Batch Informed Trees (BIT*)

James Swedeen
Department of Electrical and Computer Engineering
Utah State University
Logan, Utah 84322
Email: james.swedeen@usu.edu

Greg Droge
Department of Electrical and Computer Engineering
Utah State University
Logan, Utah 84322
Email: greg.droge@usu.edu

Abstract—Path planning through complex obstacle spaces is a fundamental requirement of many mobile robot applications. Recently a rapid convergence path planning algorithm, Batch Informed Trees (BIT*), was introduced. This work serves as a concise write-up and explanation of BIT*. This work includes a description of BIT* and how BIT* operates, a graphical demonstration of BIT*, and simulation results where BIT* is compared to Optimal Rapidly-exploring Random Trees (RRT*).

I. INTRODUCTION

The ability to plan paths through complex obstacles is a fundamental requirement of many mobile robot applications and is an NP-complete problem in general [1]. The literature has seen an explosive growth in sampling-based motion planning algorithms [2]–[6] that provide asymptotic guarantees for this NP-complete problem. These algorithms operate on the principles of dynamic programming, breaking the problem into many smaller problems that can each be solved individually and then combined to make the overall solution.

Many sampling-based motion planning algorithms are based on Optimal Rapidly-exploring Random trees (RRT*) [7]. RRT* iteratively samples the continuous space it plans over building a root search tree from the initial robot location to every other reachable part of the obstacle-free space. As RRT* searches, local optimizations are performed on the search tree shortening the length of the paths through the tree. As the number of iterations RRT* performs goes to infinity, the resulting solution path from the initial location to the target location converges to optimality [7]. RRT* and its variants have been shown to solve a large range of path planning problems rapidly. However, convergence can be slow [2], [4], [8].

Fast Marching Trees (FMT*) is a sampling-based path planning algorithm that makes use of dynamic programming principles to avoid unnecessary calculations [9]. FMT* builds a rooted search tree similar to RRT*. However, instead of iteratively sampling the state space and building the search tree simultaneously, FMT* samples a fixed number of times before starting to build a search tree. Once all of the samples have been generated, FMT* builds a search tree with the knowledge of the location of each sample from the beginning. Using this knowledge, FMT* is able to avoid many of the calculations that RRT* performs while locally optimizing its search tree. This makes FMT* faster than RRT* at generating a solution

from a set number of samples. However, FMT* is unable to continue refining its solution after that set number of samples. RRT*, on the other hand, is purely iterative and will continue to refine its solution path for as long as it is allowed.

Batch Informed trees (BIT*) combines the iterative nature of RRT* with the efficient graph searching used in FMT* [10]. BIT* iteratively samples a batch of random samples from the configuration space, then it uses a procedure similar to FMT* to incorporate the new batch of samples into the pre-existing search tree. The performance of BIT* varies with the size of each batch of samples. When the batch size of BIT* was just one, BIT* is nearly equivalent to RRT*. When the batch size is very large, BIT* is nearly equivalent to FMT* except for being able to sample subsequent batches of samples after the first is used. This allows the user of the algorithm to tune BIT* to have the performance characteristics desired for a particular application.

The rest of this work proceeds as follows. Section II describes general RRT* and BIT* notation used throughout this work. Section III gives a thorough step-by-step explanation of the BIT* algorithm. Section IV provides a demonstration of how BIT* operates, graphically, over three batches of samples. Section V provides averaged converges results using the Open Motion Planning Library (OMPL) and making comparisons to RRT*.

II. NOTATION AND BACKGROUND

Section II-A describes the notation used throughout this work. Section II-B defines some general helper procedures, to be used in the description of BIT*.

A. Nomenclature

RRT* and BIT*-based algorithms iteratively construct a rooted, out-branching tree to find a path through the state space. The tree is an acyclic directed graph denoted as $T \triangleq (V, E)$, where V is the set of nodes or vertices within the tree and $E \subset V \times V$ denotes the set of edges between vertices. The root vertex has no parent while all other vertices have exactly one parent. Each vertex can have multiple children. Each vertex within V corresponds to a state in the d-dimensional state space denoted as $X \subset \mathbb{R}^d$. The tree is initialized with solely the root node, i.e. $V = \{x_r\}$, $E = \emptyset$. Nodes are added to the tree to find a path to the target set, $X_t \subset X$. The path

must avoid the space blocked by obstacles, $X_{obs} \subset X$, staying within the obstacle-free space, $X_{free} \triangleq X \setminus X_{obs}$.

 \mathbb{R} is the set of all real numbers, and \mathbb{R}_+ is the set of all real positive numbers. The notation $X \xleftarrow{+} \{x\}$ and $X \xleftarrow{-} \{x\}$ is used to compactly represent the set updating operations $X \leftarrow X \setminus \{x\}$ and $X \leftarrow X \cup \{x\}$ respectively.

B. Common Procedures

In this section, a number of primitive sub-procedures are defined for use while describing BIT*. We now define several generic procedures that can be found in [10], [11] with notation updated to match the sequel.

Definition 1 (cost \leftarrow c(x, y)): Calculates the length of the path in X that connects $x \in X$ to $y \in X$. Note that c returns infinity if the path is blocked by an obstacle.

