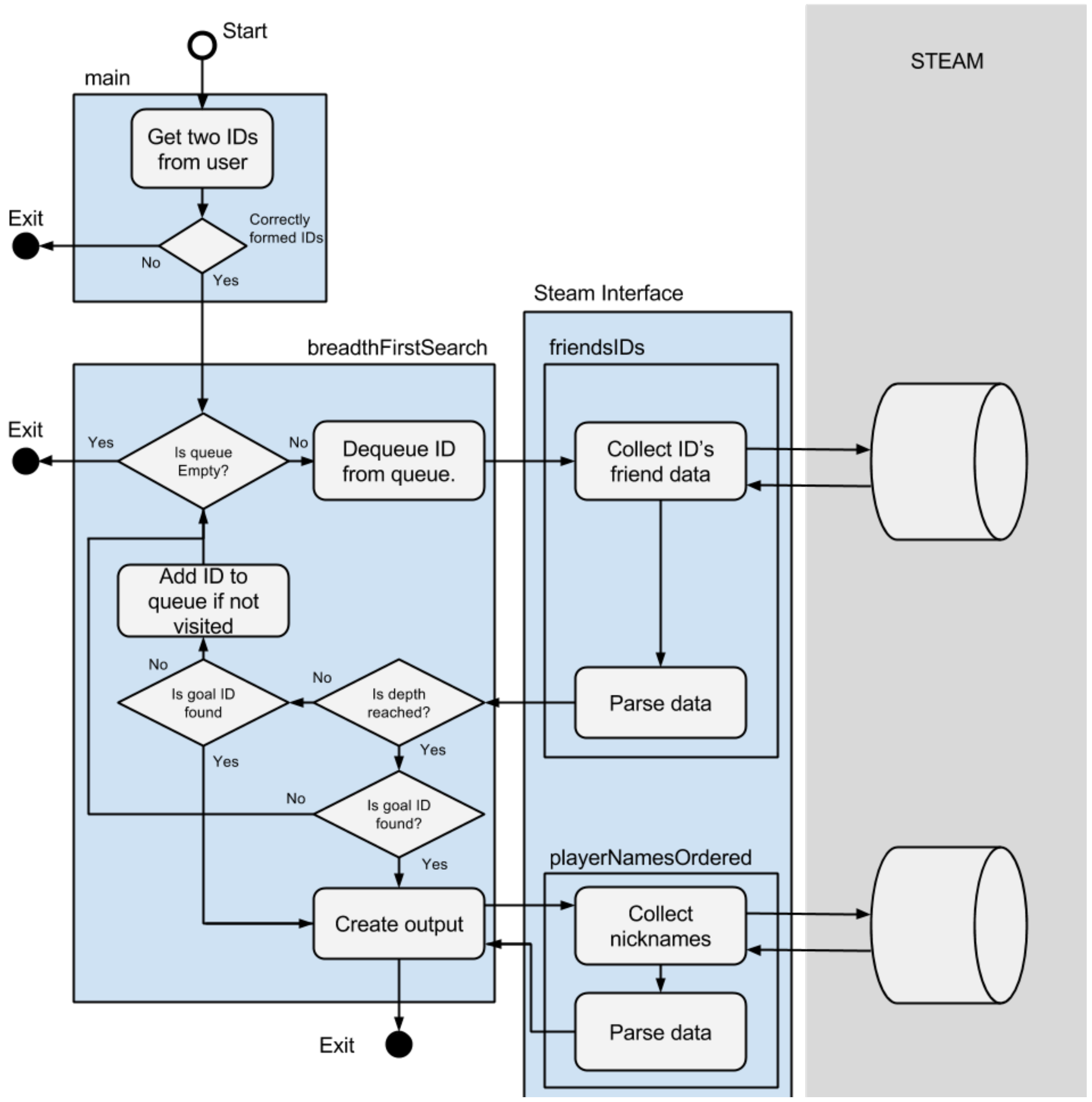

Fig. 1
Schematic overview of Steamed Bacon program.

The program is divided into three general parts.

- `main` - A main function which received input from the user.
- `breadthFirstSearch` - The Graph algorithm.
- the Steam Interface - Functions for data collection and parsing.

