Coursework Report

Jaroslav Belkov
40187218@napier.ac.uk
Edinburgh Napier University - Algorithms and Data Structures (SETSET09117)

1 Abstract

The following article describes the problem, design and implementation of the solution to an automatic draughts playing computer program. It consists of Authors Word and a report. The report consists of introduction, use of appropriate data structures and algorithms to implement the solution, critical and personal evaluations.

Keywords – Draughts, Checkers, Dama, Python, Python 3, Computer, Game

2 Authors Word

I greatly enjoyed working on this project and hope that I have met all requirements stated in the coursework task. However, I feel that some parts of the program could be written better.

The general task was to implement a draughts game in a programming language of my choice. It also was recommended to use Java or Python languages and suggested the use of a language that I never used before, to take the opportunity and learn a new programming language. I accepted the challenge and decided to write the solution in Python as my very first program in this language, and I am very glad I have done so. However, there are some aspects, that could have been better, if I was not a beginner to the Python. Specifically, I do not like the fact that almost all source code is written in one document: it makes it untidy, and hard to follow the program flow. That is because I did not fully understand the way Python treats variables to be global or local until a late stage of development. If I wrote it again, most of the functions would be nicely imported from their own module, so the main module would be nice and tidy, clean piece of code.

Also, it is worth to mention that no high-level programming language, interpreted or compiled, is the best choice when it comes down to writing a solution to the high performance demanding problem, such as an automatic play of the draughts game (i.e. AI player). As a former assembly language programmer, I tend to use programming techniques and algorithms which can be directly implemented in machine code. Of course, the main point of the existence of high-level languages is to make coding simpler, so I looked after myself and tried my best to use built-in high-level data structures, often in the expense of performance. However, I feel I might have overacted in that, and at least one aspect of the solution should have been written using Boolean algebra and basic logical operations instead of a comparison of integers and strings: that is the data structure used to record the current position (a variable main-position, mentioned below in the report).

3 Introduction

This report is written to complement the solution to the assigned task in coursework specification for SET09117. The solution for the game comprehends all features required in the task and some more, and as an add-on is provided with a graphical user interface. The truth is that the GUI in the first instance was programmed as a testing tool during the development phase, and before the submission, the code was cleaned, but it is still not the best illustration of what the HTML, CSS and JavaScript code should look like. Also, the method used to read a file from the file system on the hosting machine is quite not the best programming practice in the field of web development: if these pages were part of a site on a server, this would not be possible to implement.

Required features that are given directly by the assignment, and features identified as necessary to run the game by the author:

1. The program should be robust and easy to install. It must run on a working machine in JKCC.
2. Represent game board and figures on the board in their positions as a text-based user interface so the game can be played from the command line.
3. The user interface should be intuitive to use and comply with the Human-Computer Interaction good practice as much as it is possible on the command line.
4. Allow the user to input his choice of a move. The subfeature is: Present to the user possible legal moves at any game position.
5. Allow the user to choose the game mode: Two people, or a person versus computer, or computer versus computer.
6. Record the game and allow the user to undo and subsequently redo their moves.
7. Allow the user to automatically replay the game from the first move to the last move at any point of the game, including the last position of figures in a finished game.
8. Implement an algorithm that enables the computer to choose a move from all possible legal moves in a given

position, so it imitates an effective player; at least better than a random move picker.

Features added by the author to make the game more playable and the program more robust:

1. Game record is also written to the backup file, so it can be recovered if necessary.
2. . The user can save the game at any point.
3. . The user can load a previously saved game from a file.
4. The speed of automatic replay is adjustable.
5. The game can also be replayed move by move pressing ‘ENTER‘ key if set so.
6. When only one move is possible, it can be set to be played automatically (not waiting for the user‘s input)
7. The user can set difficulty level for the automatic player (further in this report also referred to as AI for Artificial Intelligence)
8. The user can save his settings, so they are retrieved in the next session.
9. The user can resign from his game. Can offer to his opponent to tie the game or confirm his opponent‘s offer. Get suggestion on the move using AI‘s calculation, or let AI finish the game for him. And also, the user can quit the game at any time.

Features related to the GUI:

1. The user can set whether to use the GUI or not.
2. The user can set the start position of the game, including switching the first move to have a white player.

