State space search with prioritised soft constraints

Natasha Alechina and Brian Logan
School of Computer Science, University of Birmingham
Birmingham B15 2TT UK

{n.alechina, b.s.logan}@cs.bham.ac.uk

Abstract

This paper addresses two issues: how to choose between solutions for a problem specified by multiple criteria, and how to search for solutions in such situations. We argue against an approach common in decision theory, reducing several criteria to a single 'cost' (e.g., using a weighted sum cost function) and instead propose a way of partially ordering solutions satisfying a set of prioritised soft constraints. We describe a generalisation of the A^* search algorithm which uses this ordering and prove that under certain reasonable assumptions the algorithm is complete and optimal.

1 Introduction

Consider the problem of an agent playing the game of 'hide-and-seek' which has to plan a route from its current position to the 'home' position in a complex environment consisting of hills, valleys, impassable areas and so on. The plan should satisfy a number of criteria, for example, it should be concealed from the agent's opponents, it should be as short as possible and be executable given the agent's current resources (e.g., fuel or energy). This problem is sometimes formulated as that of finding a *minimum-cost* (or low-cost) route between two locations in a digitised map which represents a complex terrain of variable altitude, where the cost of a route is an indication of its quality [1]. In this approach, planning is seen as a search problem in space of partial plans, allowing many of the classic search algorithms such as A^* [4] or variants such as A^*_ϵ [8] to be applied. However, while such planners are complete and optimal (or optimal to some bound ϵ), it can be difficult to formulate the route planning task in terms of minimising a single criterion.

One way of incorporating multiple criteria (such as time, energy, visibility) into the planning process is to define a cost function for each criterion and use, e.g. a weighted sum of these functions as the function to be minimised. However the relationship between the weights and the solutions produced is complex, and it is often not clear how the different cost functions should be combined to give the desired behaviour across all magnitude ranges for the costs. This makes it hard to specify what kinds of plans a planner should produce and hard to predict what it will do in any given situation; small changes in the weight of one criterion can result in large changes in the resulting plans. Changing the cost function for a particular criterion involves changing not only the weight for that cost, but the weights for all the other costs as well. Moreover, if different criteria are more or less important in different situations, we need to find sets of weights for each situation.

Rather than attempt to design a weighted sum cost function, it is often more natural to formulate such problems in terms of a set of constraints which a solution should satisfy. In this paper, we focus on optimisation constraints (requirements to minimise a cost) and upper bound constraints (requirements that a cost be less than or equal to some value). We allow constraints to be *prioritised*, i.e., it is more important to satisfy some constraints than others, and *soft*, i.e., constraints which can be satisfied to a greater or lesser degree. Such a framework is more general in admitting both optimisation problems (e.g., minimisation constraints) and satisficing problems (e.g., upper bound constraints), and cannot be modelled using weighted sum cost functions.

The A^* search algorithm is ill-suited to dealing with problems formulated in terms of constraints. We present a generalisation of A^*, A^* with bounded costs (ABC) [7], which searches for a solution which best satisfies a set of prioritised soft constraints. In the next section we introduce state space search and A^*. In section 3 we define a preference order on solutions based on prioritised soft constraints and in section 4 we describe the ABC search algorithm. In section 5 we show that, given certain reasonable assumptions about the constraints, ABC is both complete and optimal. Section 6 discusses some complexity issues. In section 7, we briefly describe an implemented route planning system based on ABC and show that ABC cannot be emulated by A^*.

Although ABC was originally motivated by the route planning problem described above, we believe that it can be applied in other problem domains which involve searching for a solution specified by multiple incommensurable criteria or prioritised soft constraints. In the sequel, we will use route planning as a running example.

2 State space search

Search is a universal problem-solving technique in AI. It is useful when the sequence of actions required to solve a problem are not known *a priori*, but must be determined by systematic trial and error exploration of the alternatives. In particular, route planning can be seen as a search problem.

A state space search problem consists of the following components:

- A *state* is a complete description of the world for the purposes of problem-solving. For example, in a chess game, the states might be the positions of the pieces on the board, in route planning, a location. Some states are designated as *goal states*, e.g. in route planning the goal states would be the desired destination(s).
- An *operator* is an action that transforms one state of the world into another state. In a chess game, the operators might be the legal moves for the pieces given the current board position. In route planning on a digitised map, the operators would be moves to a neighbouring cell.
- the *state space* is the set of all states reachable from the *initial state* (the state the world is in when problem-solving begins) by any sequence of operator applications.

