State space search with prioritised soft constraints

Natasha Alechina
School of Computer Science and IT,
University of Nottingham,
Nottingham NG8 1BB, UK.
email: `nza@cs.nott.ac.uk`

Brian Logan
School of Computer Science and IT,
University of Nottingham,
Nottingham NG8 1BB, UK.
email: `bsl@cs.nott.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.

Keywords: heuristic search, constraint satisfaction, multiobjective decision making, route planning.

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 [9] 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 or 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), which are difficult to model 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, 8], 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. In section 6 we briefly discuss the relative computational complexity of ABC and A^*, and show that the behaviour of ABC cannot be emulated by A^* with any cost function (or by any state space search algorithm which uses the same order to order paths and to determine which paths to discard). In section 7, we briefly describe an implemented route planning system based on ABC, and in section 8 we give a brief overview of related work in optimisation and constraint satisfaction.

In the sequel, we will use route planning as a running example. However, 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.

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 could 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 an associated cost which depends on the operator and the context in which the operator is applied. A *path cost function* $g(p)$ is the sum of costs of operator applications which constitute the path p. The path cost 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, devising a single cost function can be problematic. Instead we use two partial orders on solutions: a preference order and a dominance order on paths in the state space. 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, '$f(n) = \mathit{true}$', '$f(n) = 100$', '$f(n) < 10$', '$f(n) > 20$', 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. The rest of this section aims to make the notion of an order induced by constraints more precise.

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 $y \sqsubseteq x$.

We are interested in extensions of the pointwise order. An obvious example is MaxCSP [3], where the more constraints are satisfied, the better (e.g., satisfying any three constraints is preferred to satisfying any two constraints). Similarly, if the constraints are themselves ordered (prioritised), then the subsets of $\{C_1, \ldots, C_n\}$ could be ordered lexicographically.

Constraint order $\sqsubseteq$ 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 the $\sqsubset$ $(\sqsubseteq, \equiv)$ relation. A set of paths equivalent in the constraint order is called a *constraint equivalence class*.

Cost order In addition to the constraint order, we associate with each path a vector of cost values $\langle v_1, \ldots, v_n \rangle$ and define a partial order $\preceq$ on these vectors, 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*.

In many cases it is most natural to 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. For example, 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:

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

In the case of route planning, solutions which over-satisfy time or energy constraints are often more robust in the face of unexpected problems during the execution of the plan. Given 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. For example, if constraints are lexicographically ordered, the slack order could be a lexicographic ordering of cost vectors. However, sometimes this doesn't make sense; being one minute late is just as bad as being two hours late. For this reason, and in those cases where all solutions which satisfy the constraints are equally acceptable, we don't require a strong slack order. The only assumption which we use in the proofs in sections 5 and 6 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 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 order is used to decide which newly generated paths to keep and which to discard. Below (Theorem 3) 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. Typically the preference order will be more informative than the dominance order.

4 The *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 first node from

[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.