Definition 2 $(cost \leftarrow \hat{c}(x,y))$: Calculates a lower bounded heuristic for the cost of the path that goes from $x \in X$ to $y \in X$, i.e., $0 \le \hat{c}(x,y) \le c(x,y) \le \infty$. Note that \hat{c} typically will not consider obstacles or have to generate the edge explicitly which makes it a fast operation. In the work \hat{c} is defined using Euclidean distance as $\hat{c}(x,y) := \|x-y\|$.

Definition 3 (cost $\leftarrow g_T(x)$): Calculates the cost-to-come from the root node to $x \in X$ through the tree, T. Note that if x is not in the tree then $g_T(x) = \infty$.

Definition 4 ($cost \leftarrow \hat{g}(x)$): Calculates a lower bounded heuristic for the cost-to-come of $x \in X$, i.e., $0 \le \hat{g}(x) \le g_T(x)$. In this work \hat{g} is defined using the edge cost heuristic as $\hat{g}(x) := \hat{c}(x_r, x)$, where x_r is the initial state of the path planning problem.

Definition 5 ($cost \leftarrow \hat{h}(x)$): Calculates a lower bounded heuristic for the cost-to-go of $x \in X$. In this work \hat{h} is again defined using edge cost heuristic as $\hat{h}(x) := \hat{c}(x, x_t)$, where $x_t \in V$ is the end of the best solution that has been found by the planner so far.

Definition 6 $(X_{rand} \leftarrow Sample(m))$: Generates a set of $m \in \mathbb{R}_+$ random samples of the obstacle-free state set, X_{free}^{-1} . The sampling is an independent, identically, and uniformly distributed (i.i.u.d.) sample of X_{free}^{-2} .

Definition 7 $(X_{near} \leftarrow Near_{\rho}(x, X_{search}))$: Finds all vertices in X_{search} that are within a given edge cost radius³, $\rho \in \mathbb{R}_+$, of the point $x \in X$, i.e.

$$Near_{\rho}(x, X_{search}) := \{v \in X_{search} | c(x, v) \le \rho\}.$$

Definition 8 $(X_{sol} \leftarrow Solution(x))$: Finds the path through $T \triangleq (V, E), \ X_{sol} \subset V$, that leads from the root node to $x \in V$.

Definition 9 $(x_p \leftarrow Par(x_c))$: Returns the parent node of $x_c \in V$ in the tree $T \triangleq (V, E)$, or \emptyset if x_c is the root node.

Definition 10 $(X_{children} \leftarrow Children(x_p))$: Returns every node from the set V in $T \triangleq (V, E)$ that has x_p as its parent.

Algorithm 1: BIT*

```
Input: x_r, X_{goal}, X_t
    Output: X_{sol}
 1 V \leftarrow \{x_r\}; E \leftarrow \varnothing; T \triangleq (V, E);
 2 Q_V \leftarrow V; Q_E \leftarrow \varnothing; Q \triangleq (Q_V, Q_E);
3 X_{ncon} \leftarrow X_{goal}; X_{new} \leftarrow X_{ncon};

4 V_{exp} \leftarrow \varnothing; V_{rewire} \leftarrow \varnothing; V_{sol} \leftarrow V \cap X_t;

5 if V_{sol} = \varnothing then c_{sol} \leftarrow \infty else c_{sol} \leftarrow \min_{v \in V_{sol}} g_T(v);
 6 X_{flags} \triangleq (X_{new}, V_{exp}, V_{rewire}, V_{sol}, c_{sol});
   repeat
       if Q_V = \varnothing \wedge Q_E = \varnothing then // End of batch
           // Prune sub-optimal nodes
           \{X_{reuse}, T, X_{ncon}, X_{flags}\} \leftarrow
             Prune(T, X_{ncon}, X_{flags});
           // Generate new batch of samples
           X_{new} \leftarrow Sample(m);
10
           // Add new samples to queues
           X_{ncon} \xleftarrow{+} X_{new} \cup X_{reuse};
11
           Q_V \leftarrow V;
12
       else if BestValue(Q_V) \leq BestValue(Q_E) then
13
             / Process best vertex available
14
          \{Q, X_{flags}\} \leftarrow ExpVertex(T, Q, X_{ncon}, X_{flags});
15
       else // Process best edge available
           \{T, Q, X_{ncon}, X_{flags}\} \leftarrow
16
             ExpEdge(T, Q, X_{ncon}, X_{flags}, X_t);
17 until STOP;
18 return Solution(\arg\min_{v_t \in V_{sol}} g_T(v_t));
```

Definition 11 (cost \leftarrow BestValue(Q_i)): Finds the element in Q_i with the lowest queue cost and returns the queue cost of that element.

Definition 12 ($x \leftarrow PopBest(Q_i)$): Finds the element in Q_i with the lowest queue cost, removes it from the queue, and returns that element.

III. BATCH INFORMED TREES

BIT* functions by iteratively generating batches of samples from the state space and incorporating those new samples into the pre-existing search tree. To achieve this BIT* defines two sorted queues, the vertex queue and the edge queue. At the beginning of each batch, the vertex queue is populated with all of the nodes that are currently in the search tree. Vertices are then iteratively removed from the vertex queue and all potential edges that start from that vertex are added to the edge queue. Once all potential edges between samples of the state space have been found the search tree is updated to include them if doing so will shorten the length of the resulting path. Once both of the queues are empty a new batch is sampled and the process begins again.

