Clumsy Crucible

Roy Ratcliffe

2024-06-20

I recently encountered an amusing "elf" optimisation puzzle called the Clumsy Crucible. "Elves" want to cross a 13-by-13 square of city blocks starting at junction $(1, 1)$ and ending at junction $(13, 13)$. The little elves push a large wheeled crucible across the city blocks. The goal is to optimise the heat loss. The sum of the costs must be minimal. The elves make their merry way from $(1, 1)$ to $(13, 13)$ across the heat-loss grid shown below.

```
2413432311323
3215453535623
3255245654254
3446585845452
4546657867536
1438598798454
4457876987766
3637877979653
4654967986887
4564679986453
1224686865563
2546548887735
4322674655533
```

The elf puzzle is a classic A^* optimisation problem. The possible routes form a graph of nodes. Arcs between nodes have a cost metric. Higher values represent a higher cost of traversing between nodes. The A^* optimiser walks the graph looking for the path with the lowest total cost.

Deriving a solution requires the solving of multiple sub-problems. The following sections take them individually, starting with the grid parser and ending with heuristic tuning. I include a suite of tests, one for each stage for a test-driven approach using Programming Logic.

The solution requires the solving of multiple sub-problems. The following sections take them individually, starting with the grid parser and ending with heuristic tuning. I include a suite of tests, one for each stage for a test-driven approach using Programming Logic.

"Easy" one-liner

Relatively easy, that is. Reading the heat loss grid is the first problem. Prolog is quite clever at parsing in general. It's almost a one-liner. See the grammar below. "Cost" refers to heat loss, a more generic term.

```
:- use_module(library(dcg/high_order), [sequence//2, sequence//5]).
:- use_module(library(dcg/basics), [blank//0, blanks//0]).

cost(Cost) --> [Code], { code_type(Code, digit(Cost)), ! }.
cost(-) --> "-".

cost_grid(CostGrid) -->
    sequence([], sequence(cost), (blank, blanks), [], CostGrid).
```

```
:- begin_tests(cost_grid).
test(it, CostGrid == [[]]) :- phrase(cost_grid(CostGrid), ``).
test(it, CostGrid == [[1, 2, 3], []]) :- phrase(cost_grid(CostGrid), `123\n`).
test(it, CostGrid == [[1, -, 3]]) :- phrase(cost_grid(CostGrid), `1-3`).
:- end_tests(cost_grid).
```

The grid grammar makes use of the Prolog `sequence//5` high-order DCG predicate where

1. the first argument defines how the sequence starts, its grammar;
2. the second argument defines the grammar of the elements;
3. the third argument defines the separator grammar;
4. the fourth argument defines the sequence ending grammar.

Grammar `blank//0` matches a space or newline; `blanks//0` matches zero or more of them.

Predicate `cost//1` describes the grammar of one cost digit: one code, one digit. The cost is the value of the digit. It allows for atom `-` as the cost meaning "not allowed" rather than zero cost. Tests can use this cost to mark up no-go areas within a cost grid. The `cost_grid//1` grammar defines a sequence of cost sequences, a two-dimensional list of lists of single-digit costs, or no-gos.

Note from the tests that the sequence grammar generates a final `[]` empty list for the last row of costs. This does not matter, though it is tempting to clean it up. Nevertheless, for the sake of simplicity, the grid grammar does not validate the resulting list of lists of numbers. Sub-lists correspond to lines. The list can have any length and the sub-lists too. The simple parser merely adds an extra empty row of numbers for the last blank line. We can consider this flexibility an advantage, however. Grids do not require a full population of costs; they can contain blank lines. In general, software development benefits when validation occurs on data acceptance fails and optimisation occurs on performance overruns. In short, fix it when it's broken not before.

Next, access the grid by row and column.

