Chapter 5

Batch Informed Trees (BIT*)

or: The benefits of trying good solutions first

This chapter demonstrates how informed graph-based search can be used to address the limitations of anytime sampling-based planning. Unifying these approaches results in algorithms that build anytime approximations of continuous planning problems and search this representations in order of potential solution quality. This avoids unnecessary computational costs throughout the search and results in finding better solutions faster than existing algorithms, especially in high state dimensions.

Single-query anytime sampling-based planners, such as RRT*, often approximate continuous planning problems incrementally. This allows them to stop once a suitable solution is found but makes their search dependent on the underlying sequence of samples. Such an *unordered* search simultaneously expands towards every state in the problem domain and performs work that is not needed to find the eventual solution. This wasted computational effort can become prohibitively expensive in large problem domains and high state dimensions. The effort can be reduced by focusing the search once a solution is found (e.g., Informed RRT*; Chapter 4) but the search itself will still be inefficient.

Informed graph-based searches, such as A*, avoid unnecessary computational effort by ordering their entire search by potential solution quality. This assures that solutions are found by only considering states that could have provided a better solution and effectively solves a problem by searching the informed set that the solution *will* define (the *future informed set*). These approaches have previously been applied to *a priori* graph-based discretizations.

Section 5.1 presents a review of previous work to incorporate ordered search concepts into sampling-based planning. The review is roughly divided into efforts to add sampling to informed graph-based search (e.g., to provide anytime representations for A*) and to add heuristics to anytime sampling-based planners (e.g., to order the search of RRT*). This work either applies heuristics partially (e.g., RRT#), sacrifices anytime resolution (e.g., RA* and FMT*), or unnecessarily searches outside the future informed set due to metrics other than solution cost (e.g., SBA*).

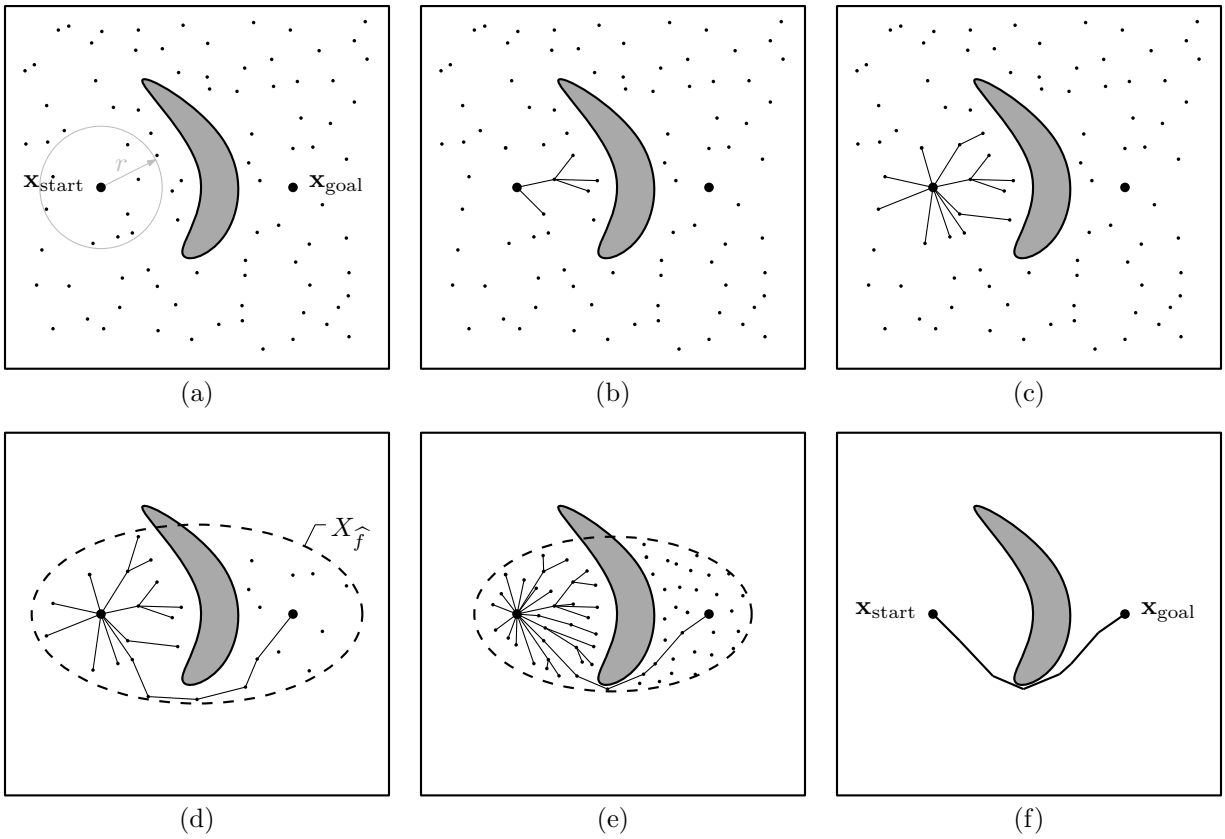


Figure 5.1: An illustration of the BIT* algorithm. The continuous planning problem is approximated with a sample-based tree rooted at the start. The tree is grown outwards using a radius of connection defined by RGG theory, (a). The search proceeds in order of increasing solution cost as estimated by an *admissible* heuristic, (b) and (c), as in A* (see Fig. 2.4) until no edges in can provide a (better) solution, (d). A batch of new samples is then added to the informed set and the search continues efficiently with incremental search techniques, (e). The ordered search assures that BIT* finds solutions without searching outside the future informed set of a batch. With an infinite number of samples, BIT* has a unity probability of both finding a solution and converging asymptotically to the optimum, if one exists, (f).

Section 5.2 presents Batch Informed Trees (BIT*) as an example of a single-query sampling-based planner that is both anytime and ordered only on potential solution quality (Fig. 5.1). It unifies informed graph-based search and sampling-based planning to efficiently search an increasingly dense implicit random geometric graph (RGG) defined by *batches* of samples with incremental search techniques (Fig. 2.2). Processing batches of multiple samples allows it to search the current approximation in order of potential solution quality, as in A*. Processing multiple batches allows it to increase the approximation accuracy of the continuous planning problem until it contains a suitable solution, as in RRT*. A version of BIT* is publicly released in OMPL.

BIT* uses heuristics in all aspects of its search to improve its efficiency. It uses informed sets to focus its approximation of the continuous planning problem to the set of states that could belong to a better solution, as in Informed RRT*. It uses RGG theory to reduce

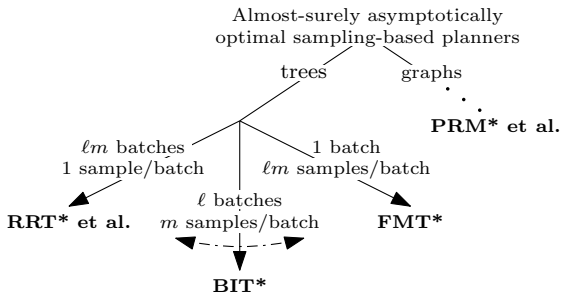


Figure 5.2: A simplified taxonomy of almost-surely asymptotically optimal sampling-based planners that demonstrates the relationship between RRT*, FMT*, and BIT*. When using a batch size of a single sample, BIT* is a version of RRT*. When using a single batch consisting of multiple-samples, BIT* is a version of FMT*.

the number of edges in its approximation while still maintaining almost-sure asymptotic convergence to the optimum, as in RRT* or FMT*. It uses incremental search to order the search of a changing approximation by potential solution quality and reuse information, as in LPA*. It uses an implicit representation of the RGG to avoid the computational cost of explicitly representing every edge, as in Bohlin (2001) and Quasi-random Lazy PRM (Branicky et al., 2001). It uses lazy collision checking to avoid considering unnecessary edges, as in Lazy Weighted A* (Cohen et al., 2014b), lazy versions of PRM (Bohlin and Kavraki, 2000; Branicky et al., 2001; Hauser, 2015), and other techniques (Helmert, 2006; Sánchez and Latombe, 2002).

BIT* only requires three user-defined options, the RGG constant, the heuristic function, and the number of samples per batch and can be be viewed as a generalization of existing sampling-based planners. With no heuristic and multiple batches of one sample, it is a version of RRT*. With no heuristic and a single batch of m samples, it is a version of FMT* (Fig. 5.2).

Section 5.3 presents a theoretical analysis of BIT* that proves it to be probabilistically complete and almost-surely asymptotically optimal. It also shows its search order to be analytically equivalent to a version of LPA* that does not propagate rewirings.

