K2: An Efficient Approximation Algorithm for Globally and Locally Multiply-Constrained Planning Problems

Andrés Santiago Pérez-Bergquist
aspb@mapache.org

Anthony Stentz
tony@cmu.edu

Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

***Abstract* - Many problems are easily expressed as an attempt to fulfill some goal while laboring under some set of constraints. Prior planning algorithms have addressed this in part, but there are few fast ways of working with more than just a few constraints. Extending algorithms designed for one constraint to multiple constraints is difficult due to the NP-complete nature of the problem, prompting a switch to an approximation algorithm. This paper presents K2, a multiply-constrained planning algorithm which is an amalgamation of parts of H_MCOP and Focussed D*. It accepts additive constraints over the path or over any fixed-length section of the path. K2 operates quickly and produces results of acceptable quality.**

Index Terms* - *Multiply-constrained path selection, replanning, approximation algorithm

INTRODUCTION

Planning is the act of taking a state space, a set of actions that define transitions within this state space, and a pair of states labeled the start and the goal, then finding a sequence of actions that lead from the start to the goal. For most problems, one is interested not only in finding a solution, but also in finding a good solution relative to some set of criteria. This is traditionally handled by attaching a cost to each action and then specifying that sequences of actions with minimal total cost are preferred.

Aside from the issue of finding a minimal-cost path, assigning costs to actions can be difficult. For many real-world tasks, preferences regarding solutions are most naturally expressed in terms of multiple criteria, which often compete with each other. Even after one has assigned numerical costs to each of these criteria, it is unclear how to proceed. Most existing planning algorithms operate on a single cost per action, so the multiple criteria need to be combined into a single cost function, such as by assigning arbitrary weights and summing them.

This sort of approach implies that one criterion can be traded off for another, which is usually not the case. Instead, each criterion often models a distinct limited resource, such as time elapsed and fuel spent, and one is only interested in plans that simultaneously meet certain cost constraints for each resource. Thus, if one wishes to sum the various cost functions, the weights need to be arrived at intelligently in a problem-dependent fashion.

Path planning is the archetypical example of such a multiply-constrained problem. One might have a robot that needs to get from point A to point B without exceeding its battery charge, overheating, taking too long, or exposing itself to excessive risks along the way. If each of these factors can be formulated as a cost map, then they can be used as inputs for the planning process.

PRIOR WORK

Several algorithms exist to address this category of problems. A* with bounded costs (ABC) relies upon prioritizing the constraints [1]. For each point in the state space, it keeps track of all non-dominated paths that reach it, and selects from among those. Although optimal and complete, it can require substantial overhead due to keeping track of and comparing several paths per point in the state space. This is unsurprising, since the underlying problem is NP-complete [2].

CD* is a computationally efficient planner and re-planner that extends D* (a re-planning version of A*) to handle one hard constraint which must not be violated in addition to the primary objective function which needs to be minimized [3]. CD* assigns weights to the objective function and the constraint function, then sums them and uses this combined cost function as the input to D*. Based on whether or not the constraint is met by the solution returned, the weights are adjusted up or down and D* is run again, converging via binary search to the optimal solution within this weight space.

The obvious approach is to extend CD* to handle multiple simultaneous constraints. Multiply-Constrained D* (MCD*) is a planning and re-planning algorithm that takes into account a single cost function to be minimized and multiple constraint functions to be kept below individual thresholds. The constraint functions are assigned weights and summed with the primary objective function to produce a combined cost function that is used for the actual planning via D*. Ideally, the resulting solution could then be assessed against the constraints and the weights adjusted in a multi-dimensional search that repeatedly creates new combined cost functions which converge upon a solution that meets all the constraints while minimizing the primary cost function.

Unfortunately, guiding the search in such a manner is problematic. The focusing of the search is predicated on the fact that the optimal solution (if it exists) will lie on the boundary between a region of weight-space whose points yield solutions that meet all the constraints (the feasible region) and a region whose points do not. However, for more than one constraint, the feasible region need not be contiguous within the weight-space (See Figure 1). This leaves no efficient means of discovering the boundaries of the feasible region. MCD* thus reduces to placing an n-dimensional grid over the weight space, sampling each of the intersections, and returning the best solution of those tested. While this is resolution-optimal within the weight-space, it requires running D* a number of times exponential in the number of constraints.

Fig. 1. Proof that the feasible region need not be contiguous.

