UPDATES TO THE COURSEWORK HANDOUT

HANDOUT UPDATES

- A couple of updates have been made to the handout.
- These are all minor, but are meant to
  a. Clarify a few aspects regarding generation of events.
  b. Tweak the output requirements to simplify the marking process.
  c. Exemplify summary statistics.
  d. Clarify output specifications for experiments and introduce delimiter requirements for the output.
  e. Introduce the requirement of a **Makefile**.

HANDOUT UPDATES - EVENTS

- Requests arrive in the system in a **stochastic** fashion.
- A request comprises:
  - desired pick-up time
  - source stop
  - target (destination) stop
- Time between requests and the delay between a request and its correspondin pick-up time are chosen by sampling exponentially distributed R.V.
- When the mean rate is given as input, the mean delay is computed as the reciprocal of the rate.
- Source and target stops chosen uniformly at random.

HANDOUT UPDATES - OUTPUT

- Specify the destination stop of a request, i.e.

```
‹time› -> new request placed from stop ‹unsigned int› to
   stop ‹unsigned int› for departure at ‹time› scheduled for ‹time›
‹time› -> new request placed from stop ‹unsigned int› to
   stop ‹unsigned int› for departure at ‹time› cannot be accommodated
```

- For example:

```
00:01:25:10 -> new request placed from stop 3 to stop 8 for departure
   at 00:01:34:00 scheduled for 00:01:36:00
```

EXAMPLE SUMMARY STATISTICS

```
---
average trip duration 14:30
trip efficiency 7.20
percentage of missed requests 0.10
average passenger waiting time 30 seconds
average trip deviation 1.60
---
```

Delimit the start and end of the statistics with a `'---'` sequence.

OUTPUT OF EXPERIMENTS

- When running experiments, disable the output of detailed information and instead only display summary statistics.
- To delimit the different experiments, precede the output of each with the following heading:

```
Experiment #‹experiment no.›: ‹param1› ‹param1-value› ‹param2› ‹param2-value› ...
---
```

OUTPUT OF EXPERIMENTS

- For example:

```
...
Experiment #4: maxDelay 10 noBuses 15
---
average trip duration 14:30
trip efficiency 7.20
percentage of missed requests 0.10
average passenger waiting time 30 seconds
average trip deviation 1.60
---
Experiment #5: maxDelay 10 noBuses 20
---
average trip duration 12:00
...
```

HANDOUT UPDATES - MAKEFILE

- Part of the functionality of your simulator will be automatically tested.
- Therefore, also include a **Makefile** with your submission.
- Running `make` in the root directory of the submitted project should produc a running application.

CSLP ASSESSMENT

ASSESSMENT CRITERIA (I)

1. Implementation of requirements:
   a. Parsing
   b. Input validation
   c. Correct simulation & correct output
   d. Summary statistics of simulation results
   e. Experimentation implementation
2. Source code documentation (comments)

ASSESSMENT CRITERIA (II)

3. Testing, including sample test input scripts
4. Maintainable code
5. Code efficiency (optimisations)
6. Any additional features
7. Written report

OBJECTIVE & SUBJECTIVE CRITERIA

- Some of the items on the above list are *objective* whilst some are *subjective*
- *Objective* criteria are those which are testable
- *Subjective* criteria are those which are, at least partially, based upon opinion

OBJECTIVE ASSESSMENT CRITERIA

This first list of implementation requirements are all relatively objective:

1. Parsing
2. Input validation
3. Correct simulation & correct output
4. Summary statistics of simulation results
5. Experimentation implementation

OBJECTIVE ASSESSMENT

- Your application will be put through my own suite of test inputs.
- Some of these test inputs will be inputs you have seen, some will be new.
- Part of the exercise is for you to foresee possible inputs for which your application would fail
  - Either by crashing, or by producing incorrect output.
- Should your application fail any tests I would have to figure out **why** this happened and objective marking would not be so straightforward.