Section 5.4 presents simple extensions of BIT* that may further improve performance in some planning situations. Extensions include delaying the rewiring of connected vertices until an initial solution is found, generating samples when necessary *just in time* (JIT), and removing unconnected samples while maintaining almost-sure asymptotic optimality. Extending the idea of removing unconnected samples also results in Sorted RRT* (SORRT*), a variation of Informed RRT* that is also publicly available in OMPL.

Section 5.5 demonstrates the benefits of BIT* on both abstract problems and experiments for the CMU Personal Robotic Lab's HERB, a 14-DOF mobile manipulation platform. The results show that BIT* outperforms other almost-surely asymptotically optimal planners while maintaining anytime performance, especially in high dimensions. The only tested planner that consistently found solutions faster than BIT* was RRT-Connect, a bidirectional algorithm that does not converge towards the optimum.

Section 5.6 reviews the benefits of unifying informed graph-based search and sampling-based planning and discusses potential future work.

In summary, this chapter makes the following novel contributions:

- Reviews existing methods to combine informed search and sampling-based representations and shows that they either provide incomplete/inefficient ordering or sacrifice anytime performance (Section 5.1).
- Develops BIT* as a unification of informed graph-based search and sampling-based planning that is almost-surely asymptotically optimal (Algs. 5.1–5.3).
- Presents extensions to BIT* (Section 5.4), including JIT sampling to allow for the direct search of unbounded problems (Alg. 5.5) and SORRT* (Alg. 5.6) to apply ordered search concepts directly to RRT*.
- Demonstrates experimentally the benefits of using ordered search to combat the curse of dimensionality (Section 5.5).

5.1 Prior Work Ordering Sampling-based Planners

Prior work to combine graph-based search and sampling-based planning can be loosely classified as either incorporating sampling into informed A*-style searches (Section 5.1.1) or adding heuristics to incremental RRT/RRT*-style searches (Section 5.1.2).

5.1.1 A*-based Approaches

A* is the optimally efficient search for a given graph and any other algorithm using the same heuristic to find the resolution-optimal solution will expand at least as many vertices (Hart et al., 1968). This is achieved by ordering the search by potential solution quality such that the optimal solution is found by only considering states that could provide a better solution (i.e., by only searching the future informed set). Applying A* to continuous planning problems requires discretizing the search domain and significant work has incorporated sampling into this process, often to avoid the need to do so *a priori*.

Sallaberger and D'Eleuterio (1995) demonstrate the advantages of including stochasticity in the representation of a continuous state space by adding random perturbations to a regular discretization. They solve planning problems for spacecraft and multilink robotic arms using dynamic programming and show that their technique finds better solutions than regular discretizations alone; however, the approximation is still defined *a priori* to the search.

Randomized A* (RA*; Diankov and Kuffner Jr., 2007) uses random sampling to apply A* directly to continuous planning problems. Vertices are expanded by randomly sampling a user-specified number of possible descendants in a local neighbourhood. If a sample is sufficiently different than the existing states in the tree and can be reached without collision it is added

as child of the expanded vertex. The resulting sampling-based search expands outwards from the start in order of potential solution quality but is not anytime. If the search does not find a (suitable) solution then it must be restarted with a different number of samples per vertex.

Hybrid Randomized A* (HRA*; Teniente and Andrade-Cetto, 2013) performs a similar search for systems with differential constraints by sampling control inputs instead of states. Vertices are expanded by generating a user-specified number of random control inputs and propagating them forward in time with a motion model. These states are used to expand the tree and, when they are sufficiently similar to an existing state, rewire the tree to improve the cost of existing vertices. The search expands vertices with a hybrid cost policy that considers path cost, the number of nearby vertices, and the distance to obstacles. This biases sampling into unexplored regions of the problem domain but may extend the search outside the future informed set and waste computational effort on states that are unnecessary to find the eventual solution.

Sampling-based A* (SBA*; Persson and Sharf, 2014) performs an ordered search by *iteratively* expanding vertices in a heuristically weighted version of EST (Hsu et al., 1997, 1999a). At each iteration a *single* random sample is generated in the local neighbourhood of the vertex at the front of a priority queue. This queue is ordered on the potential solution quality of vertices and the likelihood of sampling a *unique* and *useful* sample in their neighbourhoods. These likelihoods bias sampling into unexplored space and, as vertices are never removed from the queue, are required to avoid over expansion of vertices. They are estimated from previous collision checks and estimates of the sample density around vertices. Ordering the search on metrics other than the optimization objective can expand the search outside the future informed set and waste computational effort on states that are unnecessary to find the eventual solution.

Unlike these other A*-based methods, BIT* maintains anytime performance while ordering its search only on potential solution quality. This allows it to be run indefinitely until a suitable solution is found, return incrementally better solutions, and almost-surely asymptotically converge to the optimum. This also limits the search to the future informed set and avoids considering states that are unnecessary to find the eventual solution of each representation.

5.1.2 RRT-based Approaches

RRT and RRT* search continuous planning problems by using incremental sampling to build a tree through obstacle-free space. This avoids *a priori* discretizations of the problem domain and allows them to be run indefinitely until a (suitable) solution is found. This also makes the search dependent on the sequence of samples (i.e., makes it *unordered*) and significant work has sought ways to use heuristics to order the search by potential solution quality. Heuristics can also be used to focus the search, as in Anytime RRTs (Ferguson and Stentz, 2006) and Informed RRT*, but this does not change the order of the search.

Heuristically Guided RRT (hRRT; Urmson and Simmons, 2003) probabilistically includes heuristics in the RRT expansion step. Randomly sampled states are probabilistically added in proportion to their heuristic value relative to the tree. This balances the Voronoi bias of the RRT search with a preference towards expanding potentially high-quality paths. This improves performance, especially in problems with continuous cost functions (e.g., path length), but maintains a nonzero probability of searching outside the future informed set and wasting computational effort.

Kiesel et al. (2012) use a two-stage process to order sampling for RRT* in their *f-biasing* technique. First, a heuristic is calculated by solving a coarse discretization of the planning problem with Dijkstra's algorithm. This heuristic is then used to bias RRT* sampling to the regions of the problem domain where solutions were found. This increases the likelihood of finding solutions quickly but maintains a nonzero sampling probability over the entire problem domain for the entire search and allows search effort to be wasted outside the future informed set.

RRT$^{\#}$ (Arslan and Tsiotras, 2013, 2015) uses heuristics to order rewirings in RRT*. It uses LPA* techniques to propagate changes efficiently through the entire tree and assure that each vertex is optimally connected given the current samples. This allows it to avoid computational costs that cannot improve the solution once one is found but does not order the underlying RRT* search which may waste computational effort by searching outside the future informed set.

Fast Marching Tree (FMT*; Janson and Pavone, 2013; Janson et al., 2015b) uses a marching method to search a batch of samples in order of increasing cost-to-come, similar to Dijkstra's algorithm. This makes its search independent of the sampling order and decouples the approximation of the continuous planning problem from the search of the resulting representation but sacrifices anytime performance. Solutions are not returned until the search is finished and the search must be restarted if a (suitable) solution is not found, as with any other *a priori* discretization.

The Motion Planning Using Lower Bounds (MPLB) algorithm (Salzman and Halperin, 2015) extends FMT* with a heuristic and quasi-anytime resolution. Quasi-anytime performance is achieved by *independently* searching increasingly large batches of samples and returning the improved solutions when each individual search finishes. It is stated that this can be done efficiently by reusing information but no specific methods are presented.

Unlike these other RRT*-based methods, BIT* orders its search only on potential solution quality and still maintains anytime performance. This only searches inside the future informed sets of representations and reduces wasted computational effort. This also returns suboptimal solutions in an anytime manner while almost-surely converging asymptotically to the optimum.

5.2 Ordering Search with BIT*[1]

Any discrete set of states distributed in the state space of a planning problem, $X_{\text{samples}} \subset X$, can be viewed as a graph whose edges are given algorithmically by a transition function (an *edge-implicit* graph). When these states are sampled randomly, $X_{\text{samples}} = \{\mathbf{x} \sim \mathcal{U}(X)\}$, the properties of this graph can be described by a probabilistic model known as a RGG (Penrose, 2003).

The connections (edges) between states (vertices) in a RGG depend on their relative geometric position. Common RGGs have edges to a specific number of each state's nearest neighbours (a k-nearest graph; Eppstein et al., 1997) or to all neighbours within a specific distance (an r-disc graph; Gilbert, 1961). RGG theory provides probabilistic relationships between the number and distribution of samples, the k or r defining the graph, and specific properties such as connectivity or relative cost through the graph (Muthukrishnan and Pandurangan, 2005; Penrose, 2003).