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 constraint 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 preference order, and recurse.

```
OPEN ← [start]
CLOSED ← []

repeat
    if OPEN is empty return false

    remove n, the head of the OPEN list, 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. We assume:

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; and

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 $C_1, \dots, C_n$. For simplicity, assume that they all are upper bound or minimisation constraints corresponding to increasing cost functions $f_1, \dots, f_n$. Suppose further that a solution does exist, and that s is an optimal solution.

Recall that the paths on the OPEN list are ordered by the preference order, i.e., first with respect to their equivalence class and then each equivalence class is sub-ordered by a slack order. Suppose that s belongs 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 fixed amounts $d_1, \ldots, d_n$, respectively. 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. In the case of minimisation constraints, the cost of s on a cost function f_i 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^* [9]. □

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) often cannot easily be reformulated in terms of minimising a weighted-sum cost function. More importantly, the solutions found by ABC (using two orderings) are different from those found by A^* (using only one order to order the OPEN list and determine which paths to discard). While it is possible to replace the preference order used by ABC with an order induced by a weighted sum cost function and obtain the same ordering of the OPEN list, this weighted sum cost function cannot be used to decide which paths are dominated and should be discarded.

6.1 Limitations of *A**

Several variants of A^* which use either a weighted sum cost function (or other way to compute a single overall cost from several costs), or a partial order over vectors of costs, have been proposed (e.g., [11, 12]). In this section we show that none of these algorithms can replicate the behaviour of ABC with respect to constraint satisfaction if the same ordering is used both to order and to prune the OPEN list. Essentially, we make a simple point based on the observation that the order used to direct the search (order the paths in the OPEN list) is normally much more informative than the order used to determine which paths can be discarded (dominance order).

We say that a state space search algorithm is a *variant of* A^* if it generates solutions and keeps them in an OPEN list just like A^* but possibly uses some other criterion to order and prune the OPEN list (e.g., a partial order, or any function of multiple costs).

Theorem 3 *A^* or any variant of A^* which uses the same criterion to order paths and to determine which paths to discard, is not guaranteed to find an optimal solution even if all constraints are admissible.*

Proof. Suppose a search algorithm for constraint satisfaction is based on A^* with some preference order on vectors of costs. Only the most preferred (non dominated in the preference order) path to every state is retained. The following counterexample shows that such an algorithm may fail to find an optimal solution for a search problem with prioritised constraints.

Suppose there are two constraints, C_1 and C_2. Suppose further that two paths p_a and p_b to the same state n are found (see Figure 2). The first path satisfies both constraints with very little slack on both of them. The second path violates one of the constraints but satisfies another one with a lot of slack. Any preference order which takes account of constraint satisfaction should prefer the first path to the second one. If the same order is used to decide which paths to remember and which to discard, the second path will be discarded. However, if the heuristics are too optimistic, there may be no completion of p_a to any goal state which does not violate both constraints. At the same time there could be a continuation of p_b to a goal state which still satisfies one of the constraints. Obviously, the latter would give a better solution than a continuation of p_a, but the algorithm will not find it since it cannot backtrack to p_b, which was discarded. If p_b is ever regenerated, it will be discarded again since a 'better' path to the same state exists. If the optimal solution involves a path through n, the optimal solution will never be found. □

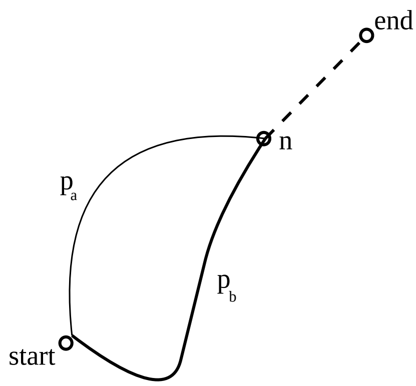

Figure 2: Plan subsumption with A^*.

6.2 Computational complexity

As might be expected, additional flexibility of ABC involves a certain overhead compared with A^*. In particular, we must remember all the non-dominated paths to each state visited by the algorithm. Slack ordering requires an additional $\log m$ comparisons of k cost values, where m is the number of paths in the equivalence class and k is the number of criteria. 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 possible solutions to 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 state, by discarding any path which is sufficiently similar to an existing path to that state. In the limit, this reduces to A^* where we only remember one path to each state.

7 A route planner based on *ABC*

