Efficient Route Planning

SS 2011

Lecture 2, Friday May 13th, 2011

(Implementation Advice, Build, Test, Style)

Prof. Dr. Hannah Bast
Chair of Algorithms and Data Structures
Department of Computer Science
University of Freiburg

Overview of this lecture

- Implementation advice
  - Graph representation
  - Dijkstra with a priority queue without decrease key
- Unit tests
  - A short HowTo
- Make / Ant + Jenkins
  - Make / Ant = build framework for C++ / Java
  - Jenkins = our continuous build system
  - I will provide a short HowTo for both
- Visualization of geo data
  - Google Maps, Google Fusion Tables, Google Earth (KML)

Graph representation

- Adjacency matrix
  - Store the arc costs in an n x n matrix, where n = #nodes
    - if arc does not exist, put some special value, e.g. ∞
  - Needs space $\Theta(n^2)$
  - Ok when m = #arcs is very large (so-called dense graphs)
- Adjacency lists
  - For each node, store an array of the outgoing arcs + their costs
  - Needs space $\Theta(n + m)$
  - Method of choice when $m << n^2$ (so-called sparse graphs)

Graph representation

- Adjacency lists: Algorithm engineering
  - The straightforward implementation is

    ```
    vector<vector<Arc> > adjacencyLists;
    ```

  - An alternative would be

    ```
    vector<Arc> adjancencyLists;  // size = #arcs
    vector<int> adjacencyListsOffsets; // size = #nodes
    ```

    where the first vector is the concatenation of all adjacency lists, and the second vector contains, for each node, the start of the adjacency list of that node in that concatenation

    - more space-efficient (each vector has space-overhead)
    - more time-efficient (all adjacency lists are contiguous in memory, hence better cache-efficiency)
    - the straightforward implement. is fine for now though