Anytime sampling-based planners can be viewed as algorithms that construct a RGG in the problem domain and search it. Some algorithms perform this construction and search simultaneously (e.g., RRT*) and others separately (e.g., FMT*) but, as in graph-based search, their performance always depends on both the accuracy of their approximation and the quality of their search. RRT* uses RGG theory to limit graph complexity while maintaining probabilistic bounds on approximation accuracy but incompletely searches the RGG in the order its constructed (i.e., performs an *unordered* search). FMT* performs a complete ordered search but uses RGG theory to define an *a priori* approximation of the problem domain (i.e., it is not anytime).

BIT* uses RGG theory to limit graph complexity while simultaneously building the graph in an anytime manner and searching it in order of potential solution quality. This is made possible by using *batches* of random samples to build an increasingly dense *edge-implicit* RGG in the informed set and using incremental search techniques to update the search (Fig. 5.3). The anytime approximation allows BIT* to run indefinitely until a (suitable) solution is found. The incremental search technique allows it to search its changing representation efficiently by reusing previous information. The ordered search assures that it only considers states from the future informed set of a given approximation and avoids unnecessary computational cost.

The complete BIT* algorithm is presented in Algs. 5.1–5.3 and Sections 5.2.1–5.2.6 with a discussion on some practical considerations presented in Section 5.2.7. For simplicity, the discussion is limited to a search from a single start state to a finite set of goal states with a constant batch size but the formulation directly extends to searches from a start or goal region, and/or with variable batch sizes. This version is publicly available in OMPL.

[1] Pronounced *Bit-Star*.

Algorithm 5.1: BIT*($\mathbf{x}_{\text{start}} \in X_{\text{free}},\ X_{\text{goal}} \subset X_{\text{free}}$)

```
1  V ← {x_start}; E ← ∅; T = (V, E);                                   ┐
2  X_unconn ← X_goal;                                                  │
3  Q_V ← V; Q_E ← ∅;                                                   │ Initialize search
4  V_sol'n ← V ∩ X_goal; V_unexpnd ← V; X_new ← X_unconn;               │ (Section 5.2.1)
5  c_i ← min_{v_goal ∈ V_sol'n} {g_T(v_goal)} ;                        ┘
6  repeat
7      if Q_E ≡ ∅ and Q_V ≡ ∅ then                                     ┐
8          X_reuse ← Prune(T, X_unconn, c_i);                          │
9          X_sampling ← Sample(m, x_start, X_goal, c_i);               │ Add new batch
10         X_new ← X_reuse ∪ X_sampling;                               │ (Section 5.2.2)
11         X_unconn ←+ X_new;                                          │
12         Q_V ← V;                                                    ┘
13     while BestQueueValue(Q_V) ≤ BestQueueValue(Q_E) do              ┐ Select edge
14         ExpandNextVertex(Q_V, Q_E, c_i);                            │ (Section 5.2.3)
15     (v_min, x_min) ← PopBestInQueue(Q_E);                           ┘
16     if g_T(v_min) + ĉ(v_min, x_min) + ĥ(x_min) < c_i then           ┐
17         if g_T(v_min) + ĉ(v_min, x_min) < g_T(x_min) then           │
18             c_edge ← c(v_min, x_min);                               │
19             if g_T(v_min) + c_edge + ĥ(x_min) < c_i then            │
20                 if g_T(v_min) + c_edge < g_T(x_min) then            │
21                     if x_min ∈ V then                               │
22                         v_parent ← Parent(x_min);                   │
23                         E ←− {(v_parent, x_min)};                   │
24                     else                                            │ Process edge
25                         X_unconn ←− {x_min};                        │ (Section 5.2.4)
26                         V ←+ {x_min}; Q_V ←+ {x_min}; V_unexpnd ←+ {x_min};
27                         if x_min ∈ X_goal then                      │
28                             V_sol'n ←+ {x_min};                     │
29                     E ←+ {(v_min, x_min)};                          │
30                     c_i ← min_{v_goal ∈ V_sol'n} {g_T(v_goal)};     │
31     else                                                            │
32         Q_E ← ∅; Q_V ← ∅;                                           ┘
33 until STOP;
34 return T;
```

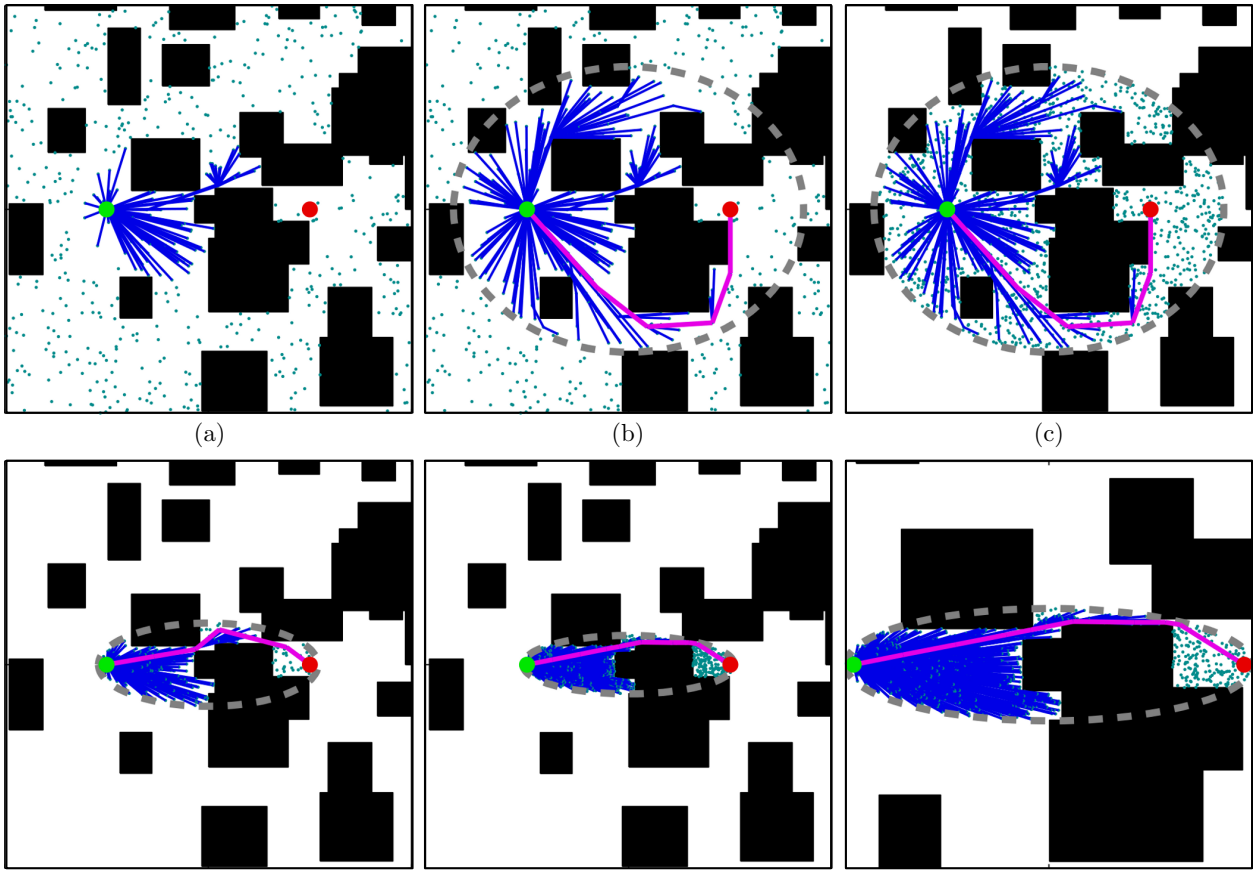


Figure 5.3: An example of how BIT* uses incremental techniques to efficiently search batches of samples in order of potential solution quality. The search starts with a batch of samples uniformly distributed in the planning domain. This batch is searched outward from the start in order of potential solution quality, (a). The search continues until the batch is exhausted or a solution is found that cannot be improved with the current samples, (b). New samples are then added to the informed set and incremental techniques are used to continue the search, (c)–(e). This results in an algorithm that performs an ordered anytime search that almost-surely asymptotically converges to the optimal solution, shown enlarged in (f). By ordering the search, BIT* finds solutions by searching the future informed set of a batch.

BIT* builds an explicit spanning tree of the implicit RGG defined by a batch of samples. The graph initially consists of only the start and goal (Alg. 5.1, Lines 1–5; Section 5.2.1) but is incrementally grown with batches of new samples during the search (Alg. 5.1, Lines 7–12; Section 5.2.2). The graph is searched in order of potential solution quality by selecting the best possible edge from a queue ordered by potential solution cost (Alg. 5.1, Lines 13–15; Section 5.2.3) and considering whether this edge improves the cost-to-come of its target vertex and could improve the current solution (Alg. 5.1, Lines 16–32; Section 5.2.4). The search continues until no edges in the implicit RGG could provide a better solution, at which point the accuracy of the RGG approximation is increased by adding a batch of new samples and the search is resumed. This process continues indefinitely until a suitable solution is found.