Cost of heat loss

Look up the xth cost within the yth grid row given a grid of costs and an `X-Y` term. See the implementation below with a simple test.

```
cost_grid_from_file(CostGrid, File) :-
    phrase_from_file(cost_grid(CostGrid), File).

cost_grid_from_clumsy_crucible_txt(CostGrid) :-
    cost_grid_from_file(CostGrid, 'clumsy_crucible.txt').

cost_at(CostGrid, X-Y, Cost) :-
    nth1(Y, CostGrid, Costs),
    nth1(X, Costs, Cost),
    Cost \== (-).

:- begin_tests(cost_at).
test(it, Costs == [2, 3]) :-
    cost_grid_from_clumsy_crucible_txt(CostGrid),
    convlist(cost_at(CostGrid), [0-0, 1-1, 13-13], Costs).
:- end_tests(cost_at).
```

The `cost_at/3` predicate does *not* always succeed, i.e. semideterministic; it fails if the grid does not define a cost at x and y or if the grid does not exist at (x, y). Hence, failure is the default non-solution. Also, notice that the coordinates have one-based values; `nth1/3` selects the nth item of a list starting at 1; e.g. `1-13` selects the first item of the thirteen row.

Next, how can the elves go?

Go elves!

The crucible-pushing elves go up, down, left or right. "Right" adds 1 to the x ordinate. "Down" adds 1 to y and so on. See the `go/3` below. Atoms `>`, `v`, `<` and `^` define the four cardinal directions. The predicate defines how the direction atoms relate to (x, y). The elves **cannot** traverse the city grid diagonally. Buildings stand in the way. The implementation utilises `succ/2` for computing and matching successive integers where `succ(N0, N)` unifies $N = N_0 + 1, \forall N \in \mathbb{Z}$.

```
go(X0-Y, >, X-Y) :- succ(X0, X).
go(X-Y0, v, X-Y) :- succ(Y0, Y).
go(X0-Y, <, X-Y) :- succ(X, X0).
go(X-Y0, ^, X-Y) :- succ(Y, Y0).

:- begin_tests(go).
test(it, all(A-B == [(>)-(2-1),v-(1-2),(<)-(0-1),(^)-(1-0)])) :- go(1-1, A, B).
:- end_tests(go).

direction(go(From, To), Direction) :- go(From, Direction, To).

:- begin_tests(direction).
test(it, [A == v, nondet]) :- go(1-1, A, 1-2).
:- end_tests(direction).
```

Next, the elves A^*-traverse the cost grid.

A-star traversal

This is how it works in a nutshell: start with an initial node and some arbitrary score typically zero. Expand the lowest-scoring node. Replace it with expanded nodes. Rinse repeat until one of the expanded nodes matches the final node. Taking the lowest score prefers finding the least-cost route.

Given a *priority heap*, the implementation of A^* becomes simple. See the listing below. You create an initial A^* heap given a starting score and node, see `a_star_initial/3`. The node is any term used to identify a location in a graph. You ask for a "final" node by removing the one with the lowest score, see `a_star_final/4`. The lowest-scoring node sits at the root of the heap. Accessing and removing, if necessary, the lower-scoring node executes very quickly.

```
:- use_module(library(heaps), [singleton_heap/3, get_from_heap/4]).

%!  a_star(+Score0, +Node0, :Goal, -Score, -Node) is nondet.
%
%   Non-deterministically expands an A-star heap until empty using the
%   given non-deterministic Goal to find all expanded nodes.

a_star(Score0, Node0, Goal, Score, Node) :-
    a_star_initial(Score0, Node0, Heap),
    a_star_(Heap, Goal, Score, Node, _).

a_star_(Heap0, _Goal, Score, Node, _Heap) :-
    a_star_final(Heap0, Score, Node, _).
a_star_(Heap0, Goal, Score, Node, Heap) :-
    a_star_expand(Heap0, Goal, Heap1),
    heap_size(Heap1, Size1),
    (   Size1 == 0
```

```
    ->  true
    ;   a_star_(Heap1, Goal, Score, Node, Heap)
    ).

a_star_initial(Score, Node, Heap) :-
    singleton_heap(Heap, Score, Node).

a_star_final(Heap0, Score, Node, Heap) :-
    get_from_heap(Heap0, Score, Node, Heap).

a_star_expand(Heap0, Goal, Heap) :-
    a_star_expand(Heap0, Goal, Nodes, Heap1),
    heaps:list_to_heap(Nodes, Heap1, Heap).

a_star_expand(Heap0, Goal, Nodes, Heap) :-
    get_from_heap(Heap0, Score0, Node0, Heap),
    findall(Score-Node, call(Goal, Score0, Node0, Score, Node), Nodes).
```

Heap expansion appears in two steps. The two-step `a_star_expansion/{3,4}` predicates use the expansion *Goal* to find all the expanded nodes based on a given *Node*. The result is a list of $Score - Node$ pairs. Expansion occurs if the "final" heap does **not** match the final node requirements, i.e. not the node we were searching for. Nothing stops A^* from continuing a search to find alternative routes.

