Juuso Käenmäki 479725
Juho Marttila 356178
Joonas Ulmanen 432335
Ville Harmaala 347873

ELEC-A7150 / sanajahti1

Documentation for Sanajahti Solver

Table of contents

1. Overview
2. Software structure
   - Overall architecture
   - External libraries and dependencies
3. Software logic and function interfaces
   - Main Algorithms
   - Important Classes
4. Build and usage instructions
   - How to build the software
   - How to use the software
5. Testing
6. Work log
   - Division of work and responsibilities
   - What was done on each week, and by whom

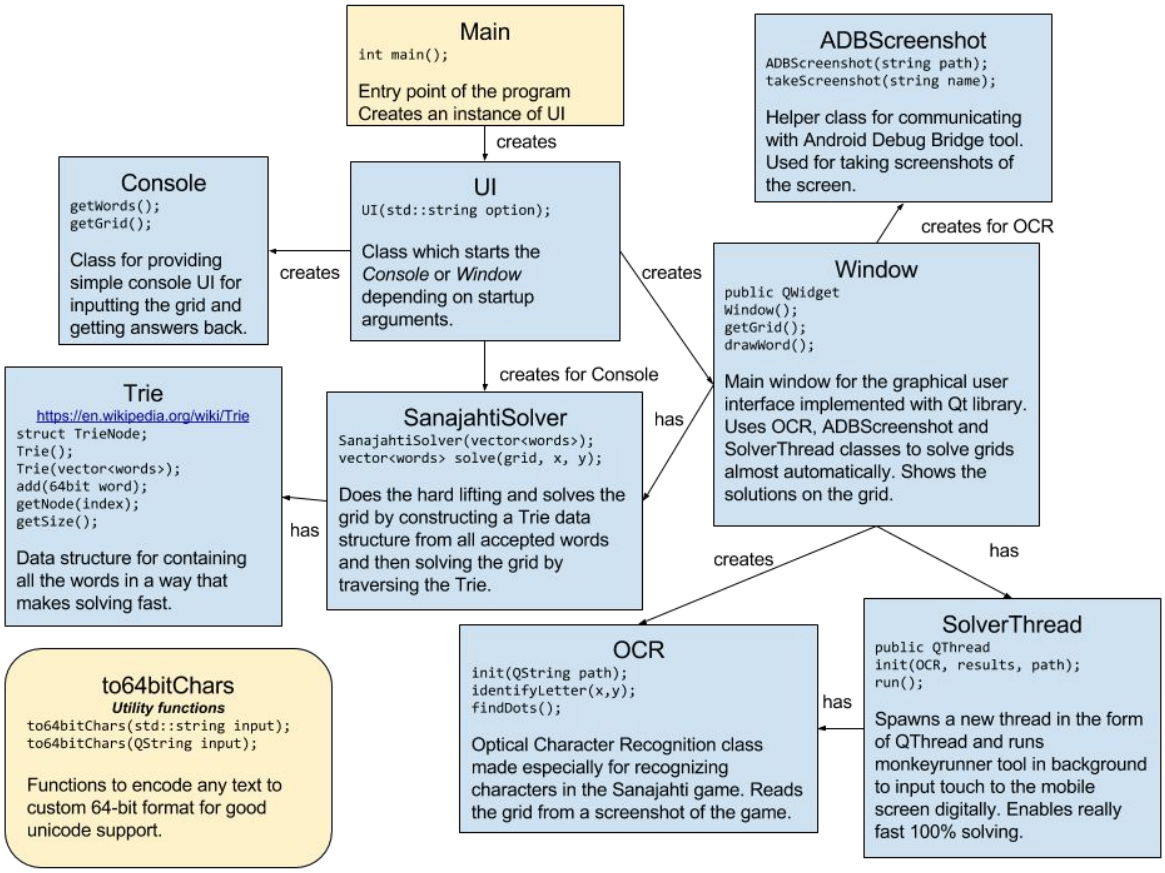
1. Overview

The software is a Qt/C++ based solver for the game “Sanajahti”. As an input it takes a grid of letters, and a wordlist. With these inputs the software calculates paths on the grid that produce words on the wordlist. The software can take input via command line, GUI, or straight from an Android phone.

Implemented features

- basic command-line UI
- Graphical UI using Qt
- UTF-8 support
- reading the solving grid directly from the Sanajahti game with an Android phone
- inputting all solutions to the Sanajahti game via Android phone

2. Software structure



Overall architecture

As the graphical user interface binds most of the functionality together, the architecture is centered around the GUI. When the program starts, at first it calls the UI class constructor, which determines, based on command line arguments, if GUI or command-line interface should be created. If GUI is chosen, the UI class then asks the Qt framework starts an Qt application with a window, and execute the program.