A *path* in the state space is any sequence of operator applications leading from one state to another. A *solution* is a path from an initial state to a goal state.

We assume that each application of an operator has a cost associated with it (which depends on the operator and the context in which the operator is applied). A *path cost function* g is the sum of costs of operator applications which constitute the path. It can be seen as a measure of quality of the path. For example, in the route planning problem, we might prefer solutions which minimise the distance travelled or the time taken to reach a goal.

A search strategy which is guaranteed to find a solution (if one exists) is said to be *complete*. If it also finds the minimum cost solution it is said to be *optimal*.

The A^* search algorithm is an example of an informed search strategy which uses additional information about the likely cost of completing a path to a goal. This is expressed as a *heuristic function*, $h(p)$, which, given a path p to some state s, returns an estimate of the cost of the minimum cost path from s to a goal state. A^* uses an estimated total cost:

$$f(p) = g(p) + h(p)$$

to guide the search, so that the most promising paths (partial solutions) are considered first. A^* is *complete* if each operator costs at least d for some positive d. A^* is *optimal* if $h(p)$ is always an underestimate of the true cost of extending the path p to a goal state.

3 Ordering paths

When solutions are evaluated on multiple criteria, coming up with a single cost function becomes problematic. Instead we use several partial orders on solutions. In this section, we introduce a preference order on paths in the state space and a dominance order on paths. Both orders are used by the ABC search algorithm introduced in the next section. First we need to introduce several notions.

Constraint order *Constraints* are bounds on *costs* of solutions (where a solution has multiple costs, one for each criterion of evaluation). A cost can be anything for which a (partial) order relation can be defined: e.g., numbers, booleans, or more generally a label from an ordered set of labels (e.g., 'tiny', 'small', 'medium', 'large', 'huge') etc. A constraint is a requirement that a cost lies within a given range of values; for example, '$f(n) = true$', '$f(n) = 100$', '$f(n) < 10$', '$f(n) > 20$', or '$f(n) \leq O + \epsilon$' (i.e. within ϵ of the optimum value O). A constraint is satisfied, if the cost is inside the required range, and violated otherwise.

An important class of constraints are upper/lower bound constraints which define an upper or lower bound on some property of the solution, such as the time required to execute a plan, its degree of visibility etc. Another kind of constraint which we consider in detail, since they allow us to formulate ABC as a generalisation of A^*, are optimisation constraints which require that some property of the solution be minimised or maximised, or more generally should lie within ϵ of the minimum or maximum value (for example that a plan should be as short as possible).

Suppose the requirements on a solution are given by a set of constraints $C_1, \ldots, C_n$. If a solution p satisfies the same constraints as a solution p' and at least one more, p should be preferred to p'. This gives a very uninformative preference relation; for example, a solution p which satisfies only C_1 and p' which satisfies $C_2, \ldots, C_n$ are incomparable. In some cases, either the constraints are prioritised (e.g., C_1 is more important than $C_2, \ldots, C_n$ taken together, and therefore p is preferred to p') or, more generally, some combinations of constraints are more important to satisfy than others.

More precisely, we associate with every path p a vector of t's and f's of length n, where the ith element of the vector is t if C_i is satisfied, and f otherwise. The value t is preferred to f ($t \sqsubseteq f$), since it is always better to satisfy a constraint than to violate it. This gives rise to the pointwise order on vectors of constraint values:

- $t \sqsubseteq f$, $t \sqsubseteq t$, $f \sqsubseteq f$;
- Let $1 \leq i \leq n$ and $a_i, b_i \in \{t, f\}$. Then $\langle a_1, \ldots, a_n \rangle \sqsubseteq \langle b_1, \ldots, b_n \rangle$ if for all i $a_i \sqsubseteq b_i$;
- As usual, $x \sqsubset y$ if $x \sqsubseteq y$ and not $y \sqsubseteq x$
- $x \equiv y$ if $x \sqsubseteq y$ and $y \sqsubseteq x$.

In general, we assume that $\sqsubseteq$ is any reflexive and transitive extension of the pointwise order defined above, for example lexicographic order or the order in which only the number of satisfied constraints matter.

