Implementing A^*in Prolog

Richard A. O'Keefe

1 October 1997

1 Introduction

The A^*algorithm is a search strategy for searching directed acyclic graphs. It is an instance of the general "informed best-first" approach. An important point about it is that the cost of a solution path is the sum of the costs of the edges in that path.

We are given the following information:

- A *start node* called Start.
- A *heuristic function* called H which estimates the cost of the cheapest path from a given node to a goal node.
- A *predicate* Goal which recognises goal nodes.
- A *node expansion function* Expander which is given a node and returns all the weighted edges emanating from that node; more precisely, it returns a set of (child,cost) pairs.

It matters that $A^*$is searching a DAG, not a tree. That means that we need a way of telling whether we have seen a node before. To be fully general, we should have an equality predicate that recognises whether two terms represent the same node or not, but to keep our code simple we shall assume that the caller uses a representation where there is a one-to-one mapping from problems states to Prolog terms.

The algorithm maintains two sets and two functions:

- The Open set contains nodes we have met and are yet to explore (or perhaps re-explore).
- The Closed set contains all the nodes we have met so far and explored.
- The Parent function maps a node to its best parent so far.
- The Cost function maps a node to its best cost so far.

The following pseudo-code for the algorithm is adapted from the book "Search in AI", edited by Kanal and Kumar. Specifically, it is adapted from the chapter "The Optimality of A^*", by Dechter and Pearl. because

```
Open := {Start}
Closed := {}
while Open is not empty loop
        find a Node in Open minimising Cost(Node)+H(Node)
        (if there are several such nodes, prefer a goal node)
        Open := Open \ {Node}
        Closed := Closed ∪ {Node}
        if Goal(Node) then
                Path := [Node]
                while Parent(Node) is defined loop
                        Node := Parent(Node)
                        Path := [Node|Path]
                end loop
                return Path
        end if
        for each (Child,StepCost) in Expander(Node) loop
                if Child in Open+Closed then
                        if Cost(Node) + StepCost < Cost(Child) then
                                Cost(Child) := Cost(Node) + StepCost
                                Parent(Child) := Node
                                if Child in Closed then
                                        Open := Open ∪ {Child}
                                        Closed := Closed \ {Child}
                                end if
                        end if
                else
                        Parent(Child) := Node
                        Cost(Child) := Cost(Node) + StepCost
                        Open := Open ∪ {Child}
                end if
        end loop
end loop
return failure
```

2 First data structure choices

In order to turn something like this into Prolog, we need to make some representation decisions. There is nothing special about Prolog here; the kinds of decisions we make apply in any programming language. The first thing to note is the key step, where we pick the best candidate from the Open set. Now there

are data structures for picking the best candidate. They are called *priority queues* and are described in all good data structures textbooks. Essentially, a priority queue is a data structure supporting the following operations:

- create an empty priority queue
- insert a (cost,item) pair
- test whether a priority queue is empty
- remove a (cost,item) pair with minimal cost

In this problem the cost is actually the pair (C, G) where C is Cost(Node)+H(Node) and G is 0 for a goal node, 1 for a non-goal. This is how we break ties in favour of goal nodes.

So we'll actually want the Open set to be a priority queue with $((C, G)$,Node) elements. But a priority queue is optimised for extracting the best element; it is not at all good at testing whether something is already an element. So we are actually going to represent Open and Closed redundantly, using three variables:

- Candidates is a priority queue, representing Open
- Closed is a set, representing itself
- AllNodes is a set, representing the union of Open and Closed.

We do have a problem here. When the Cost of a node changes, the corresponding element of the priority of the node should also change if it is in Candidates. This is actually rather nasty, because we require a new operation:

- reduce the priority of an existing element

which requires us to *find* the elment, and priority queues in general are not good at that. Typical priority queue implementations in imperative languages handle this by returning a pointer when they add an element to a priority queue, and asking you to give the pointer back when you want to change the priority.