The Qt application works through the Window class. This class, and its constructor defines the application layout and right functions for each event (such as button press). The window class owns the solver, and uses it to solve the grid. It also owns the classes that command the phone to retrieve screenshots, and input the solutions.

The Window class is the main graphical user interface, and as such, it owns many objects of other classes. Window class has its own *SanajahtiSolver* which handles solving the grid, *ADBScreenshot* (gets a screenshot of the grid from the phone), *OCR* (recognizes the characters in screenshot) and *SolverThread*, which inputs the solution to the phone. *SanajahtiSolver* is the main solving class, which handles constructing the data structure for fast solving, and solving the Sanajahti grid with specific inputs using the data structure.

SanajahtiSolver objects are used both by console UI and GUI. *SanajahtiSolver* owns a *Trie* object that handles building, storing and accessing the *Trie* data structure. Trie is only used by *SanajahtiSolver*.

External libraries and dependencies

The program is built on top of Qt5 framework, so Qt is required for building the program.

Interfacing with Android phones is done with the help of external programs from Android SDK, bundled with the repository. These are ADB (Android Debug Bridge) and Monkeyrunner. With ADB program can execute commands in the phone, and Monkeyrunner is a test automation tool, which is used to send inputs to Sanajahti game. ADB is a standalone program, but Monkeyrunner is built on top of Jython. A standalone Jython is also included in the *tools* folder. Instructions for using Auto Solve feature are in section 4.

Folders that bundle external programs: *tools, platform-tools*

3. Software logic and function interfaces

Main Algorithms

Solving algorithm
SanajahtiSolver uses the *Trie* data structure to solve the game quickly. Solver first starts from each position in grid, and starts traversing it, like a search tree. The idea for fast solving is this: solver also traverses the *Trie* structure at the same time, and goes only to neighbor grid nodes that fulfill these conditions

- node's character is a child of the current TreeNode character
    - Example: if traversed path is "aut", wordlist contains only the word "auto", and your neighboring nodes are "o, r, i, c", there is no reason to visit the three last ones.
- Node has not been visited yet

The first condition saves so many paths from being visited that a typical grid is solved only in milliseconds. Trie structure is built because we need to know which characters are the possible next ones when traversing the tree.

OCR
The letters on the Sanajahti game window are mostly single colored. The program uses certain points of the grid to locate all the letter tiles on the game window and then search for the letter color and create smallest possible rectangle around the letters. This rectangle is converted to matrix with all letter colored pixels having value 1 and all others value 0. This matrix is compared to pre-created matrices of every letter from A to Ö, which is scaled to same size as the area of the letter. After running against all the letter matrices, letter giving best response is returned for the program.

Important Classes:

Window : public QWidget

Window is the basis for the graphical user interface in the program. It's implemented with the Qt library and it defines the whole graphical layout, i.e. there are no other graphical components in this program. In addition to defining how the program looks and behaves, *Window* has some complex logic in it related mainly to showing and drawing various things on the grid. In addition to this, *Window* uses *SanajahtiSolver, OCR* and *ADBScreenshot* and mainly shows results from them.

```
Window();
//return size of the x-axis of the grid.
int getX();
//return size of the y-axis of the grid.
int getY();
//returns the path of the library given.
std::string getLibrary();
// get the inputted character grid and return a vector of it
// encodes the characters in the grid to a custom 64-bit fixed-width format
std::vector<uint64_t> getGrid();
```

ADBScreenshot

Helper class that utilizes the Android Debug Bridge tool to fetch screenshots from an Android mobile device. Gets initialized with an absolute file path, which is used when calling the ADB tools bundled with the program. Function *takeScreenshot* handles errors itself by outputting the error reason to standard error output and then returning false to mark that the function did not succeed. The screenshot is always removed from the device to not waste space.

```
ADBScreenshot(const std::string &path);
bool takeScreenshot(const std::string &name);
```

SanajahtiSolver

Solving a Sanajahti grid in this program is a two-step process. First, caller must create a SanajahtiSolver, whose constructor creates a Trie data-structure using a vector of words (strings).

```
SanajahtiSolver(const vector<QString>& words);

vector<pair<string, vector<pair<int, int>>>>
solve(const vector<uint64_t>& grid, int x, int y);
```

When the solver is created (trie constructed), caller can call the solve function with the grid to be solved, and its dimensions. Solve function returns a vector of solutions, where each

solution is a pair: first element is the word, second element is the route of the word in the grid, represented as a vector of pairs (x,y).