5.2.1 Initialization (Alg. 5.1, Lines 1–5)

BIT* begins searching a planning problem with the start, $\mathbf{x}_{\text{start}}$, in the spanning tree, $\mathcal{T} \coloneqq (V, E)$, and the goal states, X_{goal}, in the set of unconnected states, X_{unconn} (Alg. 5.1, Lines 1–2). This defines an implicit RGG whose vertices consist of all states (i.e., $V \cup X_{\text{unconn}}$) and whose edges are defined by a distance function and an appropriate connection limit. When the goal is a continuous region of the problem domain it will need to be discretized (e.g., sampled) before adding to the set of unconnected states.

The explicit spanning tree of this RGG is built using two queues, a vertex expansion queue, $\mathcal{Q}_V$, and an edge evaluation queue, $\mathcal{Q}_E$. These queues are sorted in order of potential solution quality through the current tree. Vertices in the vertex queue, $\mathbf{v} \in \mathcal{Q}_V$, are ordered by the sum of their current cost-to-come and an estimate of their cost-to-go, $g_{\mathcal{T}}(\mathbf{v}) + \widehat{h}(\mathbf{v})$. Edges in the edge queue, $(\mathbf{v}, \mathbf{x}) \in \mathcal{Q}_E$, are sorted by the sum of the current-cost-to-come of their source vertex, an estimate of the edge cost, and an estimate of the cost-to-go of their target vertex, $g_{\mathcal{T}}(\mathbf{v}) + \widehat{c}(\mathbf{v}, \mathbf{x}) + \widehat{h}(\mathbf{x})$. Ties are broken in the vertex queue in favour of entries with the lowest cost-to-come through the current tree, $g_{\mathcal{T}}(\mathbf{v})$, and in the edge queue in favour of the lowest cost-to-come through the current tree and estimated edge cost, $g_{\mathcal{T}}(\mathbf{v}) + \widehat{c}(\mathbf{v}, \mathbf{x})$, and then the cost-to-come through the current tree, $g_{\mathcal{T}}(\mathbf{v})$. These queues are initialized to contain all the vertices in the tree and an empty queue, respectively (Alg. 5.1, Line 3).

To improve search efficiency, BIT* tracks the vertices in the goal region, $V_{\text{sol'n}}$, the vertices that have never been expanded, V_{unexpnd}, the samples newly created during this batch, X_{new}, and the current best solution, c_i. These are initialized to any vertices already in the solution (empty in all but the most trivial planning problems), the existing vertices, the existing samples, and the current best solution, respectively (Alg. 5.1, Lines 4–5). As is customary, the minimum of an empty set is taken to be infinity.

Initialized, BIT* now alternately builds an increasingly accurate implicit RGG approximation of the continuous planning problem (Section 5.2.2) and searches these representations for explicit solutions in order of potential solution quality (Sections 5.2.3 and 5.2.4).

5.2.2 Batch Addition (Alg. 5.1, Lines 7–12)

BIT* alternates between building an increasingly dense approximation of the continuous planning problem and searching this representation for a solution. The approximation is updated whenever the search is finished (i.e., both queues are empty; Alg. 5.1, Line 7) by removing unnecessary states and adding a batch of new samples. This avoids the computational cost of representing regions of the problem domain that cannot provide a better solution while increasing the accuracy of approximating the regions that can (i.e., the informed set). This improving approximation allows BIT* to almost-surely converge asymptotically to the optimal solution.

The approximation is pruned to the informed set by removing any states or edges that cannot improve the *current* solution (Alg. 5.1, Line 8; Section 5.2.6). This reduces unnecessary complexity but may disconnect vertices in the informed set that cannot improve the solution solely because of their current connections. These vertices are recycled as additional 'new' samples in the batch so that they may be reconnected later if better connections are found.

The approximation is improved by adding m new randomly generated samples from the informed set (Alg. 5.1, Line 9). This can be accomplished with direct informed sampling or advanced rejection sampling (e.g., Kunz et al., 2016).

Each batch of states are labelled as X_{new} for the duration of that batch's search (Alg. 5.1, Line 10). This set is used to improve search efficiency and consists of both the newly generated samples and the recycled disconnected vertices. BIT* adds these new states to the set of unconnected states and initializes the vertex queue with all the vertices in the tree (Alg. 5.1, Lines 11–12) to restart the search (Sections 5.2.3 and 5.2.4).

5.2.3 Edge Selection (Alg. 5.1, Lines 13–15)

Traditional graph-based search techniques assume that finding and evaluating vertex connections is computationally inexpensive (e.g., given explicitly). This is not true in sampling-based planning as finding vertex connections (i.e., the edges in the edge-implicit RGG) requires performing a nearest-neighbour search and evaluating them requires checking for collisions and solving two-point BVPs (e.g., differential constraints). BIT* avoids these computational costs until required by using a lazy search procedure that delays both finding and evaluating connections in the RGG. Similar lazy techniques can be found in both advanced graph-based search and sampling-based planners (Bohlin and Kavraki, 2000; Branicky et al., 2001; Cohen et al., 2014b; Hauser, 2015; Helmert, 2006; Sánchez and Latombe, 2002).

Connections are found by using a vertex queue, $\mathcal{Q}_V$, ordered by potential solution quality. This queue delays processing a vertex (i.e., performing a nearest-neighbour search) until its outgoing connections *could* be part of the best solution to the current graph. These connections are evaluated by using an edge queue, $\mathcal{Q}_E$, also ordered by potential solution

quality. This queue delays evaluating an edge (i.e., performing collision checks and solving two-point BVPs) until it *could* be part of the best solution to the current graph.

A vertex could be part of the best solution when it could provide an outgoing edge better than the best edge in the edge queue. When the heuristic is consistent (e.g., the L^2 norm) the queue value of a vertex, $\mathbf{v} \in \mathcal{Q}_V$, is a lower-bounding estimate of the queue value of its outgoing edges,

$$\forall \mathbf{x} \in X,\ g_{\mathcal{T}}(\mathbf{v}) + \widehat{h}(\mathbf{v}) \leq g_{\mathcal{T}}(\mathbf{v}) + \widehat{c}(\mathbf{v}, \mathbf{x}) + \widehat{h}(\mathbf{x}).$$

The best edge at any iteration can therefore be found by processing the vertex queue until it is worse than the edge queue (Alg. 5.1, Line 13). This process of removing a vertex from the vertex queue and placing its outgoing edges in the edge queue is referred to as *expanding* a vertex (Alg. 5.1, Line 14; ; Section 5.2.5). Once all necessary vertices are expanded, the best edge in the queue, $(\mathbf{v}_{\min}, \mathbf{x}_{\min})$, is removed (Alg. 5.1, Line 15) and used for this iteration of the search (Section 5.2.4).

The functions `BestQueueValue` $(\cdot)$ and `PopBestInQueue` $(\cdot)$ return the value of the element at the front of a queue and pop the element off the front of a queue, respectively.

5.2.4 Edge Processing (Alg. 5.1, Lines 16–32)

BIT* also uses heuristics to avoid expensive calculations when evaluating the best edge, $(\mathbf{v}_{\min}, \mathbf{x}_{\min})$. An edge is added to the spanning tree if and only if

1. an *estimate* of its cost *could* provide a better *solution*, given the current tree,

$$g_{\mathcal{T}}(\mathbf{v}_{\min}) + \widehat{c}(\mathbf{v}_{\min}, \mathbf{x}_{\min}) + \widehat{h}(\mathbf{x}_{\min}) < c_i, \qquad \text{(Alg. 5.1, Line 16)}$$

2. an *estimate* of its cost *could* improve the *current tree*,

$$g_{\mathcal{T}}(\mathbf{v}_{\min}) + \widehat{c}(\mathbf{v}_{\min}, \mathbf{x}_{\min}) < g_{\mathcal{T}}(\mathbf{x}_{\min}), \qquad \text{(Alg. 5.1, Line 17)}$$

3. its *real* cost *could* provide a better *solution*, given the current tree,

$$g_{\mathcal{T}}(\mathbf{v}_{\min}) + c(\mathbf{v}_{\min}, \mathbf{x}_{\min}) + \widehat{h}(\mathbf{x}_{\min}) < c_i, \qquad \text{(Alg. 5.1, Line 19)}$$

4. and its *real* cost *will* improve the *current tree*,

$$g_{\mathcal{T}}(\mathbf{v}_{\min}) + c(\mathbf{v}_{\min}, \mathbf{x}_{\min}) < g_{\mathcal{T}}(\mathbf{x}_{\min}). \qquad \text{(Alg. 5.1, Line 20)}$$

Conditions 1 and 3 are always true in the absence of a solution, while Conditions 2 and 4 are always true when the target of the edge, $\mathbf{x}_{\min}$, is not in the spanning tree.