Let's recast the algorithm in terms of these variables.

```
Priority := (H(Start), !Goal(Start))
Candidates := {Start ↦ Priority}
Closed := {}
AllNodes := {Start}
while Candidates is not empty loop
        remove the best ((C,G),Node) element from Candidates
        Closed := Closed ∪ {Node}
        if G = 0 then
                Path := [Node]
                while Parent(Node) is defined loop
```

```
                Node := Parent(Node)
                Path := [Node|Path]
            end loop
            return Path
        end if
        for each (Child,StepCost) in Expander(Node) loop
            if Child in AllNodes then
                if Cost(Node) + StepCost < Cost(Child) then
                    Cost(Child) := Cost(Node) + StepCost
                    Parent(Child) := Node
                    Priority := (Cost(Child) + H(Child), !Goal(Child))
                    if Child in Closed then
                        Candidates := Candidates ∪ {Child ↦ Priority}
                        Closed := Closed \ {Child}
                    else
                        change Child in Candidates to Priority
                    end if
                end if
            else
                Parent(Child) := Node
                Cost(Child) := Cost(Node) + StepCost
                Priority := (Cost(Child) + H(Child), !Goal(Child))
                Candidates := Candidates + (Priority,Child)
            end if
        end loop
    end loop
    return failure
```

3 The Prolog implementation

I am going to present this package in top down fashion, although I actually wrote most of it bottom up. Our inspection of the algorithm has told us what operations we want to perform on the major variables.

For Closed, we need

- create empty set
- insert element known not to be there already
- try to delete element

For AllNodes, we need

- create empty set
- insert element X, where Goal(X) and H(X) are known

- check possible element X, where it is useful to know Goal(X) and H(X) afterwards.

For Candidates, we need

- create empty priority queue
- add an entry for an item not already in the queue
- remove an entry with minimal priority
- change the priority of an item already in the queue

Top down practice would have us postpone the implementation of those operations until we have written the code that uses them. Here I only describe the interfaces:

The interface for sets is modelled on the corresponding operations in the Edinburgh/Quintus library(ordsets) package.

⟨interface for sets 5a⟩ ≡

⟨types for sets 11a⟩

```
:- pred bstset_empty(      ?Set : bstset(K)).
:- pred bstset_add_element(+Set0: bstset(K), +Elt: K,
                           -Set : bstset(K)).
:- pred bstset_selectchk(  +Elt : K, +Set0: bstset(K),
                           -Set : bstset(K)).
```

◇

Macro defined by scraps 5a, 13.
Macro referenced in scrap 6b.

The interface for finite maps is modelled on the corresponding operations in the Edinburgh/Quintus library(assoc) package.

⟨interface for finite maps 5b⟩ ≡

⟨types for finite maps 14a⟩

```
:- pred bstmap_empty(?Map : bstmap(K,V)).
:- pred bstmap_put(  +Map0: bstmap(K, V), +Key: K, +Val: V,
                     -Map : bstmap(K, V)).
:- pred bstmap_get(  +Map : bstmap(K, V), +Key: K, ?Val: V).
```

◇

Macro defined by scraps 5b, 15b.
Macro referenced in scrap 6b.

The interface for priority queues is modelled on the corresponding operations in the Edinburgh/Quintus library(heaps) package, although that package has never supported changing the priority of an existing entry.

⟨interface for priority queues 6a⟩ ≡

```
⟨types for priority queues 17a⟩

:- pred dlpq_empty(     ?PQ : dlpq(I,P)).
:- pred dlpq_add_entry( +PQ0: dlpq(I,P), +Item: I, +Priority: P,
                        -PQ : dlpq(I,P)).
:- pred dlpq_remove_min(+PQ0: dlpq(I,P), ?Item: I, ?Priority: P,
                        -PQ : dlpq(I,P)).
:- pred dlpq_alter_prio(+PQ0: dlpq(I,P), +Item: I, +Priority: P,
                        -PQ : dlpq(I,P)).
◇
```

Macro referenced in scrap 6b.

Our Prolog file is going to follow the usual format. There will be an informative header, listing all the exported predicates of the module. There will be type declarations for all the types and predicates used. The module interface and types will actually be commented out, because NU Prolog does not support Quintus module declarations, nor does it exactly support DEC-10 Prolog type declarations. Then we'll have our A^*code, and finally we'd have the support code. The support predicates would normally be in other modules.