4 Rules of the game:

There are many variations of the game draughts or checkers (American English) in the world. This program, named Pythonic Draughts (addressed as PD further in this report), allows the user to play the game of English Draughts described on Wikipedia[**?**]. The article on Wikipedia is pretty complex; the section Rules consists of an exhaustive explanation of the rules. In accordance to these rules, PD determines the legal moves in any given position of pieces on the board and only allows players to make these moves (i.e. illegal moves are not offered in the game menu, therefore cannot be done).

5 Design:

When the program starts, it presents the Main Menu to the user and current settings below it. The user can start the game by choosing a game mode, can go to Settings Menu, load previously saved game or terminate PD. When the game is started, after the welcome phrase, the user is present with the board. The program was designed to run on the command line using text-based interface. That says that there is no place for any advanced graphics. On top of that, PC‘s in JKCC apparently are not configured to encode UTF-8, so originally much more pretty board‘s graphics were discarded and instead are used following symbols: lower letter ‘x‘ for red men, uppercase ‘R‘ for red kings, lower ‘o‘ for white men and upper ‘W‘ for white kings. For an empty reachable field the character dot ‘.‘ is used.

To make it easier to read the position and what moves are available, the chess notation is implemented. Possible moves (moves to choose from) are presented to the user as a numbered list, and the user should choose the move by entering the corresponding number. As the last line of the list is presented the Game Menu, which in addition to the other options offers the opportunity to quit back to the Main Menu.

If the Settings Menu is opened from the Main Menu, the user can change Settings, can save them (so the next time he runs PD settings are preserved) or quit back to the Main Menu without saving (or changing) at any time. This interface structure ensures that the user can go back to the Main Menu at any time, where he can save the game or replay it, if at least once the game started. And also to terminate the program.

6 Algorithms and Data structures used to implement the solution:

The source code for Python 3.6.2 interpreter is written in the file pythonic-draughts.py, which is supplemented with the file pd-functions.py. In pd-functions.py are functions, which may be adjusted in order to play another variation of the checkers game. Of course, it requires a good understanding of the program PD. The program starts with the initialization of main-path global variable at the very first line of the code. This is explained below.

Then the program continues by setting the ‘try-catch‘ block which wraps the whole code and catches any generic exception apart from system exceptions (such as ‘access denied‘). This is in order to provide console users with the opportunity to understand what happened before the console terminates in the case of an exception.

Installation check .
The program flow continues by importing necessary modules from standard library and makes the installation check. This feature is to improve the robustness of the program and help the user to cope with the first execution after the installation (point one of the required features). There are many variants on how the program can be run: in a Python shell, directly in the Python console using the launcher, in the operating system‘s command prompt - launching Python from the command line. They all have different behaviour when it comes down to the determination of the directory where the PD is installed. For instance, the IDLE shell returns the path to Python installation, console launched by py.exe returns the path to the launched program (PD) installation. If the path differs from the one written on the very first line of

the code (that is where the author had it installed), PD will offer to save that path, effectively overriding its own source code.

Global constants and configurable global variables . When PD has determined and changed current directory to the directory of its installation (so it can read and write files such as backup or config), it initializes (and declares at the same time) global constants, which all are auxiliary variables and are described in the comments. Then it imports global variables from a configuration file named pd-config.py. Use and how-to-edit of this file are described in README.md. Please, consider README.md to be the first Appendix to the report.

Data structure used to record the current position . As of last import from pd-config, the global variable default-main-position is imported. To the value of this variable, which never changes in the program (may be changed directly in the pd-config.py though) is the value of main-position variable assigned before every new game starts. Yes, that is the record of the start position of English Draughts game as described above. The variable main-position itself holds the current position in the game and is declared right after the importing of configurable globals. Its functionality and format are described in the comments. Please, consider not only this part of the comments in the source code but all of them to be the second Appendix to this report.

The data structure chosen to record the position in the game is not the best available (not the worst though). The best format for recording the position from the program's performance point of view would be three sets of four bytes. In Python, those could be three variables, integers. The first would hold the position of red pieces (thirty-two squares equals to thirty-two bits), where every bit assigned to 0 would be an empty position, assigned to 1 would be the position of a red piece. Similarly, the second variable would hold the position of white pieces. The third then would hold positions of kings for both sides. Considering, that only theoretically the values of the first two of these numbers would be up to $2^{32} - 2^{20} - 1 = 4,293,918,719$, Python would cope with them easily. Variables then could be tested by logical operations in the program much faster than accessing an array to retrieve an integer and then test it comparing to another integer in the array determining its index by an arithmetic operation.

