Specialization of Admissible Path-finding Heuristics

Tristan Cazenave

Laboratoire d'Intelligence Artificielle,
Département Informatique, Université Paris 8,
2 rue de la Liberté,
93526 Saint Denis, France.
cazenave@ai.univ-paris8.fr

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 patfinding 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.

1	2	3
8		4
7	6	5

8	2	4
3	7	1
6	5	

Fig. 1. The goal state, and a randomly generated state 20 moves away from the goal state

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*, because the cost of the last tree search is usually much higher than the cost of the previous tree searches[10,11]. When using IDA*, the tree search can be cut before the maximum depth is reached, whenever f gets greater than the maximum depth. Therefore, an admissible heuristic that always give greater values than the Manhattan heuristic is interesting, because it will enable IDA* to stop its search sooner. The length of the real optimal path of our example is 20 and IDA* finds it in 33219 nodes using our specialized heuristics.

3 Specialization of an admissible heuristic

In this section we explain how the Manhattan heuristic can be specialized to give higher estimations. We follow with a description of the specialization programs of Introspect and we give experimental results for the 9-puzzle.

The application of specialization techniques to problem solving is not new. For example, S. Minton has used Prodigy/EBL to generate control rules [14], given the traces of Prodigy problem solving. O. Etzioni [6] further refined the methodology by using a kind of partial evaluation that gives better results than EBL/G [15], but that is formally equivalent [20]. Another system, Introspect uses logic metaprogramming and partial deduction to generate control rules for many games [2,3,4,17]. However, we are not aware of any application of these techniques on the specialization of heuristics for path-finding problems.

3.1 Some opportunities to specialize the heuristic

The idea behind the specialization is that some moves toward the solution increase h instead of decreasing it. If we want to generate by logic program specialization the cases when it happens, we have to define situation where it is always the case. Figure 2 gives such a situation :

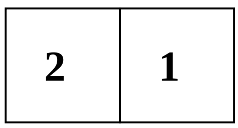

Fig. 2. A conflict between two tiles

Suppose that these situation happens in the upper left corner. Tile '2' is at the goal location of tile '1', and vice-versa. The Manhattan heuristic gives 1 for each of these tiles, resulting in 2 for the two tiles. However, if we consider the two tiles together, it is clear that the minimal number of moves to move them to their goal location is greater than 2: either tile 1 is moved first, and it moves to another location than tile 2's, increasing by one the Manhattan heuristic and by one the number of moves, either tile 2 is moved first and the same increasing holds. Therefore, we define for these two tiles the value Dh=2, corresponding to the direct conflict between them.

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:

```
conflict(0,T1,T2,2):-
          tile_on_location(L1,T1),
          tile_on_location(L2,T2),
          all_neighbors_increase_except(T1,L1,L2),
          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.

2	**?**	**1**

Fig. 3. A regressed conflict between two tiles

A specialization of this conflict can be obtained by unmoving an abstract move, that keeps the heuristic admissible. We obtain the situation in figure 3, where Dh is still 2. This specialization can be performed automatically by Introspect, which generates the corresponding program.

3.2 Logic Program Specialization with Introspect

Introspect is a logic metaprogramming system [1] that uses unfolding to specialize logic program in a similar way to other partial evaluators [19,7,13,16,8]. However, it differs from previous systems because it uses domain dependent information so as to guide the program generation. This domain dependent knowledge consists of clauses of impossibility that examine the unfolded clauses to find inconsistencies in them. A trivial and domain independent inconsistency is for example that an unfolded clause contains the atom '-1>-1'. A more domain-dependent set of impossible atoms is for example: `'number_neighbors (L,N), number_neighbors (L,N1)'` are in an unfolded clause and the conditions `'constant(N), Constant(N1), N=\=N1'` are verified. Clauses of impossibility are used to discard useless generated programs. Other domain dependent knowledge such as the statistical number of bindings of variables in some predicates is also used to generate efficient 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:

```
conflict(P,T1,T2,Dh):-
      P1 is P-1, P1 > -1,
      abstract_moves(T1,T2,L),
      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:

```
abstract_moves(T1,T2,[M1,M2]):-
      tile_on_location(L1,T1),
      tile_on_location(L2,T2),
      number_neighbors(L,2),
      neighbor(L,M1), M1\==L2,
      neighbor(L,M2), M1\==M2, M2\==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.

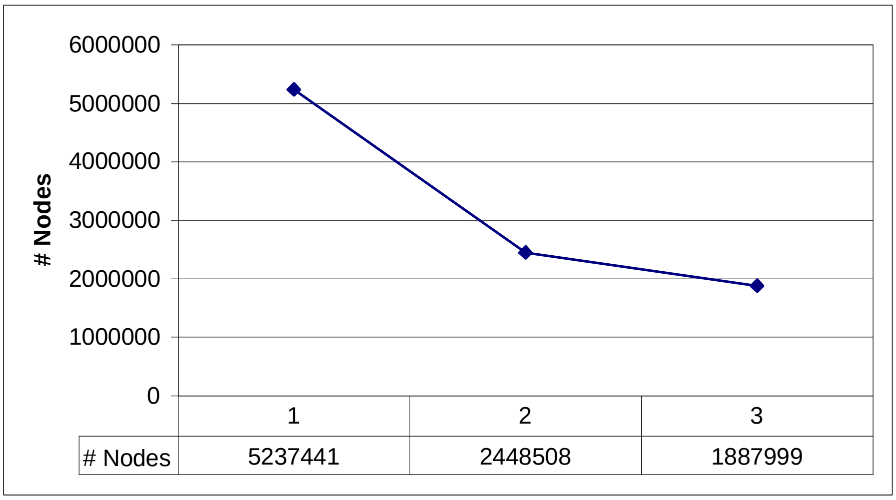

Fig. 4. Number of nodes developed by IDA* with increasing regressions

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 efficient 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

1. Barklund J.: Metaprogramming in Logic. UPMAIL Technical Report N° 80, Uppsala, Sweden, 1994.
2. Cazenave, T.: Système d'Apprentissage par Auto-Observation. Application au Jeu de Go. Ph.D. diss., University Paris 6, 1996.
3. Cazenave T.: Metaprogramming Forced Moves. Proceedings ECAI98, pp 645-649, Brigthon, 1998.
4. Cazenave T.: Controlled Partial Deduction of Declarative Logic Programs. ACM Computing Surveys, vol. 30, no 3es, 1998.
5. Culberson J.C., Schaeffer J.: Pattern Databases. Computational Intelligence, 1998.
6. Etzioni, O.: A structural theory of explanation-based learning. Artificial Intelligence 60 (1), pp. 93-139, 1993.
7. Gallagher J.: Specialization of Logic Programs. Proceedings of the ACM SIGPLAN Symposium on PEPM'93, Ed. David Schmidt, ACM Press, Copenhagen, Danemark, 1993.
8. Hill P. M. and Lloyd J. W.: The Gödel Programming Language. MIT Press, Cambridge, Mass., 1994.
9. Junghanns A.: Pushing the Limits : New Developments in Single-Agent Search. PhD thesis. University of Alberta, 1999.
10. Korf R. E.: Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, vol. 27, no 1, pp. 97-109, 1985.
11. Korf R. E.: Optimal path-finding algorithms. Search in Artificial Intelligence, L. Kanal and V. Kumar eds. New-York: Springer Verlag, 1988.
12. Korf, R.: Finding optimal solutions to Rubik's Cube using pattern databases. AAAI-97, pp. 700-705, 1997.
13. Lloyd J. W. and Shepherdson J. C.: Partial Evaluation in Logic Programming. J. Logic Programming, vol. 11 pp. 217-242., 1991.
14. Minton S., Carbonell J., Knoblock C., Kuokka D., Etzioni O., Gil Y.: Explanation-Based Learning : A Problem Solving Perspective. Artificial Intelligence 40, 1989.
15. Mitchell, T. M.; Keller, R. M. and Kedar-Kabelli S. T.: Explanation-based Generalization : A unifying view. Machine Learning 1 (1), 1986.
16. Pettorossi, A. and Proietti, M.: A Comparative Revisitation of Some Program Transformation Techniques. Partial Evaluation, International Seminar, Dagstuhl Castle, Germany LNCS 1110, pp. 355-385, Springer 1996.
17. Pitrat, J.: Games: The Next Challenge. ICCA journal, vol. 21, No. 3, September 1998, pp.147-156, 1998.
18. Reinefeld, A.; Marsland T. A.: Enhanced Iterative-Deepening Search. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 16, No. 7, July 1994, pp.701-710, 1994.
19. Tamaki H. and Sato T.: Unfold/Fold Transformations of Logic Programs. Proc. 2nd Intl. Logic Programming Conf., Uppsala Univ., 1984.

20. Van Harmelen F. and Bundy A.: Explanation based generalisation = partial evaluation. Artificial Intelligence 36:401-412, 1988.