The main function requests input from the user in the form of two Steam IDs and checks that they are correctly formed. The graph algorithm uses the *Breadth First Search (BFS)*

algorithm to search through the network of the first Steam ID for the second Steam ID. Our interface towards Steam handles data collection and parsing of the Steam data. The `friendsIDs` function returns a list of the friends of the current ID the graph algorithm is processing. And the `playerNamesOrdered` takes a list of IDs constituting the shortest path between the two initial IDs and collects the nicknames of these in order to provide a more pleasing output.
The program will exit for three possible reasons: incorrectly formed IDs from the user; the second ID was not found in the network , which includes if the *BFS* finished searching IDs with a depth of five; or the second ID was found in the network.

3.2 Main

Takes input from the user in the form of two Steam IDs and checks that they are correctly formed 17 character long numerical strings. It converts them to Integer and calls the `breadthFirstSearch` function.

To run the program, simply call the `main` function.

3.3 Breadth First Search Module

This section describes the functionality of the *BFS* Module and the datatypes and functions it relies on. The `BFS` algorithm has the following steps[3]:

Input: some node A

1. Paint A gray. Paint other nodes white. Add A to the initially empty FIFO queue of gray nodes.
2. Dequeue head node A. Paint its undiscovered (white) adjacent nodes gray and enqueue them. Paint A black. Repeat until queue is empty.

The Steamed Bacon implementation follows this fairly closely with some minor modifications. Step 1 of the above *BFS* description is performed in the main function and the rest in the *BFS* Module.

3.3.1 Breadth First Search

- `breadthFirstSearch :: SimpleQueue ((Integer,[Integer])) → Integer → Tree Integer → IO ()`

The *BFS* function parameters are a queue of the IDs as a tuple where the first value is the ID as an Integer and the second is a list of Integers representing the IDs that are the shortest route from the original ID. The integer that follows is the ID that is searched for. The tree is the list of already visited vertices (referred to as nodes painted grey in the above description).

- The `breadthFirstSearch` function checks if the queue is empty and exits if it is.
- It dequeues the next ID to investigate.
- Calls `friendsIDs` to get a list of IDs connected to it.
- Calls the `checkAndAdd` function.
  - `checkAndAdd` checks if goal ID is found in the list.
  - Adds IDs not visited (white nodes per above) to queue and list of visited vertices.

  - If the termination depth has been reached it does not add vertices to queue or list of visited.

- `breadthFirstSearch` repeats until goal ID is found or queue is empty.
  - If goal was reached it calls the `successRoute` function
  - `successRoute` calls `playerNamesOrdered` to get a list of nicknames associated with the IDs in the shortest path
  - Creates an output string with the result and prints it to the console.

3.3.2 Queue

The breadth first search algorithm relies on a *First In First Out (FIFO)* queue to organise the order of the vertices that are to be investigated. The *FIFO* queue data type has the invariant that the order in which elements in the queue have been added is the order in which they must be removed. The simplest functional way of implementing a *FIFO* queue is to use two lists [4], one from which values are removed (dequeued), from here on referred to as the front list, and one in which they are added (queued), the rear list. If a dequeue operation is made and the front list is empty, a balancing function reverses the elements in the rear list and adds them to the front list before dequeuing the first element in it.

The queue datatype is:

```
data SimpleQueue a = EmptyQ | SQ [a] [a] deriving (Show, Eq)
```

The module has the following functions:

- `createQueue :: SimpleQueue a`
  Creates an empty queue (function not used in Steamed Bacon)
- `queue :: a → SimpleQueue a → SimpleQueue a`
  Queues a value to the "end" of the queue.
- `deQueue :: SimpleQueue a → (a, SimpleQueue a)`
  Dequeues a value from the "front" of the queue
- `shiftQueue :: SimpleQueue a → SimpleQueue a`
  `deQueue` helper function that rebalances the lists in the queue datatype to ensure *FIFO* functionality.
- `isQueueEmpty :: SimpleQueue a → Bool`
  Checks if the queue is empty.

3.3.3 Binary Search Tree

This modules purpose is to receive a Tree Integer, which is a Binary Tree with Integer IDs stored in it, and a single ID. It will return a Boolean and a tree.