Algorithm 1 shows the BIT* algorithm. The inputs are the root node, x_r , a sampling of the target set, $X_{goal} \subset X_t$, and the target set, $X_t \subset X_{free}$. Line 1 initializes the search tree, T, with the root node, x_r , in its vertex set, V, and no edges in its edge set, E. Line 2 initializes the vertex queue, Q_V , and the edge queue, Q_E . Q_V is used to keep track of vertices that are under consideration for making potential edges and

 $^{^{1}}$ Because it is computationally expensive to uniformly sample an arbitrary set, all of X is sampled instead and the sample is discarded and re-sampled if it happens to be in the obstacle set.

 $^{^{2}}$ It is important that the sampling of X_{free} is i.i.u.d. to guarantee asymptotic optimality [7].

³In this work ρ is held constant, but many sampling-based algorithms vary ρ as the algorithm runs [7].

Algorithm 2: Prune

```
Input: T \triangleq (V, E), X_{ncon}, X_{flags} \triangleq (X_{new}, V_{exp}, V_{rewire}, V_{sol}, c_{sol})
Output: X_{reuse}, T, X_{ncon}, X_{flags}

1 X_{reuse} \leftarrow \varnothing;

2 X_{ncon} \stackrel{-}{\leftarrow} \left\{ x \in X_{ncon} \middle| \hat{g}(x) + \hat{h}(x) \ge c_{sol} \right\};

3 forall v \in V do

4 | if g_T(v) + \hat{h}(v) > c_{sol} then
| // Remove v from the search
| V \stackrel{-}{\leftarrow} \{v\}; E \stackrel{-}{\leftarrow} \{(Par(v), v)\};
| X_t \stackrel{-}{\leftarrow} \{v\}; X_{exp} \stackrel{-}{\leftarrow} \{v\}; X_{rewire} \stackrel{-}{\leftarrow} \{v\};
| // If v can possibility help

7 | if \hat{g}(v) + \hat{h}(v) < c_{sol} then X_{reuse} \stackrel{+}{\leftarrow} \{v\};

8 return \{X_{reuse}, T, X_{ncon}, X_{flags}\};
```

is organized in terms of the current cost-to-come plus the heuristic cost-to-go of the vertices, i.e.,

$$g_T(v) + \hat{h}(v), v \in \mathcal{Q}_V.$$

 \mathcal{Q}_E is used to keep track of edges that are under consideration for addition to T. \mathcal{Q}_E is organized in terms of the sum of the current cost-to-come of the source vertex of the edge, the heuristic cost of the edge, and the heuristic cost-to-go of the target vertex of the edge, i.e.,

$$g_T(v) + \hat{c}(v, x) + \hat{h}(x), (v, x) \in \mathcal{Q}_E.$$

Lines 3 through 6 initialize a few sets that are needed to keep track of the state of each vertex. $X_{ncon} \subset X$ is the set of all samples that are not connected to the search tree. Note that X_{ncon} is initialized with the provided samples of the target set, X_{goal} . $V_{sol} = V \cap X_t$ is the set of vertices in V that are also in the target set X_t . $X_{new} \subset X_{ncon}$ is the set of samples that are from the most recent batch of samples. $V_{nexp} \subset V$ is the set of vertices that have not been considered for expansion. $V_{nrewire} \subset V$ is the set of vertices that have not been considered for rewiring. c_i is the cost of the current best solution.

The loop on lines 7 through 17 performs the rest of the planning process and ends when a user-defined stopping condition is met⁴. The conditionals on lines 8 and 13 determine if the batch has ended and whether to process a vertex from Q_V or an edge from Q_E , respectively. Each case is discussed below.

A. Generating New Batches

When Q_V and Q_E are both empty it signifies the end of the batch, see line 8 of Algorithm 1. When that happens, all vertices that cannot contribute to the optimal solution are removed from the search tree, T, by calling the Prune procedure on line 9.

The *Prune* procedure is given in Algorithm 2. Line 2 of Algorithm 2 removes all unconnected samples with heuristic

Algorithm 3: ExpVertex

```
Input: T \triangleq (V, E), \mathcal{Q} \triangleq (\mathcal{Q}_V, \mathcal{Q}_E), X_{ncon}, X_{flags} \triangleq
                (X_{new}, V_{exp}, V_{rewire}, V_{sol}, c_{sol})
    Output: Q, X_{flags}
 1 v_b \leftarrow PopBest(Q_V);
     // Make edges to unconnected samples
 2 if v_b \not\in V_{exp} then
        V_{exp} \xleftarrow{+} \{v_b\};
        X_{near} \leftarrow Near_{\rho}(v_b, X_{ncon});
 {\sf 5} else // v_b has been expanded before
 6 | X_{near} \leftarrow Near_{\rho}(v_b, X_{new} \cap X_{ncon});
 7 Q_E \leftarrow^+
      \left\{ (v_b, x), x \in X_{near} \middle| \hat{g}(v_b) + \hat{c}(v_b, x) + \hat{h}(x) < c_{sol} \right\};  // If v_b has not been rewired before
 8 if v_b \notin V_{rewire} \land c_{sol} < \infty then
        // Make edges to connected nodes
        V_{rewire} \xleftarrow{+} \{v_b\}; \\ V_{near} \leftarrow Near_{\rho}(v_b, V);
 9
10
        Q_E \stackrel{+}{\leftarrow} \{(v_b, w), w \in V_{near} | (v_b, w) \notin E,
                    \hat{g}(v_b) + \hat{c}(v_b, w) < g_T(w),
11
                    \hat{g}(v_b) + \hat{c}(v_b, w) + \hat{h}(w) < c_{sol}  ;
12 return \{Q, V_{flags}\};
```