Nevertheless, the position in the game is recorded in an array consisting of 36 elements, four of which are not used. This is to avoid unnecessary conditioning: on the Figure 1 is shown the board numbered 1 to 32. Numbers correspond to the index of an element of the array. When a piece which moves from the bottom to the top is on any square in the bottom row, to test if it has a legal move the program needs to find the value of the element indexed square +4 and +5 (in the row above it). But, if the piece is on the square in the second row, the program needs to test squares +3 and +4. Also, squares on the edge of the board using this addition will evaluate to squares which are not a legal move: (5+3 = 8: illegal, while 6+3 = 9: legal). This would need determination whether the piece stands on an even row or an odd row and whether it is on the edge of the board or not. Therefore, more conditions are required, what brings the performance down.

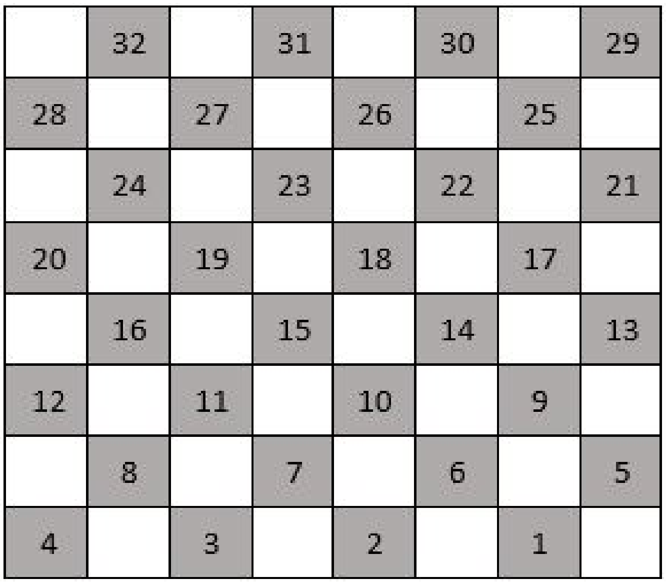

Figure 1: **Board 1 - 32:** Raises the issues

Luckily, simply not using indexes 9, 18 and 27 makes this conditioning redundant: on the Figure 2 is displayed the board with squares numbered 1 to 35, with omitted multipliers of number 9. Now, it does not matter where the piece stands: to test the adjacent positions the program always needs to test square +4 and square +5 (and -4 and -5 if it is a king). If it evaluates to one of 9, 18, 27, or greater than 35 (or smaller than 1 in case of kings): that is not a legal move. Otherwise: yes, that is a reachable square, it only needs to be empty to be a legal move. For this reason, the number three is assigned to the current position record on indexes 0, 9, 18, 27. Other values are from -2 to 2 as described in the comments in source code.

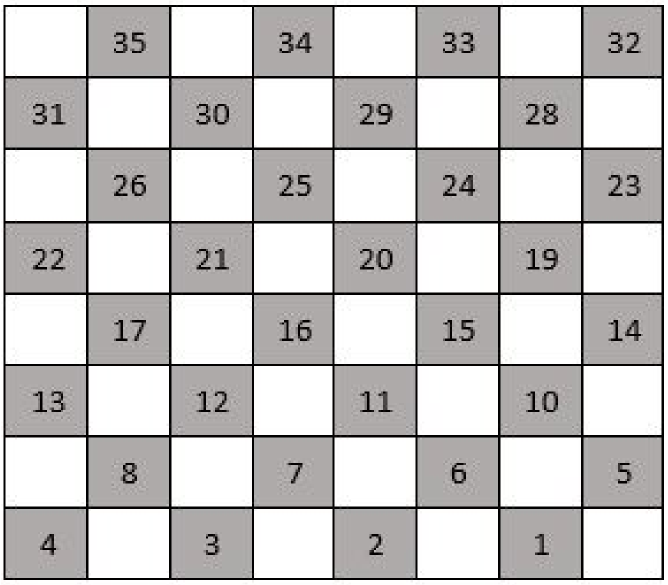

Figure 2: **Board 1 - 35:** Solves the issues

The separate array is provided to record the position from each player‘s point of view and they both are elements of main-position list. Again, this is to improve performance during calculations of the best move. If it was recorded in one array only, every time the side changes to white, the array would need to be accessed from the last index to the first, negating the values in the array and replacing unused indexes (0, 9, 18, 27) appropriately, not saying that there would need to be a condition to resolve which player is to move. While, when two arrays are provided, then none of these tests are needed.