The order $\sqsubseteq$ on the vectors gives rise to the order on paths: $p \sqsubset (\sqsubseteq, \equiv) p'$ if the corresponding vectors of constraint values are in $\sqsubset$ $(\sqsubseteq, \equiv)$ relation. A set of solutions satisfying the same constraints is called a *constraint equivalence class* (since they are in the $\equiv$ relation).

Cost order Within each equivalence class, the paths can still be distinguished on the basis of their costs. For example, consider a constraint that the time required to execute a plan should be less than 1 hour. A route which takes 50 minutes satisfies this constraint 'better' than a route which takes 59 minutes. A route which takes two hours violates it 'more' than a route which takes 1 hour 10 minutes. In general, given a constraint, it is sometimes possible to say which values of the corresponding cost function are 'better'. If v_1 and v_2 are values and k_1, k_2 constants, then v_1 is preferred to v_2 ($v_1 \prec v_2$) if, for example:

Form of constraint on cost v	Conditions on costs
$v < O_e + \epsilon$	$v_1 < v_2$
$v < k_1$	$v_1 < v_2$
$v > k_1$	$v_1 > v_2$
$v = k_1$	$\lvert k_1 - v_1 \rvert < \lvert k_2 - v_2 \rvert$

We can associate with each path a vector of cost values $\langle v_1, \ldots, v_n \rangle$ and define a partial order $\preceq$ on them, which we again assume to be at least the pointwise order. For two paths p and p', $p \preceq p'$ if $\preceq$ holds between the corresponding vectors of costs. We call $\preceq$ the *slack order*. We may prefer paths which over-satisfy the constraints, i.e., where there is some 'slack' between the cost of a path and the bound on the cost defined by a constraint. In the case of route plans, solutions which over-satisfy time or energy constraints are often more robust in the face of unexpected problems during the execution of the plan. If constraints are prioritised, the slack order can be a lexicographic ordering of cost vectors. The only assumption which we use in the proofs throughout the paper is that the slack order is at least the pointwise order.

Preference order Finally, we define the combination of the two orders which will be used to order the paths in the search space. Given the two orders $\sqsubseteq$ and $\preceq$, *the preference order* on paths is uniquely determined by by first ordering the paths with respect to $\sqsubseteq$ and then sub-ordering the equivalence classes with respect to $\preceq$. We denote the preference order by $\leq_{pref}$. Note that $\leq_{pref}$ might still be a partial (not total) order.

Dominance order Another order used in the algorithm is the *dominance order*. A path p *dominates* a path p' if both paths terminate in the same state and p is preferred to p' in the pointwise order of costs.

The preference order on paths is used to direct the search and control backtracking.[1] The dominance ordering is used to decide which newly generated paths to keep and which to discard. Below we show that all non-dominated paths to a state should be kept by the algorithm, even if some of them are below others in the preference order.

4 ABC algorithm

In the remainder of this paper we describe a generalisation of A^* search algorithm, A^* with bounded costs (ABC), which uses the preferences order defined above to search for a solution which best satisfies a set of prioritised soft constraints, rather than the solution with lowest cost on a single cost function [7].

We define an ABC search problem as consisting of:

- a set of states and operators as for A^*;
- a set of *cost functions*, one for each criterion on which solutions are to be evaluated;
- a set of *constraints* on acceptable values for each cost;
- an order $\sqsubseteq$ over vectors of constraint values; and
- an order $\preceq$ over vectors of cost values.

A *solution* to an ABC search problem is a path from the start state to a goal state.

The search strategy of ABC is similar to A^* (see Figure 1). We use two lists, an OPEN list of unexpanded nodes (paths) ordered using the preference order, and a CLOSED list which records all non-dominated expanded paths to each state visited by the algorithm. At each step, we take the

[1] Favouring paths which over-satisfy the constraints has the additional advantage of reducing the likelihood that the path will violate the constraint as the length of the path increases, reducing the amount of backtracking.