"astar.pl" 6b ≡

```
%   Package: astar
%   Author : Richard A. O'Keefe
%   Updated: %G%
%   Defines: astar/5
/*
:- module(astar, [astar/5]).

:- pred astar(+Start: T, +Goal: void(T), +H: void(T,number),
              +Expander: void(T,list(pair(T,number)))).

⟨interface for sets 5a, ... ⟩
⟨interface for finite maps 5b, ... ⟩
⟨interface for priority queues 6a⟩
⟨private declarations of astar 10⟩
*/

⟨implementation of astar 7b, ... ⟩
⟨implementation of sets 11b⟩
⟨implementation of finite maps 14b⟩
⟨implementation of priority queues 17b⟩
◇
```

4 Closures

One extremely important thing we need to design is the interface of the predicate. I have been asked to make it look like the Lisp version in `/public/320/search/graph.cl`. The Lisp version accepts functions (more properly, closures) as arguments. Lisp is a functional language, and it is very good with closures. Prolog is a relational language, and is a bit clumsier. Some Prolog systems follow the lead of the DEC-10 Prolog type checker (and the DEC-10 Prolog library) in providing built-in or library predicates

call($p(X_1, \ldots, X_m)$, Y_1, …, Y_n) :-
$p(X_1, \ldots, X_m, Y_1, \ldots, Y_n)$.

If you have to deal with a Prolog system that does not support these operations, you can implement call/2 and call/3, which we need, using this less efficient but more portable code.

⟨unused code 7a⟩ ≡

```
call(Closure, X1) :-
    Closure =.. [Pred|Args0],
    append(Args0, [X1], Args),
    Goal =.. [Pred|Args],
    call(Goal).

call(Closure, X1, X2) :-
    Closure =.. [Pred|Args0],
    append(Args0, [X1,X2], Args),
    Goal =.. [Pred|Args],
    call(Goal).
```

◇

Macro never referenced.

5 The A^* predicate

Recall our declaration of astar/5:

```
:- pred astar(+Start: T, +Goal: void(T), +H: void(T,number),
              +Expander: void(T,list(pair(T,number)))).
```

The code actually follows its imperative model very closely. Whenever there was a loop, we have a tail recursive predicate, which has to carry around the state variables that it updates and the non-local parameters that it uses. The derivation of this code from the pseudo-code was pretty mechanical.

⟨implementation of astar 7b⟩ ≡

```
astar(Start, Goal, H, Expander, Path) :-
    call(H, Start, HVal),
    ( call(Goal, Start) -> IsGoal = 0 ; IsGoal = 1 ),
    dlpq_empty(Candidates0),
    dlpq_add_entry(Candidates0, Start, (HVal,IsGoal), Candidates1),
    bstset_empty(Closed0),
    bstmap_empty(AllNodes0),
    bstmap_put(AllNodes0, Start, (HVal,IsGoal), AllNodes1),
    bstmap_empty(Current0),
    astar_loop(Candidates1, Closed0, AllNodes1, Current0,
                      Goal, H, Expander, Path).

astar_loop(
        /*state*/ Candidates0, Closed0, AllNodes0, Current0,
        /*functions*/ Goal, H, Expander,
        /*result*/ Path
) :-
    dlpq_remove_min(Candidates0, Node, (Estimate,IsGoal), Candidates1),
    (   IsGoal =:= 0 ->
        astar_result(Current0, Node, [], Path)
    ;/* IsGoal =:= 1 (is not a goal) */
        call(Expander, ChildrenWithCosts),
        astar_update(ChildrenWithCosts, Node, Estimate,
                            Candidates1, Closed0, AllNodes0, Current0,
                            Goal, H, Expander, Path)
    ).
```

◇

Macro defined by scraps 7b, 8, 9.
Macro referenced in scrap 6b.

This is clumsy because we are passing around a lot of variables. They are the state variables Candidates, Closed, AllNodes, and Current, the functions Goal, H, and Expander, and the result Path.

The Current function is undefined for the Start node, which is how we know when we've traced the Path back to the beginning.

⟨implementation of astar 8⟩ ≡

```
astar_result(Current, Node, Path0, Path) :-
    Path1 = [Node|Path0],
    (   bstmap_get(Current, Node, (_,Parent)) ->
        astar_result(Current, Parent, Path1, Path)
    ;/* Node must be Start because it has no Parent */
        Path = Path0
    ).
```

◇

Macro defined by scraps 7b, 8, 9.

