A^* with Bounded Costs

Brian Logan and **Natasha Alechina**
School of Computer Science, University of Birmingham
Birmingham B15 2TT UK
{b.s.logan,n.alechina}@cs.bham.ac.uk

Abstract

A key assumption of all problem-solving approaches based on utility theory, including heuristic search, is that we can assign a utility or cost to each state. This in turn requires that all criteria of interest can be reduced to a common ratio scale. However, many real-world problems are difficult or impossible to formulate in terms of minimising a single criterion, and it is often more natural to express problem requirements in terms of a set of constraints which a solution should satisfy. In this paper, we present a generalisation of the A^* search algorithm, A^* with bounded costs (ABC), which searches for a solution which best satisfies a set of prioritised soft constraints, and show that, given certain reasonable assumptions about the constraints, the algorithm is both complete and optimal. We briefly describe a route planner based on ABC and illustrate the advantages of our approach in a simple route planning problem.

Introduction

Heuristic search is one of the classic techniques in AI and has been applied to a wide range of problem-solving tasks including puzzles, two player games, and path finding problems. A key assumption of all problem-solving approaches based on utility theory, including heuristic search, is that we can assign a *utility* or *cost* to each state. This in turn requires that all criteria of interest can be reduced to a common ratio scale. For example, in a game of chess it is assumed that all the pieces and their positions on the board can be given a value on a common scale. Similarly, in decision theory, it is assumed that, for example, the inconvenience of carrying an umbrella and the discomfort of getting wet can be expressed as commensurable (dis)utilities. However, many real-world problems are difficult or impossible to formulate in terms of minimising a single criterion, and it is often more natural to view such problems in terms of a set of prioritised soft constraints. By prioritised we mean that it is more important to satisfy some constraints than others. For example, while not getting wet and not carrying an umbrella is clearly preferable, we may also prefer being dry with an umbrella to being wet without one. Soft constraints are constraints which can be satisfied to a greater or lesser degree, for example how long we have to spend in the rain, or the number of items we have to carry.

In this paper, we present a generalisation of the A^* search algorithm, A^* with bounded costs (ABC), which searches for a solution which best satisfies a set of prioritised soft constraints, and show that, given certain reasonable assumptions about the constraints, the algorithm is both complete and optimal. We briefly describe an implemented route planning system based on ABC and illustrate the advantages of ABC compared to A^* in a simple route planning problem.

The work reported in this paper was originally motivated by difficulties in applying classical search techniques to agent route planning problems, and we shall use this as a motivating example throughout the paper. However, the problems we identify with utility based approaches, and the solutions we propose, are equally applicable to other search problems.

An example: route planning

Consider, for example, the problem of an agent playing the game of hide-and-seek which has to plan a route from its current position to home base 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).

The route planning task can be formulated as the problem of finding a *minimum-cost* (or low-cost) route between two locations in a digitised map, where the cost of a route is an indication of its quality (Campbell *et al.* 1995). 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^* (Hart, Nilsson, & Raphael 1968) or variants such as A^*_ϵ (Pearl 1982) 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 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. For example, we can define a 'visibility cost' for being exposed and combine this with cost functions for the time and energy required to execute the plan to form a composite function which can be used to evaluate alternative plans. 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.

At best the amount of, e.g., time or energy, we are prepared to sacrifice to remain hidden is context dependent. In general, the properties which determine the quality of a solution are incommensurable. For example, the criteria may only be ordered on an ordinal scale, with those criteria which determine the feasibility of a solution being preferred to those properties that are merely desirable. It is difficult to see how to convert such problems into a multi-criterion optimisation problem without making ad-hoc assumptions.

State space search with prioritised soft constraints

In the remainder of this paper we describe a new search algorithm, A^* with bounded costs (ABC), which searches for a solution which best satisfies a set of prioritised soft constraints (Logan 1997). In effect, we replace the optimisation problem solved by A^* with a satisficing or constraint satisfaction problem which allows optimisation as a special case. For example, rather than finding the least cost plan on the basis of a weighted sum of the time required to execute the plan and its visibility, we might specify a route that takes time less than t and is at least 50% concealed, or a route that requires no more than e units of energy to execute and minimises visibility. This approach provides a means of more clearly specifying problems and more precisely evaluating solutions. For example, a plan can be characterised as satisfying some constraints and only partially satisfying or not satisfying others.

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;
- a preference ordering over sets of constraint values; and
- a preference ordering over costs.

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

Path constraints

A *cost* is a measure of path quality relative to some criterion, and can be anything for which an ordering 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 *cost function* is a function which takes a path and returns an estimate of the cheapest completion of the path to a goal state. A cost function is *admissible* if it never over-estimates the true cost of the cheapest completion of a path to a goal state. A cost function is *increasing* (resp. *decreasing*) if every operator application costs at least some minimum positive (resp. negative) amount d.