- `searchTuple :: Tree Integer -> Integer -> (Tree Integer, Bool)`
  This is the top function in this module. It takes a Tree and an ID as inputs and returns a tuple with a Tree and a bool.
- `searchIDTree :: Tree Integer -> Integer -> Bool`
  This function searches a tree for a possible match with a supplied ID, returns a Boolean.

- `isInIDSearchTree :: Tree Integer -> Integer -> Tree Integer`
  This function will simply search a Tree with a supplied ID and use its Aux function to establish if a match can be found or not. If a match is found it will initiate a insertion through `insertIDSearchTree`. Otherwise it returns tree it was supplied, that tree is now the current most updated tree.
- `isInIDSearchTreeAux :: Tree Integer -> Integer -> Bool`
  This function searches for ID matches for `isInIDSearchTree`.
- `insertIDSearchTree :: Tree Integer -> Integer -> Tree Integer`
  This function will insert an ID to the tree in its correct place.

The purpose of the IDSearchTree module is to determine if an ID has been spotted before or if it is a new never before seen ID. The way this is handled is by structuring IDs in a Binary Tree. This is done in two steps. First `searchIDTree` looks through the existing tree to determine if the provided ID is in the tree already or if an insertion is needed. If an ID is not found it will go to step two and initiate a insertion and an additional tree search will be done to determine the correct position for the new ID in the Binary Tree. When the insertion is complete a tuple with the newly updated tree and a Boolean; `False` will be returned as a tuple.
However if the ID is found during this search the current tree and a Boolean; `True` will be returned as a tuple.

data Tree
The data structure for the Tree structure used is a simple Binary Tree:

```
data Tree a = Empty | Branch a (Tree a) (Tree a)
```

The tree can either be `Empty`, e.g the tree or branch contains no information, or it can contain information in the node and two sub trees of type Tree. In the code these subtrees are referred to as left and right, these sub trees can of course be either `Empty` or Branches with information in their respective nodes.

3.4 KeyVal

The Steam API contains a lot of information regarding a Steam account. This information needs to be stored in a convenient way for the program to be able to pass it along down the line of functions. One could store it as a tuple but since the second part that is tied to the ID can be of either Type `Integer` or `String`. Therefore it is crucial that a data structure is implemented so that the code works regardless if it gets passed an `Integer` or a `String`.
The KeyVal data type solves this by enabling data to be stored as either `(KVStr String String)` or as `(KVInt String Integer)`.

```
data KeyVal = KVStr String String | KVInt String Integer
```

The `KeyVal` data type will most often be found in lists of `KeyVal`, referred to KeyVal-list.

- `findKVString :: [KeyVal] -> String -> Maybe String`
  This function is used to find a `String` that is paired with the supplied `String`.
- `findKVInt :: [KeyVal] -> String -> Maybe Integer`
  This function is used to find a `Integer` that is paired with the supplied `String`.

When data for a specific user ID is requested the KeyVal-lists and ID will be sent to either `findKVString` or `findKVInt` depending on what the expected return data is. Depending on if the program is expecting an `Integer` result or an `String` these function will look for the provided ID, represented as the first element in the data structure and return the information stored as the second element, either `Just String` or `Just Integer`. If an the given ID is not represented in the KeyVal-list it will find no relevant information and return `Nothing`.

3.5 Data acquisition

The data acquisition part of the program is responsible for requesting data from the *Steam Web API* as well as parsing the response into a workable data type/data format.

3.5.1 Steam API

Steam data is acquired using the publicly available *Steam Web API*[5]. The API contain multiple endpoints, each with their own purpose, which is accessible once a free API key is created using a valid Steam account. The program utilises only two of these endpoints as it only requires information regarding friends of a user and the summary of a user (mainly for accessing the nickname). These endpoints used are `GetFriendList` and `GetPlayerSummaries`.

Privacy settings
The first problem that may occur with the data acquisition is the privacy level of the steam accounts. Steam allows its users to select their level of privacy in three categories, *profile status*, *comment permissions* and *inventory.* The available privacy levels are *Public* (viewable by anyone), *friends only* (viewable by friends only) and *private* (only the user can view his/her info).