Macro referenced in scrap 6b.

We use recursion to iterate over the elements of the list of edges. There are a number of if-then-elses here, and we have to be careful that all state variables are updated, even when they do not change.

⟨implementation of astar 9⟩ ≡

```
astar_update([], _, _,
        Candidates, Closed, AllNodes, Current,
        Goal, H, Expander, Path
) :-
    astar_loop(Candidates, Closed, AllNodes, Current,
                      Goal, H, Expander, Path).

astar_update([(Child,StepCost)|ChildrenWithCosts], Node, NodeCost,
        Candidates0, Closed0, AllNodes0, Current0,
        Goal, H, Expander, Path
) :-
    NCost is NodeCost + StepCost,
    (   bstmap_get(AllNodes0, Child, (HVal,IsGoal)) ->
        bstmap_get(Current0,  Child, (OCost,_)),
        (   NCost < OCost ->
            NewEstimate is NCost + HVal,
            bstmap_put(Current0, Child, (NCost,Node), Current1),
            (   bstset_selectchk(Closed0, Child, Closed1) ->
                dlpq_add_entry(Candidates0, Child, (NewEstimate,IsGoal),
                               Candidates1)
            ;   Closed1 = Closed0,
                dlpq_alter_prio(Candidates0, Child, (NewEstimate,IsGoal),
                                Candidates1)
            )
        ;/* NCost >= OCost */
            Current1 = Current0,
            Closed1 = Closed0,
            Candidates1 = Candidates0
        ),
        AllNodes1 = AllNodes0
    ;/* this is a completely new node */
        call(H, Child, HVal),
        ( call(Goal, Child) -> IsGoal = 0 ; IsGoal = 1 ),
        NewEstimate is NCost + HVal,
        bstmap_put(Current0, Child, (NCost,Node), Current1),
        bstmap_put(AllNodes0, Child, (HVal,IsGoal), AllNodes1),
        dlpq_add_entry(Candidates0, Child, (NewEstimate,IsGoal),
                       Candidates1),
        Closed1 = Closed0
    ),
```

```
        astar_update(ChildrenWithCosts, Node, NodeCost,
                         Candidates1, Closed1, AllNodes1, Current1,
                         Goal, H,  Expander, Path).
```

◇

Macro defined by scraps 7b, 8, 9.
Macro referenced in scrap 6b.

For internal type checking, we need

⟨private declarations of astar 10⟩ ≡

```
:- type apq(T) == dlpq(T, (number,number)).
:- type amap(T) == bstmap(T, (number,number)).
:- type fmap(T) == bstmap(T, (number,T)).

:- pred
    astar_loop(apq(T), bstset(T), amap(T), fmap(T),
               void(T), void(T,number), void(T,list((T,number)))),
               list(T)),
    astar_update(list((T,number)), T, number,
                 apq(T), bstset(T), amap(T), fmap(T),
                 void(T), void(T,number), void(T,list((T,number)))),
    astar_result(fmap(T), T, list(T), list(T)).
```

◇

Macro referenced in scrap 6b.

6 Implementing sets

We have decided to implement the Closed set as a (simple) binary search tree. This is a tradeoff between speed and simplicity. The simplest method would be to use an unordered list, but that is very slow. The best method might be to use some kind of balanced search tree, or perhaps a splay tree. Binary search trees are a good compromise.

The obvious way to represent a binary search tree using Prolog terms is

```
:- type bst(K) --> empty | nonempty(K,bst(K),bst(K)).
```

where an `empty` term represents an empty set and a `nonempty(K,L,R)` term represets the the set $L' \cup \{K\} \cup R'$ where `L` represents L' and `R` represents R'.

The only problem with that representation is its space cost. The rule I use for estimating the space cost of Prolog data structures is 2 words per list element and $n+1$ words per term with functor f/n. To represent a set with N elements, a list would take $2N$ words, but this representation takes $4N$ words. A good rule of thumb for estimating the speed of Prolog code is to count the number of words touched. Now in a typical binary search tree, about half of the nodes are leaves. We can save space by not explicitly spelling out that the children of a leaf are empty. If we do that, the average cost will be something like $3N$ words.