Who is to move? The next variable declared in the source code after main-position is the variable called who-am-I. It is an integer, which only holds two values throughout the program‘s execution. Number one and number minus one. This variable is seen in every function, many expressions and other places of the program. Its value determines, which side has the current move, i.e. for which side is currently calculated whatever is calculated. Here is used the Python‘s ability to index an array, list, tuple or string from the last position using negative indexes. However, in languages which do not support such an indexing, it would be perfectly usable too, just a bit differently: adding one to the value of this variable would point to index 0 or 2. Across the code are many variables, most often lists, but also tuples and strings, which have assigned three elements, one of which is unused. That is because who-am-I is used to accessing them, and it accesses either index 1 or -1 (i.e. the last element).

History and Undo, Redo, Replay .
Follow the declaration of game-history and redo-stack lists. These are used to record the game positions and positions on Undo respectively. Here can be said that there is no more appropriate data structure to use a LIFO method in the programming world of high-level languages than an elastic array (an array, which does not need to have declared its size prior to use, but may adjust its size in accordance of program‘s needs), in Python referred to as ‘list‘. Use of this two lists in the program is straightforward: to the game-history is appended a string which holds reformatted position after each move. When the user uses ‘Undo‘, the last element from the game-history is removed and appended to the redo-stack. Similarly, when the user uses ‘Redo‘, the last element from redo-stack is removed and appended back to the game-history list. Things in the real program are not that simple though: if on ‘Undo‘ only one move was undone, and the opponent is AI, AI would move it back. For this reason, always two moves are undone, and on the redo, just one is redone. But on the first redo, the game mode is switched to Person vs Person mode (the user can ‘Hand over‘ the play to AI at any time when he is done with ‘Undoes‘ and ‘Redoes‘).

Both variables are also used to provide the ‘Replay‘ feature. When the user enters this option, all elements apart from the first one (the oldest position, i.e. start position) from the game-history list is moved to the redo-stack list and reversed, so the second position is at the last index, the third at the index before that and so on. In fact, after this assigning of the variables, 'Redo' feature is used to replay the game.

Content of the redo-stack list is held until the last element is popped out, when 'Redo' option becomes unavailable, or until the user makes a move. This situation is considered that the user started a new sequence of moves, different from the previous sequence and redo-stack is flushed.

Recursive functions used to get possible legal moves .
The program flow continues importing functions from pd-functions.py. As was mentioned above, these functions supposed to be changed if a different variation of the game is to be implemented. The first of them, get-possible-moves(), and the last, update-main-position are the most often accessed functions in the program. The four between them in the source code, are supplementary functions to get-possible-moves() function.

get-possible-moves() is the head of the algorithm written to determine legal moves. One of the positions array is passed to the function as the parameter: this is the reason, why red‘s position and white‘s position are recorded separately and both from the point of view of the respective player. get-possible-moves() only calculates position as it was red‘s position, i.e. the moving direction is from the lower indexes to the highest. It starts by adding additional ‘unused‘ elements, which have a value of the int(3). Actually, in this particular function and its supplementary functions, these elements are used: to determine illegal square (in other words: a position out of the board) just in the matter of one condition, instead of long compound conditions. This one condition first time occurs just a few lines below in the compound condition: if piece ¡= 0 or piece == 3, which evaluates to False when on the index ‘start‘ is element int(1) or int(2), i.e. player‘s piece, not the opponent‘s, so the flow can continue to the cascading, long condition below it (also, good programming practice was sacrificed in expense of better performance). These cascading conditions determine whether a piece can move from its square at all, and if it is a jump, there is always a possibility for a multi-jump of uncertain length. So writing an additional loop to effectively determine the multi-jump is almost impossible. Thus recursion is used.

When get-possible-moves() determines, that the piece on the tested square can jump, calls get-mans-jumps() or get-kings-jumps() depending on what the tested figure is. These two functions could be written as one, in fact, all four supplementary functions could be written in one function just adding a few conditions. But taking into account, that the whole function would be recursive, it again comes as wise decision to keep them separate to improve performance as much as possible.

Both functions called from get-possible-moves() are intermediate functions, which main reason of existence is that in the early stages of development the author did not know how globals and locals in Python work. They were left in the program, just to make it more readable at this particular place, where two recursions are written. The recursive functions get-mans-leaps(), and get-kings-leaps() do the same thing: they determine all possible jumps in the given position for the given piece. They recur until the base case is not met: no more legal jumps can be added to the jump stroke. Then, unwinding phase is really fast: just an empty return to the previous stage of the recursion. Of course, get-mans-jumps() and leaps() are much simpler functions than those for kings. Not only kings have two times more directions to jump, but it is also necessary to remove captured pieces on the go: so king does not jump over one piece forward and backwards in an infinite recursion.