For the program to be able to retrieve data about a steam account, it needs to have their profile status privacy settings set to *public*, otherwise no data is returned for that account. For example, if the friends list of an account is requested via the API and the account has 10 friends in the friend list, but say 4 of the accounts are set as *friends only* or *private*, then the API returns a list of 6 friends as they are publicly viewable. As we cannot know the amount of friends linked to an account before the request to the API, this is almost impossible to catch and therefore simply ignored in the program.

3.5.2 Parsing

As mentioned in the user case, the program uses a third party JSON library, called `Text.JSON`, to parse the raw JSON data returned by the *Steam Web API.* To parse JSON data using this library, custom data types must be created in accordance of the structure in the JSON data that is to be parsed. An instance of the JSON data class must then be created for each of the custom data types.

Custom data types
The custom data types created in accordance with the JSON data are created as record type data types. This way of creating data types in haskell automatically creates functions that allow for fetching specific parts from the data type. For example, in a data type of the following structure:

```
data MyRecord = Record {
    name :: String,
    age  :: Int
    } deriving (Show)
```

we can easily extract "name" simply by writing:

```
name person
```

where `person` is of type `MyRecord`. These custom data types could be recursive or, as in the program, contain other custom data types.

Parsing the data

The library supplies a function called `decode`, which invokes the parsing functionality. This, in turn, invokes a function defined in the instances of the JSON data class on the data types it parses the data into. The function of the instance is called `readJSON` and defines how the data should be passed to the data type[6].

This is where it gets a bit tricky, since *Applicatives* will be used to correctly parse the data into the custom data types in combination with a function called `valFromObj` supplied by the JSON library. Firstly, a simplification of functions is defined to more easily pass the data correctly. This function is created as an infix operator which simply takes the `valFromObj` function, the passed object and search term and flips them correctly as to allow for chaining the applicative functors.

```
(.:) :: (JSON a) => JSObject JSValue -> String -> Result a
(.:) = flip valFromObj
```

The JSON object is then passed as an applicative using the above explained helper function and the infix functions `<$>` and `<*>` found in `Control.Applicative`.

An example instance of the data type Player (which is used for parsing player summaries, explained later in this document) would be:

```
instance JSON Player where
    readJSON (JSObject obj) =
        Player                  <$>
        obj .: "steamid"        <*>
        obj .: "personaname"    <*>
        obj .: "lastlogoff"     <*>
        obj .: "profileurl"     <*>
        obj .: "avatar"         <*>
        obj .: "avatarmedium"   <*>
        obj .: "avatarfull"

    readJSON _ = mzero   -- Base case / Fallback
```

Custom parsing

In addition to the JSON parsing library, a set of custom functions have been created as to convert the, from `Text.JSON` returned, Response data type to the custom KeyVal data type. These functions reside in the modules in which they are used (`SteamAPI.Friends` and `SteamAPI.Summaries`), as they are created for specific data types and responses. These modules will be explained later in the report.

3.5.3 Structure

The data acquisition part of the program is built as a set of modules accessible through an abstraction layer. This abstraction layer is located in the file `steam.hs` as the module `Steam`. The abstraction layer contains functions that work as proxies to the underlying modules for ease of use as well as assuring no clashing functions or data types.

The modules underneath the abstraction layer reside in a sub folder called *SteamAPI* and are named `SteamAPI.Requests`, `SteamAPI.Friends` and `SteamAPI.Summaries`.

Steam (Abstraction layer)
The abstraction layer, as mentioned, is created for ease of use against the data collection part of the program, allowing for a single include to access the full `SteamAPI` collection of modules. The available functions of the abstraction layer are:

Custom data types / aliases
The Steam module, as all of the SteamAPI modules, use the custom types (aliases) `SteamID` and `AppID` for Integer, as to clarify what data is supposed to be supplied to the its functions.