cost-to-come plus heuristic cost-to-go value greater than the current best solution. Note that this can be thought of as removing all nodes that fall outside of the "informed set" that Informed RRT* (I-RRT*) defines [2] and as such provably cannot contribute to the optimal solution. The loop on lines 3 though 7 removes any vertices in the search tree that cannot contribute to the optimal solution. Line 4 checks if each vertex in the tree has the potential to contribute to the optimal solution given its current connection to the search tree. If this check fails, the vertex is removed from the search tree. Line 7 checks if the vertex has the potential to contribute to the optimal solution given the ideal cost-to-come. This is effectively the same condition that is used on line 2. If there is a chance of the sample contributing to the optimal solution, the vertex is added to X_{reuse} to be reused as an unconnected sample in the next batch. X_{reuse} is added to the algorithm to maintain uniform sample density in the "informed set".

After Prune is finished, line 10 of Algorithm 1 generates m new samples of the obstacle-free state space, X_{free} . Line 11 adds the new samples and reused vertices to X_{ncon} . Line 12 adds all vertices in the search tree to the vertex queue. This insures all vertices in the search tree will be considered when looking for ways to connect the new samples to the search tree.

B. Expanding Vertices

Lines 13 and 14 of Algorithm 1 find potential edges to add to Q_E from the vertices in Q_V . The condition on line 13 evaluates to true until it is impossible for the best vertex in Q_V to produce an edge of lower heuristic cost then the best

⁴Common stopping conditions include achieving a desirable solution cost or expending the extent of the planning time given.

edge in Q_E . This can be seen by noting that

$$\forall v \in \mathcal{Q}_V, \forall x \in X, g_T(v) + \hat{h}(v) \le g_T(v) + \hat{c}(v, x) + \hat{h}(x)$$

as $\hat{h}(v)$ is an under estimate of the true cost-to-go of vertex v. Thus, the vertex queue cost, $g_T(v) + \hat{h}(v)$, is a lower bound on the edge queue cost, $g_T(v) + \hat{c}(v,x) + \hat{h}(x)$ of any edge that can be made from that vertex.

ExpVertex removes the lowest cost vertex in Q_V and adds edges to Q_E for every neighbor that might be part of the optimal solution. The ExpVertex procedure is given in Algorithm 3. Line 1 of Algorithm 3 pops the lowest cost vertex in Q_V . Lines 2 through 7 adds edges to Q_E that start from v_b and go to samples that are not connected to the tree. The condition on line 2 checks if this is the first time v_b has been considered for expansion. In that case, all unconnected samples are considered for connection to v_b , see line 4. If it is not the first time v_b has been considered for expansion, only the samples that are new this batch are considered for connection, see line 6. This prevents redundant calculations as the samples that are not new this batch have already been considered for connection to v_b . Once X_{near} has been found, line 7 adds all edges in the set $\{(v_b, x), x \in X_{near}\}$ that have potential to improve the current solution to Q_E .

Lines 8 through 11 handle the case when v_b might be a better parent for its neighbors then their current parent, i.e., rewiring. Note that if BIT* has not found a solution the condition on line 8 will always evaluate to false. This is done to reduce the time it takes to find an initial solution to the problem by skipping any potential tree rewirings. Once the first solution is found, all rewirings that were skipped previously are considered. The condition on line 8 also evaluates to false if this is not the first time v_b has been considered for rewiring. This prevents redundant calculations as the potential to perform rewirings around v_b has already been considered. Line 10 finds all vertices in the search tree that are near v_b . Line 11 adds all edges from v_b to V_{near} that are not already part of the tree, have the potential to improve the cost of the neighbor, and have the potential to improve the current solution.

Note that everything done in *ExpVertex* is done completely with heuristic values and without obstacle checking. This keeps the procedure computationally lightweight and fast.

C. Evaluating Possible Edges

In the case that the best heuristic cost edge possible has been generated from Q_V , the condition on line 13 of Algorithm 1 evaluates to false and ExpEdge is called. ExpEdge removes the most promising edge from Q_E and considers it for addition to the search tree.

The ExpEdge procedure is given in Algorithm 4. Line 1 removes the lowest queue cost edge from Q_E . Line 2 checks if there is a chance that the edge under consideration will improve the current solution. Note that this condition is true only if the most promising edge in Q_E , and by extension all