This is how the possible moves (and jumps) are determined in the PD game. Please, feel free to use edit-main-position.html and stage any position to check if the function get-possible-moves() always returns all possible moves. It was heavily tested by the author. The only drawback of this function is, that in the case when a king can make a jump in two different ways, but capturing the same opponent's pieces and landing on the same square, as it started, the function will return two possible moves instead of one of them. This is not a big harm to the play-ability: the resultant position will be the same regardless what move out of two is chosen, and positions like this are happening extremely rarely in the real play if they are real at all. To stage the position like four white pieces on the squares 11, 12, 20, 21 and the red king to move from the square 7, would require a very close cooperation of both sides if they obey to the rules of the game: red king would need to have had a possible jump one move before that, what means another figures have to be present on the board, at least one red, which jumped instead of the king in the previous move. And what move white has to had done before that to get to this situation? For this reasons, the conditions, which were written to avoid this drawback, were dropped for the sake of better performance.

The function update-main-position then updates the record of the current position. Using parameter the-move (one element of the return from get-possible-moves() function), which has been chosen by the player, and parameter who-am-I to determine for which side the position is being updated, it updates and returns the new version of the main-position record to the caller. The source code of this function is well commented explaining its functionality.

Main (function) .
After imported functions, the program continues through the functions declared in the main module. The important are: get-humans-move(), get-ais-move(), and my-spider() which are explained below. Then the program continues on the line just under the comment main. Yes, in a language like Java this section would be the part of main() function. In Python, it does not need to be signified.

The first part of main() consists of boring, from a programming point of view, getting and processing of user's input. Scrolling down to the comment ' End of user's input processing.', opens the part where the game 'starts'. In other words, here is the heart of the program. Firstly, all necessary variables need to be set to the game starting values. Then the backup file is initialized, and the game starts right under the line announcing this fact.

Of course, an infinite loop is set, to be broken out when the game ends or the user decides to quit. Here follows the condition resolving if that is AI's turn to make a move, or human's. Obviously, if the game is set to person vs person mode, the first part of the if clause never evaluates to True, and similarly, when AI vs AI is set, the 'else' part of the clause is always False. If the game is set to the mode AI versus a person, the first part of the condition is evaluated to be True, the AI's move calculation starts. After printing the current position to the console for the user, function get-ais-move is called. No parameters are passed but called function does make use of some global variables such as main-position and who-amâĂŞI. Therefore, the program continues in the get-ais-move() function, which can be found scrolling up the source code roughly to the middle of the document, to the line around 567.

The first thing after declaring globals is that the function gets all possible moves in the current position. If there are not possible moves, it means the game is lost for the player. Next condition sets global will-draw to the value of main-position[0][0], where is stored the number of moves since the last capture in the game. This is done to resolve the drawn position by so-called forty move rule (sometimes sixty move rule). The rule says, that if in forty consecutive moves a capture did not happen, the game supposed to be declared drawn. In AI vs AI mode there is a good chance that the program stacks in an infinitive loop of repeated moves if neither side can win. So, the condition was written to stop this loop, and the number of moves was shortened for testing purposes.

The interesting part starts below these conditions, by initializing so-called 'collectors'. Here is the time to say that PD uses the algorithm referred to as Minimax to choose the best move. It calculates all possible positions a few moves in advance (depending on the difficulty level set by the user), using recursive function my-spider() and collects information from the farthest leafs of the imaginary tree of a possible way the game progresses. It is the tree because from each possible move in the position zero, are several possible moves in the position one and so on. But more important is the evaluation expression, the Minimax itself. So, what information and how it is used to choose the best move is explained below.

Minimax .
PD initialize some global variables (global in the scope of get-ais-move() function and all functions called from this function), mostly lists with integers and one integer called my-status. My status is assigned to the cumulative values of elements of the array holding record of current position (part of main-position), depending on the value of who-am-I global, i.e. cumulative value from the reds or whites point of view depending for which side the move is being calculated. Then it shortly resolves the 'depth' of recursion and starts its journey from the first possible move to the last (expressed in the for the-move-number loop). After updating current position for the move (but not the main-position global variable itself, so the game comes back where it was before the recursion), function my-spider is called to resolve positions from the point of view of the opponent. My-spider again does the same: gets possible moves, and if there are possible moves, traverses the tree for every possible move for opponent's side from the scope of my-spider(). It does so by calling function.. my-spider(). So, obvious recursive function, which will finally

meet its base case, that is when the value of variable 'level' is smaller than 0. And it sure be after certain number of calls, because my-spider() calls itself with lowered value of 'level' by one. Once the base case condition is met, my-spider() does not return immediately: first it collects the information to the collectors and then loops for the next move. Just when there are no more possible moves, it returns to its parent. And finally, the first child of the get-ais-move() will similarly return to get-ais-move() .