first node from the OPEN list and put it on CLOSED. Call this node n. If n is a valid solution we return the path and stop. Otherwise we generate all the successors of n, and for each successor we cost it and determine its equivalence class. We remove from OPEN and CLOSED all paths dominated by any of the successors of n and discard any successor which is dominated by any path on OPEN or CLOSED. We add any remaining successors to OPEN, in order, and recurse.

```
OPEN ← [start]
CLOSED ← []

repeat
    if OPEN is empty return false

    remove n, the least member of the first non-empty equivalence
    class, from OPEN and place it on CLOSED

    if n is a solution then return n

    otherwise for every successor, n', of n

        cost n' and determine its equivalence class

        remove from OPEN and CLOSED all paths dominated by n'

        if n' is dominated by any path on OPEN or CLOSED, discard n'

        otherwise add n' to OPEN, in preference order
```

Figure 1: The *ABC* algorithm

5 Completeness and optimality of *ABC*

In this section we prove that, given some reasonable assumptions, *ABC* is both complete and optimal. By an *optimal solution* we mean a solution p such that there is no solution p' which is strictly preferred to p. Note that there may be several different optimal solutions.

As for A^*, completeness and optimality for *ABC* hold only under some assumptions about operators and cost functions. Here we formulate them for increasing cost functions; it is straightforward to formulate analogous conditions for decreasing cost functions.

1. There are finitely many operators, and each application of an operator increases the cost of a path by at least some minimal positive amount d.
2. Heuristic components in cost functions never overestimate the actual cost of the completion of a path.

We call a constraint *admissible* if it is an upper bound or minimisation constraint on an increasing cost function satisfying the conditions above (or a lower bound or maximisation constraint on a decreasing cost function satisfying analogous conditions).

Theorem 1 *ABC with admissible constraints is complete.*

Proof. Suppose that the problem consists in finding a solution satisfying admissible constraints $C_1, \ldots, C_n$. For simplicity, assume that they all are upper bound or minimisation constraints corresponding to increasing cost functions $f_1, \ldots, f_n$. Suppose further that a solution does exist, and that the optimal solution(s) belong to the kth equivalence class.

It suffices to show that (1) all equivalence classes preceding the kth equivalence class are finite and hence the kth equivalence class will be searched after a finite number of steps. Note that the kth equivalence class itself need not be finite; if the search space is infinite, the last equivalence class for the given problem is infinite. So, we also need to show that (2) within the kth equivalence class, a solution will be found after a finite number of steps.

Recall that there are finitely many operators and an application of each of them increases the cost on $f_1, \ldots, f_n$ by at least a fixed amount d_i. A path consisting of m application of operators costs at least $m \cdot d_i$ for every function f_i. For every upper bound constraint $f_i(s) \leq r$ there are therefore only finitely many paths which cost less than r. Hence all equivalence classes which satisfy at least one upper bound constraint are finite. The cost of the solution on a cost function f_i (even though it is not known in advance and might be overestimated) gives an upper bound on the optimum for f_i and hence on the number of paths satisfying a minimisation constraint on f_i. We assume that the order of constraint equivalence classes is at least pointwise and hence every class preceding the kth equivalence class satisfies some constraint which is not satisfied by the kth class. So, there are only finitely many paths in the preceding equivalence classes. We have proved (1).

If the kth class itself satisfies at least one admissible constraint, it is finite as well, and a solution will be found regardless of the ordering of the class. In this case, the ordering of the class only matters for finding the optimal solution. If the kth equivalence class is infinite, we use the fact that it is ordered by the pointwise ordering over costs: a path which is cheaper on all cost functions is preferred. Eventually a path leading to a goal will have lower costs than any other path and will be chosen for expansion. This proves (2). □

Theorem 2 *ABC with admissible constraints is optimal.*

Proof. The proof of the previous theorem did not rely on the slack ordering (apart from the pointwise ordering on costs) or the fact that cost functions never overestimate the true cost of a path. It only used the fact that costs are finite and increasing by a discrete amount at every step, so that even a solution with overestimated cost will eventually become cheaper than any other path expanded so far. If, in addition, the slack ordering is used and the true cost is never overestimated, the first solution found will be the cheapest. The formal argument is the same as for A^* [8]. □

6 Comparison of ABC and A^*

In the worst case, A^* requires exponential space (and hence exponential time) to find a solution. ABC is a strict generalisation of A^*: with a single admissible optimisation constraint its behaviour is identical to A^*, and in this case its worst-case performance is identical to that of A^*. However, in general, the performance of ABC and A^* are not directly comparable, since the problems solved by ABC (e.g., problems involving upper-bound constraints or multiple constraints) cannot be re-formulated in terms of weighted sum cost functions (see section 7).

