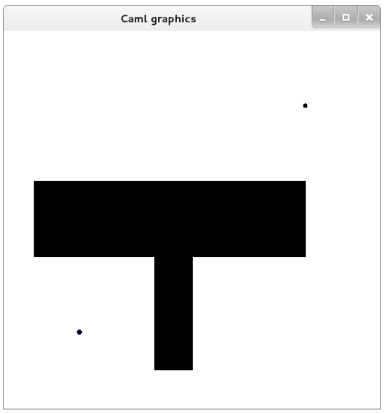
Find the shortest path using an A^* (A-star) algorithm

N. Durand, D. Gianazza

How to get started:

- Download file `tdastar.zip` (or `.tgz` or `.gz`) from `http://e-campus.enac.fr` (course IP-403 – Intelligence Artificielle) and decompress it (unzip, or tar -zxf, or gunzip). This should create an Astar subdirectory under your current working directory.
- Go in `Astar` directory and enter command `make`. This creates an executable file `findpath` and an HTML documentation (in subdirectory `doc`).
- Read the HTML doc using your web browser (`file:///your_path/Astar/doc/index.html`)
- The objective is to write an A^* algorithm and to use it to find the shortest path on a grid starting from an origin O and ending at a destination D, while avoiding an obstacle.



Work to be done

1. Implement the `search` function in file `astar.ml`,
2. Modify the `main` function if file `main.ml` in order to test your algorithm on the proposed path-finding problem.

   This means implementing the cost and heuristic functions, the function producing the next nodes (neighbours, or 'sons' of the current 'father' node), and the function checking if the current node is a goal.

   When implementing these functions, you have a choice between two possibilities for the movement on the grid :

- move only horizontally or vertically,
- move horizontally, vertically, or following a diagonal.

3. Test your program. Try it with the following heuristic function `(fun v -> 0.)`. What happens ? What kind of tree search is performed in that case ?

4. How could you modify your code to perform a depth-first search ? Would that be more efficient ?

Implementation issues

The A^* algorithm could be implemented as in your course, using lists G and D, and an array to memorize the costs $g(u)$ and the predecessors $father(u)$. However, several remarks will lead us to choose a slightly different implementation:

- More efficient implementations can be achieved using binary trees, with $O(log(n))$ complexity for the insertion and extraction operations instead of $O(n)$.

- Allocating memory for an array containing the costs and 'fathers' of all possible states is costly and not necessary. Not all states are visited during the A^* search.

Following these remarks, we replace the "open list" G with a priority queue that contains all states that have been visited, but not expanded yet. We also choose to implement $D+G$ as a hash table instead of representing D as a list. A hash table is an association table containing $key \rightarrow data$ bindings. This will allow us to efficiently store and retrieve all relevant information such as the cost and father of any node in D or G.

With this implementation, the "closed list" D is not needed anymore: we only need to store a boolean value indicating if the node has been expanded (developed) or not.

In the following, Q denotes the priority queue replacing the list G, and M the "memory" (implemented as a hash table) that replaces $D + G$.

Proposed A^* implementation

Algorithm 1 Proposed implementation for A^* algorithm.

1: $cost_0 \leftarrow 0$
2: $f_0 \leftarrow cost_0 + h(u_0)$
3: Initialize memory M with u_0 and associated data $(cost_0, f_0)$
4: Initialize priority queue Q by inserting u_0 with priority f_0
5: **while** priority queue Q not empty **do**
6:     Extract u from Q
7:     $Q \leftarrow Q - u$
8:     **if** `is_goal`(u) (terminal state) **then**
9:         **Exit and return** path from u_0 to final state u
10:     **end if**
11:     **if** u has never been expanded before **then**
12:         Memorize u as an expanded node
13:         $ls \leftarrow$ `next`(u)
14:         **for** all v in ls **do**
15:             **if** $v \notin M$ or $cost(v) > cost(u) + k(u, v)$ **then**
16:                 $cost_v \leftarrow cost(u) + k(u, v)$
17:                 $f_v \leftarrow cost_v + h(v)$
18:                 $father_v \leftarrow u$
19:                 Store v in memory M with data $(cost_v, father_v)$
20:                 Insert v in Q with priority f_v
21:             **end if**
22:         **end for**
23:     **end if**
24: **end while**
25: Raise exception (no solution)

Note that for consistent heuristics there is no need to implement lines 11, 12, and 23. If h is consistent, the path leading from u_0 to u built by the A^* algorithm is necessarily of minimum cost. As a consequence, this path has the lowest achievable value of $f(u) = cost(u) + h(u)$, and there is no way that we could go back to u later (i.e. through one of its successors) and re-expand it.

Suggestions

Some useful code is provided to you in order to complete this training exercise in approximately 2 hours. For question 1, you will need the following modules:

- `Pqueue`, implementing functions to create and handle the priority queue Q,
- `Memory`, with functions allowing you to handle the "memory" M (i.e. the hash table) that replaces $D + G$ in the initial algorithm.

Just browse the HTML documentation (or the `.mli` files) to select the useful functions in these modules, and see how to use them. You can have a look at how they are implemented later on, *once your work is completed.* The other modules (`Problem`, `Draw`) are used in `main.ml` to run the A^* algorithm on the path-finding problem (question 2), and to display the resulting path.

Hints for question 1: open file `astar.ml` with `emacs` and write the code of function `search` implementing algorithm 1:

```
let search user_fun u0 is_goal next k h =


...
```

The `search` function should have several arguments and should comply to the signature of the function given in `astar.mli`. These arguments are listed below:

- `user_fun`, a record containing two functions `do_at_extraction` and `do_at_insertion`. To use these functions:
  - insert `user_fun.do_at_extraction !q m u` just after extracting the current state u from the priority queue `!q` (i.e. between lines 7 and 8 of algorithm 1),
  - insert `user_fun.do_at_insertion u v` just before inserting a new state v in the priority queue.
- `u0` is the initial state (or node, in a graph representation),
- `is_goal` is a function such that `is_goal u` is `true` when `u` is a terminal state, `false` otherwise,
- `next` is a function returning the list of successors of a given state,
- k is a function such that `k u v` is the cost of the path between `u` and `v`.
- h is the heuristic function.

When implementing `search`, you just have to use these arguments, knowing their types described in the signature of `search` in `astar.mli`. If you want to see how `search` is called, you can look at the `main` function in file `main.ml`.