Important functions:

- `friends :: SteamID → IO [[KeyVal]]`
  This function takes a single steam ID as argument. An IO list of KeyVal-lists is then returned representing the friends of the steam user.
- `friendsIDs :: SteamID → IO [Integer]`
  This function takes a single steam ID as argument. An IO list of Integers is then returned, containing only the ID:s of the friends of the steam user.
- `playersExist :: [SteamID] → IO Bool`
  This function takes a list of maximum 100 steam ID:s as argument and returns an IO Bool. If all steam users could be fetched it returns IO True, otherwise it returns IO False. If the response is IO False and the user exists, it's an indication that the user has chosen to keep their profile private and thereby not allowing access to their information.
- `playerSummaries :: [SteamID] → IO [[KeyVal]]`
  This function takes a list of maximum 100 steam ID:s as argument. An IO list of KeyVal-lists is then returned representing the summaries of the users whose ID:s were supplied as argument. **However**, the data is not necessarily returned in the same order as the supplied list of ID:s due to the response from the *Steam Web API.*
- `playerNamesOrdered :: [SteamID] → IO [Maybe String]`
  This function takes list of maximum 100 steam ID:s as argument. An IO list of `Maybe String` is then returned representing the the nickname (or "personaname") of the users whose ID:s were supplied as argument. The list is ordered according to the supplied list of ID:s.

SteamAPI.Requests
The `Requests` module is the lowest level module in the *SteamAPI*, as it is invoked by all other modules. It utilises the `Network.HTTP` module and the `Data.List` module that comes with the haskell platform.

It houses the functionality for creating- and performing the GET-requests to the *Steam Web API* as well as a predetermined API key. The module has been fitted with most available API-calls, even though only two are used, as to enable further development of the program.

Custom data types / aliases
`SteamID` and `AppID` as mentioned in *Steam (Abstraction Layer).*

Important functions:

- `get :: String → IO String`
  Takes a string containing a URL as argument. It then passes the URL to the `Network.HTTP` module and retrieves the response body as an IO String, which it then returns.

- `concatIDs :: [Integer] → String`
  Simple helper function that takes a list of Integers as argument. It recurses over the list, converting the integers to strings and concatenates them into a single string with commas separating the ID:s.

- `getFriendList :: SteamID → IO String`
  Takes a SteamID as argument. Builds URL to `GetFriendList` endpoint by concatenating `apiBase`, endpoint URL segment, api `key` and supplied `SteamID` and passes URL to the `get` function. Returns data from `get` without modifications.

- `getPlayerSummaries :: [SteamID] → IO String`
  Takes a list of SteamID:s as argument. Builds comma separated of ID:s as string by sending list of ID:s to `concatIDs`. Builds URL to `GetPlayerSummaries` endpoint by concatenating `apiBase`, endpoint URL segment, api `key` and concatenated ID list and passes URL to the `get` function. Returns data from `get` without modifications.

SteamAPI.Friends
The `Friends` module contains functions- and data types specific to requests that use the `GetFriendsList`-endpoint of the *Steam Web API* and it utilises the `Control.Applicative`, `Control.Monad`, `Text.JSON`, `KeyVal` and `SteamAPI.Requests` modules.

Custom data types / aliases
`SteamID` and `AppID` as mentioned in *Steam (Abstraction Layer).* In addition there are custom data types according to the JSON structure returned by Steam. These are created to represent the data structure accurately, thereby prompting the use of record type data types since the JSON data contains objects with named variables. These custom data types are:

```
data ResponseWrapper =
    ResponseWrapper {
        response :: Players
    } deriving (Show)

data Players =
    Players {
        players :: [Player]
    } deriving (Show)
```

```
data Player =
        Player {
        steamid :: String,      -- SteamID is returned as string by steam API
        personaname :: String,
        lastlogoff :: Integer,
        profileurl :: String,
        avatar :: String,
        avatarmedium :: String,
        avatarfull :: String
    } deriving (Show)
```

Important functions:

- `getIDList :: SteamID → IO [Integer]`
  Takes a single SteamID as argument which it sends to the `Request` module. The response is then sent to parsing and the parsed data sent to `makeListOfIDs`. This result is then returned.
- `getRawList :: SteamID → IO [[KeyVal]]`
  Takes a single SteamID as argument which it sends to the `Request` module. The response is then sent to parsing and the parsed data sent to `extractFriends`. This result is then returned.
- `makeListOfIDs :: Result FriendsList → [SteamID]`
  Takes a JSON parsing result, checks that it is valid and and extracts ID:s of all friends.
- `extractFriends :: Result FriendsList → [[KeyVal]]`
  Takes a JSON parsing result, checks that it is valid and and extracts all friends into a list of KeyVal-list.

