Juuso Käenmäki
Juho Marttila
Joonas Ulmanen
Ville Harmaala

# Project plan for Sanajahti Solver

## Scope of the project

The goal of this project is to write a program that automatically finds all the possible words from a matrix. It takes as parameters a wordlist and an integer that tells how many rows and columns there are in the matrix. The program then presents letters line by line in an NxM matrix, after which the found words are printed each on their own line. The program pretty much consists of clever use of different algorithms, such as trie, in order to optimize the program to be fast enough.

Our goal is to adjust the scope of the project to the available time. We plan to implement the mandatory requirements first, and only then move on to the more advanced features. If we have time, we plan to implement UTF-8 support, graphical UI, and computer vision features.

**Example of program output**

```
aro  (2,0)  (3,1)  (2,2)
duo  (1,2)  (2,3)  (2,2)
edus  (0,3)  (1,2)  (2,3)  (1,3)
eka  (0,3)  (0,2)  (1,1)
kaarros  (0,0)  (1,1)  (2,0)  (3,0)  (3,1)  (2,2)  (1,3)
kaaso  (0,0)  (1,1)  (2,0)  (2,1)  (2,2)
.
.
.
tuoda  (3,3)  (2,3)  (2,2)  (1,2)  (1,1)
tuossa  (3,3)  (2,3)  (2,2)  (2,1)  (1,0)  (1,1)
tuska  (3,3)  (2,3)  (1,3)  (0,2)  (1,1)
ussa  (2,3)  (3,2)  (2,1)  (1,1)
```

## Architecture

Our plan is to, at first, divide the project into three parts: The trie data structure, the solver algorithm, and the main program that includes the graphical user interface. This is a clear division with clear boundaries. In practice this means creating two classes: Trie and Solver, and in addition to that, the main program which uses this solver. The solver then uses the Trie to solve the puzzle.

At first we are going to implement a simple command-line UI, which will ask the user to type number of rows and columns of the grid and string containing letters from the grid, starting from the upper left corner continuing to the right. If the length of the string does not match the grid size, the program will ask user to fill the parameters again. After successful

parameters are given, the program will print every possible word on its own line, followed by coordinates of every letter. Below is an example conversation with the program. Black represents the program output and red the user input:

```
Enter wordlist: kotus.txt
Enter the Sanajahti grid, row-by-row, separated by enter, empty
row ends the entry:
sdiu
daas
fujf
gefj
k

The grid is not rectangular.
Please type the asked parameters again.
Enter the Sanajahti grid, row-by-row, separated by enter, empty
row ends the entry:
aretu
ferie
datni

Parameters are ok.
Computing possible words...
```

## Summary of the main elements

### Main

Uses a class called *solver*. It gets user input, constructs the solver with a vector of words and the Sanajahti Grid. Finally the results are displayed.

### Solver Class

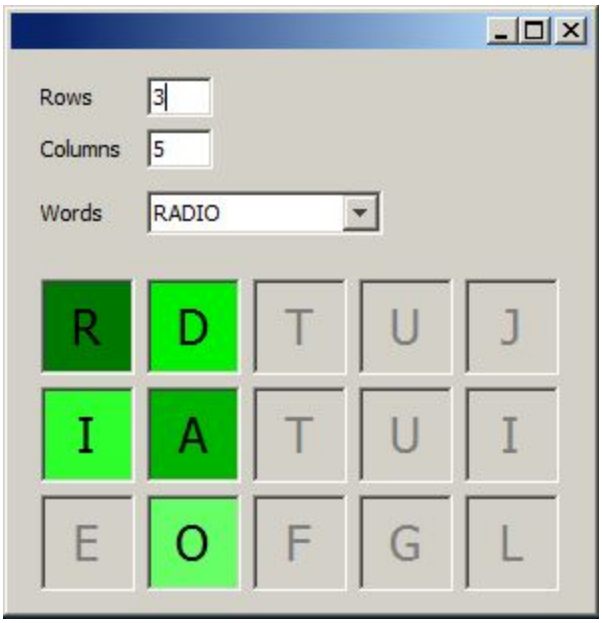
Uses a trie structure and delivers the puzzle solution to the caller of the solve method. The solver constructs the trie and adds the words into it. The trie structure is used to solve the puzzle.

### Trie Class

A constructor that initializes the trie structure. It has methods to add words to the trie. It also provides an interface to work with trie nodes themselves.

### Graphical UI

When we start creating the graphical user interface, we need to use some UI library. The architecture of the program will depend heavily on which UI library is used. As we don't know these libraries yet, the architecture of the GUI code needs to be determined at a later time.