Consider a planning problem with A, B, C, and D as the only four possible paths, with the total values of the paths relative to three cost functions being:

	f_0	f_1	f_2
A	0	9	9
B	1	0	12
C	10	1	1
D	1	12	0

B minimizes f_0+f_1. D minimizes f_0+f_2. Any linear combination of (f_0+f_1) and (f_0+f_2) is minimized by one of B or D. If our constraints are that neither f_1 nor f_2 may exceed 10, then we have a solid line bisecting the weight space all of whose optimal solutions do not meet all constraints. A minimizes f_0, which lies on one side of the line, and C minimizes $f_0+2f_1+2f_2$, which lies on the other side of the line. Both A and C meet all constraints. Thus, the feasible space is not contiguous. The cost metrics f_0, f_1, and f_2 could all be simple additive functions.

Unlike simply minimizing the cost of a path, finding a path subject to multiple constraints is not a dynamic programming problem. In dynamic programming, the optimal solution to a problem is dependent only on the current position in the state space, allowing for the recursive construction of solutions. However, in multiply constrained planning, although the minimum cost path (relative to an arbitrary metric) from any given point to the goal is not dependent on the path taken to reach that point, the admissibility relative to a given set of constraints of a complete path from the start to the goal passing through that point can be dependent on the path taken to reach that point. (This is why ABC must keep track of all non-dominated partial paths.) Expressed mathematically, if we have n cost criteria each with a corresponding function f_i which gives the cost of a path p with respect to one of the criteria, and for each criterion we have a constraint c_i, such that we do not wish $f_i(p)$ to exceed c_i, then attempting to find a path which minimizes the cost function $max_i(f_i(p)/c_i)$ will yield a path which meets all constraints (if such a path exists). This is not a dynamic programming function, however.

Given the computational complexity of the problem, several approximation algorithms exist. H_MCOP is a two-pass approximation algorithm whose computational workload grows only linearly as the number of constraint functions increases [4]. In the first pass, it works backwards from the goal, finding the optimal path according to a cost function which is the average of the individual $f_i(p)/c_i$ values. It then uses the costs to reach the goal according to this function as a heuristic to guide a forward search from the start using the actual $max_i(f_i(p)/c_i)$ cost function. Since this heuristic is not guaranteed to be admissible, the resulting algorithm is neither complete nor optimal, but, in practice, it performs well and yields solutions of acceptable quality.

THE K2 ALGORITHM

Our algorithm, K2, is derived from H_MCOP, and its basic operation is nearly identical. The problem space is defined in terms of a state space V and a set of transitions E from one state to another. Associated with each transition in E is an n-dimensional vector of non-negative real numbers which give the costs of taking that transition relative to n different metrics. For any path p composed of a sequence of transitions in E, $f_i(p)$ gives the sum of the costs of each transition in p with respect to the ith cost metric. The vector w is composed of n real numbers such that $f(p) = \sum_i(f_i(p) \bullet w_i)$ defines an objective cost function that should be minimized.

For a specific instance of a problem, we have two elements of V given as the START and the GOAL. In addition, we are given a vector c of n real numbers such that element c_i states the maximum cost acceptable with respect to the ith metric over the total path. The desired output is a path p from START to GOAL such that $\forall i(f_i(p) \leq c_i)$ and which minimizes $f(p)$ over all such paths.

For much of the algorithm, the cost functions are normalized to $f_i(p)/c_i$ so that a normalized path cost of 1.0 or less corresponds to meeting the constraint. This allows us to compare in a meaningful fashion how well the various constraints are being met.

Conceptually, the algorithm works backwards from GOAL using the sum of the normalized cost functions as an objective function for A*. For each point in V, it uses the path determined by this process to calculate the cost to the GOAL along that path using the maximum of the normalized cost functions as an objective function. It then uses these costs as a heuristic for a forward search from START with additional logic to select between favoring minimizing the maximum of the normalized cost functions and minimizing the overall objective function f.

In H_MCOP, the first pass exhaustively computes the heuristic values for every point in the state space, and the second pass computes the path from the START to the GOAL. In K2, we have switched to a time-saving lazy implementation that computes the heuristic on demand only for points that are examined by the second pass. Once the heuristic for a point has been computed, even in the process of computing the heuristic for another point, it is saved and never recomputed.