Lists score-collector and weight-collector now hold calculated values of possible positions in the farthest leafs. By value is meant the cumulative value of elements of the array which holds the record of the current position. And it holds positive integers one or two for friendly-side man and king respectively, and similarly negative integers int(-1) and int (-2) for the opponent's side. When this value is greater than the value of the my-status variable, the absolute difference of the two is added to the score-collector[move-number][2], and weight-collector[move-number][2] is increased by one effectively counting how many times my-status was better than resultant position status. Similarly, collectors with indexes [1] are added the absolute difference of cumulative values of original position and the positions in the last leafs. Also, the number of situations when the position is not better or worse than the original position are counted in weight-collector with the index [0]. There are also some tweaks, such as extra score added for capturing opponent's kings and promoting. Also, when the game is won or loss, big values are added to respective collectors, because this is already an ultimate move.

Once the tree is traversed to the required depth, there is the need to evaluate gathered 'information'. This is ensured by weighted average between the 'good' moves, which lead to better positions than original position, and 'bad' moves. This expression proved be better than a simple average by testing AI's play against AI with a different evaluation expression.

7 Enhancements:

If I had more time, I would improve the Minimax algorithm (the expression evaluating gathered information). As of now, weighted average ensures that PD plays better than random player: even difficulty level one beats difficulty level zero in hundred percent tested cases (tested more than a hundred cases). The results are less satisfying when is tested the play of level one against level three or higher: the higher level wins very closely, six to eight games out of twenty with many games resulting in draws and also has some lost games. This is largely due to bad play in the game ending when just a few moves ahead are not enough. Seven moves ahead make a difference, but one single move on this depth of recursion takes too long.

Feeling from playing against the AI may differ from person to person, depending on their skill in the game. I do not consider myself a good player of English Draughts, so at the beginning of the game and the mid part, I see the AI on levels around three as a good player. It does not let the opponent to expand too much; it even can stage traps to capture two opponents pieces sacrificing one of its own. But once the game goes closer to an end, fewer pieces are on the board; it starts to make 'stupid' moves and loses.

The fix to it may be another improvement I have in mind: building database with pre-calculated positions. Of course, the program will have to be re-written. If not in a lower level programming language, at least position record should be done differently, so complete integers can be stored in the database and sorted right at the insertion point, so then they can be retrieved using binary search. Also, the database would be divided into smaller pieces using hashing algorithms. One of hashing function would be the number of pieces on the board: so once a piece is captured, the position will never repeat in this game. Therefore the part of the database in memory may be flushed and overridden by the appropriate part from a file. Another hashing function would be the number of men in the first twelve or even sixteen squares: once a man leaves this area, it never gets back.

The best working part of the program is the function get-possible-moves() and its supplementary functions, specifically recursive functions get-mans-leaps() and get-kings-leaps. It never makes a mistake; it is very quick, in the boundaries of the high-level interpreted language and returns the results in the data structure appropriate for further processing by this program (appropriate to be processed with the main-position arrays, which choice of the data structure was discussed at the beginning of this report).

8 Personal evaluation:

Working on this problem I have learned a lot from the Python 3 programming language, which I see as a good alternative to languages like PHP for server-side scripts writing. Particularly, when there is no need for a big database or NoSQL database is required, Python 3 would be even better choice. Also, I had the opportunity to try some algorithms thought on lectures of this module. Unfortunately, I did not get to the database design, so sorting, searching and hashing algorithms are not present in the program. There is a kind of sorting algorithm which chooses the max number out of a list of numbers in the get-ais-move() function, but only finding the max value is quite an elementary programming skill. In overall, I feel I performed well in the development stage, less positive feelings I have about this report.

To conclude, Pythonic Draughts meets requirements set in the task, probably the algorithm choosing the best move is a bit less of success, on the other hand, there are good points in functions from pd-functions.py and added features like a configuration file, opportunity to save a game and replay step by step.