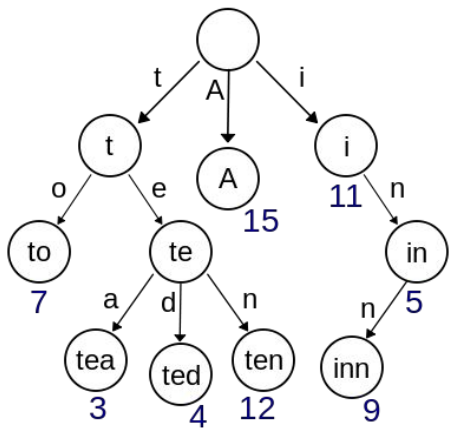



*possible implementation for the GUI*

We are also interested in a computer vision that captures a screenshot, which is used to determine the grid. This is naturally one of the last voluntary parts to be implemented because it can either be pretty straightforward and simple or very challenging. We found a library called Tesseract, which is an OCR Engine that was originally developed by HP Labs and currently by Google. It is advertised to be the most accurate open source OCR engine available so it could be a match for the problem.

# Algorithms

The main algorithmic challenge is how to solve the puzzle efficiently. For that, we plan to use a recursive algorithm that uses a Trie data structure. Trie is a data structure that encodes a word list in a tree where each node is a character (root node excluded). The idea is that all words can be constructed by following the tree from root to each leaf and concatenating the characters found in nodes.




*Trie data structure (source: https://en.wikipedia.org/wiki/Trie)*

The idea of the Sanajahti solver algorithm is as follows: For each starting block in input array the algorithm recursively traverses all possible routes. Because this alone is too inefficient, we plan to use the Trie structure to leave out routes that do not contain a valid word. It works as follows: At the same time as the input array is traversed, the trie is traversed too. If a trie node (leaf or not) contains is the end of a word, a result is found and pushed back. Then the algorithm continues the recursion to only those (not visited) tiles with character same as one of the children in the current node of the trie.

# Testing

All the functionalities will be tested thoroughly. To begin with, the group familiarizes itself with testing by studying the tests used in the course exercises. They should give a pretty good idea how to take different scenarios in the code into account. Google Test will most likely be used to make the unit testing process a bit easier. To check that all the possible words have been found, we'll use a web service *Wordamental* to compare our library to *Wordamental's* library. If *Wordamental* finds a word that can be found from our word library but not from our solution, our program doesn't work properly. An important note is that this method works only for 4x4 matrices so any other sizes of the matrices cannot be tested. In addition, it only supports an English character set. If we haven't found any better solution until the first meeting with our assistant, we'll ask for tips.

However, testing cannot catch every error in the program, since it cannot evaluate every execution path in any but the most trivial programs. In addition to unit testing, the program will be executed with the intent of finding software bugs, i.e. causing situations that the program should handle instead of crashing. Below are listed some of the properties (according to Wikipedia) that the program should fulfill.

- meets the requirements that guided its design and development
- responds correctly to all kinds of inputs
- performs its functions within an acceptable time
- is sufficiently usable
- can be installed and run in its intended environments
- achieves the general results its stakeholders desire

# Schedule

Our main principle is to compose a schedule that we can follow as strictly as possible. This way we'll be able to get the biggest amount of tips during the first code review meeting. The minimum requirements are the first goal and as soon as everything compulsory works as it should, we'll move on to the additional features. The first one of them will be implementing a support for UTF-8 and special characters. This enables us to use a wider range of words because now also umlauts can be recognized. The UI will be implemented first as a command line version and later on improved to a graphical version.

**Detailed schedule**

- Week 45 (2.11.-8.11.)
    - finish the project plan
    - share the tasks

- Week 46 (9.11.-15.11.)
    - Improve the performance of the solving algorithm by implementing trie structure
    - A robust command line UI

- Week 47 (16.11.-22.11.)
    - Basic functionalities so we can get as much feedback as possible
    - Support for UTF-8
    - Support for special characters

- Week 48 (23.11.-29.11.)
    - Graphical UI

- Week 49 (30.11.-6.12.)
    - Graphical UI
    - OCR if possible

- Week 50 (7.12.-13.12.)
    - Finishing
    - Polishing the code
    - Writing instructions for the program

# Division of labor

Joonas: Tasks will be given as soon as Joonas isn't ill anymore
Juho: Main responsibility of the solving algorithm
Juuso: Main responsibility of the command line UI and the graphical UI
Ville: Main responsibility of testing

# Communication

The group communicates via Skype and all the code will be stored in Git. Most of the project is done as remote work and therefore Collabedit (www.collabedit.com) is used to collaborate in real time. The idea is that Skype allows us to use voice communication and Collabedit helps us demonstrate different parts of the code and modify it such that the other group can see the changes immediately. Skype also offers a quick chat, which is handy when some links need to be shared, or if someone needs to send information to the other group.

# Risks

Most of the risks are related to the availability of time. We all have a lot of courses and therefore multiple projects going on so it will be difficult to follow the schedule. As a solution we try to implement a minimum viable product first, after which we start developing the extra features as much as we have time. In addition, there is a risk that some of us might fall behind if one is absent for a while because of other deadlines. We'll try to solve this by sharing the tasks carefully and by communicating regularly.