The first half of the algorithm, the backwards pass (initialized by lines 1-7 and processed by lines 52-66), is simply Dijkstra's algorithm with $\sum f_i(p)$ as the cost function. It works outwards, finding for every vertex a path to the GOAL that minimizes the total normalized cost across all cost functions f_i and recording the total cost in each function f_i separately for each node, for use in the second half of the algorithm. If the constraints are all being met to a similar degree, as evidenced by the normalized costs with respect to each of the constraint functions being similar, then the paths generated will likely be feasible (meet all constraints), if such paths exist. If, instead, a path requires substantial increases in several cost functions in order to accommodate an otherwise troublesome constraint, then the paths generated at this point will likely not resemble the end result of stage two of the algorithm.

For the second, forward pass (lines 8-51), the algorithm runs modified A* with unusual selection logic. Vertices on the OPEN list are all kept in two queues simultaneously, one sorted by the total $f(p)$ objective cost of the traversed path plus the heuristic path from the first half of the algorithm, and the other sorted by the $max_i(f_i(p)/c_i)$ cost of the path, thus preferring paths that are deemed to be furthest from violating the constraint with respect to which they are doing worse. (In H_MCOP, this second list is sorted by the sum of powers of the normalized constraint functions; if the exponent is infinite, this is equivalent to taking the maximum of the functions, which usually yields the best result.)

Fig. 2. The workings of the K2 algorithm for simple additive cost metrics. (Starred lines indicate where changes are needed to support windowed constraints.)

```
#   Initialize the Backward Search by placing the GOAL on the heuristic OPEN list

1   for each u in V
2       u.BackwardState := NEW

3   GOAL.BackwardState := OPEN
4   for p := 1 to n
5       GOAL.CostToGoal[p] := 0
6   GOAL.CombinedCostToGoal := 0

7   Enqueue(BackwardsQueue point:GOAL withCombinedCostToGoal:0 withCostToGoal:GOAL.CostToGoal
                                                                                    withSuccessor:NULL)

#   Initialize the Forward Search by placing the START on the new, doubly-sorted OPEN list

8   for each u in V
9       u.ForwardState := NEW

10  for p := 1 to n
11*     START.CostToReach[p] := 0
12  ComputeHeuristicForState(START)
13  objCost := 0
14  maxCost := 0
15  for p := 1 to n
16*     objCost += w[p] * START.CostToGoal[p]
17*     maxCost := MAX(maxCost, START.CostToGoal[p])
18  Enqueue(ObjectiveCostQueue point:START withEstimatedTotalCost:objCost
                                            withCostToReach:START.CostToReach withPredecessor:NULL)
19  Enqueue(MaxCostQueue       point:START withEstimatedTotalCost:maxCost
                                            withCostToReach:START.CostToReach withPredecessor:NULL)
20  START.ForwardState := OPEN

#   Do the Forward Search by repeatedly expanding the minimum objective cost path if it is predicted
#   to be feasible, and the minimum maximum normalized constraint cost path otherwise

21  while QueueIsNotEmpty(ObjectiveCostQueue) & QueueIsNotEmpty(MaxCostQueue) & GOAL.ForwardState != CLOSED
22      if PeekMinFromQueue(ObjectiveCostQueue).ForwardState == CLOSED
23          PopMinFromQueue(ObjectiveCostQueue)
24      else if PeekMinFromQueue(MaxCostQueue).ForwardState == CLOSED
25          PopMinFromQueue(MaxCostQueue)
26      else
27          (u, _, uCostToReach, uPredecessor) := PeekMinFromQueue(ObjectiveCostQueue)
28          minObjFeasible := true
29          for p := 1 to n
30*             if uCostToReach[p] + u.CostToGoal[p] > 1.0
31*                 minObjFeasible := false
32          if minObjFeasible
33              (u, _, uCostToReach, uPredecessor) := PopMinFromQueue(ObjectiveCostQueue)
34          else
35              (u, _, uCostToReach, uPredecessor) := PopMinFromQueue(MaxCostQueue)
36          u.ForwardState := CLOSED
37          u.BackPointer := uPredecessor
38          u.CostToReach := uCostToReach
39          for each v in V such that <u, v> is in E
40              if v.ForwardState != CLOSED
41                  ComputeHeuristicForState(v)
42                  objCost := 0
43                  maxCost := 0
44                  for p := 1 to n
45*                     vCostToReach[p] := u.CostToReach[p] + <u, v>.Cost[p] / Constraint[p]
46*                     pathTotalForPlane := vCostToReach[p] + v.CostToGoal[p]
47*                     objCost += w[p] * pathTotalForPlane
48*                     maxCost := MAX(maxCost, pathTotalForPlane)
49                  Enqueue(ObjectiveCostQueue point:v withEstimatedTotalCost:objCost
                                                        withCostToReach:vCostToReach withPredecessor:u)
50                  Enqueue(MaxCostQueue       point:v withEstimatedTotalCost:maxCost
                                                        withCostToReach:vCostToReach withPredecessor:u)
51                  v.ForwardState := OPEN

subroutine: ComputeHeuristicForState(r)

#   Do the Backward Search using Dijkstra's algorithm on the sum of the normalized cost functions

52  while r.BackwardState != CLOSED
53      (s, sCombinedCostToGoal, sCostToGoal, sSuccessor) := PopMinFromQueue(BackwardsQueue)

54      if s.BackwardState != CLOSED
55          s.BackwardState := CLOSED
56          s.ForwardPointer := sSuccessor
57          s.CombinedCostToGoal := sCombinedCostToGoal
58          s.CostToGoal := sCostToGoal

59          for each t in V such that <t, s> is in E
60              if t.BackwardState != CLOSED
61                  tCombinedCostToGoal := 0
62                  for plane := 1 to n
63*                     tCostToGoal[plane] := s.CostToGoal[plane] + <t, s>.Costs[plane] / Constraint[plane]
64*                     tCombinedCostToGoal += tCostToGoal[plane]
65                  Enqueue(BackwardsQueue point:t withCombinedCostToGoal:tCombinedCostToGoal
                                                        withCostToGoal:tCostToGoal withSuccessor:s)
66                  t.BackwardState := OPEN
```