So the representation we are going to use is

⟨types for sets 11a⟩ ≡

```
:- type bstset(K) -->
    empty         % represents the empty set
 |  leaf(K)       % represents {K}
 |  fork(K,L,R).  % represents L' U {K} U R'
```

◇

Macro referenced in scrap 5a.

Recall that the operations we want are

⟨implementation of sets 11b⟩ ≡

⟨make an empty set 11c⟩
⟨add a new element to a set 11d⟩
⟨try to remove an element from a set 12a, ... ⟩

◇

Macro referenced in scrap 6b.

Making an empty set is trivial.

⟨make an empty set 11c⟩ ≡

```
bstset_empty(empty).
```

◇

Macro referenced in scrap 11b.

Adding an element is more interesting. We use a common Prolog pattern: check the principal functor of the term, then compare a search item against a key and do a three-way dispatch. We have assumed in the design of astar/5 that nodes are represented by ground terms, so we can use Prolog term comparison safely on them.

⟨add a new element to a set 11d⟩ ≡

```
bstset_add_element(empty, Key, leaf(Key)).
bstset_add_element(leaf(K), Key, Result) :-
    compare(O, Key, K),
    bstset_add_element_case(O, Key, Result, K, empty, empty).
bstset_add_element(fork(K,L,R), Key, Result) :-
    compare(O, Key, K),
    bstset_add_element_case(O, Key, Result, K, L, R).

bstset_add_element_case(<, Key, fork(K,L1,R), K, L, R) :-
    bstset_add_element(L, Key, L1).
bstset_add_element_case(>, Key, fork(K,L,R1), K, L, R) :-
    bstset_add_element(R, Key, R1).
bstset_add_element_case(=, Key, fork(Key,L,R), _, L, R).
```

◇

Macro referenced in scrap 11b.

Trying to remove an element from a set is the only really tricky operation. When we remove the key from a subtree with children, we have to rejoin its children to make one tree. We want to do this without making the tree any deeper, so we remove the smallest element from the right subtree and make that the root of the new tree. If the element we are trying to remove is not present, this predicate quietly fails without requiring any special action.

⟨try to remove an element from a set 12a⟩ ≡

```
bstset_selectchk(Element, Set0, Set) :-
    bstset_selectchk_loop(Set0, Element, Set).

bstset_selectchk_loop(leaf(Key), Key, empty).
bstset_selectchk_loop(fork(K,L,R), Key, Result) :-
    compare(O, Key, K),
    bstset_selectchk_case(O, Key, Result, K, L, R).

bstset_selectchk_case(<, Key, fork(K,L1,R), K, L, R) :-
    bstset_selectchk_loop(L, Key, L1).
bstset_selectchk_case(>, Key, fork(K,L,R1), K, L, R) :-
    bstset_selectchk(R, Key, R1).
bstset_selectchk_case(=, _, Result, _, L, R) :-
    (   R == empty ->
        Result = L
    ;   L == empty ->
        Result = R
    ;   bstset_remove_min(R, K, R1),
        Result = fork(K,L,R1)
    ).
```

◇

Macro defined by scraps 12ab.
Macro referenced in scrap 11b.

This needs an auxiliary predicate which would be part of the exported interface of a general set package.

⟨try to remove an element from a set 12b⟩ ≡

```
bstset_remove_min(leaf(K), K, empty).
bstset_remove_min(fork(K,L,R), Min, Rest) :-
    bst_remove_min_case(L, Min, Rest, K, R).

bstset_remove_min_case(empty, Min, Rest, Min, Rest).
bstset_remove_min_case(leaf(Min), Min, fork(K,empty,R), K, R).
bstset_remove_min_case(fork(K1,L1,R1), Min, fork(K,L,R), K, R) :-
    bstset_remove_min_case(L1, Min, L, K1, R1).
```

◇

Macro defined by scraps 12ab.
Macro referenced in scrap 11b.

All that's left for this section is describing the types of the local predicates. There are type checkers for Prolog: DEC-10 Prolog and Quintus Prolog have one; NU Prolog has three. Prolog doesn't normally use types, and there are several rival type systems. Therefore the types are commented out.