In this section, we present an example application of the ABC algorithm. 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. The time cost is simply the number of timesteps necessary to execute the plan. 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. The energy cost C_i for step i is of the form:

$$C_i = l_i \times \begin{cases} ((100 \times l_i) + 1.0)^{1.5} & \text{if } g_i > 0 \\ 1.0 & \text{otherwise} \end{cases}$$

where l_i is the length of step i and g_i is the gradient. The heuristic function for the time constraint is simply the number of timesteps required to traverse the 2D distance from the current position to the goal and for the energy constraint is the 3D distance from the current position to the goal.

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 3 (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.[4] 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.

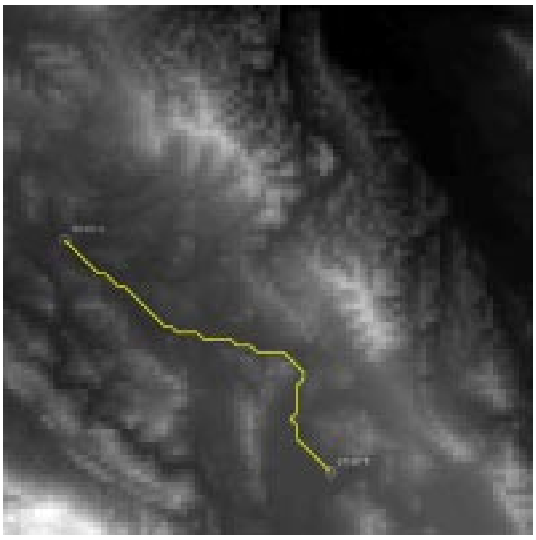

Figure 3: Planning with two constraints.

[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.

[4]For reasons of efficiency, the lexicographic ordering of costs necessary to ensure that the solution returned has maximum slack, is optional.

Figure 3 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^* using a single preference order, as Theorem 3 from the previous section shows.

Finding the plan requires the generation 29,107 nodes and 9,195 insertions into the OPEN list, and takes about about 40 seconds of CPU time on a Sun UltraSparc (300 MHz). As a rough comparison, with only the energy constraint (i.e., equivalent to A^* with energy as the cost function), the planner requires about 2.5 seconds of CPU time to find a plan, generates 6,110 nodes and performs 2,363 insertions into the OPEN list.

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). Below we briefly compare ABC with some of the related approaches.

8.1 Optimisation

Pareto optimisation Our 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, ABC will return the most preferred solution from the region of the Pareto surface bounded by the constraints which are satisfied in the highest constraint equivalence class.

Multiobjective A^* There are several extensions of heuristic search techniques to multiobjective search problems. Stewart and White [11] describe a generalisation of A^*, *Multiobjective* A^* (MOA^*), which handles multiple conflicting and incommensurable objectives by returning the set of all non dominated solutions.[5] MOA^* associates each path with a vector of costs, in which the ith component of the vector corresponds to the degree to which the path achieves the ith objective (criterion). The OPEN list is ordered using a pointwise order on cost vectors in a manner similar to that described in section 3. MOA^* expands the nodes with overall lowest costs first; if the costs are incomparable, MOA^* decides which nodes to expand by using a domain-specific selection rule. Under the usual assumptions about cost functions (positive, bounded) and finite number of operators, MOA^* is complete. Given the stronger assumption that the heuristics are admissible, MOA^* is guaranteed to produce all non-dominated solutions. (In the case in which there is a single objective, MOA^* is equivalent to A^* modified to find all optimal paths.)

A MOA^* problem with n criteria can be reformulated as an ABC problem with n minimisation constraints and no order over the constraints (i.e., preference order = pointwise order). Conversely, an ABC problem with, e.g., n unordered upper bound constraints, can be reformulated as a MOA^* problem with n criteria and n discontinuous cost functions. Where the constraints are not equally important, ABC's preference order can be represented by MOA^*'s 'domain specific preference rule'. However, MOA^* does not appear to have been used to solve problems formulated as a set of prioritised soft constraints. Rather it takes a more conventional optimising approach, in which the aim is to minimise costs. Non dominated solutions are preferred, and all criteria implicitly have equal importance. MOA^* returns all non dominated solutions (i.e., the solutions on the Pareto surface), avoiding the problem of choosing between incommensurable criteria. While it would be possible to extend ABC to return all non dominated solutions, it is not clear how the resulting set of solutions could be used to improve overall solution quality, and, in general, the computational cost of generating all non dominated solutions would be prohibitive.[6]

[5] We are grateful to Patrice Perny for drawing our attention to this work.

[6] See also [11, page 813].

U^* Another approach to multiobjective heuristic search which has similar aims to ABC is U^* [12]. U^* combines elements of both branch and bound search and dynamic programming. It is a best-first search algorithm which returns a single, most preferred, solution. Each path is associated with a vector of 'rewards', in which each element of the vector corresponds to one of the problem criteria. Larger rewards are 'better'. The OPEN list is ordered using a multiattribute preference function, u, which assigns a single real value to each reward vector. u is assumed to be any isotone function; i.e., u agrees with the pointwise order on reward vectors. u is therefore a generalisation of a weighted sum cost function, and, in particular, it allows non-additive accumulation of rewards.[7]