Checking if the edge could ever provide a better solution or improve the current tree (Conditions 1 and 2) allows BIT* to reject edges without calculating their true cost (Alg. 5.1, Lines 16–17). Condition 1 also provides a stopping condition for searching the current RGG. When an edge fails this condition so does the entire queue and both queues can be cleared

to start a new batch (Alg. 5.1, Line 32). If the edge fails Condition 2 it is discarded and the iteration finishes. If the edge passes both these conditions its true cost is calculated by performing a collision check and solving any two-point BVPs (Alg. 5.1, Line 18). Edges in collision are considered to have infinite cost.

Checking if the real edge could provide a better solution given the current tree (Condition 3), allows BIT* to reduce tree complexity by rejecting edges that could never improve the current solution (Alg. 5.1, Line 19). Checking if the real edge improves the current tree (Condition 4), assures the cost-to-come of the explicit tree decreases monotonically (Alg. 5.1, Line 20). If the edge fails either of these conditions it is discarded and the iteration finishes.

An edge passing these conditions is added to the spanning tree. If the target vertex is already connected (Alg. 5.1, Line 21), then the edge represents a *rewiring* and the current edge must be removed (Alg. 5.1, Lines 22–23). Otherwise, the edge represents an *expansion* of the tree and the target vertex must be moved from the set of unconnected states to the set of vertices, inserted into the vertex queue for future expansion, and marked as a never-expanded vertex (Alg. 5.1, Lines 25–26). The new vertex is also added to the set of vertices in the goal region if appropriate (Alg. 5.1, Lines 27–28).

The new edge is then finally added to the tree (Alg. 5.1, Line 29) and the current best solution is updated as necessary (Alg. 5.1, Line 30). The search then continues by selecting the next edge in the queue (Section 5.2.3) or increasing the approximation accuracy if the current RGG cannot provide a better solution (Section 5.2.2).

5.2.5 Vertex Expansion (Alg. 5.1, Line 14; Alg. 5.2)

The function `ExpandNextVertex` $(\mathcal{Q}_V, \mathcal{Q}_E, c_i)$ removes the front of the vertex queue (Alg. 5.2, Line 1) and adds its outgoing edges in the RGG to the edge queue. The RGG is defined using the results of Karaman and Frazzoli (2011), (3.4) and (3.5), to limit graph complexity while maintaining almost-sure asymptotic convergence to the optimum. Edges exist between a vertex and the k_{BIT^*}-closest states or all states within a distance of r_{BIT^*}, with

$$r_{\text{BIT}^*} > r^*_{\text{BIT}^*},$$
$$r^*_{\text{BIT}^*} \coloneqq \left(2\left(1+\frac{1}{n}\right)\left(\frac{\min\left\{\lambda\left(X\right), \lambda\left(X_{\widehat{f}}\right)\right\}}{\zeta_n}\right)\left(\frac{\log\left(|V|+|X_{\text{unconn}}|-m\right)}{|V|+|X_{\text{unconn}}|-m}\right)\right)^{\frac{1}{n}}, \quad (5.1)$$

and

$$k_{\text{BIT}^*} > k^*_{\text{BIT}^*},$$
$$k^*_{\text{BIT}^*} \coloneqq e\left(1+\frac{1}{n}\right)\log\left(|V|+|X_{\text{unconn}}|-m\right), \quad (5.2)$$

where m is the number of samples added in the last batch.

Algorithm 5.2: `ExpandNextVertex` $(\mathcal{Q}_V \subseteq V,\ \mathcal{Q}_E \subseteq V \times (V \cup X),\ c_i \in \mathbb{R}_{\geq 0})$

1 $\mathbf{v}_{\min} \leftarrow$ `PopBestInQueue` $(\mathcal{Q}_V)$;
2 **if** $\mathbf{v}_{\min} \in V_{\text{unexpnd}}$ **then**
3 $\quad X_{\text{near}} \leftarrow$ `Near` $(X_{\text{unconn}}, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
4 **else**
5 $\quad X_{\text{near}} \leftarrow$ `Near` $(X_{\text{new}} \cap X_{\text{unconn}}, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
6 $\mathcal{Q}_E \xleftarrow{+} \left\{(\mathbf{v}_{\min}, \mathbf{x}) \in V \times X_{\text{near}} \;\middle|\; \widehat{g}(\mathbf{v}_{\min}) + \widehat{c}(\mathbf{v}_{\min}, \mathbf{x}) + \widehat{h}(\mathbf{x}) < c_i\right\}$;
7 **if** $\mathbf{v}_{\min} \in V_{\text{unexpnd}}$ **then**
8 $\quad V_{\text{near}} \leftarrow$ `Near` $(V, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
9 $\quad \mathcal{Q}_E \xleftarrow{+} \left\{(\mathbf{v}_{\min}, \mathbf{w}) \in V \times V_{\text{near}} \;\middle|\; (\mathbf{v}_{\min}, \mathbf{w}) \notin E,\ \widehat{g}(\mathbf{v}_{\min}) + \widehat{c}(\mathbf{v}_{\min}, \mathbf{w}) + \widehat{h}(\mathbf{w}) < c_i,\ \widehat{g}(\mathbf{v}_{\min}) + \widehat{c}(\mathbf{v}_{\min}, \mathbf{w}) < g_{\mathcal{T}}(\mathbf{w})\right\}$;
10 $\quad V_{\text{unexpnd}} \xleftarrow{-} \{\mathbf{v}_{\min}\}$;

This connection limit is calculated from the cardinality of the graph *minus* the m new samples to simplify proving almost-sure asymptotic optimality (Section 5.3). This lower bound will be large for the initial sparse batches but it can be thresholded with a maximum edge length, as done by RRT* in (3.3). The function `Near` returns the states that meet the selected RGG connection criteria for a given vertex.

Every vertex in the tree is either expanded or pruned in every batch. Processing all the outgoing edges from vertices would result in BIT* repeatedly considering the same previously rejected edges. This can be avoided by using the sets of never-expanded vertices, V_{unexpnd}, and new samples, X_{new}, to add only *previously unconsidered* edges to the edge queue.

Whether edges to unconnected samples are new depends on whether the source vertex has previously been expanded (Alg. 5.2, Line 2). If it has not been expanded then no outgoing connections have been considered and all nearby unconnected samples are potential descendants (Alg. 5.2, Line 3). If it has been expanded then any connections to 'old' unconnected samples have already been considered and rejected and only the 'new' samples are considered as potential descendants (Alg. 5.2, Line 5). The subset of these potential edges that could improve the current solution are added to the queue in both situations (Alg. 5.2, Line 6).

Whether edges to connected samples (i.e., *rewirings*) are new also depends on whether the source vertex has been expanded (Alg. 5.2, Line 7). If it has not been expanded then all nearby connected vertices are considered as potential descendants (Alg. 5.2, Line 8). The subset of these potential edges that could improve the current solution *and* the current tree are added to the queue (Alg. 5.2, Line 9) and the vertex is then marked as expanded (Alg. 5.2, Line 10).

If a vertex has previously been expanded then no rewirings are considered. Improvements in the tree may now allow a previously considered edge to improve connected vertices but considering these connections would require repeatedly reconsidering infeasible edges. This lack of *propagated rewiring* has no effect on almost-sure asymptotic optimality, as in RRT*.

Algorithm 5.3: `Prune` $(\mathcal{T} = (V, E),\ X_{\text{unconn}} \subset X,\ c_i \in \mathbb{R}_{\geq 0})$

```
1  X_reuse ← ∅;
2  X_unconn ←⁻ { x ∈ X_unconn | f̂(x) ≥ c_i };
3  forall v ∈ V in order of increasing g_T(v) do
4      if f̂(v) > c_i or g_T(v) + ĥ(v) > c_i then
5          V ←⁻ {v}; V_sol'n ←⁻ {v}; V_unexpnd ←⁻ {v};
6          v_parent ← Parent(v);
7          E ←⁻ {(v_parent, v)};
8          if f̂(v) < c_i then
9              X_reuse ←⁺ {v};
10 return X_reuse;
```

5.2.6 Graph Pruning (Alg. 5.1, Line 8; Alg. 5.3)

The function, `Prune` $(\mathcal{T}, X_{\text{unconn}}, c_i)$, reduces the complexity of both the approximation of the continuous planning problem (i.e., the implicit RGG) and its search (i.e., the explicit spanning tree) by limiting them to the informed set. It removes any states that can *never* provide a better solution and disconnects any vertices that cannot provide a better solution *given the current tree*. Disconnected vertices that *could* improve the solution with a better connection are reused as new samples in the next batch to maintain uniform sample density in the informed set. This assures that in each batch every vertex is either expanded or pruned as assumed by `ExpandNextVertex` to avoid reconsidering edges (Section 5.2.5).

