Search

This lab is about search techniques. Namely, BFS, DFS and UCS.

Running example: Romania

The example we'll be using is the (simplified) map of Romania as given in the textbook. The graph has been converted to a prolog-usable form in romania.prolog.

A successor function (succ/2) has been defined as a wrapper to access the (road/3) facts as undirected edges. Please use this in your code instead of the facts directly since the facts are directional but the roads are not.

succ(+From, -To, -Dist):-

> There is an (undirected) edge from ***From*** to ***To*** of distance ***Dist***

The image contains the 'crow flies' distance from various cities to bucharest. A function heuristic/2 has been provided which returns this value wherever available. (If one of the cities is not bucharest, It returns 0).

heuristic(+From, +To, -Dist):-

> Dist is the 'crow flies' distance between From and To.

Notes / Implementation details:

State representation

In all cases, We want the actual path taken along with the distance. We also want to avoid expanding nodes we've already expanded.

To achieve this, Each node will have to store –

1. The current node
2. its predecessor
3. The cost of reaching the current state.

A node 'A' is the predecessor of a node 'B' if A was added to the fringe while 'A' was being expanded. Using the predecessors of a state recursively, We can rebuild the path taken. Hence, We shoud use a representation similar to the following for each 'state' in the search process:

state(current, predecessor, cost)

Efficiency

For DFS & BFS, We will not bother much about efficiency of the fringe but focus on writing a declarative style using lists and the append function.

For UCS & A*, We will use the heaps library for efficient implementation of the fringe.

The heap predicates you will have to use are below. For a full list, see https://github.com/SWI-Prolog/swipl/blob/master/library/heaps.pl

- empty_heap(?Heap) is semidet.
  - True if Heap is an empty heap.
  - Complexity: constant.
- get_from_heap(?Heap0, ?Priority, ?Key, -Heap) is semidet.
  - Retrieves the minimum-priority pair Priority-Key from Heap0.
  - Heap is Heap0 with that pair removed.
  - Complexity: logarithmic (amortized), linear in the worst case.
- add_to_heap(+Heap0, +Priority, ?Key, -Heap) is semidet.
  - Adds Key with priority Priority to Heap0, constructing a new heap in Heap.

For the sake of reusability, all operations to the fringe will take place using the following predicates:

- fringe_init/1
- fringe_push/4

- fringe_pop/4.

For efficient lookup, we can simply assert a state within a seen/1 predicate and look them up. Since we will always lookup with the current node, it is advisable to use the current node as first element of the structure.

Tasks

1. Open search.prolog. Skeleton code has been provided. Fill in the functions and test your code. Sample output is provided in samples.prolog
    1. For A*, The heuristic can be worked into the fringe_push predicate. Use an assert to make the goal globally accessible.
2. Define the succ/3 and heuristic/3 for the eightpuzzle in a file named 'eightpuzzle.prolog'.
    1. Initially let the heuristic be 0.
    2. Once that works, try new heuristics to improve the search efficiency.