2010 IEEE International Conference on Robotics and Automation
Anchorage Convention District
May 3-8, 2010, Anchorage, Alaska, USA

Rapid Updating for Path-Planning using Nonlinear Branch-and-Bound

Alison Eele†
Department of Engineering
Cambridge University
United Kingdom

Arthur Richards*
Department of Aerospace Engineering
University of Bristol
United Kingdom

***Abstract*— This paper develops and tests a novel rapid updating technique for use with a nonlinear branch and bound optimisation method, tailored for finding optimal trajectories for a vehicle constrained to avoid fixed obstacles. The key feature of the rapid updating technique developed is the ability to increment and re-arrange the existing search tree reducing the amount of computation taken to find a new plan. The rapid updating techniques developed are combined into a receding horizon control and compared to full cold start method. The rapid updating technique demonstrated an average 62% solve time improvement over a cold start. The rapid updating method is also demonstrated for removal of obstacles from the environment and very large scale problems with 60 obstacles.**

NOMENCLATURE

$(r_x(t), r_y(t))$	Position at time t
$(v_x(t), v_y(t))$	Velocity at time t
$(r_{x0}, r_{y0}, v_{x0}, v_{y0})$	Initial position and velocity
$(r_{x,m}, r_{y,m})$	m^{th} Collocation point
(r_{xT}, r_{yT})	Target position
t_0	Start time
t_f	Final time
$(r^{\text{obs}}_{x,p}, r^{\text{obs}}_{y,p})$	Position of centre of obstacle p
R_p	Radius of obstacle p
N	The total number of obstacles
$v_{\max}, v_{\min}$	Maximum and minimum speeds.
$a_{\max}$	Maximum acceleration.
P_s	Optimised path for a particular subproblem
P_{inc}	Incumbent path

I. INTRODUCTION

This paper develops a rapid updating technique for single vehicle trajectory optimisation from obstacle branch-and-bound (OBB) path optimisation, previously developed by the authors [1]. The OBB method's key unique property is the ability to solve for the globally optimal solution of the nonconvex problem whilst retaining the full nonlinear dynamics model of the vehicle. The original OBB method however assumed all obstacles positions and radii were known at the time of solving. This paper seeks to remove that assumption allowing updating of the path with addition and removal of obstacles as they are detected. These problems are important for the control of autonomous Uninhabited Aerial Vehicles (UAVs) and air traffic management in free flight operations [2], [3].

Many approaches have been investigated for solving path-planning problems, all attempting some form of simplification to achieve practical computation. The fastest of these include randomised searches [4]–[6] which rapidly find feasible paths through fields of obstacles, retaining accurate dynamics models but having no guarantee of optimality. Also graph-based methods, such as Voronoi diagrams [7], [8] or visibility graphs [9], approximate the trajectories as joined line segments. The Mixed-Integer Linear Programming (MILP) [10] approach indirectly uses branch-and-bound optimisation, reformulating the problem in linearised form and using powerful, commercial software to solve the MILP problem. Commercial software such as CPLEX [11] uses various branching strategies guided by finely-tuned heuristics to solve generic MILP problems rapidly.

There are two elements to a generic branch-and-bound method [12]: branching, where the problem is iteratively divided into subproblems, partitioning the search space and generating a search tree for the algorithm to traverse; Bounding, where the amount of the search tree required to be solved is reduced by putting a lower bound on any branch's solution and fathoming out branches where this lower bound is worse than the best feasible solution found so far. The global optimality of any solution obtained from a branch-and-bound method is guaranteed assuming that the globally optimal solution to each evaluated subproblem is found [13]. This globally optimal solution in a receding horizon problem will only apply for the current horizon as information about obstacles existing beyond the horizon is unavailable. In receding horizon setups the problem is solved in detail to a planning horizon, and then has a rough plan from the planning horizon to the goal. The solution found is then executed until the execution horizon which is within or equal to the planning horizon. Once the execution is complete the problem is then resolved from the vehicle's new state within its updated planning horizon. Receding horizon techniques have commonly been employed alongside MILP trajectory optimisation [9].