OBJECTIVE ASSESSMENT

- A good part of the testing will be automated, so you either pass or fail a test.
- **Follow the given input/output specification strictly** to avoid unpleasant surprises.

PARSING

- Your parser should be able to parse all syntactically valid input scripts.
- I cannot say it much simpler than that.
- There will not be any deliberately tricky tests.

INPUT VALIDATION

- This is the first task which is not finely specified.
- You have to demonstrate some ingenuity to devise your own rules for what should and should not be valid input.
- You also have to decide which kinds of inputs result in *warnings* or *errors*.
    - Specifically those in which the simulation could be **started** but may resul in an error.
    - This may depend upon the structure of your simulator.

CORRECT SIMULATION & OUTPUT

- Here I will be testing whether your simulator follows the requirements correctly.
- The simulator is tested via its output, so these are tested at the same time.
- Having said that, where the output is not correct, the code is inspected to determine why.
    - This is part of the reason your code must compile on DiCE

SUMMARY STATISTICS

- This will test for correctly calculating and reporting the specified summary statistics.
- It is possible to get the simulation incorrect but the summary statistics correct.
- A small tip is to make sure your reported statistics are consistent with each other.
- It might be that you are getting inconsistent results because your simulation incorrect, in which case you should note this in your README.

EXPERIMENTATION IMPLEMENTATION

- Whether or not you correctly implement the experimentation of maximum admissible pick up delay and number of buses.
- As before it is possible to get this correct, without getting either (or both) of the simulation and the summary statistics correct.
- As before, if you are getting inconsistent results you should at least note that in your README.

CODE EFFICIENCY

- Implementing some code optimisations will lead to shorter run times.
- It is possible that you implement everything above correctly, but your simulations take a very long time to complete.
- On the other hand, your code may run fast, but will not have implemented al requirements. This is not considered to be efficient.

NOTING DEFICIENCIES

- Use your README file to record any deficiencies you are aware of.
- In general any implementation errors will be treated more indulgently if they are known about.
- Remember, it is generally worse to produce incorrect output than no output at all.

SUBJECTIVE ASSESSMENT

The remaining items are mostly judged subjectively

- Source code documentation
- Testing, including sample test input scripts
- Maintainable code
- Any additional features
- Written report

DOCUMENTATION

- Use appropriate comments to document your code.
- You may develop additional features which, if you do not document, I may nc even know about.
- Clear mark and explain the code that you have not authored yourselves.
- Remember that code sharing is not allowed.

TESTING

- The practical is intended to write a good simulator.
  - You can at least strive for *“half decent”*.
- Either way, running one test input, is woefully insufficient.
- You also need to be able to investigate the performance of the "on-demand transport operation", what parameters affect this and how.

MAINTAINABLE CODE

- ***Highly*** subjective.
- Remember, reusable code is more difficult to understand.
- But, reusable code is easier to reuse and maintain.
- What is an inexperienced developer to do?
- Try to imagine what you might wish to do in the future.

MAINTAINABLE CODE

- Trying to justify your choices is likely a good thing.
- Even if your reasoning is flawed, it demonstrates that you have thought abou how to design your source code.
- It also shows that you probably could have implemented things in other way, but specifically chose not to.
- A future maintainer at least knows why you made that choice, if they disagre they can change the code without fear of some other reason they have not yet uncovered.

ADDITIONAL FEATURES

- This is your chance to be creative and go beyond the implementation of the requested features.
- It perhaps requires some imagination, but imagine you were really going to use your simulator to investigate some real (or other) logistics operation.
- What would be useful to you?

README

- Don't forget to provide me with a README.
- In general this can only help your grade:
    - It lets me know good things are deliberate and not fortunate.
    - It lets me know that deficiencies are at least known about.

WRITTEN REPORT

You should produce a written report that discusses:

- the key building blocks of your design,
- the results of the analyses you performed with different inputs,
- insights gained into system's performance,
- a summary of the most important findings.