Elven wisdom

The elves are wise. They have to be. They do not want to flail around, going around helplessly in circles. They apply the following rules to their movements.

1. They never return to the same place. That's just sensible.
2. They never go upwards more than once. That would take them too far from the bottom row, wasting time and heat more importantly.
3. They never go leftwards more than once. That would wastefully take them too far from the righthand column where they desire to be.
4. They never go three blocks in the same direction because the crucible gets too hot and they have no mittens.

In logic, these rules translate as follows.

```
elven_star(Initial, Final, CostGrid, Score, Where) :-
    a_star(0, elven(Initial, CostGrid, []),
           elven_expansion,
           Score, elven(Final, _, Where0)),
    reverse(Where0, Where).

elven_expansion(Score0, elven(From, CostGrid, Where),
                Score, elven(To, CostGrid, [go(From, Direction, To)|Where])) :-
    go(From, Direction, To),
    \+ memberchk(go(_, _, To), Where),
    elven_direction(<, Direction, Where),
    elven_direction(^, Direction, Where),
    \+ elven_thrice_in_same_direction(Where, Direction),
    cost_at(CostGrid, To, Cost),
    Score is Score0 + Cost.

elven_direction(Direction, Direction, Where) :-
```

```
    !,
    (   memberchk(go(_, Direction, _), Where)
    ->  fail
    ;   true
    ).
elven_direction(_, _, _).

elven_thrice_in_same_direction(Where, Direction) :-
    maplist(arg(2), Where, [Direction, Direction, Direction|_]).
```

Some points to note:

- The elven nodes are `elven(X-Y, CostGrid, Where)` terms.
- The *Where* argument is a list of `go(From, Direction, To)` terms that grow at the list head for every successful expansion.
- The `elven_star/5` predicate reverses the *Where* list so that its `go/3` terms start at the beginning rather than at the end.

Optimum loss of heat

Run and time the elven quest as follows.

```
?- time((cost_grid_from_clumsy_crucible_txt(CostGrid), elven_star(1-1, 13-13, CostGrid, Score, Where)))
% 313,918,202 inferences, 142.328 CPU in 185.423 seconds (77% CPU, 2205595 Lips)
Score = 102
```

The `elven_star` predicate solves non-deterministically. It delivers the best solution first but thereafter continues searching for other solutions if the user backtracks by pressing semicolon (`;`) to redo the predicate. Indeed, it continues non-deterministically until the A^* heap drains to nothing, producing successively worse but plausible solutions along the way.

```
2>>>43^>>>323
321v>>>53v623
325524565v>54
3446585845v52
4546657867v>6
14385987984v4
44578769877v6
36378779796v>
465496798688v
456467998645v
12246868655<v
25465488877v5
43226746555v>
```

Heat loss of 102 in about three minutes. Clever elves.

Post-mortem

A^* has its limitations. It is **not** a magic bullet. The priority heap quickly grows. Unless the host machine has unlimited amounts of memory, the solution will inevitably hit out-of-memory unless node expansion applies constraints. Expansion requires *some* amount of leeway but within reasonable bounds.

Tweaking the expansion heuristics does the trick. Choose the function wisely. It determines the complexity of the search optimisation. Without some form of smart heuristic, complexity quickly multiplies. The expansion

function must smuggle in just enough knowledge to permit a wise balance between freedom to find a solution and sensible smarts to avoid exploding the available computing resources.

For the elves, the expansion must allow for *some* measure of counter-intuitive backtracking away from the goal. Upwards, but only once. Leftwards, but only once.

The solution runs in a single thread. This is necessary since each node expansion requires access to a common search tree. But it does add performance limitations.

> "Not all those who wander are lost."—J.R.R. Tolkien