SteamAPI.Summaries
The `Summaries` module contains functions- and data types specific to requests that use the `GetPlayerSummaries`-endpoint of the *Steam Web API* and it utilises the `Control.Applicative`, `Control.Monad`, `Text.JSON`, `KeyVal` and `SteamAPI.Requests` modules.

Custom data types / aliases
`SteamID` and `AppID` as mentioned in *Steam (Abstraction Layer).* In addition there are custom data types according to the JSON structure returned by Steam. These are created to represent the data structure accurately, thereby prompting the use of record type data types since the JSON data contains objects with named variables. These custom data types are:

```
data FriendsList =
    FriendsList {
        friendslist :: Friends
    } deriving (Show)

data Friends =
    Friends {
        friends :: [Friend]
    } deriving (Show)
```

```
data Friend =
        Friend {
        steamid :: String,      -- SteamID is returned as string by steam API
        relationship :: String,
        friend_since :: Integer
    } deriving (Show)
```

Important functions:

- `extractList :: Result ResponseWrapper → [Player]]`
  Takes a parsed JSON result and extracts the list of players from the custom data type, then returns said list.

- `playersExist :: [SteamID] → IO Bool`
  Checks whether all supplied Steam ID:s resolve to an existing, publicly available, account on Steam's servers. This is done by retrieving the summaries of all supplied ID:s then checking whether the number of summaries match the number of supplied ID:s.

- `playerList :: Result ResponseWrapper → [[KeyVal]]`
  Takes a parsed JSON result from which the list of summaries is extracted using `extractList`, then extracts the summary data into a list of KeyVal-lists.

- `orderedNames :: Result ResponseWrapper → [SteamID] → [Maybe String]`
  Takes a parsed JSON result and a list of Steam ID:s. It then recurses over the ID list and extracts the nickname (or personaname) of each ID, using `nameOflayer`, in order. This is due to that the *Steam Web API* does not always return the results in the same order as requested.

- `nameOfPlayer :: SteamID → [[KeyVal]] → Maybe String`
  Takes a single Steam ID and a list of KeyVal-lists containing player summaries. It then extracts- and returns the nickname (or personaname) from the KeyVal-list with matching Steam ID. Since there may not be a matching Steam ID, it returns the nickname as a Maybe String.

- `getPlayerList :: [SteamID] → IO [[KeyVal]]`
  Takes a list of Steam ID:s as argument, which it then sends to `Request` module. The response is then parsed and sent to `playerList`. The result is then returned.

- `getOrderedNames :: [SteamID] → IO [Maybe String]`
  Takes a list of Steam ID:s as argument, which it then sends to `Request` module. The response is then parsed and sent to `orderedNames` along with the original list of SteamID:s. The result is then returned.

4 Discussion

The time it takes for the program to run is 0.5s per ID that is investigated. Possible optimisations are discussed.

4.1 Optimising the code

Several improvements could be made to make the program run faster such as optimising the *BFS* functions and the choice of using a Binary Search Tree for storing visited vertices.

4.1.1 Breadth First Search

Every ID investigated by the program takes 0.5s to process. This time is split fairly evenly between the network call to Steam and the BFS functions. Graph functions usually have adjacency list or adjacency matrix representation of the data which is then stored in an array(ref). We instead decided to use a binary search tree as it is a data structure we have experience using.

4.1.2 Binary Search Tree

The code is somewhat inefficient due to the fact that it sometimes needs to go through the binary tree twice. A better way of doing this would be to let a insert function do both the search for- and insert of IDs, however this creates a problem with returning a complete tree. The problem is that whenever the search algorithm works its way down the branches of the tree, it will discard all elements that are eliminated from possible candidates. When this is done and the function is asked to return the tree and a Boolean, it will only return what is left of the tree and ignore everything that has been cut out. The way, implemented in the program, to ensure that a complete tree is returned is to first search for the ID in order to establish if or if not an insertion is needed and if an insertion is needed, it will go through a regular insertion process that is returned with the Boolean. If no insertion is needed, the tree is returned without additional calculations.

