

Performance and Test, Spring 2008

Project description

Project: KRAK Route planner

In this project you will be provided with the actual data set that KRAK uses to provide its route planning services on www.krak.dk as well as a basic code framework for loading this data set. Additionally you will be given specifications for a route planning system and asked to analysis, design, implement, and test it.

The problem to be solved

The final implementation should allow a user to input a source and destination address. The output is a descriptive route plan of the shortest possible route from source to destination, describing which roads to follow and where and in which direction to turn. In terms of user interface you can freely choose to implement it as a command-line, web, or window (AWT/Swing etc.) based interface.

The shortest path functionality must be implemented based on both Dijkstra's shortest path algorithm and the A* search algorithm, and it has to be possible to specify at runtime which one is to be used.

Approach

You should follow the approach of an *incremental, iterative software development process* with two increments. Each increment should contain the four phases: Analysis, Design, Implementation, and Test.

After the first increment the program should be able to read in the KRAK data and provide the capability to read user input, map it to the graph, find some route from the source to destination, and output a route-plan describing that route. All this functionality should have automated tests to indicate that things work as expected.

In the second increment you are required to add the shortest path algorithms in order to guarantee that the shortest possible route is found and complete the user interface. The automated tests for the previous increment should be used as a regression test, and additional tests should be provided to test the new functionality.

Tasks

Below you will find a list of tasks that we suggest you solve and integrate into your project report. They are intended to guide you through the process as well as ensure that you make use of the techniques covered in this course, but it doesn't cover everything that should be in the report. Make sure to document your work on these tasks in the report but don't treat them like a list of questions to be answered one by one, just make sure each one is covered naturally in the report.

Your report should have a table of contents containing at least the following elements:

- First increment
 - Analysis
 - Design
 - Implementation
 - Test
- Second increment
 - Analysis
 - Design
 - Implementation
 - Test

You should use the suggestion for a test plan from test lecture 3 (see the slides from the lecture, available from <http://www.itu.dk/courses/SPT/F2008/Episode03.2/>).

Time planning

You should hand in the first increment to your project supervisor on **16/4, 15:00** including the part of your report related to the first increment. This is a preliminary version of the report, which you will be given feedback on from your supervisor. Try to make an initial estimate and break down of how to use the time from 16/4 to 21/5 in order to meet the deadline. A challenge is not to get stuck in one of the phases but be sure you get through to testing. On the other hand, not spending enough time in Analysis and Design can be a bad idea too.

The final report with increment two is to be handed in **21/5, 15:00** at the exam office. This version will be part of the exam for the course.

Notice There is a programming contest with **great prizes** for the best performing implementation. The description will be made available on the course homepage. The **deadline for handing in your contribution to the contest is 30/4**.

Analysis

Elaborate the specifications to explain what you will do in each increment. Based on the specifications given, write a set of use-cases.

Design

Come up with an initial design for the system and document the structure using UML class diagrams. Check the design with guided inspection.

Identify primitive and non-primitive classes in the design and provide functional unit and interaction tests for them prior to starting the actual implementation. Make at least one interaction test with mock objects.

Implementation

Implement the code for the first increment. Identify useful class invariants and pre- and post-conditions of methods. Write unit tests to test (some of) them.

OPTIONAL: Identify invariants in the search algorithms and other relevant parts of the code and implement these as Java assertions (similar to JUnit's assert methods, but intended to be used not in

the actual program code) as additional checks besides unit tests. You can read more about java assertions here <http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html>. The lab installation notes explain how to do this from inside Eclipse.

Use a profiler to document the actual coverage of the unit and integration tests. Add tests to improve the coverage if needed.

Use Ant to create a deploy script that creates a package for the end user. This could for example be an executable jar file or similar, containing all the necessary auxiliary data as well as the compiled program.

Testing

As the final testing, you should rerun all the automated tests and document the outcome in the report. Define from the use cases, some system tests. It is ok to make the manually. Document the outcome in the report.

Performance of implementation (in the second increment)

Briefly describe the utility data structures used in the program as well as their performance characteristics (examples could be hash tables, priority queues etc).

Describe and analyze the performance of the approach taken to transform the textual input from the user into the appropriate nodes in the graph.

Describe and analyze Dijkstra's shortest path algorithm when used for a single-source single-destination search. Do the same for the A* algorithm. In addition to the usual worst case analysis also relate the performance to the structure of the graph and relevant properties of the source and destination.

Perform appropriate benchmarks on the number of nodes expanded in addition to the time and space usage and compare with the analysis. Use a profiler to track where the CPU time is used, and consider optimizations based on this.

OPTIONAL: If you decided to use Java assertions above, perform benchmarks with assertions enabled and disabled and compare.

Imagine that the system was to be used on a much larger roadmap, say the roadmap of USA. Try to estimate the absolute time various searches would take on such a map, based on the empirical results from the Danish roadmap.

OPTIONAL: Consider how to find the *fastest* path instead of the shortest. What needs to be changed for A* and Dijkstra?

Online resources

At the course webpage <http://www.itu.dk/courses/SPT/F2008> you will find the KRAK data under Resources.

Good luck, Kristian and Dan