As might be expected, this additional flexibility involves a certain overhead compared with A^*. In particular, we must remember all the non-dominated paths to each state visited by the algorithm rather than just the minimum cost path as with A^* since: (a) it may be necessary to 'trade off' slack on a more important constraint to satisfy another, less important constraint; and (b) it may not be possible to satisfy all the constraints, in which case we must backtrack to a path in a lower equivalence class. If slack ordering is used, we must also perform an additional $\log m$ comparisons of k cost values, where m is the number of paths in the equivalence class. In addition, we must update the constraint values of the paths in the OPEN list when we obtain a better estimate of the optimum value for an optimisation constraint.

In some cases remembering all the non-dominated paths can be a significant overhead. However, there are a number of ways round this problem, including more intelligent initial processing of the constraints and discretising the Pareto surface. For example we can require that the algorithm retain no more than n paths to any given point, by discarding any path which is sufficiently similar to an existing path to that point. In the limit, this reduces to A^* where we only remember one path to each point.

7 A route planner based on ABC

In this section, we present an example application of the *ABC* algorithm and compare it to conventional approaches based on weighted sum cost functions. We describe a simple route planner based on *ABC* for an agent which plays the game of 'hide-and-seek' in complex environments. The goal of the agent is to get from a given position to the 'home' position subject to a number of constraints, e.g., that the route should take less than t timesteps to execute or that the route should be hidden from the agent's opponents, and the function of the planner is to return a plan which best satisfies these constraints.

The current implementation of the route planner supports seven constraint types which bound the time and effort taken to execute the plan or require that certain cells be visited or avoided, for example, *concealed route* constraints enforce a requirement that none of the steps in the plan be visible by the agent's opponents.[2] However, for reasons of brevity, we shall consider only time and energy constraints here. Time constraints establish an upper bound on the time required to execute the plan assuming the agent is moving at a constant speed of one cell per timestep. Energy constraints bound a non-linear 'effort' function which returns a value expressing the ease with which the plan could be executed—the cost function is based on the 3D distance travelled with an additional non-linear penalty for going uphill.

In the following example, we consider the problem of planning from coordinates (50, 10) to (10, 45) in an 80×80 grid of spot heights representing a 10km $\times$ 10km region of Southern California. The terrain model is shown in Figure 2 (lighter shades of grey represent higher elevations).[3] We use a lexicographic ordering over constraints and costs, with the time constraint being more important than the energy constraint. The time taken to execute the plan should be less than 100 timesteps ($t < 100$) and the energy cost should be less than 15,000 units ($e < 15,000$). There is a conflict between the two constraints, in that shorter plans involve traversing steeper gradients and so require more energy to execute.

Figure 2 shows the plan returned by the *ABC* planner. The plan requires 63 timesteps and 14,736 units of energy to execute, i.e. it just satisfies the energy constraint. A straight line path would have given maximum slack on the first (time) constraint, but the planner has traded slack on the more important constraint to satisfy the second, less important, constraint (energy). The plan is optimal in the sense that there is no plan which takes less time to execute and still satisfies the energy constraint.

It is important to stress that this plan could not be found by a planner based on A^*. *ABC* will never prefer a route which satisfies only the second constraint to a route which satisfies the first constraint. If we attempted to obtain the same behaviour with A^* using a weighted sum cost function of the form $w_1 t + w_2 e$ we must ensure that the ratio of w_1 to w_2 is greater than the maximal value of

$$\frac{|e(p_a) - e(p_b)|}{|t(p_a) - t(p_b)|}$$

for any two plans p_a and p_b. But then a planner minimising $w_1 t + w_2 e$ will never trade off slack on the first constraint to satisfy the second one. The following example illustrates this point, and also explains the necessity to store all non-dominated paths to a state in *ABC* (see Figure 3).

Suppose that there are two plans, p_a and p_b to a point n, both satisfying the time and energy constraints, that is, $t(p_a) < T$, $e(p_a) < E$, and $t(p_b) < T$, $e(p_b) < E$, where T and E are upper bounds on time and energy respectively. Suppose further that $t(p_a) < t(p_b)$ and $e(p_a) > e(p_b)$. Given that

$$\frac{w_1}{w_2} > \frac{e(p_a) - e(p_b)}{t(p_b) - t(p_a)},$$

[2] Note that the current implementation of the planner does not support optimisation constraints.