This, however, means that the function will recurse over the tree twice when a new ID is to be added and only once if the ID is already present in the tree.
Using an array would probably be a different way to improve the time to check if IDs had already been visited.

4.1.3 Data acquisition

Since all acquired data is fetched and directly returned, it's not stored on disk. This means that every request for a specific ID will cause a new request to the Steam API, which takes a long time compared with reading a local file from disk.

An improvement (or optimisation) of this would be to cache all data collected from the Steam API locally so that when the same ID is requested a second time, it simply reads- and returns the locally cached data. A timed refresh rate would accompany the local storage as to refresh the steam data every now and then.

4.2 Further development

The program could eventually be expanded to include the information and statistics that we initially planned to implement. It could also contain a more user friendly interface. No specific data type associated with graph algorithms were used such as an adjacency list, which is a list of vertices connected to a specific vertex, even though the data received from Steam is a list of IDs and easily could have been put in this format. Storing the predecessor and distance from the source vertex instead of a list of vertices would have been a more general implementation of the algorithm and could have been useful in case the mapped network needed to be used again. Since the data is not stored in any persistent way, the function needs to map the network, including calling Steam, even if the origin ID of subsequent searches is the same. Caching the results of the first search would cut down on the time used in all cases except when the searched for ID was discovered early in the preceding search.

4.3 Scope of the project

The goals partly changed during the project, primarily in order to adjust the difficulty and results of the project to the goals of the course. Even though the general aim of using the Steam API for the project stayed more or less the same the use of it was changed. Since the program fetches information from a web API and needs this information parsed into Haskell syntax a suitable parser was needed. Initially a decision between using a third party parser or create one was considered, but after some research the time and difficulty of creating a custom parser was considered too great and a third party module was used.

The received information that got through the JSON parsing was initially meant to be used with different Haskell function to get statistics and other fun information regarding users. Several functions were made to enabled the user to search for the most compatible steam friend, compare hours spent playing and other quite simple haskell comparisons. The aim of the project was instead shifted to creating a Kevin Bacon Index calculator for Steam users. It was a more interesting problem to solve.

In the end the project turned out to be quite successful and it contained a balanced, as well as challenging, level of problems that needed to be solved. The workflow, as always, is subject to improvement but overall the work went smooth and all members of the group did their best to complement each other. Communication worked well and all group members showed up on time to meetings.

5 Summary

We created a degree of separation game for the Steam community that calculates the number of users connecting two Steam IDs. The program consists of three parts, a main function for user input, a breadth first search algorithm and a data collection module that retrieve JSON data from the Steam web API which use a third party JSON parser to prepare the data for the graph algorithm. The possibility of writing a JSON parser was considered in the beginning of the project but abandoned after it was determined to require too much time.

Further improvements could include using an array instead of a binary search tree in the graph algorithm, caching the resulting networks to improve search times on already mapped networks and more types of information that could be collected from the Steam Web API and presented.

References

[1] (2015) Steam community. Wikipedia. [Accessed 5-March-2015]. [Online]. Available: http://en.wikipedia.org/wiki/Steam_(software)

[2] (2015) Six degrees of separation. Wikipedia. [Accessed 5-March-2015]. [Online]. Available: http://en.wikipedia.org/wiki/Six_degrees_of_separation

[3] D. Clark, "Algorithms: Elementary graph algorithms," Lecture notes.

[4] C. Okasaki, "Simple and efficient purely functional queues and deques," *J. Functional Programming 5(4)*, pp. 583–592, October 1995. [Online]. Available: http://www.westpoint.edu/eecs/SiteAssets/SitePages/Faculty%20Publication%20Documents/Okasaki/jfp95queue.pdf

[5] Steam API. [Accessed 5-March-2015]. [Online]. Available: https://developer.valvesoftware.com/wiki/Steam_Web_API

[6] Stack Overflow. [Accessed 5-March-2015]. [Online]. Available: http://stackoverflow.com/questions/17844223/json-parsing-in-haskell