Like ABC, U^* allows a preference ordering over criteria to be defined. However, while ABC uses different orderings to order the OPEN list (preference order) and to decide which paths to discard (dominance order), the multiattribute preference function, u, is used both to select which solution to expand next, and to decide which paths to discard. So by Theorem 3, U^* cannot emulate behaviour of A^* with respect to constraint satisfaction.

8.2 Constraint satisfaction

ABC also has a number of features in common with constraint satisfaction techniques. Conventional algorithms for constraint satisfaction problems (CSPs) usually 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. Generalisations of classical constraint satisfaction problems include partial constraint satisfaction problems (PCSP), e.g., [3], and fuzzy constraint satisfaction problems (FCSP) e.g., [2]. These generalised constraint satisfaction problems can be more easily translated into ABC problems than into, e.g., heuristic search problems using a weighted sum cost function.[8]

Partial constraint satisfaction In a partial constraint satisfaction problem the aim is to find a solution satisfying the maximal number of most important constraints. A preference order on solutions is defined (solutions satisfying a better set of constraints are preferred) which is a special case of the constraint order defined in this paper. This means that ABC can be used to find a solution to a PCSP problem (and is a complete and optimal algorithm for PCSP under the standard assumptions).

Fuzzy constraint satisfaction Fuzzy constraint satisfaction also supports prioritisation of constraints: each constraint has a numerical degree of importance associated with it, ranging from 0 (totally unimportant) to 1 (hard constraint). The degree to which a solution satisfies a constraint is also characterised by a number between 0 (violated) and 1 (completely satisfied). A vector of degrees of satisfaction is associated with every solution in which the ith component is the degree to which the ith constraint is satisfied. To take the relative importance of constraints into account, the actual degree of satisfaction is adjusted. For example, a solution is assumed to satisfy a constraint which has importance α to a degree which is the maximum of $1 - \alpha$ and the actual degree of satisfaction. That is, if a constraint with importance 0.7 is totally violated, it is assumed to be satisfied with degree 0.3. The objective in FCSP is to find a solution which satisfies *all constraints* to the maximal degree. Several approaches to combining degrees of satisfaction have been proposed, e.g., a solution which satisfies the constraints with degrees $u_1, \ldots, u_n$ is preferred to a solution which satisfies the constraints with degrees $v_1, \ldots, v_n$ if $min(u_1, \ldots, u_n) > min(v_1, \ldots, v_n)$ [2]; see also [10]. While this objective is different from the applications ABC was designed for, ABC can be modified to work with the preference order arising from the FCSP approach.

Iterative techniques 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

[7] It is possible to generalise ABC to work with non-additive (but increasing) cost functions as well.

[8] We are not claiming that ABC is necessarily an efficient way to solve conventional constraint satisfaction problems; ABC is best suited to constraint satisfaction problems where the number of variables is not known in advance and solution has to be constructed stepwise from some initial state.

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.

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 a 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. The preference ordering $\leq_{pref}$ blurs the conventional distinction between absolute (hard) constraints and preference (soft) constraints. In our approach, all constraints are preferences that the problem-solver will try to satisfy, trading off slack on a more important constraint to satisfy another, less important, constraint.

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. Like A^* and multiobjective search techniques, ABC requires monotonic cost functions and good heuristics. However it has many of the advantages of PCSP/FCSPs and iterative constraint satisfaction techniques. 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). We are grateful to Patrice Perny for drawing our attention to the work on MOA^* and U^*.

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] D. Dubois, H. Fargier, and H. Prade. Possibility theory in constraint satisfaction problems: Handling priority, preference and uncertainty. *Applied Intelligence*, 6:287–309, 1996.

[3] E. C. Freuder and R. 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] V. Liatsos and B. 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] B. Logan and N. Alechina. A^* with bounded costs. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence, AAAI-98*, pages 444–449. AAAI Press/MIT Press, 1998.

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

[10] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95*, volume 1, pages 631–639. Morgan Kaufmann, 1995.

[11] B. S. Stewart and C. C. White III. Multiobjective A^*. *Journal of the Association for Computing Machinery*, 38:775–814, 1991.

[12] C. C. White III, B. S. Stewart, and R. L. Carraway. Multiobjective, preference-based search in acyclic or-graphs. *European Journal of Operational Research*, 56:357–363, 1992.