Trie

Trie constructor creates the trie structure, by first creating the root node and then for each word, using the *add* method. This is the most performance-sensitive part of the solving algorithm, creating the trie takes 50-100 ms (with kotus.txt Finnish dictionary)

```
Trie(const std::vector<QString>& words);
```

The Trie data is accessed by SanajahtiSolver using this method, which returns a TrieNode struct representing all the data if a tree node.

```
const TrieNode& getNode(const int nodeIdx) const;
```

OCR

OCR-class is used to search certain elements in the screenshot image file provided by ADBScreenshot-class. The grid contains 9 single color dots, which are in every crossing of 4 grid-tiles. These dots are used to locate all 16 tiles of the grid. After that the program marks every pixel containing the text color it matches

```
OCR();
void init(QString path);
std::string identifyLetter(int x, int y);
void getGridSize();
bool findDots();
std::pair<int,int> getTileCoordinate(int x, int y);
```

SolverThread : public QThread

SolverThread-class handles sending the touch inputs to the phone. The running the class takes quite long, so it is run on a parallel thread, so the window doesn't hang on the class's runtime. Touch commands are sent using MonkeyRunner program in a jython script file, *commands.py*, which the program also creates. When creating instance of the class, class checks on which platform the program was run and uses corresponding files.

```
SolverThread();
//initializing function
void init(OCR ocr, vector<pair<string, vector<pair<int, int>>>> results,
          string path);
//overrides function from QThread
//executes the thread
void run();
```

4. Build and usage instructions

How to build the software

The software is built using CMake, which generates the required makefiles to build the software. As the program is written using Qt GUI framework, Qt (version 5) must be installed to build the software. Build instructions:

1. Set environment variable QTDIR to your Qt installation directory (with right compiler), for example “/Users/jma/Qt/5.5/clang_64/” on OS X or “/home/jma/Qt/5.5/gcc_64/” on Ubuntu. Example command: `export QTDIR="/Users/jma/Qt/5.5/clang_64/"`
2. Use CMake in the main directory, where CMakeLists.txt is located, with command “cmake ./”
3. Build the program using command “make” in the same directory
4. The program is now built and ready to be used. Note that folders “tools” and “platform-tools” should be in the same folder as the executable, as these include binaries for android-interoperability

In addition, unit tests can be built using first command “cmake ./” and then command "make" in the directory called "tests" and run the tests by typing "./sanajahti_test".

How to use the software

1. After starting the program you need to specify the library file containing all the words needed by the solver. This is done by pressing button “Library”, and the selecting the correct file.
2. Next you can let the program automatically solve game by using button “Auto Solve”. This feature takes screenshot of the phone screen, so make sure the game is running and the grid is visible. Alternatively you can manually fill the grid and press button “Start” to show all the words found by the program on the screen.
3. If you want to solve another grid you can use the Auto Solve feature straight away, but for manual inputting, you need to press the button “Restart” first.

To be able to use Auto Solve, USB Debugging Mode must first be enabled on the phone. Below are short instructions how to do it (source : *https://www.kingoapp.com/root-tutorials/how-to-enable-usb-debugging-mode-on-android-5-lollipop.htm)*.

Step 1: Home Screen > App Drawer > Settings
Step 2: Settings > About Phone
Step 3: About Phone > Build number > Tap 7 times to become developer
Step 4: Settings > Developer Options > USB Debugging
Step 5: Click to enable USB Debugging mode on Android [version number & name]

In addition, when first connecting the phone to computer, user must approve the computer to command the phone. A prompt is shown to the user on the phone when this must be done.

Console UI

If the program is called with argument “console”, the program doesn’t launch the GUI, and instead the user is presented with a console-based interface. The user is first asked to provide the wordlist file and then the grid. The program solves the grid with wordlist and prints the solutions, along with their paths on the grid.

5. Testing

The testing was mostly done using the solver itself. This is because the program is significantly based on its graphical user interface, which can be used to see if different functionalities really work. Unit testing of graphical interfaces may not necessarily reveal all of the problems (bugs and user interface problems), but are instead found more easily when the program is actually used.

The *Auto solve* feature makes the manual testing process a lot quicker as enables direct communication between the solver and the Sanajahti application. By using this feature, it can be quickly noticed if the solver doesn't find all the correct words that exist in the wordlist. Therefore the *Auto solve* feature makes unit testing less important because the program can automate some testing by itself by using the actual game as a reference. No problems were found in the solver algorithm during manual testing cases.