The rapid updating technique for OBB accommodates changes in the environment while the path is being followed. Specifically, if an obstacle is added or removed, the revised globally optimal path for the remainder of the problem can be found without starting OBB from scratch (or "cold start").

†Research Assistant, Email: aje46@cam.ac.uk
*Senior Lecturer, Email: arthur.richards@bristol.ac.uk

The key to rapid updating is the efficient management of the search tree, enabling a search to be re-started on the addition or removal of an obstacle. The incumbent node is re-evaluated first, since the result is a good lower bound on path length. This approach tries to maximize the preservation of useful information from the previous solution, thus reducing the number of other nodes to be revisited. More savings in computation can be achieved by re-ordering the tree between updates, as a good tree order reduces further the number of nodes to be evaluated.

The paper is organised as follows. Section II describes the problem statement. Section III briefly reviews the branch-and-bound optimisation algorithm. Section IV details tree re-ordering strategy. Section V details obstacle addition and Section VI details obstacle removal. Section VII presents the results of the rapid updating method and a comparison to a cold start method. Finally, Section VIII presents conclusions.

II. Problem Statement

This paper considers the problem of finding the minimum time path for a vehicle modelled as a "Dubins-like" car, *i.e.* moving in two dimensions with limited rate of turn, though with bounded speed. The velocity and positioning relationships are linked by the following constraints:

$$v_x(t) = \dot{r}_x(t), \quad v_y(t) = \dot{r}_y(t)$$

Speed is constrained between both minimum and maximum bounds,

$$v_{\min}^2 \leq v_x^2(t) + v_y^2(t) \leq v_{\max}^2$$

The acceleration is also bounded, effectively restricting the turning rate:

$$|\dot{v}_x(t)| \leq a_{\max}, \quad |\dot{v}_y(t)| \leq a_{\max}$$

The initial state is completely specified by the following constraints:

$$\begin{aligned} r(t_0) &= (r_{x0}, r_{y0}) \\ v(t_0) &= (v_{x0}, v_{y0}) \end{aligned}$$

and the terminal constraint is that the vehicle should reach a specified target regardless of velocities at time t_f:

$$r(t_f) = (r_{xT}, r_{yT})$$

Finally, the obstacle avoidance is expressed as a minimum distance from each obstacle centre. The method has been demonstrated throughout with the use of circular obstacles:

$$(r_x(t) - r_{x,p}^{\text{obs}})^2 + (r_y(t) - r_{y,p}^{\text{obs}})^2 \geq R_p^2 \ \forall t, \ \forall p$$

Overlaps can be accommodated [1] and arbitrary shapes can be approximated by unions of ellipsoids. The final time t_f is constrained such that

$$t_f \geq 0$$

The objective is to find the minimum time path where the elapsed time, $t_f - t_0$ is minimum:

$$J^* = \min t_f - t_0$$

This paper assumes that the objective is the shortest-time path without loss of generality. The cost function is independent of the branch-and-bound method and can be substituted to solve a variety of problems such as those of minimum fuel use and minimum risk [14]. The dynamics are discretized using direct global collocation with Gaussian Radau points for constraint enforcement [15], [16]. The position and velocity profiles are parameterized by

$$\begin{aligned} r(t) &= \sum_{m=1}^{n_{\text{col}}} \psi_m \left(\frac{t - t_0}{t_f - t_0} \right) r_m \\ v(t) &= \sum_{m=1}^{n_{\text{col}}} \psi_m \left(\frac{t - t_0}{t_f - t_0} \right) v_m \end{aligned}$$

where $r_m = (r_{x,m}, r_{y,m})$ and $v_m = (v_{x,m}, v_{y,m})$ represent the values of the associated variables at collocation point m. Function ψ_m is the mth Lagrange polynomial of order n_{col} satisfying

$$\psi_m(\tau_r) = \delta_{m,r}$$

where τ_r is the rth collocation point and $\delta_{m,r}$ is the Kronecker delta and n_{col} is the number of collocation points used to discretise the path. Thus, the path optimisation is approximated by the finite dimensional optimisation with decision variable $P = \{r_{x,1}..r_{x,n_{\text{col}}}, r_{y,1}..r_{y,n_{\text{col}}}\}$. The corresponding conversion of the constraints is familiar and is covered in detail in Refs. [15], [16] for example.