If the vertex at the head of the objective cost queue appears to lie on a feasible path (lines 39-43), then it is expanded (line 45); if not, then the vertex at the head of the max cost queue is expanded (line 47). Intuitively, if we seem to be meeting the requirements, we can focus on keeping down the objective cost, while if we are in danger of violating the requirements, we should focus on the one closest to the limit or most over it. This selection process is not an admissible heuristic, and it is here that non-optimality and incompleteness enter the algorithm. The rationale for this algorithm is explained in detail by Korkmaz and Krunz [4].

We then extended this algorithm by adding a new, more complicated form of cost function. The original functions are additive costs over the entire path. For many applications, some of the constraints are best formulated as a restriction that should apply to any given segment of the path. For example, to avoid overheating, one might need to limit one's exposure to direct sunlight over any given period of time, or an agent may need to communicate with another party periodically, limiting how long it should be out of radio contact with a base. These sorts of constraints can be expressed as an additive cost function whose value over any given window should not exceed some constant. The obvious way to express this window is over a number of state transitions, but for most such constraints, they are more naturally expressed in terms of elapsed time. If one of the costs associated with each action is the time it requires to accomplish, then the windowed constraints can be expressed in terms of that time expenditure.

The primary problem with such windowed constraint functions is the question of how to guide the search. For simple additive functions, the total cost to reach a given point increases monotonically as one moves down the path. However, the total cost over the window ending at any given point along the path can vary up or down. To turn this into a monotonically increasing function, we can treat the maximum sum over any window up to this point as the cost upon which to base the search. While this works, it is not particularly well-informed. Specifically, unless the next possible step along a path would result in the sum over the current window exceeding the previous maximum, there is no feedback regarding embarking upon a high-cost section of a path. Furthermore, after the highest-cost window the path will ever encounter has been traversed, this function will provide no additional feedback for the rest of the path.