Algorithm 4: ExpEdge

```
Input: T \triangleq (V, E), \mathcal{Q} \triangleq (\mathcal{Q}_V, \mathcal{Q}_E), X_{ncon}, X_{flags} \triangleq
              (X_{new}, V_{exp}, V_{rewire}, V_{sol}, c_{sol}), X_t
    Output: T, Q, X_{ncon}, X_{flags}
 1 (v_b, x_b) \leftarrow PopBest(\mathcal{Q}_E);
 2 if g_T(v_b) + \hat{c}(v_b, x_b) + h(x_b) \ge c_{sol} then
        // The best edge in the edge queue
             cannot improve the solution
       Q_E \leftarrow \varnothing; Q_V \leftarrow \varnothing;
 3
 4
       return \{T, Q, X_{ncon}, X_{flags}\};
 5 if x_b \in X_{ncon} then // x_b is unconnected
       if g_T(v_b) + c(v_b, x_b) + \hat{h}(x_b) < c_{sol} then
           // Add x_b to the tree
          X_{ncon} \leftarrow \{x_b\}; V \leftarrow \{x_b\}; E \leftarrow \{(v_b, x_b)\};
 7
           \mathcal{Q}_V \xleftarrow{+} \{x_b\};
          if x_b \in X_t then
10
           V_{sol} \xleftarrow{\top} \{x_b\}; c_{sol} \leftarrow \min_{v_{sol} \in V_{sol}} g_T(v_{sol});
11 else // x_b is connected
      if g_T(v_b) + \hat{c}(v_b, x_b) < g_T(x_b) then
12
          if g_T(v_b) + c(v_b, x_b) + \hat{h}(x_b) < c_{sol} then
13
             if g_T(v_b) + c(v_b, x_b) < g_T(x_b) then
14
                 E \leftarrow \{(Par(x_b), x_b)\}; E \leftarrow \{(v_b, x_b)\};
15
                 c_{sol} \leftarrow \min_{v_{sol} \in V_{sol}} g_T(v_{sol});
16
17 return \{T, Q, X_{ncon}, X_{flags}\};
```

of the edges in Q_E , cannot contribute to the optimal solution. For this reason Q_E and Q_V are cleared on line 3.

Line 5 checks if x_b is already in the tree. If x_b is not part of the search tree, line 6 checks if connecting x_b to the tree through v_b can improve the current solution. If it can, x_b is added to the search tree with v_b as its parent. Lines 9 and 10 check if x_b is in the target set and adds x_b to V_{sol} if so. If x_b is already part of the search tree at line 5, extra checks are performed before adding the edge under consideration to the tree. Line 12 checks if connecting x_b through v_b can improve the cost of x_b . Line 13 checks that the edge under consideration can improve the current solution. Line 14 checks that connecting x_b through v_b will improve the cost of x_b . If all checks pass, x_b is rewired to have v_b as its parent and the current solution cost is updated if needed.

IV. DEMONSTRATION

Figures 1 through 4 graphically show how BIT* operates over three batches of samples. In this example, each batch consists of five samples, i.e., m=5.

Figure 1 shows what we call batch 0. This is the part of the planning process where the only node in Q_V is the root node and no samples have been made from the state space. This part of the algorithm checks for the trivial case where it is possible to directly connect the root node to the target set.

Figure 2 shows the process of generating the first batch of samples and starting to build a search tree. Note that when batch 1 completes a path to the target set has not been found. This is a common occurrence in BIT* where the search tree is unable to grow much until the sample density in X_{free} grows for a few batches.

Figure 3 shows how the second batch of samples is incorporated into the search tree. By the end of the batch a path to the target set has been found and the "informed ellipse" is defined. This enables pruning to be performed before batch 3 begins.

Figure 4 shows how a new batch of samples is generated within the informed ellipse and used to refine the current solution. Note that as the solution length reduces in the third batch, the size of the informed ellipse also shrinks. This leads to more nodes being pruned and the search process becoming more focused on the area that can improve the current solution.

V. SIMULATION

Simulation results are now presented to demonstrate the effectiveness of the BIT* algorithm. Comparisons are made between BIT* and RRT* planning with straight-lines.

A. Simulation Details

To test the capabilities of BIT*, it is used to plan paths for a simulated UAV through an urban environment. The UAV simulation is shown in Figure 5. The buildings in the UAV simulation are modeled off of the real buildings in Manhattan, New York. The placement and height of the buildings are from New York's Open Data project [12]. The initial position of the UAV is located in Central park. Maintaining an altitude of 10 meters, the UAV plans a path through the buildings to the goal location on Governors Island.

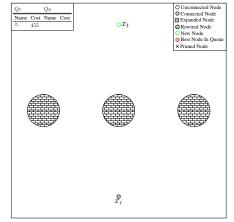
While planning a path, obstacles are represented with an occupancy grid with each pixel corresponding to ten square centimeters. Every building that is taller than 10m is considered an obstacle in the occupancy grid. Figure 6 shows the resulting occupancy grid with black representing obstacles. The occupancy grid covers a $10.7km \times 5.65km$ area of Manhattan. The initial location of the UAV in the coordinate frame used for this problem is $x_r = \begin{bmatrix} 0 & 0 \end{bmatrix} km$, with the UAV orientation defined in the direction of the target set. The target set, X_t , is a circle of radius 0.001m centered at |-9|-3.8| km. The samples of the target set that are provided to BIT*, X_{goal} , is the singleton set of the center of X_t . The neighborhood search radius, ρ , is 500m. Paths are generated and checked for obstacles four times per every meter of path length. When using BIT*, the batch size, m, is set to 1500 samples.

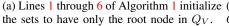
When using RRT* there are three additional parameters, η , α , and b_t , that are described in [11] but only given values here for brevity. The steering constant, η , is 500m. The max number of neighbors to search, α , is 100 neighbors. The check target period, b_t , is 1 out of every 50 samples.