The manual testing consisted of several different test cases. An essential part of the solver is the wordlist that the solver uses as a helping tool to find the words. Therefore one of the test cases was to give the solver a wordlist that is of a different file type than a proper .txt file, for example an object file. The solver handles the situation well although no error messages are shown. The solver simply does nothing when a false type of a wordlist is given.

The truthfulness of the solver was tested by giving an input, of which no words can be constructed, such as only consonants or special characters. In this case the solver works properly and doesn't construct words that don't exist. As a result the list of found words remains empty. The next test case was naturally to try a fully or partly empty grid, where some of the boxes are intentionally left empty. The program then opens a notification box that tells the user to complete the grid. Along with this test the different sizes of the grid were tested from the smallest 1x1 grid to the largest 20x20 grid. No problems either.

An important part of the communication between the solver and a phone is Android Debug Bridge, which enables the solver to send and receive data from the phone. The connectivity was tested first such that *Auto solve* was pressed when the the phone wasn't connected. The program successfully tells about the problem and keeps functioning properly. After the phone is connected and *Auto solve* is pressed again, the solving process finishes without

errors. The next test case was to interrupt the program by unplugging the program during the solving process. Again the error window pops up and tells the same error message as before. If the phone is then connected again and *Auto solve* pressed, the solver can still solve all the words.

Finally, playing the game offered important feedback. The found words were compared to the used wordlist *kotus.txt* and in most cases all the words were found.
The only minor problem was that *kotus.txt* didn't include all the words used in the Sanajahti game. It is noticeable though that the solving algorithm can find all the words that the list includes. Therefore adding these missing words to the list would have solved the problem if the same word showed up in an another game.

In addition, unit tests were used as was already mentioned in the project plan. The group implemented the tests using Google test, which turned out to be a bit complicated to install. There were a lot of manuals around the Internet but none of them answered why the compiler kept saying there were problems with dependencies. After countless hours the problems were solved by installing the Google test directory straight to the test folder.

The unit tests consist of nine tests that are described below in the same order as in the tester *test_sanajahti.cpp*. The first test *TestLength* checks that the length of the grapheme is correct. It first initializes four different test strings and gives them as a parameter to the function *graphemeLength()*, which is located in the source file *to64bitchars.cpp*. Finally, it compares the returned to the correct length and tells if the tested function returns an incorrect length.

The same source file has also a function *to64bitChars* that converts a QString to a custom 64-bit format. The test begins with initializing a test string and its correct counterpart in ASCII format. A new grid is then created by calling the function *to64bitChars()*, which gets the test string as a parameter. This newly created grid is lastly compared to the expected result and an error message is printed if the tested function didn't return the correct string.

The source file *console.cpp* is tested pretty thoroughly as all the functions apart from the function *run()* are tested. The reason for not testing the function *run()* is the fact that it doesn't return anything. However, all the functionalities can already be tested by implementing the unit tests for the rest of the functions. The tests *GetX* and *GetY* check that the corresponding functions *getX()* and *getY()* from the source file *console.cpp* return the correct *x* and *y* sizes for the grid. It is done by simply calling the functions and checking that the returned values are equal to the initialized values.

The tests *GetLibrary* and *GetGrid* test the functions *getLibrary()* and *getGrid()* similarly, i.e. giving a proper parameter to the function, which leads to a desired behaviour of the program. For example, the variable *library* is initialized as an empty string in the source file *console.cpp* and therefore the returned value is compared to an empty string in the test *GetLibrary*. *GetGrid* uses a similar method and initializes an empty vector of 64-bit unsigned integers and compares the returned value to the vector. This is because the grid is originally empty.

In the case of *IsValidGrid* the comparison values are initialized as an empty grid and a non rectangular grid to see if the function *isValidGrid()* can complete its checks and return false. In the beginning of the test two vectors of strings are initialized, one of which is the empty grid and the other the non rectangular grid. Now if the tested functions are working properly, the tests will print the error messages defined in the functions and the test will be passed. Otherwise, if the input is a non empty or rectangular grid, the test fails and the error message of the test will be printed.

The second to last test is *GetNodeSIze*, which tests the function *getSize()* from the source file *trie.cpp*. The goal is to make sure that the function returns the correct amount of nodes and in this case the amount is originally '1'. The test calls the function and compares the returned value to the integer '1'. The last test *GetNode,* which is also located in the source file *trie.cpp*, tests that the function *getNode()* returns successfully a node from the trie. It initializes the trie with similar words *auto* and *auta*, after which it initializes the desired values as ASCII characters and lastly initializes the correct indices of the nodes. The output of the function *getNode* is then solved recursively by giving the previous index of the node as a parameter to the current node. Finally both the indices of the nodes and the ASCII characters that the node includes are compared to the previously initialized correct values.