A *constraint* is a relation between a cost and a set of acceptable values for the cost, for example the boolean value '*true*', '$= 100$', an open interval such as '< 10', '> 20', or '$\leq O + \epsilon$' (i.e. within ϵ of the optimum value O).

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).

Upper bound/minimisation constraints on increasing admissible cost functions and lower bound/maximisation constraints on decreasing admissible cost functions are termed *admissible*. The latter are precise mirror images of the former, and for the ease of exposition we assume that admissible constraints bound increasing functions.

Path ordering

A path which satisfies all the constraints is termed *valid*. If the problem is over-constrained, there will be no solution which satisfies all the constraints. In such situations, it is often possible to distinguish among the invalid solutions, as the violation of some (sets of) constraints will be preferable to others.

Combinations of possible constraint values define a set of *path equivalence classes*, with those paths which satisfy the same constraints falling in the same equivalence class. We assume that there exists a preference ordering over these equivalence classes. A path p_a is *preferred* to a path p_b if the equivalence class of p_a precedes the equivalence class of p_b in this ordering. For example we may prefer paths which satisfy the greatest number of constraints, or paths which satisfy constraints which determine the feasibility of the solution to those which satisfy constraints defining properties which are merely desirable. In what follows, we assume that this ordering is at least a pointwise ordering, that is, if a path p_a satisfies the same constraints as p_b plus at least one more constraint, it is preferred to p_b.

We can use this ordering over equivalence classes without having an explicit ordering over the constraints. However, in many situations, it is often more natural to *prioritise* the constraints and use this ordering to generate the ordering on the path equivalence classes. For example, we could define a total order over the constraints and use this to partition paths into equivalence classes on the basis of the number of important constraints they satisfy, by comparing the value of each constraint in order until we find a constraint which is satisfied by only one of the paths. This is essentially lexicographic ordering on fixed length boolean strings in which *true* is preferred to *false*.[1]

If the problem is under-constrained, there may be many valid solutions. In such cases, it is often possible to define a notion of how well a path satisfies a constraint, which can be used to order the solutions. For example, 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.

The preference ordering on costs depends on the constraints associated with the costs. In general, if v_1 and v_2 are values and k_1, k_2 constants, then v_1 is preferred to v_2 if:

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

Combined orderings on cost values define a pointwise ordering over costs, i.e., a path p_a is preferred to a path p_b if it has the same or 'better' values on all cost functions. A special case of the pointwise ordering is a dominance ordering. One path p_a *dominates* another path p_b if both paths terminate in the same state, and there is at least one cost f_i such that $f_i(p_a) < f_i(p_b)$ and there is no cost f_j such that $f_j(p_a) > f_j(p_b)$.

Many refinements of the pointwise ordering of costs are possible. For example, we could order the equivalence classes using the costs for the most important constraint or the cost for the most important violated constraint. If the constraints are ordered lexicographically, it is often more natural to use a lexicographic ordering over the costs which reflects the constraint ordering. We will refer to all such refinements as *slack orderings*.

The slack ordering allows us to sub-order paths within a path equivalence class, with those paths which have the greatest slack being the most preferred. Conversely, for violated constraints, the sub-ordering may favour paths which are closer to satisfying the constraint. This can be useful in the case of 'soft' constraints, where minor violations are acceptable.

Slack ordering also allows us to define *relative optimisation constraints*, which are requirements that some cost f be minimised or maximised, given that some more important constraints are satisfied. A relative minimisation constraint has the form $f < \infty$ and is assumed to be always satisfied, but the slack ordering associated with the constraint prefers the paths with the minimal cost on f within every equivalence class.

The constraint and slack orderings over paths are used to direct the search and control backtracking.[2] The dominance ordering is used to decide which newly generated paths to keep and which to discard.

[1]It is clear that, in the general case, this ordering cannot be produced using a weighted sum cost function.

A^* with bounded costs

The search strategy of ABC is similar to A^*. We use two lists, an OPEN list of unexpanded nodes (paths) ordered using the preference ordering, and a CLOSED list containing all non-dominated expanded nodes. At each step, we take the first node from the OPEN list and put it on CLOSED. Call this node n. If n is a valid solution and all the constraints are admissible 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 (see Figure 1).

If the constraints are admissible, the first solution found will satisfy the greatest number of more important constraints; if slack ordering is used, this solution is also the most preferred with respect to the slack ordering. If the constraints are not admissible, we can never be sure we have found the optimum solution without an exhaustive search: even if we have a solution which satisfies all the constraints, there may be another solution which is preferable with respect to the slack ordering.

```
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 order
```

Figure 1: The ABC algorithm

[2]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. (If the cost functions are admissible, the estimated cost of a path will typically increase as the path is expanded.)