[3] We are grateful to Jeremy Baxter at DERA Malvern for providing the terrain model.

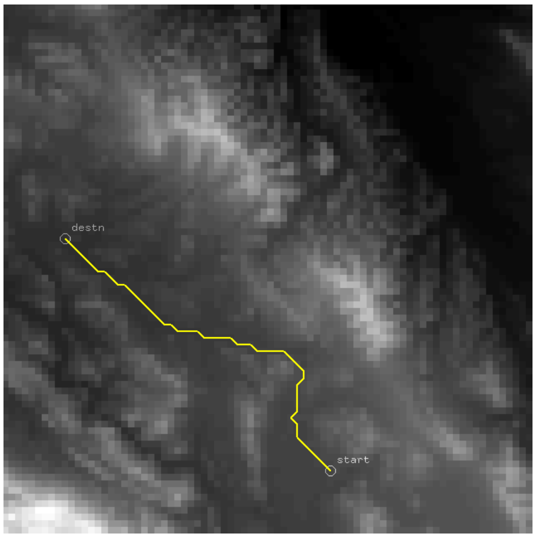

Figure 2: Planning with two constraints.

we have

$$w_1 t(p_a) + w_2 e(p_a) < w_1 t(p_b) + w_2 e(p_b),$$

that is, p_a is cheaper than p_b.

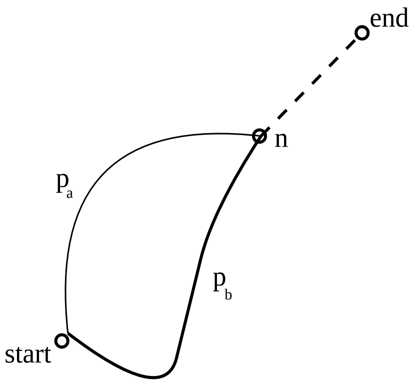

Figure 3: Plan subsumption with A^*.

However if it subsequently turns out that no completion of p_a through n will satisfy the energy constraint but there exists a completion of p_b which satisfies both constraints, we cannot backtrack to p_b since A^* retains only the (estimated) cheapest solution through n. A^* collapses both costs into a single value which is used to determine both the preference ordering and whether one plan dominates another. The resulting loss of completeness means we cannot use A^* to trade one constraint off against another [7].

Another possible way of attempting to solve the example problem using A^* would be to use a single partial order on the set of plans. Suppose we have some partial order on plans, which is at least the dominance ordering. Given two plans to the same point, p_a and p_b such that p_a satisfies the time and energy constraints, and p_b takes less time to execute but violates the energy constraint, then if p_a and p_b are comparable in this ordering, then p_a is preferred to p_b. If A^* uses this ordering to decide which plans to discard, then only p_a will be retained. However, if all extensions of p_a violate the first constraint, while there exists an extension of p_b which satisfies it, then the optimal solution will never be found. Conversely if A^* uses only the dominance ordering then the first solution found may not be optimal.

8 Related work

Our work has similarities with work in both optimisation (e.g., heuristic search for path finding problems and decision theoretic approaches to planning) and constraint satisfaction (e.g., planning as satisfiability). ABC is a strict generalisation of A^*: with a single optimisation constraint its behaviour is identical to A^*. However unlike heuristic search and decision theoretic approaches, we do not require that all the criteria be commensurable. The emphasis on non-dominated solutions has some similarities with Pareto optimisation which also avoids the problem of devising an appropriate set of weights for a composite cost function. However the motivation is different: the aim of Pareto optimisation is to return some or all of the non-dominated solutions for further consideration by a human decision maker. In contrast, when slack ordering is used, ABC will return the most preferred solution from the region of the Pareto surface bounded by the the constraints which are satisfied in the highest equivalence class. If an optimal solution is not required (i.e., a slack ordering is not used), the algorithm will return any solution which satisfies the constraints; such a solution will not necessarily be Pareto optimal.

