Abstract. Automatic program generation using logic programming can be used to improve existing problem solving programs. An important class of problems in AI are optimal path-finding problems. These problems are usually solved using the IDA* algorithm with an admissible heuristic. An heuristic is admissible if it never overestimates the cost of solving a problem. An admissible heuristic is better than another one if it always gives higher results, the better the heuristic, the fewer nodes are developed for solving the problem. We propose a metalogic programming framework that specializes heuristics on abstract representation of problems. The specialized heuristics are improvements on the original heuristic. Some experiments in simple path-finding problems like the 9-puzzle give encouraging results.

1 Introduction

Problems like the 9-puzzle, the Rubik's cube [12] or Sokoban [9] are path-finding problems. They belong to a general class of problems related to heuristic single-agent search techniques. These problems are solved building a decision tree in order to find the best of several alternative by searching. They are related to perception problems, theorem proving, robot control, pattern recognition, knowledge based systems and some combinatorial optimization problems.

Finding a solution path is easy for the puzzle and the Rubik's cube, using macro-moves. However, finding the shortest path to the desired state is much harder. The algorithm of choice for this kind of problems is Iterative Deepening A* (IDA*). IDA* has to compute an admissible heuristic at each node. An heuristic is admissible if it never overestimates the distance to the desired state. The hard problem when writing an optimal path-finding problem solver is to find a good admissible heuristic. A commonly used heuristic is the Manhattan distance.

We propose a logic program specialization framework that operates on an abstract domain theory in order to improve existing heuristics. This framework is based on the Introspect system that has already been used to generate powerful game programs using logic metaprogramming. To generate programs, Introspect
uses a theory of the problem to be solved expressed in Prolog, and some metaknowledge on the problem used to remove useless generated programs, and to improve the efficiency of the useful generated programs.

The second section describes our path-finding problem solver. The third section uncovers a way to specialize path-finding heuristics with Introspect and gives experimental results.

2 An Optimal Path-finding Problem Solver

We use the 9-puzzle to test our system. The goal state of the 9-puzzle is represented on the left of figure 1. On the right of the same figure, a randomly generated problem is given. Our problem are generated by playing 100 random moves from the goal state. A* computes two functions at each node of its search: g and h. The g function gives the cost of the moves already played, in the case of the 9-puzzle it is the number of moves. The h function gives an underestimation of the cost of the remaining moves to reach the goal state. A commonly used heuristic is the Manhattan heuristic. It consists in computing for each tile the minimal number of moves necessary to move it to its goal location, with the hypothesis that the tile can move on other tiles.

![Fig. 1. The goal state, and a randomly generated state 20 moves away from the goal state](image-url)

In our example, we therefore have \( h = 1 + 0 + 1 + 3 + 2 + 3 + 1 + 1 = 12 \) with the Manhattan heuristic. At each node a function \( f = g + h \) is computed that represent the minimal cost of the path going through that node. IDA* is an iterative deepening A*. It begins with developing a tree of maximum depth 1, if the solution is not found, it develops a tree of maximum depth 2, and so on, increasing the maximum depth after each unsuccessful tree search. The advantage of IDA* on A* is that it uses an amount of memory that increases linearly with the depth of the problem, whereas A* has exponential requirements and cannot solve complex problems. Another advantage of IDA* is that information can be obtained from previous searches to speed it up [18]. Moreover IDA* is not much more time consuming than A*, be-
Dh is added to the result of the Manhattan heuristic in order to improve it. In a more general way, we can define a direct conflict between two tiles with the following logical rule:

\[
\text{conflict}(0, T1, T2, 2):\neg
\text{tile_on_location}(L1, T1),
\text{tile_on_location}(L2, T2),
\text{all_neighbors_increase_except}(T1, L1, L2),
\text{all_neighbors_increase_except}(T2, L2, L1).
\]

The signification of the argument of the head predicate are conflict (Regression, Tile1, Tile2, Dh), and the predicate all_neighbors_increase_except (T1,L1,L2) indicates that the Manhattan heuristic increases for all the neighbors of tile T1 on location L1, except for location L2 where it decreases.