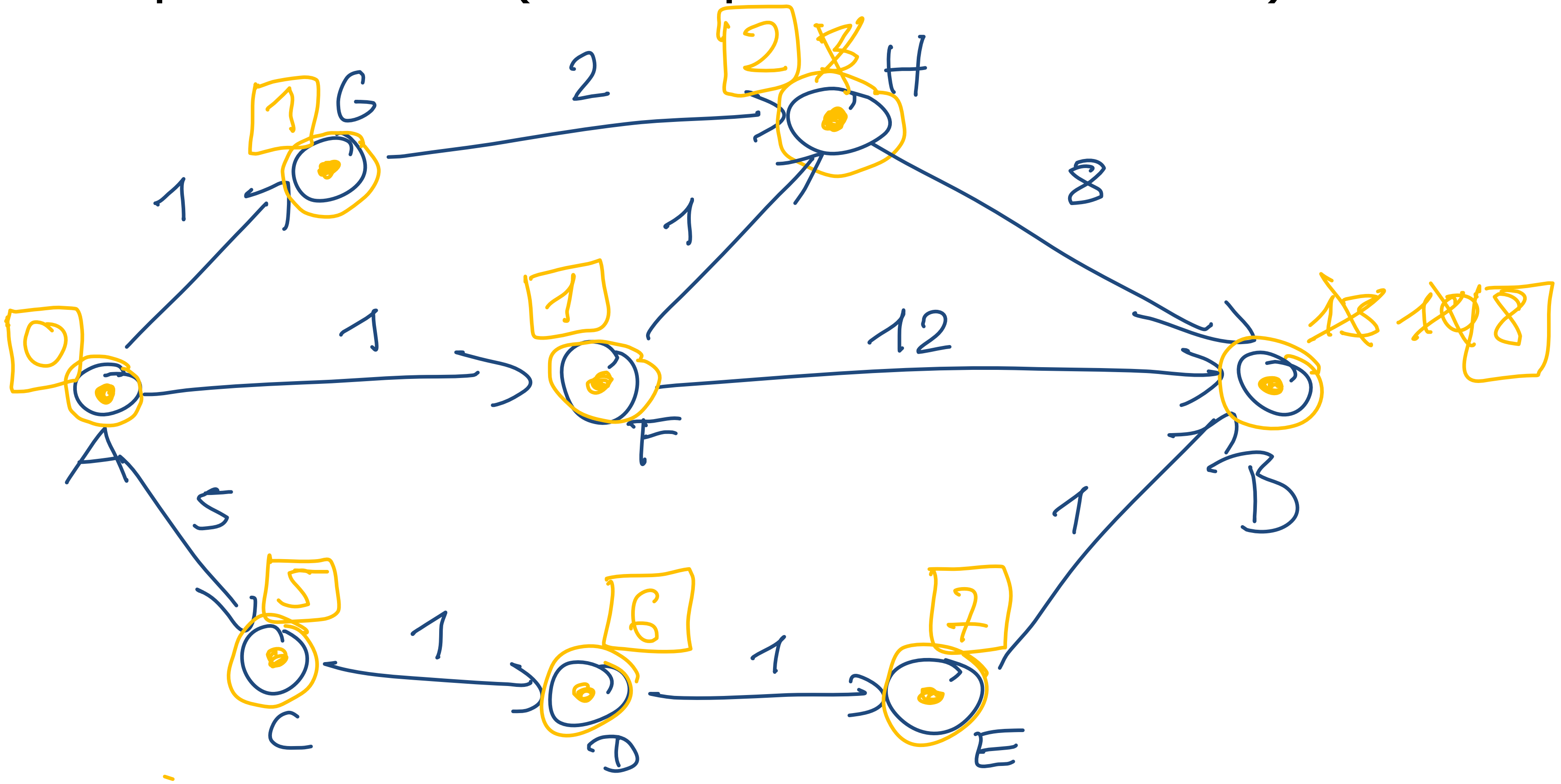
Dijkstra without decrease key

- Ordinary Dijkstra
  - The tentative distance of a node in the priority queue (PQ) can decrease several times over the course of the execution

    → seems we need a PQ with a decrease-key operation
  - However, the std::priority_queue does not support this
  - There is a simple trick to avoid this operation
    - instead of a decrease-key, insert the node (again) with the smaller tentative distance
    - whenever a node with key larger than the already known tentative distance is removed from the PQ, ignore it
    - works fine as long as there are relatively few decrease-key operations, which is the case for road networks **why?**

Dijkstra's algorithm

- Example execution (slide copied from last lecture)



● active

◉ settled

Make / Ant + Jenkins

- Jenkins is our continuous build system
  - See link on your Daphne page (linked from the Wiki)
  - Will build your code (from the SVN) on one of our servers
    - Important to ensure that it does not only work on your local machine
  - Jenkins assumes a build file in your SVN
    - Makefile for C++, build.xml for Java
    - The build file should provide four targets
      - compile, test, checkstyle, clean
    - Let's see an example of this ...

Unit Tests

- Test functionality of each (major) method
  - Otherwise 1: debugging becomes a nightmare
  - Otherwise 2: no trust in your experimental results
  - Let's write a unit test together ...
- Problem: testing equality of complex objects
  - For example, a whole road network object
  - Simple solution: for each class provide a method DebugString which outputs the object in a simple human-readable form
  - Then your test can check simple string equality, e.g.

```
rn.readFromOsmFile("RoadNetworkTest.TMP.osm");
ASSERT_EQ("[3,2,{(1,2)},{2,3},{3,1}]", rn.DebugString());
```

Unit Tests

- Which framework to use?
  - For C++ please use gtest
  - For Java please use JUnit

Style checking

- A consistent style is important when you write code
  - Especially when you work in a team, but not only then
  - Please use the following style checkers:
    - cpplint.py for C++
      - implements Google's code conventions
      - you find cpplint.py in your SVN subdirectory
    - checkstyle for Java
      - highly configurable, i.p. to SUN's code conventions
      - you find checkstyle.jar and sun_checks.xml in your SVN directory (do svn update to get it)
  - Set the checkstyle target in your build file accordingly

Visualizing your data

- How to visualize nodes / arcs?
  - Later in the course we will see how to write a simple UI using the Google Maps API
  - In the meantime, the following might be useful
    - In Google Maps you can just enter a coordinate (latitude, longitude) into the search field
    - With Google Fusion Tables you can import CSV files with geo information and visualize them
    - With Google Earth you can visualize KML files (an XML dialect for representing simple geo information)

References

- Unit testing frameworks
  - http://code.google.com/p/googletest/
  - http://www.junit.org/
- Make / Ant
  - http://www.gnu.org/software/make/
  - http://ant.apache.org/
- Checkstyle
  - http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml
  - http://checkstyle.sourceforge.net/
- Data Visualization
  - http://maps.google.com/
  - http://www.google.com/fusiontables
  - http://code.google.com/apis/kml/