III. Obstacle Branch and Bound Planning Algorithm Review

This section presents full details of the sub steps of the planning algorithm. Optimisation of the paths and choice of branching strategy are available in [1]. Define a *subproblem* (A, P) consisting of a set of active obstacles $A \subseteq \{1, \ldots, M\}$ and an initial "diverted" path P that satisfies the initial and terminal constraints and the avoidance constraints for the active obstacles A but is not necessarily dynamically feasible. The presence of P defines the clockwise/anti-clockwise decisions for the active obstacles in that subproblem. P_{inc} represents the incumbent path, *i.e.* the best feasible path found so far as the algorithm progresses. Algorithm 1 details how these are used and updated.

Under the assumption that each path P_s is the optimal solution to its corresponding subproblem, the properties of the generic branch-and-bound algorithm hold [13] and at termination, P_{inc} is the globally-optimal path.

IV. Search Tree Ordering

This section first illustrates the importance of tree ordering for updating and then presents an algorithm to achieve good ordering. Fig. 1(a) shows the optimal path from SI to SF avoiding the obstacles, with the associated solution tree in Fig. 1(b) and the optimal leaf node ringed. The branching strategy has been carefully chosen to minimize solution time [1] and in this case, obstacle 3 has entered the tree last. Suppose now that the vehicle has reached point SP and a change in the size or position of obstacle 2 has been

Algorithm *1. Branch and Bound Path Planning*

1. $P_{inc} \leftarrow$ NULL
2. Set list of active subproblems to $(\emptyset, P_0)$ where path P_0 connects the start to the goal in a straight line
3. **while** list of active subproblems is not empty
4. **do** select a subproblem (A, P_i) from the list of active subproblems with the least incursion on an active obstacle and remove it from that list
5. solve the subproblem (A, P_i) for the shortest feasible path P_s avoiding active obstacles A
6. **if** path P_s intersects inactive obstacles $\{1, \ldots, N\} \setminus A$ and P_s is shorter than P_{inc}
7. **then** select the first obstacle p_b (the "branching obstacle") intersected by path P_s
8. add $(A \oplus \{p_b\}, P_a)$ to the list of active sub-problems where P_a is a diverted path avoiding anti-clockwise around obstacle p_b
9. add $(A \oplus \{p_b\}, P_c)$ to the list of active sub-problems where P_c is a diverted path avoiding clockwise around obstacle p_b
10. **else if** path P_s is shorter than P_{inc}
11. **then** $P_{inc} \leftarrow P_s$
12. **return** P_{inc}