Fig. 3. To enable the use of windowed cost metrics, make the following changes:

```
Replace Line 11 with:
Case: p is a windowed constraint function
    START.CostToReach[p] := (0, 0)
Case: p is a simple additive constraint function
    START.CostToReach[p] := 0

Replace Lines 16-17 with:
Case: p is a windowed constraint function
    (m, c) := START.CostToGoal[p]
    objCost += w[p] * m
    maxCost := MAX(maxCost, m)
Case: p is a simple additive constraint function
    objCost += w[p] * START.CostToGoal[p]
    maxCost := MAX(maxCost, START.CostToGoal[p])

Replace Lines 30-31 with:
Case: p is a windowed constraint function
    (uBackwardsMax, _) = uCostToReach[p]
    (uForwardsMax, _) = u.CostToGoal[p]
    if uBackwardsMax > 1.0 | uForwardsMax > 1.0
        minObjFeasible := false
Case: p is a simple additive constraint function
    if uCostToReach[p] + u.CostToGoal[p] > 1.0
        minObjFeasible := false

Replace Lines 45-48 with:
Case: p is a windowed constraint function
    window := <u, v>.Costs[windowLengthDefinedInTermsOf[plane]]
    windowCost := <u, v>.Costs[p] / Constraint[p]
    x := u
    while window < windowSize[p] & x != START
        windowCost += <x.BackPointer, x>.Cost[p] / Constraint[p]
        window += <x.BackPointer, x>.Cost[windowLengthDefinedInTermsOf[p]]
        x := x.BackPointer
    (maxWindow, uWindowCost) := u.CostToReach[p]
    maxWindow := MAX(maxWindow, windowCost)
    vCostToReach[p] := (maxWindow, windowCost)
    (vForwardEstimateMax, _) := v.CostToGoal[p]
    maxWindow := MAX(maxWindow, vForwardEstimateMax)
    vCombinedCost = (maxWindow * d + windowCost) / (d+1)
    objCost += w[p] * vCombinedCost
    maxCost := MAX(maxCost, vCombinedCost)
Case: p is a simple additive constraint function
    vCostToReach[p] := u.CostToReach[p] + <u, v>.Cost[p] / Constraint[p]
    pathTotalForPlane := vCostToReach[p] + v.CostToGoal[p]
    objCost += w[p] * pathTotalForPlane
    maxCost := MAX(maxCost, pathTotalForPlane)

Replace Lines 63-64 with:
Case: plane is a windowed constraint function
    window := <t, s>.Costs[windowLengthDefinedInTermsOf[plane]]
    windowCost := <t, s>.Costs[plane] / Constraint[plane]
    x := s
    while window < windowSize[plane] & x != GOAL
        windowCost += <x, x.ForwardPointer>.Cost[plane] / Constraint[plane]
        window += <x, x.ForwardPointer>.Cost[windowLengthDefinedInTermsOf[plane]]
        x := x.ForwardPointer
    (maxWindow, sWindowCost) := s.CostToGoal[plane]
    maxWindow := MAX(maxWindow, windowCost)
    tCostToGoal[plane] := (maxWindow, windowCost)
    tCombinedCostToGoal += (maxWindow * d + windowCost) / (d+1)
Case: plane is a simple additive constraint function
    tCostToGoal[plane] := s.CostToGoal[plane] +
                                    <t, s>.Costs[plane] / Constraint[plane]
    tCombinedCostToGoal += tCostToGoal[plane]
    maxCost := MAX(maxCost, pathTotalForPlane)
```

To get around this, we can define a new sort of cost value which is a vector rather than a scalar. Let the cost over a path with a windowed cost function be an ordered pair of non-negative real numbers *(M, C)*, where *M* is the maximum cost over any window on this path and *C* is the cost over the window ending with the path. When comparing two such pairs, $(M1, C1) \leq (M2, C2)$ iff $M1 < M2$ or $M1 = M2$ and $C1 \leq C2$ (a lexicographical ordering). This has the desired effect of treating as cheaper a path with a lower maximum and, in cases where the maximum is identical, treating as cheaper the path that is currently traversing a lower-cost zone. Although this is non-monotonic, it is so in a bounded manner; if at any point the value along the path is *(M, C)*, it can never drop below *(M, 0)* afterwards. Furthermore, since the overall algorithm is already just an approximation due to its use of a non-admissible heuristic, use of this sort of cost function does not lead to a loss of any of the guarantees we previously had.

However, the algorithm as a whole relies upon summing the individual cost functions into a unified cost function suitable for classical planners to act upon. To transform these ordered pairs into scalars that preserve the same ordering relation, we can simply convert *(M, C)* to $M*d+C$ for some constant *d* which is larger than the maximum possible sum over any window (which can be computed in advance by multiplying the maximum cost of any transition in the map by the maximum number of actions that constitute a window). Alternatively, if one is concerned about the fact this is a non-monotonic function, one could just use the maximum cost over any window up to this point as the cost metric; while less informed, this is monotonic. The changes contained in the sidebar enable windowed constraints using the costs-as-ordered pairs approach.

Finally, to provide a potential speed increase when replanning after making small changes to the problem, such as after discovering that the map is incorrect, we can apply the same principles that transform A* into D*. D* behaves identically to A* when first run, but it caches path cost information for all explored states to avoid having to recompute unchanged information after small updates to the problem [5].

For the first half of the algorithm, which employs no heuristic, we simply substitute D* for Dijkstra's algorithm, with the sum of the normalized constraints as the cost function. To avoid inefficiency, the algorithm should compute and store the cost for each function separately, then make its decisions based on the sum, as before.