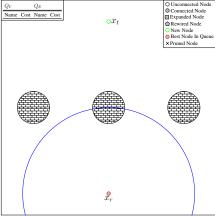
Results are gathered using the Open Motion Planning Library (OMPL) [13]. As the sampling is random, each simulation consists of over 100 individual simulations with the average results being presented. The results were gathered on an AMD RyzenTM ThreadripperTM 3970X processor. Convergence plots were made by fitting a 15th-order polynomial using a least-squares fitting algorithm as described in [14].

A least-squares approach is used as the sampling times for path length are not uniform across all simulations and not all simulations find the initial path at the same time. Note that while the path length for any one run will be monotonically decreasing with time, the least squares fitted plot does not always have the same monotonic property. The reason is that a particular run may not find a solution until well after other runs and the initial solution it finds may be much larger than the current solution of the other runs, effectively causing the average to increase at the time the run first produces path length data.

Simulation code can be found in our open-source repository https://gitlab.com/utahstate/robotics/fillet-rrt-star.

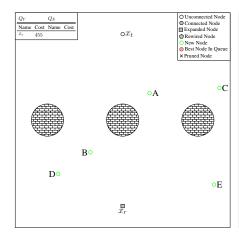


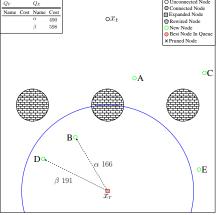


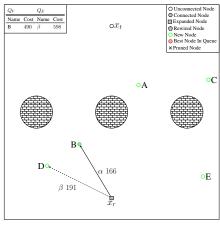


(a) Lines 1 through 6 of Algorithm 1 initialize (b) x_r is the vertex with the lowest cost in Q_V so it is removed from the queue and considered for expansion. x_r does not have any neighbors so no edges are added to Q_E .

Fig. 1: Batch 0 of BIT*.

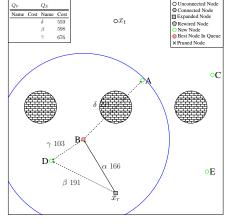


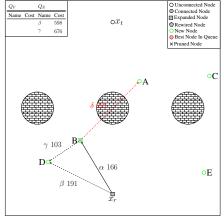


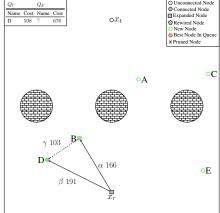


(a) Q_E and Q_V are empty so a new batch (b) The lowest cost vertex, x_r , is re-(c) The vertex queue is empty so the of samples is made. Pruning is performed moved from Q_V and used for expansion ExpEdge procedure starts. α is the lowest but has no effect until a solution is found. in ExpVertex. Edges α and β pass the cost edge in Q_E so it is removed from Q_E Samples A through E are generated. Q_V is heuristic cost tests and are added to Q_E . filled with all connected nodes.

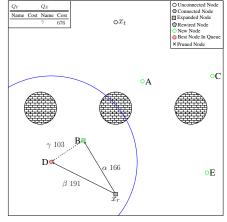
and, after passing all validity and cost tests, added to the tree. B is now connected to the tree and as such added to Q_V .

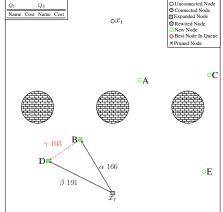






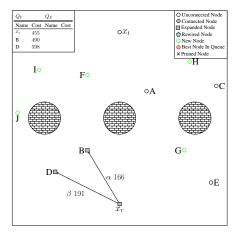
(d) The cost of B is less then the cost of (e) Q_V is empty so we move to ExpEdge. (f) Edge β is removed from Q_E and, after β so the ExpVertex procedure is called. Edge δ is removed from Q_E . δ passes the passing all tests, added to the tree. Node D is B is removed from Q_V and edges γ and δ heuristic cost tests but fails the true cost tests now connected to the tree and added to Q_V . are added to Q_E . Note that an edge is not because of the obstacle it passes through. δ made from B to x_r because that edge fails is not added to the tree. the heuristic cost tests.

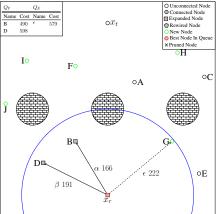


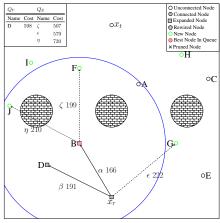


(g) The cost of node D is less then γ so (h) With Q_V empty we move back ExpVertex begins. Node D is removed from ExpEdge. Edge γ is removed from Q_E but Q_V and used for expansion. However, a fails the heuristic cost tests so is not added solution has not been found so connected to the tree. neighbors are not considered and D has no unconnected neighbors.

Fig. 2: Batch 1 of BIT*.

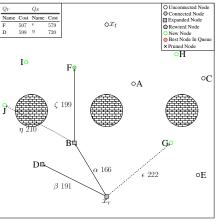


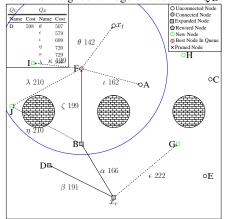


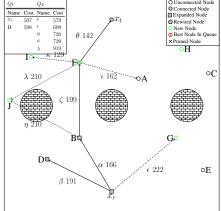


(a) Q_E and Q_V are empty so a new batch of (b) In ExpVertex, x_r is removed from Q_V . (c) Node B is removed from Q_V . Node B nodes.

samples is made. Samples F through J are Because x_r is still part of the expanded set has already been expanded so only edges to generated. Q_V is filled with all connected from last batch, we only search over the new new nodes are considered. Edges η and ζ are nodes for neighbors. Edge ϵ is added to Q_E added to Q_E .