The set of recycled vertices is initialized as an empty set (Alg. 5.3, Line 1) and all unconnected states that cannot provide a better solution (i.e., are not members of the informed set) are removed (Alg. 5.3, Line 2). The connected vertices are then incrementally pruned in order of increasing cost-to-come (Alg. 5.3, Line 3) by identifying those that can never provide a better solution or improve the solution given the *current* tree (Alg. 5.3, Line 4). Vertices that fail either condition are removed from the tree by disconnecting their incoming edge and removing them from the vertex set and any labelling sets (Alg. 5.3, Lines 5–7).

Any disconnected vertex that could provide a better solution (i.e., is a member of the informed set) is reused as a sample in the next batch (Alg. 5.3, Lines 8–9). This maintains uniform sample density in the informed set and assures that vertices will be reconnected if future improvements allow them to provide a better solution. This set of disconnected vertices is returned to BIT* at the end of the pruning procedure (Alg. 5.3, Line 10).

5.2.7 Practical Considerations

Algs. 5.1–5.3 describe an *implementation-agnostic* version of BIT* and leave room for a number of practical improvements depending on the specific implementation.

Many of the sets (e.g., X_{new}, V_{unexpnd}) can be implemented more efficiently as labels. Searches (e.g., Alg. 5.2, Line 3) can be implemented efficiently with appropriate datastructures that do not require an exhaustive global search (e.g., k-d trees). Pruning (Alg. 5.1, Line 8; Alg. 5.3) is expensive and should only occur when a new solution has been found or limited to *significant* changes in solution cost.

Ordered containers provide efficient sorted queues but the order of elements in the vertex and edge queues change when vertices are rewired. In practice, there appears to be little difference between efficiently resorting the affected elements in these queues and only lazily resorting the queue to assure no elements have been missed when reaching the end.

Depending on the datastructure used for the edge queue, it may be beneficial to remove unnecessary entries when a new edge is added to the spanning tree, i.e., by adding

$$\mathcal{Q}_E \xleftarrow{-} \{(\mathbf{v}, \mathbf{x}_{\min}) \in \mathcal{Q}_E \mid \widehat{g}(\mathbf{v}) + \widehat{c}(\mathbf{v}, \mathbf{x}_{\min}) \geq g_{\mathcal{T}}(\mathbf{x}_{\min})\}$$

after Alg. 5.1, Line 29.

5.3 Analysis

The performance of BIT* is analyzed theoretically using the results of Karaman and Frazzoli (2011). It is shown to be probabilistically complete (Theorem 5.1) and almost-surely asymptotically optimal (Theorem 5.2). Its search order is also shown to be equivalent to a lazy version of LPA* (Lemma 5.3).

Theorem 5.1 (Probabilistic completeness of BIT*)**.** *The probability that BIT* finds a feasible solution to a given planning problem, if one exists, when given infinite samples is one,*

$$\liminf_{q \to \infty} P\left(\sigma_{q,\text{BIT}^*} \in \Sigma,\, \sigma_{q,\text{BIT}^*}(0) = \mathbf{x}_{\text{start}},\, \sigma_{q,\text{BIT}^*}(1) \in X_{\text{goal}}\right) = 1,$$

where q is the number of samples, σ_{q,BIT^*} is the path found by BIT from those samples, and Σ is the set of all feasible, collision-free paths.*

Proof. Proof of Theorem 5.1 follows from the proof of Theorem 5.2. □

Theorem 5.2 (Almost-sure asymptotic optimality of BIT*)**.** *The probability that BIT* converges asymptotically towards the optimal solution of a given planning problem, if one exists, when given infinite samples is one,*

$$P\left(\limsup_{q \to \infty} c\left(\sigma_{q,\text{BIT}^*}\right) = c\left(\sigma^*\right)\right) = 1,$$

where q is the number of samples, σ_{q,BIT^*} is the path found by BIT from those samples, and σ^* is optimal solution to the planning problem.*

Proof. Theorem 5.2 is proven by showing that BIT* considers at least the same edges as RRT* for a sequence of states, $X_{\text{samples}} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_q)$, and connection limit, $r_{\text{BIT}^*} \geq r_{\text{RRT}^*}$.

RRT* incrementally builds a tree from a sequence of samples. For each state in the sequence, $\mathbf{x}_k$, it considers the neighbourhood of earlier states that are within the connection limit,

$$X_{\text{near},k} \coloneqq \left\{ \mathbf{x}_j \in X_{\text{samples}} \;\middle|\; j < k,\ \|\mathbf{x}_k - \mathbf{x}_j\|_2 \leq r_{\text{RRT}^*} \right\}.$$

It selects the connection from this neighbourhood that minimizes the cost-to-come of the state and then evaluates the ability of connections from this state to reduce the cost-to-come of the other states in the neighbourhood.

Given the same sequence of states, BIT* groups them into batches of samples, $X_{\text{samples}} = (Y_1, Y_2, \ldots, Y_\ell)$, where each batch is a set of m samples, e.g., $Y_1 \coloneqq \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m\}$. It incrementally builds a tree by processing this batched sequence of samples. For each state in the sequence, $\mathbf{y} \in Y_k$, it considers the neighbourhood of states from the same or earlier batches within the connection limit,

$$X_{\text{near},k} \coloneqq \{\mathbf{x} \in Y_j \mid j \leq k,\ \|\mathbf{y} - \mathbf{x}\|_2 \leq r_{\text{BIT}^*}\}.$$

The connection from this neighbourhood that minimizes the cost-to-come of the state is the one that will be added to the tree and all its outgoing edges will be considered in connecting its neighbours. This contains all the edges considered by RRT* for an equivalent connection limit, $r_{\text{BIT}^*} \geq r_{\text{RRT}^*}$. As (5.1) uses the same connection radius for a batch that RRT* would use for the first sample in the batch and the connection radius of both are monotonically decreasing, this shows that BIT* is almost-surely asymptotically optimal and proves Theorem 5.2. □

BIT* uses a search order that is equivalent to a version of LPA* that *lazily* estimates the edge cost between states in the queue (Lemma 5.3). As BIT* does not reconsider outgoing connections from rewired vertices (i.e., it does not propagate rewirings), it is not a complete LPA* search of the RGG but instead a search similar to TLPA* (Aine and Likhachev, 2016).

Lemma 5.3 (Equivalence of BIT* to a lazy LPA* search)**.** *An edge queue ordered first on the sum of a vertex's estimated cost-to-go, estimated incoming edge cost, and current cost-to-come of its parent,*

$$g_{\mathcal{T}}(\mathbf{u}) + \widehat{c}(\mathbf{u}, \mathbf{v}) + \widehat{h}(\mathbf{v}),$$

then the estimated cost-to-come of the vertex,

$$g_{\mathcal{T}}(\mathbf{u}) + \widehat{c}(\mathbf{u}, \mathbf{v}),$$

and then the cost-to-come of its parent,

$$g_{\mathcal{T}}(\mathbf{u}),$$

is an equivalent ordering to LPA (Koenig et al., 2004).*

Proof. LPA* uses a queue of vertices ordered lexicographically first on the solution cost constrained to go through the vertex and then the cost-to-come of the vertex. Both these terms are calculated for a vertex, $\mathbf{v} \in V$, considering all its possible incoming edges (referred to as the *rhs-value* in LPA*), i.e.,

$$\min_{(\mathbf{u},\mathbf{v})\in E} \{g_{\mathcal{T}}(\mathbf{u}) + c(\mathbf{u},\mathbf{v})\} + \widehat{h}(\mathbf{v})$$

and

$$\min_{(\mathbf{u},\mathbf{v})\in E} \{g_{\mathcal{T}}(\mathbf{u}) + c(\mathbf{u},\mathbf{v})\}.$$

This minimum requires calculating the true edge cost between a vertex and all of its possible parents. This calculation is expensive in sampling-based planning (e.g., collision checking, differential constraints, etc.), and reducing its calculation is desirable. This can be achieved by incrementally calculating the minimum in the order given by an admissible heuristic estimate of edge cost. Considering edges into the vertex in order of increasing *estimated* cost calculates a running minimum that finds the true minimum when the estimated cost through the next edge to consider is higher than the current value.

BIT* combines the minimum calculations for individual vertices into a single edge queue. It simultaneously calculates the minimum cost-to-come for each vertex in the current graph while expanding vertices in order of increasing estimated solution cost. □

5.4 Modifications and Extensions

The basic version of BIT* presented in Algs. 5.1–5.3 can be modified and extended to include more advanced features that may improve performance for some planning applications. Section 5.4.1 presents a method to delay rewiring the tree until an initial solution is found. This prioritizes exploring the RGG to find solutions and may be beneficial in time-constrained applications. Section 5.4.2 presents a method to delay sampling until necessary. This avoids approximating regions of the planning problem that are never searched, improves performance in large planning problems, and avoids the need to define *a priori* limits in unbounded problems.