We end this section by stating two theorems about the formal properties of ABC, the proofs of which we omit due to lack of space.

Given reasonable assumptions about the constraints, it can be shown that ABC is both complete and optimal. By *complete* we mean that if a solution exists, it will be found after a finite number of steps. By an *optimal solution* we mean a solution in the highest non-empty equivalence class with respect to the constraint ordering which is also most preferred with respect to the slack ordering. An algorithm is optimal if it returns an optimal solution. Note that there may be several different optimal solutions.

Theorem 1 *ABC with admissible constraints is complete.*

Theorem 2 *ABC with admissible constraints is optimal.*

Proofs of these theorems can be found in (Logan & Alechina 1998)

Comparison of ABC and A^*

ABC is a strict generalisation of A^*: with a single admissible optimisation constraint its behaviour is identical to A^*. Indeed, ABC can be seen as A^* with two partial orderings on paths: a dominance ordering to determine which paths to discard and a preference ordering to determine which paths to expand first.

As might be expected, the additional flexibility of ABC involves a certain overhead compared with A^*. The preference ordering of paths requires the comparison of k constraint values for each pair of paths, where k is the number of constraints. 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.

There is also a storage overhead associated with this approach. For each path we must now hold k constraint values in addition to the k costs from which the constraint values are derived. More importantly, 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. 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 l 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.

Route planning with prioritised soft constraints

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 the start point to home base 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).[3] 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).[4] 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). In fact this plan has the greatest slack on the time constraint while still satisfying the energy constraint. 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

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

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

CPU time to find a plan, generates 6,110 nodes and performs 2,363 insertions into the OPEN list.

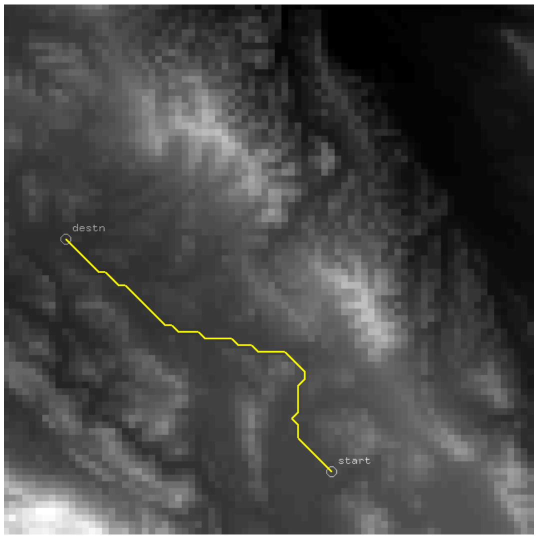

Figure 2: Planning with two constraints.

Unfortunately, it is not possible to compare the performance of ABC with A^* in other than trivial cases, e.g., when there is a single optimisation constraint, because we can't reduce the dominance and preference orderings used by ABC to the single ordering required by A^*. If, for example, we attempted to solve the above problem with A^* using a weighted sum cost function of the form $w_1t + w_2e$, 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 $w_1t + w_2e$ will never trade off slack on the first constraint to satisfy the second one. The following example illustrates this point, and also explains why ABC must remember all non-dominated paths to each visited state.

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

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

we have

$$w_1t(p_a) + w_2e(p_a) < w_1t(p_b) + w_2e(p_b),$$

that is, p_a is cheaper than p_b.

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 (Logan 1997).

Another possible way of solving the example problem using A^* would be to use a 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.

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., (Freuder & Wallace 1992), where the aim is to find a solution satisfying the greatest number of most important constraints. Dubois et al. (Dubois, Fargier, & Prade 1996) introduce Fuzzy Constraint Satisfaction Problems (FCSP), a generalisation of boolean CSPs, 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 (Kautz & Selman 1996; Liatsos & Richards 1997), have described iterative techniques which can be applied when the number of variables is unknown. However, to date, these techniques have been applied to problems which are considerably smaller than the route planning problems we consider, which typically involve more than 100,000 states and plans of more than 500 steps. Moreover these techniques are incapable of handling prioritised or soft constraints.

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.

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

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

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

Freuder, E. C., and Wallace, R. J. 1992. Partial constraint satisfaction. *Artificial Intelligence* 58:21–70.

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

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

Liatsos, V., and Richards, B. 1997. Least commitment—an optimal planning strategy. In *Proceedings of the 16th Workshop of the UK Planning and Scheduling Special Interest Group*, 119–133. University of Durham.

Logan, B., and Alechina, N. 1998. A^* with bounded costs. Technical Report CSRP-98-09, School of Computer Science, University of Birmingham.

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

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