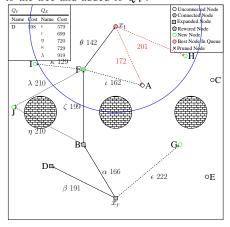


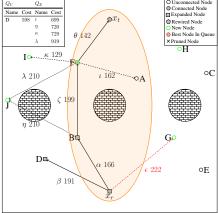


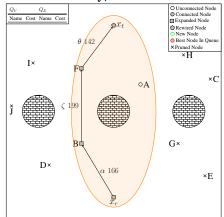


(d) Edge ζ has a lower heuristic cost then (e) In ExpVertex, node F is removed from (f) Edge θ has a lower heuristic cost then node D so ExpEdge starts. Edge ζ is re- Q_V . Node F has not been expanded so edges node D so ExpEdge is called. Edge θ is moved from Q_E and, after passing all tests, to all unconnected nodes are considered removed from Q_E and, after passing all tests, added to the tree. Node F is now connected Edges λ , κ , θ , and ι are added to Q_E . to the tree and added to Q_V .

added to the tree. Node x_t is now part of the tree and added to Q_V .

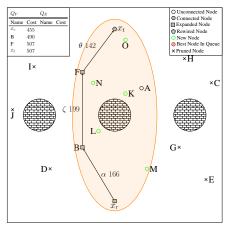


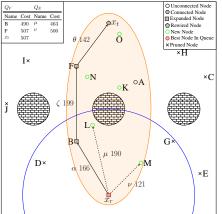


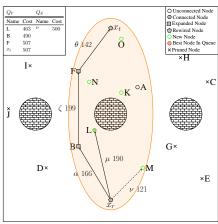


(g) Node x_t has a lower heuristic cost then (h) In ExpEdge, edge ϵ is removed from (i) Before generating batch 3, all nodes in the edge ϵ so ExpVertex is used. Node x_t is Q_E and considered for addition to the tree, tree or unconnected are checked to make sure removed from Q_V . Node x_t has not been Edge ϵ fails the primary heuristic cost check, they fall within the informed ellipse. All that expanded so edges to all near nodes are meaning that ϵ has no chance of improving fall outside of the ellipse are pruned and not considered. Edges from x_t to nodes A and H the solution. The rest of the nodes and edges considered moving forward. fail to pass the heuristic cost test and as such in Q_V and Q_E also can not help the solution can not improve the current solution, and are so both queues are cleared. Note the orange not added to Q_E .

ellipse that shows the informed set of states, and how G falls outside of the ellipse. This is a visual way of seeing why node G was not added to the tree.



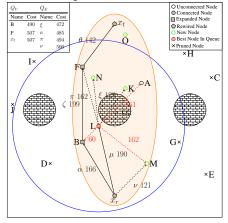


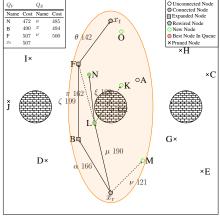


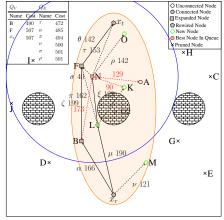
to-come through the tree.

(a) Nodes K through O are sampled from (b) In ExpVertex, node x_r is removed from (c) Edge μ has a lower heuristic cost then the informed ellipse. All connected nodes are Q_V . Node x_r has been expanded but not node B so we move to ExpEdge. Edge μ is added to Q_V . Note that node F is before x_t rewired so only edges to new and connected removed from Q_E and, after passing all tests because ties are broken based on true cost-nodes are considered. Edges μ and ν are added to the tree. Node L is now part of the added to Q_E .

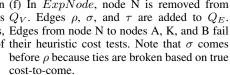
tree and added to Q_V .

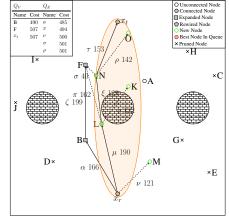


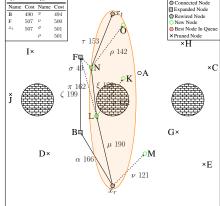


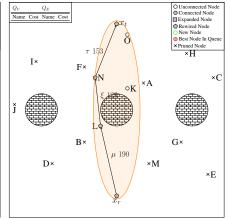


(d) In ExpNode, node L is removed from (e) Edge ξ has a lower heuristic cost then (f) In ExpNode, node N is removed from Q_V . The is the first time Node L is used for node B so we move to ExpEdge. Edge ξ is Q_V . Edges ρ , σ , and τ are added to Q_E . expansion so all non-pruned nodes are under removed from Q_E and, after passing all tests, Edges from node N to nodes A, K, and B fail consideration. Edges ξ , ϕ , and π are added to added to the tree. Node N is now a part of their heuristic cost tests. Note that σ comes Q_E . Edges from L to M, B, and A fail the tree and added to Q_V . heuristic cost test and are not added to Q_E .