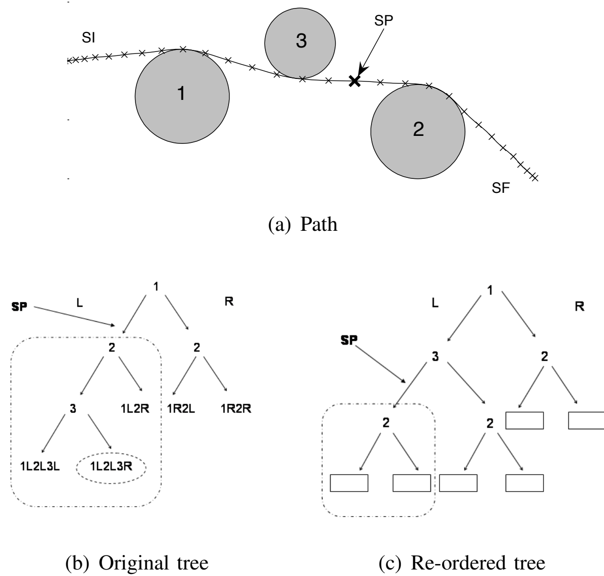


Fig. 1. Re-ordering tree to suit updating

detected. The search must restart at the point in the tree where obstacle 2 was introduced, marked SP in Fig. 1(b). Proceeding downwards, it will revisit decisions regarding obstacle 3, despite the fact that they are irrelevant to the remaining problem. However, suppose instead that the tree has been re-ordered to the form in Fig. 1(c), in which the obstacles appear in the order in which they are encountered along the path. Now, the change found at SP involves a smaller subtree, concerning only the relevant obstacles to the remaining problem.

It is impossible to ensure that the initial solution generates a tree ordered as in Fig. 1(c). For example, in Fig. 1(a), only once constrained to pass left of both obstacles 1 and 2 does the path intersect obstacle 3. Instead, it is necessary to re-order the tree after the solution. This has no effect on the final path: stating the vehicle must go to the left of obstacle 1 and to the right of obstacle 2 is the same as stating the vehicle must go to the right of obstacle 2 and the left of obstacle 1. A node can be moved up the tree without any resolving as long as the obstacle in question is totally present. Obstacles which are not apparently present in subtrees are considered in a state of flux. They exist within the problem but have not been activated during Algorithm 1. We can therefore if required enumerate and solve them to fill out the tree where re-ordering requires.

Algorithm 2 defines the method behind re-ordering a tree to a desired order. The algorithm is performed on the order for the optimal path branch. Essentially, it swaps nodes until the desired order is achieved. Once the initial solution is complete, re-ordering can be performed as a background task while the vehicle is moving, providing it is complete before any changes are detected requiring an update to the path.

V. Rapid Updating for Addition of Obstacles

Algorithm 3 describes the method used to add an obstacle to a solved problem. The key advantage here is that fathoming and tree restriction is used to try and minimise the amount of the overall search tree that needs to be re-evaluated. Previously discovered information is not discarded by updates. A change to the incumbent path is only required if the new obstacle intersects the path (Line 4). First, a simple diversion of the incumbent either side of the new obstacle is tried (Lines 5 and 6). Then, the subtree for the remaining path is re-tested, to see if an alternative route could now be better (Lines 7–12).

VI. Rapid Updating for Removal of Obstacles

This case, solved by Algorithm 4, is more complicated than that of obstacle addition. Even if the removed obstacle was not in contact with the incumbent path, an update may be necessary: the removal may open up a route that was previously suboptimal. Therefore the condition for an update (Line 4) is based on whether the removed obstacle was *active* or not, *i.e.* whether its presence had ever influenced the solve process. It would then be possible to go back to the nodes associated with the removed obstacles and resolve those subtrees from scratch, but this would discard potentially useful information regarding other obstacles in the old subtrees. Instead, the algorithm tries to limit the amount of data lost when pruning the obstacle out of the search tree. It re-attaches re-usable subtrees if it can (Line 9) and re-activates leaf nodes to resolve only the lowest level of problems (Lines 12–15), rather than down a whole subtree.