For the latter half of the algorithm, with its complex heuristic, we require a modified form of Focussed D*, which is to ordinary D* what A* with a heuristic is to Dijkstra's algorithm [6]. Focussed D* orders states that are candidates for expansion based on the projected total cost of the best path passing through them. If, as a result of advancing along the planned path, the agent discovers that

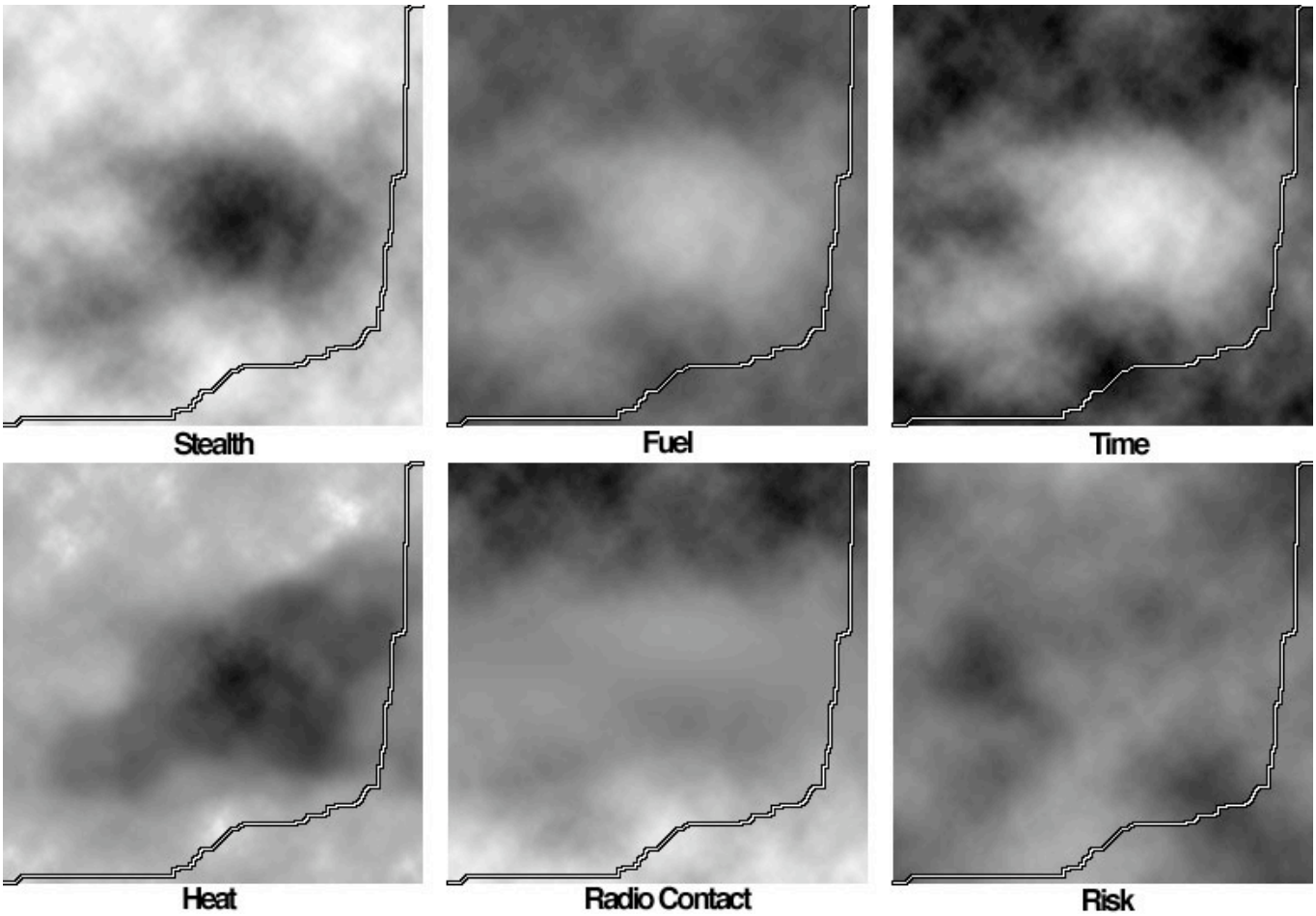

Fig. 4. Example path planning problem. Each map represents the cost to traverse areas relative to one of six labeled metrics; darker shades are higher costs. The path shown was generated by K2 trying to meet all six constraints while minimizing the total chance of being detected (stealth cost).

the cost information it had was incorrect, it modifies the cost map and propagates the changed values outward. At this point, the agent has moved, so the START state has changed, altering the expected costs of all paths. However, in most cases (and certainly those where rapid replanning is most necessary) the agent has moved a few states at most, introducing only small errors, which Focussed D* corrects upon expanding the erroneous states. The end result is an efficient replanning algorithm.