6. Work log

Division of work and responsibilities

Juuso Käenmäki

Juuso Käenmäki focused on the graphical user interface. He did by far most of the work regarding the Qt GUI. Juuso also implemented the OCR functionality that recognizes the characters from the screenshot downloaded from the connected Android device. Juuso's third area of work in this project was the *SolverThread* class which inputs the computed Sanajahti solution to the phone by using external program Monkeyrunner.

As it is mentioned in the project plan, Juuso Käenmäki's main responsibility was the user interface. When project was being planned, we did not assign people features that we might not make at all. When the project was going forward smoothly and we estimated we can implement these additional features, Juuso took responsibility of developing first the OCR functionality and secondly the automated input functionality.

Files of responsibility:
ui.hpp / ui.cpp
window.hpp / window.cpp
console.hpp / console.cpp
ocr.hpp / ocr.cpp
solver_thread.hpp / solver_thread.cpp

Juho Marttila

As it is mentioned in the project plan, Juho's work was the solving algorithm itself. He implemented all of the code regarding the solving algorithm (*SanajahtiSolver*) and the underlying *Trie* data structure. Juho also designed the necessary algorithms and data structures. In addition to the main solving algorithm, Juho also developed unicode support for the algorithm, and developed the procedures, data structures and algorithms for it. In addition, Juho did most of the work regarding building the project, making the CMake-based build system to work etc. Juho also did general organization and established the communication channels.

Juho's responsibility area, as it is mentioned in the project plan, was the main solving algorithm, and in addition to that, unicode support.

Files of responsibility:
solver.hpp / solver.cpp
trie.hpp / trie.cpp
to64bitchars.hpp / to64bitchars.cpp

Joonas Ulmanen

Joonas was sick during the first two weeks but contacted the group to tell about the situation. As soon as Joonas was able to start the project, he implemented the ADB screenshot functionality that commands the phone to take a screenshot and transfer it to the computer. One of the general tasks of Joonas was to clean the code base every now and then to maintain the quality of the code. He also tested the program in Linux and Android environments as other members of the group mostly used Windows or OSX. Joonas' last task was to write a part of the document and create the diagram that illustrates class relationships.

Files of responsibility:
adb_screenshot.hpp / adb_screenshot.cpp

Areas of responsibility:
ADB screencap functionality
diagram that illustrates class relationships
writing a part of the document

Ville Harmaala

Ville's task was to implement the unit tests using Google test and take responsibility of other general testing of the program. However, due to the great amount of work with other courses, Ville didn't have enough time to implement the tests early enough. Therefore the tests couldn't be used to verify the validity of changes that were made to the algorithms

along the way. At last during the last week the unit test were implemented but at this point the group already had the knowledge that the program works as it should. Ville tested the program one more time to make sure no bugs were left and wrote the chapter 5 that handles testing. The existing tests come in handy if the program is improved later on.

Files of responsibility:
test_sanajahti.cpp

Areas of responsibilty:
primary writer of the project plan and document

What was done on each week, and by whom

	Juuso	Juho	Joonas	Ville
Week 1 2.-8.11.	3 hours Project Plan	8 hours Project Plan Basic non-trie solver with generic test ui	0 hours (was sick)	3 hours Project plan
Week 2 9.-15.11.	10 hours Basic console-UI Setting Qt up Early version of the GUI	6 hours Trie-based fast solver Build system development	0 hours (was sick)	0 hours Other school tasks took all the time
Week 3 16.-22.11.	14 hours Better GUI OCR functionality	0 hours (was busy)	0 hours	0 hours Other school tasks took all the time
Week 4 23.-29.11.	3 hours Merged console UI and GUI to same main.cpp Project meeting	7 hours Unicode support Project meeting	5 hours ADB screenshot support Project meeting	0 hours Other school tasks took all the time
Week 5 30.11.-6.12.	6 hours Monkeyrunner functionality and ADB without installs	6 hours General fixes and testing	5 hours OCR support with ADB screenshot	10 hours Configuration of QT and Google Test Planning the tests
Week 6 7.-13.12.	10 hours Bug fixing Documentation Commenting and code quality	18 hours Build system fixes Commenting and code quality Testing and bug fixes Documentation Helping Ville with unit tests	5 hours Documentation Bug fixing Commenting and code quality	20 hours Implementing the unit tests Testing the program in general Being the primary writer of the document