Algorithm *2. Tree Reordering Algorithm*

1. $O_e \leftarrow$ the desired order of nodes
2. $O_c \leftarrow$ the current order of nodes
3. **while** $O_e \neq O_c$
4. **do for** $i = length(O_c) : 1$
5. **if** $O_c(i) < O_c(i-1)$
6. **then** disconnect the child nodes of the nodes $O_c(i)$ and $O_c(i-1)$.
7. swap the places of the nodes in the tree labelling $O_c(i-1)$ the new left child of $O_c(i)$
8. create an additional node N_a branching on the same obstacle as $O_c(i-1)$ and label it as the right child of $O_c(i)$
9. reconnect the two out of the three children disconnected in step 6 to their appropriate parents either N_a or $O_c(i-1)$
10. **if** remaining unattached child node's subtree does not contain the obstacle described in $O_c(i-1)$
11. **then if** is the obstacle always passed anti-clockwise or clockwise throughout the subtree
12. **then** enumerate the obstacle to the side it is always passed on in the subtree and attach the child node to the relevant parent out of N_a or $O_c(i-1)$. Create an additional node N_b as the remaining child node.
13. **else** duplicate the subtree headed by the unattached child node and activate the obstacle described in $O_c(i-1)$ and N_a attach the two child nodes one to each parent node and resolve leaf nodes.
14. **else** discard the subtree headed by the remaining unattached child node and create nodes N_c and N_d to place in the remaining child slots of N_a and $O_c(i-1)$
15. swap $O_c(i)$ and $O_c(i-1)$ in the overall list of O_c

Algorithm *3. Obstacle Addition Algorithm*

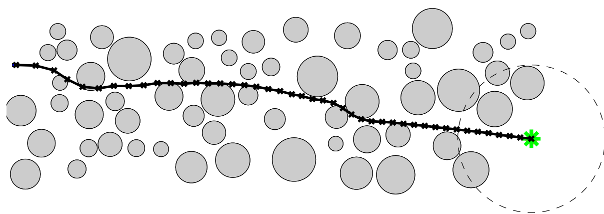
1. $p_d \leftarrow$ the newly detected obstacle
2. $N_{inc} \leftarrow$ the leaf node corresponding to the incumbent path P_{inc}
3. $N_{head} \leftarrow$ the node corresponding to the head of the subtree describing the remaining stretch of the incumbent path P_{inc}
4. **if** p_d intersects the incumbent path P_{inc}
5. **then** branch on obstacle p_d from node N_{inc} and evaluate the two newly created nodes N_a and N_c.
6. $P_{inc} \leftarrow$ the shorter of the two paths generated by N_a and N_c
7. **for** all leaf nodes with N_{head} as an ancestor
8. select a leaf node N_i
9. **if** the path described by N_i is longer than P_{inc} or the N_i is an infeasible path
10. **then** the node N_i is fathomed/inactive again and does not need to be resolved.
11. **else** the node N_i is reactivated to be resolved.
12. Resolve all active nodes using the standard branching algorithm with P_{inc} as the starting incumbent path.
13. **return** P_{inc}

Algorithm *4. Obstacle Removal Algorithm*

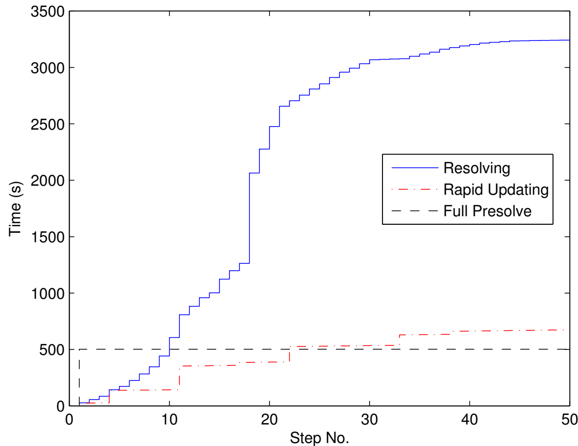
1. $p_d \leftarrow$ the obstacle to be removed
2. $N_{inc} \leftarrow$ the leaf node corresponding to the incumbent path P_{inc}
3. $N_{head} \leftarrow$ the node corresponding to the head of the subtree describing the remaining stretch of the incumbent path P_{inc}
4. **if** p_d was active in the incumbent path P_{inc}
5. **then** $N_{remove} \leftarrow$ the node which branched on p_d and is an ancestor of N_{inc}
6. **if** N_{remove} has child nodes (to be known as N_a and N_c)
7. **then** $N_{anc} \leftarrow$ be the node out of either N_a or N_c which is the ancestor of N_{inc}
8. **if** N_a and N_c branch on the same obstacle
9. **then** replace N_{remove} in the tree with the subtree headed by N_{anc} and activate the leaf nodes of the subtree.
10. **else** $p_{next} \leftarrow$ the next obstacle encountered by the path for N_{remove} if p_d does not exist
11. **if** either N_a or N_c branches on p_{next}
12. **then** replace N_{remove} in the tree with the subtree headed by either N_a or N_c respectively and activate the leaf nodes of the subtree..
13. **else** replace N_{remove} in the tree with the subtree headed by N_{anc} and activate the leaf nodes of the subtree.
14. Remove any other nodes with p_d in the subtree headed by N_{head} and activate any nodes which have changed.
15. Resolve all active nodes using the standard branching algorithm.
16. **return** P_{inc}