USEFUL THINGS TO INCLUDE IN YOUR WRITTEN REPORT

- Produce graphs based on the numerical output of your simulations to suppor your findings, especially for experimentation.
- Explain the purpose of the tests carried out and whether the results met your initial expectations. Any lessons learned?
- Motivate your choice(s) of route planning algorithms implemented and discuss their impact on the performance of the system.

FINAL POINTS

- The report will have a **25% weight** of the final mark.
- There is no minimum number of pages required for the report.
- Present your findings and results clearly.
- Submit the report as a PDF file.
- Students are often worried about *losing* marks.
- Indeed our own assessment descriptions often talk of losing marks.
- But let's not forget, you start with zero.

OPTIMISING COMPILATION

OPTIMISING COMPILATION

- We already discussed about code optimisation.
  - Benchmarking
  - Profiling
- It is possible to further optimise your code at compilation
  - try to minimise program's execution time;
  - try to minimise the amount of memory occupied (less common);
  - minimise the consumed power (for mobile devices).

COMPILING AND LINKING

- Compiling is not the same as creating an executable.
- Building an executable involves compilation and linking.
- Your code may compile without errors, but it may fail during the linking phase.

COMPILING AND LINKING

Compilation

- Turning the source code into an 'object' file.
- This is not executable, it only contains the corresponding machine language instructions.
- If you have multiple files, you will have multiple objects

```
# gcc -c -o "simulator.o" "simulator.c"
# gcc -c -o "utils.o" "utils.c"
```

The "-c" flag specifies that no linking should be done at this stage.

COMPILING AND LINKING

Linking

- The process of creating a single executable from multiple object files.
- Finds references for the functions that are used in one file but were defined i
  another.

```
# gcc -o "simulator" simulator.o utils.o
```

COMPILING AND LINKING

- This approach allows building large programs without having to redo the compilation every time a file is changed.
- Conditional compilation --compile only source files that have changed;
- Conditional compilation works well when you use some Integrated Development Environment (IDE).
- Otherwise you will have to manually create a `makefile` and use the `make` utility, which determines what needs to be recompiled.

OPTIMISING COMPILATION

- When you compile your code, you can set some flags that instruct the compiler to perform some optimisation.
- Note that this often takes more time and requires more memory, but your executable may run faster.
- Example:

  ```
  # gcc -O3 -o "simulator.o" "simulator.c"
  ```

  `-O<level>` instructs the compiler to perform some optimisation.

OPTIMISING COMPILATION

- `-O1` - tries to reduce code size and execution time, without performing optimisations that increase compilation time significantly.
- `-O2` - performs several optimisations that do not involve a space-speed trade off. Increases both compilation time and the performance of the generated code.
- `-O3` - optimises even more.
- `-O0` - reduces compilation time and makes debugging produce the expected results (default).

Check the GCC manual page for more details.

MULTIPLE FILES

- Question: Should you spread your implementation across multiple source code files?
- There may be *some* good reasons to do so:
  - Increase code reusability
  - Reduces compilation time
  - Could help navigating source code faster

MULTIPLE FILES

- Not suggesting you should not, but do so for a good reason.
- Given the size of this project, you could try to use as few files as possible.
- Move type definitions, functions, etc. to separate files when that seems necessary.

SHOULD I DEVELOP CODE WITH OR WITHOUT AN IDE?

- This shouldn't make a difference, but you may have good reasons for choosin one of the two approaches.
- Coding using a plain text editor (e.g. vi, nano)
  - You can easily code remotely (over ssh) on e.g. a DiCE machine
  - Write a **makefile** yourself (essential if working with multiple files).
  - Better control on compilation optimisation.

SHOULD I DEVELOP CODE WITH OR WITHOUT AN IDE?

- Using IDEs
    - Nicer keyword highlighting;
    - Some auto complete braces/brackets/parenthesis;
    - Some may have integrated help for functions;
    - Some warn about certain syntax errors as you type;
    - Perhaps easier if you are not very experienced in C;
- If you decide to code using an IDE, it's entirely up to you which one you choose (NetBeans C/C++ pack, CodeLite, Eclipse CDT, etc.)