The specialized heuristic consists in computing all the possible Dh for each pair of tiles, and then in counting the maximal Dh for each tile, taking into account no more than one Dh for each tile, so that the specialized heuristic is still admissible. The resulting set of Dh is then summed, and the result is added to the h resulting from the Manhattan heuristic.

\textbf{Fig. 3.} A regressed conflict between two tiles

A specialization of this conflict can be obtained by unmoving
cient programs. Moreover, the termination of unfolding can be tailored to a particular problem rather than using the same strategy for every program. The goal of the program generation is to express the same knowledge in a different way so that similar computations are shared, and that useless computations are avoided.

The domain theory used to specialize an admissible heuristic is particular in the sense that it is not a theory of the real moves played in the problem. It is rather a theory of the abstract moves that can be played. The abstract moves keep the admissibility of the heuristic because they always underestimate the number of real moves necessary to perform the action. In the 9-puzzle, an abstract move consists in moving a tile on any of its neighbors, providing that the neighbor does not contain the other conflicting tile. In practice, a tile can only move on one of its neighbor if it is empty.

The clause used to generate the program by specialization is a recursive one that defines conflict regression, P being the depth of regression of the conflict between T1 and T2:

\[
\text{conflict}(P,T1,T2,Dh):- \\
P1 \text{ is } P-1, \quad P1 > -1, \\
\text{abstract_moves}(T1,T2,L), \\
\text{moves_increase_h_or_conflict}(P1,T2,L).
\]

The end of the unfolding process is assured because the maximum depth of regression is fixed in advance. The abstract moves are defined by clauses of the type:

\[
\text{abstract_moves}(T1,T2,[M1,M2]):- \\
\text{tile_on_location}(L1,T1), \\
\text{tile_on_location}(L2,T2), \\
\text{number_neighbors}(L,2), \\
\text{neighbor}(L,M1), \quad M1\neq L2, \\
\text{neighbor}(L,M2), \quad M1\neq M2, \quad M2\neq L2.
\]

Once the program is unfolded, the conditions of the unfolded clauses are ordered using domain dependent knowledge. They are then collected together in a tree of conditions. This tree is compiled into C, so as to be linked to the problem solver. This is one of the reasons why the approach works: instead of computing many times the same things, the specialized program shares the computations in the tree of conditions.

3.3 Results

Our test set contains 100 randomly generated 9-puzzle problems. All of them are optimally solved with 24 moves or less. During problem solving we compute the number of nodes developed by IDA* on each problem. The number of nodes is only an approximation of the real efficiency of a problem solver. However, it is independent of a particular implementation and it gives insights on the possible improvements due to specialization on other more difficult problems.
The first number is the number of nodes using only the Manhattan heuristic, the second one using the direct conflict heuristic and the third one using both the direct and the regressed conflict heuristic.

<table>
<thead>
<tr>
<th># Nodes</th>
<th>5237441</th>
<th>2448508</th>
<th>1887999</th>
</tr>
</thead>
</table>

Fig. 4. Number of nodes developed by IDA* with increasing regressions

On the 9-puzzle it of no use to regress the conflict heuristic further because of some particularities of the problem. However, on the 15-puzzle, the heuristic can be specialized one step further, and on more complex problem like Sokoban, it can be regressed much more.

4 Conclusion

We have presented a technique that uses a kind of logic program generation to specialize admissible heuristics for path-finding problems.

It is of interest to apply this technique to more complex path-finding problems such as the Rubik's cube or Sokoban. This approach can be compared to other knowledge generation approaches like retrograde analysis of patterns [5]. The advantage of the representation of heuristics by a program is that abstract knowledge of the domain can be easily represented. This abstract information might be more powerful than usual pattern-based representation in that it enables flexible and non-local properties to be matched together (for example two stones separated by a long tunnel in Sokoban form a deadlock that does not fit in a pattern).

Another practical issue is the comparison of the cost of the computation of an elaborate heuristic that cuts down a lot of nodes, and the cost of a much cheaper heuristic that cuts less nodes but solves the problems in less time. This comparison is usually problem dependent. On some simple problems where a cheap and effi-
cient heuristic already exists, it may not be of practical interest to generate elaborate heuristics, whereas on more complex and difficult problems a very specialized heuristic may well give excellent results.

5 References