VII. RESULTS

The results in this section were generated using MATLAB [17] for the algorithm implementation, and SNOPT [18] as the nonlinear optimiser through an AMPL [19] interface on a 2.4GHz PC with 2GB RAM. The obstacle addition technique is combined with a receding horizon method with varying sensor ranges. This is then compared to an alternative method where instead of updating



(a) Final Path



(b) Solve Time Comparison

Fig. 2. Cumulative Solve Time Comparisons for 60 Obstacle Problem

the existing tree the problem is cold started from the new position every time a new obstacle is sensed.

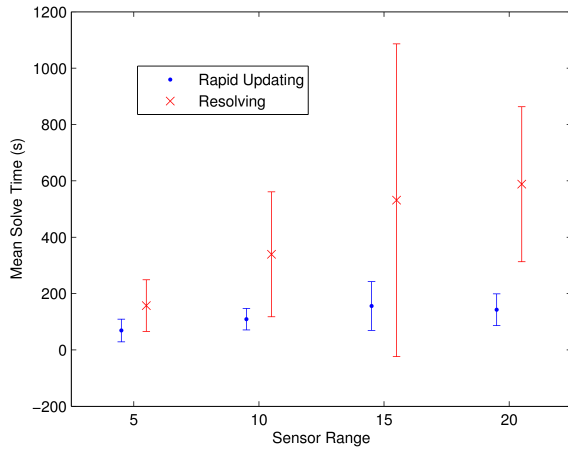
The results shown in Fig. 2 demonstrate a large scale example where a 60 obstacle case with a sensor range of 10 units has been used. The path shown in Fig. 2(a) is the final path found by the rapid updating method with the detection radius at the current point shown by the dashed line. Fig. 2(b) shows a cumulative time plot of the solve time for this problem when solved by three individual methods. The first using the rapid updating method has a low gradient with many steps requiring no significant computation, due to the added obstacles not intersecting the current path. Conversely the second method, that of a cold start from the current point each time a new obstacle is sensed, follows a far steeper gradient and requires computation at every step, though this tails off near the end as a clear path lies between the current point and the goal. The full presolve uses the original nonlinear branch and bound technique proposed in [1] and assumes all obstacles are known before the problem begins. All computation is done in the first step and no updates take place after that. The overall solution time of the rapid updating method remains closer to that of the full presolve than that of the cold start. The rapid updating and cold start methods will find the same path, however are not guaranteed to find the same path as that of the full presolve since only sensor information is known when solving.