Section 5.4.3 presents a method for BIT* to occasionally remove unconnected samples while maintaining almost-sure asymptotic optimality. This avoids repeated connection attempts to infeasible states and may be beneficial in problems when many regions of the free space are unreachable. Section 5.4.4 extends the idea of reducing the number of connections attempted per sample during an ordered search to develop Sorted RRT* (SORRT*). This version of RRT* uses batches of samples to order its search by potential solution quality, as in BIT*, but only makes one connection attempt per sample, as in RRT*.

Algorithm 5.4: `ExpandNextVertex` $(\mathcal{Q}_V \subseteq V,\ \mathcal{Q}_E \subseteq V \times (V \cup X),\ c_i \in \mathbb{R}_{\geq 0})$

1 $\mathbf{v}_{\min} \leftarrow$ `PopBestInQueue` $(\mathcal{Q}_V)$;
2 **if** $\mathbf{v}_{\min} \in V_{\text{unexpnd}}$ **then**
3 $\quad X_{\text{near}} \leftarrow$ `Near` $(X_{\text{unconn}}, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
4 $\quad V_{\text{unexpnd}} \xleftarrow{-} \{\mathbf{v}_{\min}\}$;
5 $\quad V_{\text{delayed}} \xleftarrow{+} \{\mathbf{v}_{\min}\}$;
6 **else**
7 $\quad X_{\text{near}} \leftarrow$ `Near` $(X_{\text{new}} \cap X_{\text{unconn}}, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
8 $\mathcal{Q}_E \xleftarrow{+} \left\{(\mathbf{v}_{\min}, \mathbf{x}) \in V \times X_{\text{near}} \;\middle|\; \widehat{g}(\mathbf{v}_{\min}) + \widehat{c}(\mathbf{v}_{\min}, \mathbf{x}) + \widehat{h}(\mathbf{x}) < c_i\right\}$;
9 **if** $\mathbf{v}_{\min} \in V_{\text{delayed}}$ **and** $c_i < \infty$ **then**
10 $\quad V_{\text{near}} \leftarrow$ `Near` $(V, \mathbf{v}_{\min}, r_{\text{BIT}^*})$;
11 $\quad \mathcal{Q}_E \xleftarrow{+} \left\{(\mathbf{v}_{\min}, \mathbf{w}) \in V \times V_{\text{near}} \;\middle|\; (\mathbf{v}_{\min}, \mathbf{w}) \notin E,\ \widehat{g}(\mathbf{v}_{\min}) + \widehat{c}(\mathbf{v}_{\min}, \mathbf{w}) + \widehat{h}(\mathbf{w}) < c_i,\ \widehat{g}(\mathbf{v}_{\min}) + \widehat{c}(\mathbf{v}_{\min}, \mathbf{w}) < g_{\mathcal{T}}(\mathbf{w})\right\}$;
12 $\quad V_{\text{delayed}} \xleftarrow{-} \{\mathbf{v}_{\min}\}$;

5.4.1 Delayed Rewiring

Many robotic systems have a finite amount of computational time available to solve planning problems. In these situations, improving the existing graph before an initial solution is found only reduces the likelihood of BIT* solving the given problem. A method to delay rewirings until a solution is found is presented in Alg. 5.4 as simple modifications to `ExpandNextVertex`, with changes highlighted in red (cf. Alg. 5.2). The rewirings are still performed once a solution is found and this method does not affect almost-sure asymptotic optimality.

Rewirings are delayed by separately tracking whether vertices are unexpanded to nearby unconnected samples, V_{unexpnd}, and unexpanded to nearby connected vertices, V_{delayed}. This allows BIT* to prioritize finding a solution by only considering edges to new samples and then once a solution is found improving the graph by considering rewirings from the existing vertices.

A vertex is moved from the never-expanded set to the delayed set when edges to its potential unconnected descendants are added to the edge queue (Alg. 5.4, Lines 4–5). This allows future expansions of the vertex to avoid old unconnected samples while tracking that the vertex's rewirings have not been considered. Vertices in the delayed set are expanded as potential rewirings of other connected vertices once a solution is found (Alg. 5.4, Line 9) and the delayed label is removed (Alg. 5.4, Line 12).

This extension requires initializing and resetting V_{delayed} along with the other labelling sets (e.g., Alg. 5.1, Line 4 and Alg. 5.3, Line 5). This extension is included in the publicly available OMPL implementation of BIT*.

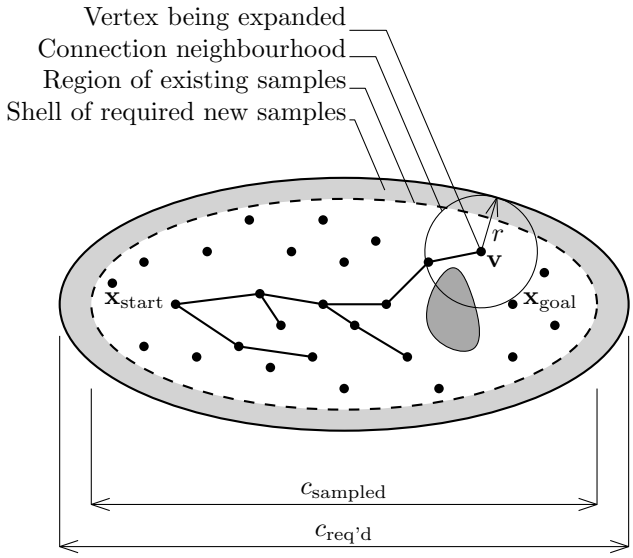


Figure 5.4: An illustration of just-in-time (JIT) sampling. Samples are generated only when they are necessary for the expansion of a vertex, $\mathbf{v}$, into the edge queue. This is accomplished while maintaining uniform sample density by an expanding informed set. The informed set is expanded by adding uniformly distributed samples in the prolate hyperspheroidal shell defined by the difference between the maximum heuristic value of the neighbourhood, $c_{\text{req'd}}$, and the currently sampled informed set, c_{sampled}.

5.4.2 Just-in-Time (JIT) Sampling

Many robotic systems operate in environments that are unbounded (e.g., the outdoors). These problems has previously required using *a priori* search limits to make the problem domain tractable. Selecting these limits can be difficult and may prevent finding a solution (e.g., defining a domain that does not contain a solution) or reduce performance (e.g., defining a domain too large to search sufficiently). A method to avoid these problems in BIT* by generating samples *just in time* (JIT) is presented in Alg. 5.5 and accompanying modifications to the main algorithm. This modification generates samples only when needed by BIT*'s search while still maintaining uniform sample density and almost-sure asymptotic convergence to the optimum. This avoids approximating regions of the problem that are not used to find a solution and allows BIT* to operate directly on large or unbounded planning problems.

BIT* searches a planning problem by constructing and searching an implicit RGG defined by a number of uniformly distributed samples (vertices) and their relative distances (edges). These samples are given explicitly in Algs. 5.1–5.3 but are not used until they could be a descendant of a vertex in the tree. This occurs when the samples are within the local neighbourhood of the vertex (Fig. 5.4). Samples therefore can be generated as needed by incrementally sampling an expanding informed set. The size of the informed set, $c_{\text{req'd}}$, necessary to contain the neighbourhood of a vertex, X_{near}, for planning problems seeking to

Algorithm 5.5: `UpdateSamples` $(\mathbf{v} \in V,\ c_{\text{sampled}} \leq c_i,\ c_i \in \mathbb{R}_{\geq 0})$

1 $c_{\text{req'd}} \leftarrow \min\left\{\widehat{f}(\mathbf{v}) + 2r_{\text{BIT}^*}, c_i\right\}$;
2 **if** $c_{\text{req'd}} > c_{\text{sampled}}$ **then**
3 $\quad \lambda_{\text{sample}} \leftarrow \lambda_{\text{PHS}}(c_{\text{req'd}}) - \lambda_{\text{PHS}}(c_{\text{sampled}})$;
4 $\quad m' \leftarrow \rho\lambda_{\text{sample}}$;
5 $\quad X_{\text{new}} \xleftarrow{+}$ `Sample` $(m', c_{\text{sampled}}, c_{\text{req'd}})$;
6 $\quad X_{\text{unconn}} \xleftarrow{+} X_{\text{new}}$;
7 $\quad c_{\text{sampled}} \leftarrow c_{\text{req'd}}$;

minimize path length,

$$c_{\text{req'd}} \coloneqq \max_{\mathbf{x} \in X_{\text{near}}} \left\{\widehat{f}(\mathbf{x})\right\},$$

is bounded from above as,

$$c_{\text{req'd}} \leq \widehat{f}(\mathbf{v}) + 2r_{\text{BIT}^*}.$$

JIT sampling only generates samples when necessary to expand vertices by incrementally sampling this growing informed set. It does this while maintaining uniform sample density by tracking the previously sampled size of the set, c_{sampled}, and only generating the new samples necessary to increase it. The function `UpdateSamples` $(\mathbf{v}, c_{\text{sampled}}, c_i)$ generates JIT samples for vertex expansion in geometric planning problems (Alg. 5.5). The required size of the sampled informed set, $c_{\text{req'd}}$, is a function of the neighbourhood and the maximum size of the informed set (Alg. 5.5, Line 1). If it is higher than the previously sampled cost, c_{sampled}, the local neighbourhood requires new samples (Alg. 5.5, Line 2). If it is less than the sampled cost the neighbourhood has already been sampled and no samples are generated.