⟨interface for sets 13⟩ ≡

```
:- pred
    bstset_add_element_case(order, K, bstset(K),
                            K, bstset(K), bstset(K)),
    bstset_selectchk_loop(bstset(K), K, bstset(K)),
    bstset_selectchk_case(order, K, bstset(K),
                          K, bstset(K), bstset(K)),
    bstset_remove_min(bstset(K), K, bstset(K)),
    bstset_remove_min_case(bstset(K), K, bstset(K),
                           K,bstset(K)).
```

◇

Macro defined by scraps 5a, 13.
Macro referenced in scrap 6b.

In practice, you would not write these predicates yourself, but would find them in a library. The library might have been supplied by the Prolog vendor, as Quintus and SICS do, or it might be a local library. It is also possible to implement balanced binary search trees in Prolog, and there again, your Prolog library should already contain such routines. It would be quite reasonable to use balanced binary trees for these data structures. Some kind of hash table would be even better, but that would require a hashing function for nodes, which is not in our interface. Term comparison is defined for all ground terms.

7 Implementing finite maps

We have decided to implement the AllNodes set as a (simple) binary search tree with satellite data. We are also going to implement the Current function the same way. A binary search tree with satellite data is just like an ordinary binary search tree, except that the nonempty nodes have an Info field as well as a Key field, and some of the predicates have an Info argument as well.

Although it complicates the code, distinguishing leaf from fork nodes is still a worthwhile space saving. So the representation we are going to use is

⟨types for finite maps 14a⟩ ≡

```
:- type bstmap(K,V) -->
    empty          % represents the empty map
 |  leaf(K,V)      % represents {K:->V}
 |  fork(K,V,L,R). % represents L' U {K:->V} U R'
```
◇

Macro referenced in scrap 5b.

Recall that the operations we want are

⟨implementation of finite maps 14b⟩ ≡

⟨make an empty map 14c⟩
⟨put a new or replacement maplet in 14d⟩
⟨get the value associated with a key 15a⟩
◇

Macro referenced in scrap 6b.

Making an empty finite map is trivial.

⟨make an empty map 14c⟩ ≡

```
bstmap_empty(empty).
```
◇

Macro referenced in scrap 14b.

Adding or replacing a maplet is just the same as adding an element to a set, except for the Info argument.

⟨put a new or replacement maplet in 14d⟩ ≡

```
bstmap_put(empty, Key, Info, leaf(Key,Info)).
bstmap_put(leaf(K,V), Key, Info, Result) :-
    compare(O, Key, K),
    bstmap_put_case(O, Key, Info, Result, K, V, empty, empty).
bstmap_put(fork(K,V,L,R), Key, Info, Result) :-
    compare(O, Key, K),
    bstmap_put_case(O, Key, Info, Result, K, V, L, R).

bstmap_put_case(<, Key, Info, fork(K,V,L1,R), K, V, L, R) :-
    bstmap_put(L, Key, Info, L1).
bstmap_put_case(>, Key, Info, fork(K,V, L,R1), K, V, L, R) :-
    bstmap_put_case(R, Key, Info, R1).
bstmap_put_case(=, Key, Info, fork(Key,Info,L,R), _, _, L, R).
```
◇

Macro referenced in scrap 14b.

This time we don't have to worry about deleting. Lookup is even easier than adding.

⟨get the value associated with a key 15a⟩ ≡

```
bstmap_get(leaf(Key,Info), Key, Info).
bstmap_get(fork(K,V,L,R), Key, Info) :-
    compare(O, Key, K),
    bstmap_get_case(O, Key, Info, K, V, L, R).

bstmap_get_case(<, Key, Info, _, _, L, _) :-
    bstmap_get(L, Key, Info).
bstmap_get_case(>, Key, Info, _, _, _, R) :-
    bstmap_get(R, Key, Info).
bstmap_get_case(=, Key, Info, Key, Info, _, _).
```

◇

Macro referenced in scrap 14b.

All that's left for this section is describing the types of the local predicates.