(g) In ExpEdge, edge τ is removed from (h) Edge \emptyset is removed from Q_E and consid- (i) Before generating batch 4, all nodes ei- Q_E and, after passing all tests, added to the ered for addition to the tree. Edge \emptyset fails the ther in the tree or unconnected are checked tree. In order for node x_t to only have one primary heuristic cost tests and is not added to make sure they fall within the informed parent, edge θ is removed from the tree. Note to the tree. Q_V and Q_E cleared and batch 2 ellipse. All that fall outside of the ellipse are that this changes the current solution cost and ends. the informed ellipse.

pruned and not considered moving forward.

Fig. 4: Batch 3 of BIT*.

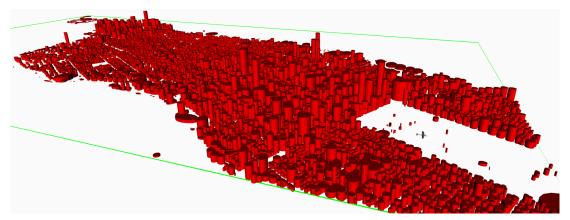


Fig. 5: The UAV simulation with Manhattan's buildings shown in red.



Fig. 6: The resulting paths from running BIT* in the Manhattan world. The Path from BIT* is shown in red.

B. Results

The convergence results from benchmarking RRT* and BIT* in the Manhattan environment are shown in Figure 7. RRT* and BIT* are shown in blue and red respectively. Clearly BIT* outperforms RRT* in terms of converging to a near optimal-value rapidly. BIT* initially finds a solution much shorter than RRT* and proceeds to converge to near optimality by the time 30 seconds have passed. This comes from BIT*'s use of cost-to-come and cost-to-go heuristics to focus their search efforts in directions that are most likely to improve the solution cost. RRT* on the other hand starts with a long initial solution. Despite converging for five minutes, the solution that RRT* produces is still substantially longer than that of BIT*.

REFERENCES

- [1] S. M. LaValle, Planning Algorithms. Cambridge, U.K.: Cambridge University Press, 2006, available at http://planning.cs.uiuc.edu/.
- J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2014, pp. 2997-3004.
- [3] A. H. Qureshi and Y. Ayaz, "Intelligent bidirectional rapidly-exploring random trees for optimal motion planning in complex cluttered environments," *Robotics and Autonomous Systems*, vol. 68, pp. 1– 11, 2015. [Online]. Available: https://www.sciencedirect.com/science/ article/pii/S0921889015000317
- [4] J. Nasir, F. Islam, U. Malik, Y. Ayaz, O. Hasan, M. Khan, and M. Muhammad, "Rrt*-smart: A rapid convergence implementation of rrt*," International Journal of Advanced Robotic Systems, vol. 10, 2013.
- I. Noreen, A. Khan, and Z. Habib, "Optimal path planning using rrt* based approaches: A survey and future directions," International Journal of Advanced Computer Science and Applications, vol. 7, no. 11, 2016. [Online]. Available: http://dx.doi.org/10.14569/IJACSA.2016.071114

Convergence Trends

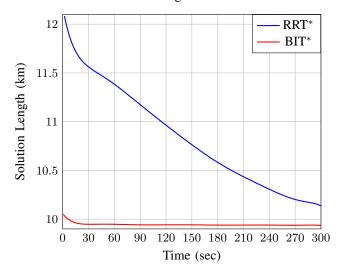


Fig. 7: The convergence plots of the Manhattan world over 5 minutes. RRT* and BIT* planning with straight-line motion primitives are shown in blue and red respectively.

- [6] K. Yang, S. Moon, S. Yoo, J. Kang, N. L. Doh, H. B. Kim, and S. Joo, "Spline-based rrt path planner for non-holonomic robots," *Journal of Intelligent & Robotic Systems*, vol. 73, no. 1-4, pp. 763–782, 2014a.
- [7] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011. [Online]. Available: https://doi.org/10.1177/0278364911406761
- [8] I.-B. Jeong, S.-J. Lee, and J.-H. Kim, "Quick-rrt*: Triangular inequality-based implementation of rrt* with improved initial solution and convergence rate," *Expert Systems with Applications*, vol. 123, pp. 82–90, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417419300326
- [9] L. Janson, E. Schmerling, A. Clark, and M. Pavone, "Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions," *The International journal of robotics research*, vol. 34, no. 7, pp. 883–921, 2015.
- [10] J. D. Gammell, T. D. Barfoot, and S. S. Srinivasa, "Batch informed trees (bit*): Informed asymptotically optimal anytime search," *The International Journal of Robotics Research*, vol. 39, no. 5, pp. 543–567, 2020. [Online]. Available: https://doi.org/10.1177/0278364919890396
- [11] J. Swedeen, G. Droge, and R. Christensen, "Fillet-based rrt*: A rapid convergence implementation of rrt* for curvature constrained vehicles," 2023. [Online]. Available: https://arxiv.org/abs/2302.11648
- [12] "Building footprints," May 2016. [Online]. Available: https://data.cityofnewyork.us/Housing-Development/Building-Footprints/nqwf-w8eh
- [13] M. Moll, I. A. Sucan, and L. E. Kavraki, "Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization," *IEEE Robotics Automation Magazine*, vol. 22, no. 3, pp. 96–102, 2015.
- [14] W. N. Venables and B. D. Ripley, Modern Applied Statistics with S, 4th ed. New York: Springer, 2002, iSBN 0-387-95457-0. [Online]. Available: https://www.stats.ox.ac.uk/pub/MASS4/