To employ Focussed D*, we need to make the same changes we made to A*. Namely, instead of a single queue, the algorithm must maintain two queues, one sorted by projected objective function cost and the other sorted by projected maximum normalized constraint function cost. If the path through the minimal objective-function-cost state is projected to be feasible, then it is expanded; otherwise, the minimal state from the other queue is expanded. Individual costs for each metric need to be tracked separately and only combined for the purpose of inserting states into the queues.

Unfortunately, this turns out to be of little practical value. The computations made in the second half of the algorithm are dependent on the START state, and when that changes, the cached values are invalidated and need to be recomputed. The calculations made in the first half are not dependent on the START state, but they are dependent on the constraint thresholds due to the normalization employed. For nearly any problem of interest, replanning is needed because the agent has learned about the world after taking actions, in which case its available reserves of the resources represented by the constraint thresholds will have changed, invalidating all the cached values. Thus, replannning is only valid if the agent doesn't move between planning attempts. We are still investigating possible means of being able to reuse some of the values.

EXAMPLE APPLICATIONS

Figure 4 shows the sort of problem which K2 can be used to solve. Imagine a scenario where a robot needs to travel from the southwest corner of a region to the northeeast corner in order to investigate some object of interest. The region is filled with light vegetation and dominated by an open clearing. There are also unspecified adversaries in the region, so the robot should be stealthy in its movements.

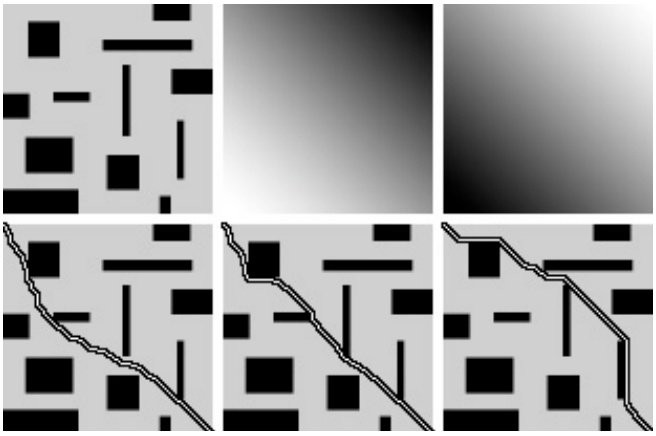

Fig. 5. The effects of constraints. This demonstrates K2 with three cost metrics (top row); the leftmost is used as the objective function. Depending on the relative importance set for the other two constraints, different paths are produced (bottom row).

In addition, the robot has a finite quantity of fuel and time available in which to complete its traversal task. The environment is hot and the sun is shining, and overheating poses a risk to the robot as well. Lastly, the robot should try to communicate periodically with an operator located to the south of the region, in case new orders need to be assigned or new information about the environment becomes available.

The six cost maps in the figure quantify these constraints. Darker parts represent higher costs and lighter parts lower costs. Stealth is harder to maintain in the central clearing, but it takes less time to traverse than the surrounding vegetation. Having to move more slowly through the vegetation also results in greater fuel expenditure to move the same distance. Heat is worse in the clearing and the area immediately east of it, as the sun is shining from the western sky. Vegetation has an effect on radio communication, as the robot uses a narrow-beam system to avoid broadcasting its position to adversaries, but the primary factor affecting radio contact is distance from the operator. Lastly, there is a map that indicates the calculated risk associated with passing through an area based on known information about the terrain and adversarial action.

We gave this problem to K2, setting hard constraints on the total amount of time and fuel available and the amount of risk we were willing to tolerate. The amount of heat we were willing to let build up and the length of time we were willing to risk being out of radio contact were set up as windowed constraints. Lastly, we designated stealth as the sole objective function, because we would like to minimize exposure, but have no pre-set limits on how much it is safe to be visible. The resulting path is drawn on all six cost maps, to show how it interacts with each of them.