⟨interface for finite maps 15b⟩ ≡

```
:- pred
    bstmap_put_case(order, K, V, bstmap(K,V),
                    K, V, bstmap(K,V), bstmap(K,V)),
    bstmap_get_case(order, K, V,
                    K, V, bstmap(K,V), bstmap(K,V)).
```

◇

Macro defined by scraps 5b, 15b.
Macro referenced in scrap 6b.

In practice, you would not write these predicates yourself, but would find them in a library. The library might have been supplied by the Prolog vendor, as Quintus and SICS do, or it might be a local library. It is also possible to implement balanced binary search trees in Prolog, and there again, your Prolog library should already contain such routines. It would be quite reasonable to use balanced binary trees for these data structures. Some kind of hash table would be even better, but that would require a hashing function for nodes, which is not in our interface. Term comparison is defined for all ground terms.

8 Representing the functions

Whether a Node in AllNodes is a Goal or not, and what its H estimate is, are invariant properties. We might as well keep them in AllNodes to save recomputation. But there are two properties that change: Parent and Cost. We note that they both change together; we never change one without changing the

other. That suggests that we keep a Current function that maps a Node to a (Parent,Cost) pair.

The finite maps described in the previous section are all we need.

Current will initially be empty, because the Start node has no parent and cost 0. We can never find a cheaper route to the Start than that!

9 Implementing priority queues

The operations we have to implement are

- make an empty priority queue
- add an entry for a node not already in the queue
- remove an entry with minimal priority
- change the priority of a node

This would be very easy without the last operation. There are many different ways to implement priority queues: heaps, deaps, treaps, leftist trees, pagoda trees, binomial heaps, Fibonacci heaps, and more. Some of them support an efficient "change priority" operation, but none of them implement a "find item" operation. That means that when you add an item to a priority queue, you get back something which acts like a pointer, and you pass that back when you want to change the priority, or even delete the item. That doesn't really fit well into Prolog. Some day I may think of a clever way of doing it.

An obvious way to implement a priority queue with change is to implement a simple priority queue and then just do a brute force search to find the node to change, and somehow repair the priority queue data structure invariant. I was given very short notice for preparing this package, so I have chosen a simple scheme that should do about $O(\log N)$ for normal insertions and deletions and $O(N)$ for changes.

This is the scheme. We represent a priority queue Q by a list $[L_1, \ldots, L_m]$ where

$$(\forall 1 \leq i < j \leq n)(\forall x \in L_i)(\forall y \in L_j)\, x \leq y$$

(the sublists form an ascending partition of Q) and

$$(\forall 1 \leq i \leq n)(\exists H_i)(\exists T_i)\, L_i = [H_i|T_i] \wedge (\forall z \in T_i)\, H_i \leq z$$

(each sublist is non-empty and its first element is its smallest). These conditions guarantee that a non-empty priority queue is represented by a term $[[H_1, \ldots], \ldots]$ where H_1 is the minimum element of the priority queue.

It turns out not to matter very much whether we think of a priority queue of simple elements, or a priority queue of (Priority,Item) pairs. Accordingly, we have

⟨types for priority queues 17a⟩ ≡

```
:- type dlpq(K) == list(list(K)).
:- type dlpq(I,P) == dlpq(pair(P,I)).
```

◇

Macro referenced in scrap 6a.

⟨implementation of priority queues 17b⟩ ≡

⟨make an empty priority queue 17c⟩
⟨remove the smallest element 17d⟩
⟨insert a new element 18a⟩
⟨change an existing element 18b⟩
◇

Macro referenced in scrap 6b.

As usual, making an empty structure is trivial.

⟨make an empty priority queue 17c⟩ ≡

```
dlpq_empty([]).
```

◇

Macro referenced in scrap 17b.

Removing the smallest element is more interesting. *Finding* the first element is trivial, by design. The trick is to repair the data structure invariants. If T_1 is empty, we just pop the first sublist off. If T_1 is not empty, we use the 'partition' operation of quicksort just often enough to find the smallest element. We pick an arbitrary element of T_1 and pivot around it, getting S, X, and G, where X is our pivot, S is the smaller elements, and G is the greater or equal elements. If S is empty, we've found the new minimum. If S is not empty, we can push $[X|G]$ back onto the front of the priority queue and keep looking for the minimum in S.

