Tut-Lab Week 6

Aims:

- Consolidate knowledge of basic search algorithms
- Reinforce usage of design patterns

Preparation:

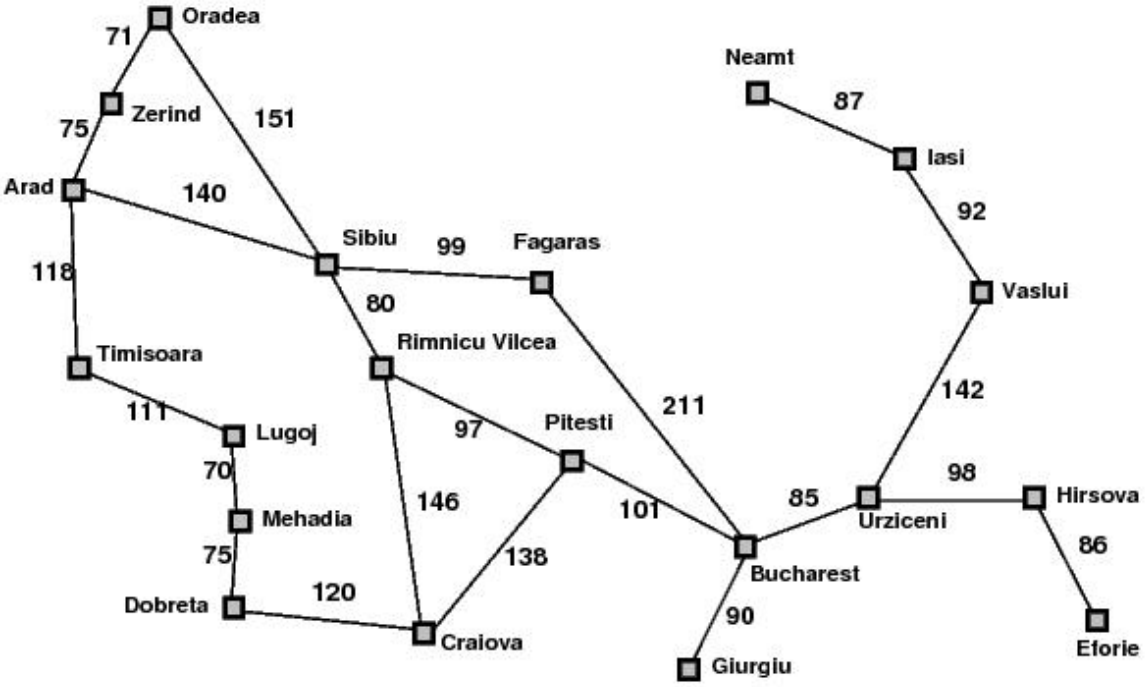
- Review graphs and basic search algorithms from COMP1927
- Review the `Strategy` pattern
- Review the use of the **Comparator** interface type for sorting collections

Graphs

- A *directed graph* is a set of nodes and a binary relation over this set (that node *a* is related to node *b* can be represented by an arrow from *a* to *b*)
- Define a generic **Graph<E>** interface type that can handle nodes of a generic type **E**
    - What are suitable accessors and mutators that would be needed with this interface type?
    - Apart from the basic operations, can you think of some more complex graph operations that could be part of the interface?
- As you will recall from COMP1927, the two standard ways of representing a graph are the *adjacency matrix* and the *adjacency list*
    - If a graph has *n* nodes, an adjacency matrix is an $n \times n$ matrix with entry 1 (or some other data value) in the *i*, *j* element if node *i* is related to node *j*, while the adjacency list is a list whose elements are lists, where for element *i* of the main list, the associated list is of those nodes *j* such that node *i* is related to node *j*
- Choose one of the graph representations and provide a class that implements the **Graph<E>** interface type in this way
- Implement a test class that creates and performs some basic operations on graphs
    - Verify that the implementation really is generic by creating and manipulating two graphs whose nodes have different types
- Define some suitable class invariants for your implementation of the **Graph<E>** interface type and show that they are satisfied

Basic Search Algorithms

- Consider the route-finding problem using the following Romania map as an example (ignore the straight-line distances for now)



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

- Define the route-finding problem (from Arad to Bucharest) as a state space search problem
- What order are nodes in the state space expanded for each of the following algorithms when searching for a path between Arad and Bucharest (when there is a choice of nodes, take the one earliest in the alphabetical ordering)?
    - Depth-first search (efficient use of space but may not terminate)
    - Breadth-first search (space inefficient, guaranteed to find a solution)
- Implement breadth-first search for route-finding in the Romania map
    - Use a class that implements the **Graph<E>** interface type to store the map
    - Use a class that implements the **Queue<E>** interface type to store the nodes generated in the search: what information, besides the state, needs to be stored at each node?
    - Store the successors of a node in an **ArrayList** and make sure they are sorted in alphabetical order of the city name before adding them to the **Queue**
    - How did you handle the problem of infinite looping in the search space?
- Refactor your design
    - Apply the `Strategy` pattern so that the successors of a node are sorted using an anonymous class that implements the **Comparator** interface type: in either the method that computes the successors of a node or the method that adds those successors to the **Queue** – is either of these options better?
    - Draw a UML class diagram of your program, making sure your design and code conforms to the `Strategy` pattern