The next set of results compare the differences in overall solve time and the number of steps required for rapid-updating and cold start implemented on 20 instances of 30 obstacles with sensor ranges varied between 5 to 20 units in 5 unit steps (80 cases considered overall). Fig. 3 shows the mean and standard deviations of the two methods over the sensor ranges. Fig. 3(a) displays the differences in the mean solve times of the 20 cases for each method and each sensor range. The mean solve time for the rapid updating method varies only marginally between the different sensor ranges, because a large number of the obstacles added as the sensor ranges grow do not require a full updated solve as they do not interact with the path. Fig. 3(b) firstly highlights that predominantly the number of update steps required to solve the cases does not vary between the two methods (any fractional differences relate to numerical rounding in the simulation of the sensor). An update step occurs each time a vehicle detects a new obstacle and since both methods find the same solution they would encounter the same number of obstacles from start to goal. Fig. 3(b) highlights the trade off of sensor range and mean number of update steps for the solution. Whilst the sensor range is small very few obstacles are known initially so the initial solution will need more updates during the trip to the goal. However a small sensor range also will restrict the vehicle from detecting outlying obstacles and thus reduce the number of updates to reach the goal. Fig. 3(c) combines the previous two figures to determine the mean time per update step, including the initial step where multiple obstacles are detected and solved. Other update steps will only introduce one obstacle at a time and thus as the sensor range increases the standard deviation of the time per step increases. Overall the rapid updating method showed an average improvement of 62% of solve time compared to the cold start method.

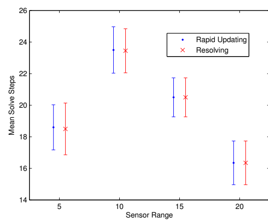
The final example shown in Fig. 4 demonstrates the update steps of an obstacle problem involving obstacle removal. It is presumed the vehicle either has an initial, inaccurate map and a more accurate short range sensor. In the example shown the problem was presolved with all the obstacles available. Then as it progressed the short range sensor updates the map and if needed the path will update using the obstacle removal method proposed. Fig. 4(a) shows the initial solution of the presolve with all the obstacles apparently known in the map. The vehicle proceeds along this path until it detects the absence of obstacle 1 (Fig. 4(b)). The obstacle removed has no effect on the path so the vehicle continues until it detects the absence of obstacle 2 (Fig. 4(c)). Obstacle 2 was effectively blocking a hole in a wall of obstacles and with it gone a faster route is presented. The vehicle adjusts its path and proceeds to the goal (Fig. 4(d)). The problem took 6 seconds to update from the initial presolve compared to the cold start update method which took 15 seconds to update the same problem.

VIII. Conclusions

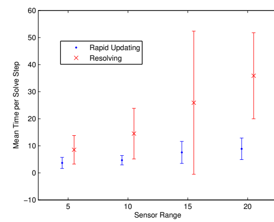
In this paper we have implemented a rapid path updating technique, returning the optimal trajectory with dramatically less solve time than a cold start. Re-ordering of the search tree enables efficient restarting when needed. We have con-



(a) Total Solve Time

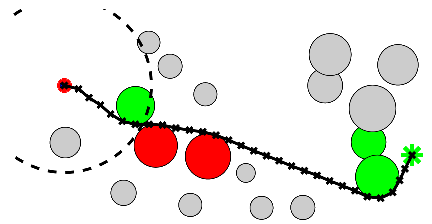


(b) Solve Steps Required

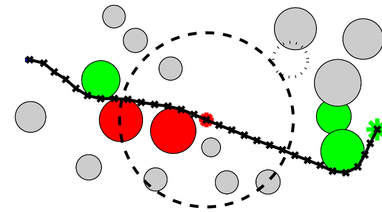


(c) Mean Time per Solve Step

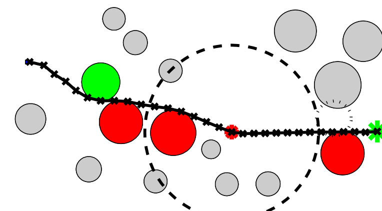
Fig. 3. Comparisons between Rapid Updating and Cold Start Techniques with Variable Sensor Range



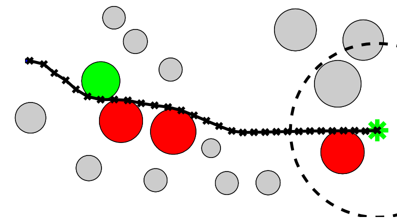
(a) Presolve Result



(b) First Obstacle Removal



(c) Second Obstacle Removal



(d) Resultant Path

Fig. 4. Snapshots of Path Updating with Obstacle Removal

sidered the effects of sensor range on the effectiveness of the technique and shown a trade off between solve time per step and number of steps solved. We have demonstrated the applicability of the technique on solving very large problems and on obstacle removal.