⟨remove the smallest element 17d⟩ ≡

```
dlpq_remove_min(Q0, I, P, Q) :-
    dlpq_remove_min(Q0, P-I, Q).

dlpq_remove_min([[Min|T]|L], Min, Q) :-
    dlpq_restore(T, L, Q).

dlpq_restore([], Q, Q).
dlpq_restore([X|T], L, Q) :-
    dlpq_partition(T, X, S, G),
    dlpq_restore(S, [[X|G]|L], Q).
```

```
dlpq_partition([], _, [], []).
dlpq_partition([Y|Ys], X, S, G) :-
    (   Y @< X -> S = [X|S1], dlpq_partition(Ys, X, S1, G)
    ;/* Y @>=X */ G = [X|G1], dlpq_partition(Ys, X, S, G1)
    ).
```
◇

Macro referenced in scrap 17b.

To insert a new element X, try each sublist in turn until you find $X \leq H_i$, and add X at the front of that sublist. If there is no such H_i, either add X as the *second* element of the last sublist, or if the priority queue is empty, add it as a new sublist. The interesting point here is that *inserting* an item never increases the length of the top level of a non-empty priority queue; only *removing* does that.

⟨insert a new element 18a⟩ ≡

```
dlpq_add_entry(Q0, I, P, Q) :-
    dlpq_add_entry(Q0, P-I, Q).

dlpq_add_entry([], X, [[X]]).
dlpq_add_entry([S|L], X, Q) :-
    dlpq_add_entry_loop(S, X, Q, L, S).

dlpq_add_entry_loop([H|T], X, Q, L, S) :-
    (   X @< H -> Q = [[X|S]|L]
    ;   L == [] -> Q = [[H,X|T]]
    ;   L = [S1|L1], Q = [S|Q1],
        dlpq_add_entry_loop(S1, X, Q1, L1, S1)
    ).
```
◇

Macro referenced in scrap 17b.

Changing is tricky. We actually implement it in two stages: delete the old entry completely, and insert a new entry. It is clearly possible to do better, but in the worst case we will still have to do $O(N)$ work, because we have no clue where the entry is. This is the only operation where it matters that the entries are (Priority,Item) pairs, because we are given an Item and not its old Priority.

⟨change an existing element 18b⟩ ≡

```
dlpq_alter_prio(Q0, I, P, Q) :-
    dlpq_delete_item(Q0, I, Q1),
    dlpq_add_entry(Q1, I, P, Q).

⟨other deletion support 19a, ... ⟩
```
◇

Macro referenced in scrap 17b.

To delete an entry, we could use the commonly available predicates

```
select(L0, Q0, L, Q),
select(_-I, L0, L)
```

but for one thing: if we delete the *first* element of a sublist, we have to repair the data structure invariant by finding the new minimum of that sublist. The neat thing is that we already have a way of doing that: given a list we can find its minimum and repair the priority queue using dlpq_restore/3, which we derived for removal. Repeated removals will have a tendency to break the queue into lots of little sublists, which is bad. We could have a tree of sublists, which would make insertion fast again, but if you have a lot of removals, you will be spending a lot of time on the removals, so there isn't a lot of point in worrying.

⟨other deletion support 19a⟩ ≡

```
dlpq_delete_item([S1|L1], I, Q) :-
    S1 = [H1|T1],
    H1 = _-P1,
    (   P1 = P ->
        dlpq_restore(T1, L1, Q)
    ;   select(_-P, T1, T2) ->
        Q = [[H1|T2]|L1]
    ;   Q = [S1|Q1],
        dlpq_delete_item(L1, I, Q1)
    ).
```

◇

Macro defined by scraps 19ab.
Macro referenced in scrap 18b.

It is possible that your Prolog system might be lacking the widely used select/3 predicate. In case it is, here's the code:

⟨other deletion support 19b⟩ ≡

```
select(X, [X|R], R).
select(X, [H|T], [H|R]) :-
    select(X, T, R).
```

◇

Macro defined by scraps 19ab.
Macro referenced in scrap 18b.

10 Index of predicate symbols

Unfortunately, the literate programming tool used to write this document doesn't really understand Prolog, so the cross reference below is based on predicate *symbols* only instead of predicate *functors*.