ABC also has a number of features in common with boolean constraint satisfaction techniques. However, algorithms for boolean CSPs assume that: (a) all constraints are either true or false, (b) all constraints are equally important (i.e., the solution to an over-constrained CSP is not defined), and (c) the number of variables is known in advance. Considerable work has been done on partial constraint satisfaction problems (PCSP), e.g., [3], where the aim is to find a solution satisfying the greatest number of most important constraints. Dubois et al. [2] introduce Fuzzy Constraint Satisfaction Problems (FCSP), a generalisation of boolean PCSPs, which support prioritisation of constraints and preference among feasible solutions. In addition, FCSPs allow uncertainty in parameter values and ill-defined CSPs where the set of constraints which define the problem is not precisely known. However, in common with more conventional techniques, both PCSP and FCSP assume that the number of variables is known in advance. In many cases this assumption is violated, for example, in route planning the number of steps in the plan is not normally known in advance. Several authors, for example [5, 6], have described iterative techniques which can be applied when the number of variables is unknown. However, these techniques are incapable of handling prioritised or soft constraints, and the problems to which they have been applied are considerably smaller than the route planning problems which have been solved by ABC which typically involve more than 100,000 states and plans of more than 500 steps.

Like A^*, ABC requires monotonic cost functions and good heuristics. However it has many of the advantages of PCSP/FCSPs and iterative techniques: it can handle prioritised and soft constraints (though not uncertain values or cases in which the set of constraints which define the problem is not precisely known) and problems where the number of variables is not known in advance.

9 Conclusions and further work

In this paper, we have presented a new approach to formulating and solving multi-criterion search problems with incommensurable criteria.

We have argued that it is often difficult or impossible to formulate many real world problems in terms of minimising a single weighted sum cost function. By using an ordered set of prioritised soft constraints to represent the requirements on the solution we avoid the difficulties of formulating an appropriate set of weights for a composite cost function. Constraints provide a means of more clearly specifying problem-solving tasks and more precisely evaluating the resulting solutions: a solution can be characterised as satisfying some constraints (to a greater or lesser degree) and only partially satisfying or not satisfying others.

We have described a new search algorithm, A^* with bounded costs, which searches for a solution which best satisfies a set of prioritised soft constraints, and shown that for an important class of constraints the algorithm is complete and optimal. The utility of our approach and the feasibility of the ABC algorithm has been illustrated by an implemented route planner which is capable of planning routes in complex terrains satisfying a variety of constraints.

The present work is the first step in the development of a hybrid approach to search with prioritised soft constraints. It raises many new issues related to preference orderings over solutions ('slack ordering') and the relevance of different constraint orderings for different kinds of problems. More work is also necessary to characterise the performance implications of ABC relative to A^*. However, we believe that the increase in flexibility of our approach outweighs the increase in computational cost associated with ABC.

Acknowledgements

We wish to thank Aaron Sloman and the members of the Cognition and Affect and EEBIC (Evolutionary and Emergent Behaviour Intelligence and Computation) groups at the School of Computer Science, University of Birmingham for useful discussions and comments. This research is partially supported by a grant from the Defence Evaluation and Research Agency (DERA Malvern).

References

[1] C. Campbell, R. Hull, E. Root, and L. Jackson. Route planning in CCTT. In *Proceedings of the Fifth Conference on Computer Generated Forces and Behavioural Representation*, pages 233–244. Institute for Simulation and Training, 1995.

[2] Didier Dubois, Helene Fargier, and Henri Prade. Possibility theory in constraint satisfaction problems: Handling priority, preference and uncertainty. *Applied Intelligence*, 6:287–309, 1996.

[3] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[4] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC–4(2):100–107, 1968.

[5] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence, AAAI-96*, pages 1194–1201. AAAI Press/MIT Press, 1996.

[6] Vassilis Liatsos and Barry Richards. Least commitment—an optimal planning strategy. In *Proceedings of the 16th Workshop of the UK Planning and Scheduling Special Interest Group*, pages 119–133. University of Durham, Dec 1997.

[7] B. Logan. Route planning with ordered constraints. In *Proceedings of the 16th Workshop of the UK Planning and Scheduling Special Interest Group*, pages 133–144. University of Durham, Dec 1997.

[8] J. Pearl. A^*_ϵ — an algorithm using search effort estimates. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(4):392–399, 1982.