REFERENCES

[1] Eele, A. J. and Richards, A. G., "Path-Planning with Avoidance Using Nonlinear Branch-and-Bound Optimization," *AIAA Journal of Guidance, Control and Dynamics*, Vol. 32, No. 2, March 2009, pp. 384–394.

[2] Pachter, M. and Chandler, P. R., "Challenges of Autonomous Control," *IEEE Control Systems Magazine*, Vol. 18, No. 4, 1998, pp. 92–97.

[3] Perry, T. S., "In Search of the Future of Air Traffic Control," *IEEE Spectrum*, Vol. 34, No. 8, 1997, pp. 19.

[4] Barraquand, J., Kavraki, L., Latombe, J.-C., Motwani, R., and Raghavan, P., "A Random Sampling Scheme for Path Planning," *Intl. Journal of Robotics Research*, Vol. 16, No. 6, 1997, pp. 759–774.

[5] Lee, Y. I. and Kouvaritakis, B., "Constrained Receding Horizon Predictive Control for Systems with Disturbances," *International Journal of Control*, Vol. 72, No. 11, 1999, pp. 1027–1032.

[6] Frazzoli, E., Dahleh, M., and Feron, E., "Real-Time Motion Planning for Agile Autonomous Vehicles," *Proceedings of the AIAA Guidance, Navigation and Control Conference*, No. AIAA-2000-4056, Denver, Colorado, August 2000.

[7] Bortoff, S. A., "Path-Planning for UAVs," *Proceedings of American Control Conference*, Chicago, Illinois, June 2000, pp. 364–368.

[8] Chandler, P. R., Pachter, M., and Rasmussen, S., "UAV Cooperative Control," *Proceedings of American Control Conference*, Vol. 1, Arlington, Virginia, June 2001, pp. 50–55.

[9] Bellingham, J., Richards, A., and How, J. P., "Receding Horizon Control of Autonomous Aerial Vehicles," *Proceedings of the American Control Conference*, Vol. 5, Anchorage, May 2002, pp. 3741 – 3746.

[10] Richards, A., Schouwenaars, T., How, J., and Feron, E., "Spacecraft Trajectory Planning with Avoidance Constraints Using Mixed-Integer Linear Programming," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 25, No. 4, 2002, pp. 755–764.

[11] ILOG CPLEX Website, `http://www.ilog.com/products/cplex/`, visited July 2007.

[12] Bertsimas, D. and Tsitsiklis, J. N., *Introduction to Linear Optimization*, Athena Scientific, Belmont, Massachusetts, 1997.

[13] Dakin, R. J., "A tree-search algorithm for mixed integer programming problems," *The Computer Journal*, Vol. 8, No. 3, 1965, pp. 250–255.

[14] Lawler, E. L. and Wood, D. E., "Branch-And-Bound Methods: A Survey," *Operations Research*, Vol. 14, No. 4, 1966, pp. 699–719.

[15] Huntington, G., *Advancement and Analysis of a Guass Pseudospectral Transcription for Optimal Control Problems*, Ph.D. thesis, Department of Aeronautics and Astronautics, Massachusetts Institue of Technology, Cambridge, Massachusetts, May 2007, 2007.

[16] Ross, M. and Fahroo, F., "Direct Trajectory Optimization by a Chebyshev Pseudospectral Method," *Proceedings of the American Control Conference*, Vol. 6, Chicago, Illinois, June 2000, pp. 3860 – 3864.

[17] MATLAB Website, `http://www.mathworksl.com/`, Visited July 2007.

[18] SNOPT Website, `http://www.sbsi-sol-optimize.com/asp/sol_product_snopt.htm`, Visited June 2008.

[19] AMPL Website, `http://www.ampl.com/`, Visited July 2007.