Figure 5 is a much simpler case of navigating between block-like obstacles with two competing contraints consisting of directly opposed gradients, and demonstrates how adjusting the constraints produces different paths without any changes to the cost maps.

PERFORMANCE RESULTS

The algorithm's strength lies in its ability to handle multiple cost metrics easily. When given a map of 250 cells by 250 cells of uniform cost with only one cost metric, K2 required an average of 0.61 seconds over ten runs on a dual-processor 2.5 GHz G5 with 1 GB RAM to find a path from one corner to the opposite. In contrast, running it on a map of equal size with six synthetically-generated fractal cost maps took an average of 1.01 seconds. That was with all simple additive constraints; windowed constraints involve slightly more work since computing their current value requires checking a chain of pointers after each step. With one of the constraints windowed, the average planning time was 1.21 seconds, and with two of the six constraints windowed, it was 1.38 seconds. Even when handling excessive numbers of constraints, the algorithm scales well; with fifty distinct cost metrics, one-third of them windowed, K2 still took only 6.93 seconds.

Gauging the quality of the solutions generated is substantially more difficult. We do not have access to an

efficient, complete, and optimal algorithm against which to compare the paths produced by K2. The best we can do is to compare with the results of other algorithms in cases where both are applicable. CD* can handle a single additive constraint, and MCD* can theoretically handle an arbitrary number of simple additive constraints, but its exponential runtime and certain implementation issues limit it to just a few constraints in practice.

While K2 is not complete in the sense that it can fail to find a feasible path when one it exists, it always returns a path which, even if it is not feasible, is as close as K2 can get. Since it does this even when a feasible path does not exist, it is not sound, but such a resulting solution may still be of value to the user, depending on the circumstances.

In trials (Fig. 6), on singly-constrained problems, CD* was faster than K2 and the paths it produced were slightly better in terms of total costs, but this is to be expected. CD* makes efficient use of binary search to obtain optimal results, and the version we have is a production-grade one refined over the years for speed, while K2 is still a fresh implementation without much code optimization. On multiply-constrained problems, we had to switch to MCD*, which is optimal to whatever resolution it explores the space of possible weights for the individual cost functions but performs much slower, because it requires orders of magnitude more passes to find an answer. It has not had any extensive optimization, either. The paths it produced for multiply-constrained problems were similar, though not identical, to those produced by K2.

One Objective & One Constraint			One Objective & Two Constraints		
	100x100	250x250		100x100	250x250
CD*	0.02	0.12	MCD*	2.77	19.75
K2	0.09	0.68	K2	0.10	0.83

Fig 6. Average runtimes in seconds of different algorithms on problems involving cost maps of two different numbers of cells.

CONCLUSION

This paper presents K2, a synthesis of existing approximation and replanning algorithms for constrained planning tasks. It also describes a novel form of more localized constraint over segments of the plan which can be used to express useful real-world properties. Although directed towards robotic motion planning, the algorithm itself is general-purpose and suitable for any sort of planning task, similar to A* and its brethren.

There is the potential to introduce rapid replanning capabilities to the algorithm, but in its current form, altering the problem in real-world ways invalidates all work previously done computing solutions.

ACKNOWLEDGMENTS

This work was sponsored in part by the U.S. Army Research Laboratory, under contract "Robotics Collaborative Technology Alliance" (contract number DAAD19-012-0012) and by the National Science Foundation via a Graduate Research Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements of the U.S. Government.

REFERENCES

[1] B. Logan & N. Alechina, "A* with Bounded Costs," *Proceedings of the 15th national/10th conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, p. 444-449, July 1998.

[2] F. Kuipers, T. Korkmaz, M. Krunz, & P. Van Mieghem, "Performance Evaluation of Constraint-Based Path Selection Algorithms," *IEEE Networks*, to appear 2004.

[3] A. Stentz, "CD*: A Real-time Resolution-Optimal Re-planner for Globally Constrained Problems," *18th national conference on Artificial Intelligence*, 2002.

[4] T. Korkmaz & M. Krunz, "Multi-Constrained Optimal Path Selection," *Proceedings of the IEEE INFOCOM 2001 Conference*, p. 834-843, 2001.

[5] A. Stentz, "Optimal and Efficient Path Planning for Partially-Known Environments," *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 4, p. 3310-3317, May 1994.

[6] A. Stentz, "The Focussed D* Algorithm for Real-Time Replanning," *Proceedings 1995 International Joint Conference on Artificial Intelligence*, p. 1652-1659, August 1995.