The number of required new samples, m', can be calculated from the chosen batch sample density, ρ, and the volume of the shell being sampled (Alg. 5.5, Lines 3–4). These samples are added to the set of new states and the set of unconnected states (Alg. 5.5, Lines 5–6). Finally, the sampled cost is updated to reflect the new size of the sampled informed set (Alg. 5.5, Line 7).

The function $\lambda_{\text{PHS}}(\cdot)$ calculates the measure of the prolate hyperspheroid defined by the start and goal with the given cost using (3.8). The function `Sample` $(m', c_{\text{sampled}}, c_{\text{req'd}})$ generates samples within the cost interval $[c_{\text{sampled}}, c_{\text{req'd}})$ and may be implemented with rejection sampling.

Using Alg. 5.5 requires modifying Algs. 5.1 and 5.2. The `Sample` function of a batch (Alg. 5.1, Line 9) is replaced with an initialization of the sampled cost variable, $c_{\text{sampled}} \leftarrow 0$, and `UpdateSamples` $(\mathbf{v}, c_{\text{sampled}}, c_i)$ is added to the front of `ExpandNextVertex` (Alg. 5.2). This extension is included in the publicly available OMPL implementation of BIT*.

5.4.3 Sample Removal

BIT* approximates continuous planning problems by building an implicit RGG. It efficiently increases the accuracy of this approximation by focusing it to the informed set and alternately adds new samples to increase density and reduces the size of the informed set by searching the existing approximation for better solutions. This builds an explicit spanning tree that contains all states that could *currently* provide a better solution but may not use every state in the RGG.

States in an informed set may not be able to improve a solution for many reasons. The approximation may be insufficiently accurate (i.e., low sample density) to capture difficult features (e.g., narrow passages) or represent sufficiently optimal paths. The informed set may also include regions of the problem domain that cannot improve the solution due to unconsidered problem features (e.g., barriers separating free space) or because it is otherwise poorly chosen (i.e., low precision). Unconnected samples in the first situation may later be beneficial to the search but samples in the second represent unnecessary computational cost. Periodically removing these samples would reduce the complexity of the implicit RGG and avoid repeatedly attempting to connect them to new vertices in the tree.

Unconnected samples can be removed while maintaining the requirements for almost-sure asymptotic optimality by modifying the RGG connection limits to consider only uniformly distributed samples. This can be accomplished by using the number of uniformly distributed samples added since the last removal of unconnected states in (5.1) and (5.2). This simple extension is included in the publicly available OMPL implementation of BIT*.

5.4.4 Sorted RRT* (SORRT*)[2]

Approaching sampling-based planning as the search of an implicit RGG motivates BIT* to consider multiple connections to each sample. Section 5.4.3 presents a method to limit this number of attempts by periodically removing samples. The natural extension of this idea is to consider only a single connection attempt per sample, as in RRT*. This motivates the development of Sorted RRT* (SORRT*), a version of RRT* that orders its search by potential solution quality by sorting batches of samples.

SORRT* is presented in Alg. 5.6 as simple modifications of Informed RRT*, with changes highlighted in red (cf. Alg. 4.1). Instead of expanding the tree towards a randomly generated sample at each iteration, SORRT* extends the tree towards the best unconsidered sample in its current batch. It accomplishes this by using a queue samples, $\mathcal{Q}_{\text{Samples}}$, ordered by potential solution cost, $\widehat{f}(\cdot)$. This queue is filled with m samples (Alg. 5.6, Lines 5–6) and the search proceeds by expanding the tree towards the best sample in the queue (Alg. 5.6, Line 7). This orders the search for the m iterations required to use the samples in a batch, at which point a new batch of samples is generated and the search continues.

[2]Pronounced *Sort-Star*.

Algorithm 5.6: SORRT*($\mathbf{x}_{\text{start}} \in X_{\text{free}},\ X_{\text{goal}} \subset X$)

$V \leftarrow \{\mathbf{x}_{\text{start}}\};\ E \leftarrow \emptyset;\ \mathcal{T} = (V, E)$;
$V_{\text{sol'n}} \leftarrow \emptyset;\quad \mathcal{Q}_{\text{Samples}} \leftarrow \emptyset$;
for $i = 1 \dots q$ **do**
  $c_i \leftarrow \min_{\mathbf{v}_{\text{goal}} \in V_{\text{sol'n}}} \{g_{\mathcal{T}}(\mathbf{v}_{\text{goal}})\}$;
  if $\mathcal{Q}_{\text{Samples}} \equiv \emptyset$ **then**
    $\mathcal{Q}_{\text{Samples}} \leftarrow \texttt{Sample}(m, \mathbf{x}_{\text{start}}, X_{\text{goal}}, c_i)$;
  $\mathbf{x}_{\text{rand}} \leftarrow \texttt{PopBestInQueue}(\mathcal{Q}_{\text{Samples}})$;
  $\mathbf{v}_{\text{nearest}} \leftarrow \texttt{Nearest}(V, \mathbf{x}_{\text{rand}})$;
  $\mathbf{x}_{\text{new}} \leftarrow \texttt{Steer}(\mathbf{v}_{\text{nearest}}, \mathbf{x}_{\text{rand}})$;
  if $\texttt{CollisionFree}(\mathbf{v}_{\text{nearest}}, \mathbf{x}_{\text{new}})$ **then**
    if $\mathbf{x}_{\text{new}} \in X_{\text{goal}}$ **then**
      $V_{\text{sol'n}} \xleftarrow{+} \{\mathbf{x}_{\text{new}}\}$;
    $V \xleftarrow{+} \{\mathbf{x}_{\text{new}}\}$;
    $V_{\text{near}} \leftarrow \texttt{Near}(V, \mathbf{x}_{\text{new}}, r_{\text{rewire}})$;
    $\mathbf{v}_{\min} \leftarrow \mathbf{v}_{\text{nearest}}$;
    forall $\mathbf{v}_{\text{near}} \in V_{\text{near}}$ **do**
      $c_{\text{new}} \leftarrow g_{\mathcal{T}}(\mathbf{v}_{\text{near}}) + c(\mathbf{v}_{\text{near}}, \mathbf{x}_{\text{new}})$;
      if $c_{\text{new}} < g_{\mathcal{T}}(\mathbf{v}_{\min}) + c(\mathbf{v}_{\min}, \mathbf{x}_{\text{new}})$ **then**
        if $\texttt{CollisionFree}(\mathbf{v}_{\text{near}}, \mathbf{x}_{\text{new}})$ **then**
          $\mathbf{v}_{\min} \leftarrow \mathbf{v}_{\text{near}}$;
    $E \xleftarrow{+} \{(\mathbf{v}_{\min}, \mathbf{x}_{\text{new}})\}$;
    forall $\mathbf{v}_{\text{near}} \in V_{\text{near}}$ **do**
      $c_{\text{near}} \leftarrow g_{\mathcal{T}}(\mathbf{x}_{\text{new}}) + c(\mathbf{x}_{\text{new}}, \mathbf{v}_{\text{near}})$;
      if $c_{\text{near}} < g_{\mathcal{T}}(\mathbf{v}_{\text{near}})$ **then**
        if $\texttt{CollisionFree}(\mathbf{x}_{\text{new}}, \mathbf{v}_{\text{near}})$ **then**
          $\mathbf{v}_{\text{parent}} \leftarrow \texttt{Parent}(\mathbf{v}_{\text{near}})$;
          $E \xleftarrow{-} \{(\mathbf{v}_{\text{parent}}, \mathbf{v}_{\text{near}})\}$;
          $E \xleftarrow{+} \{(\mathbf{x}_{\text{new}}, \mathbf{v}_{\text{near}})\}$;
    $\texttt{Prune}(V, E, c_i)$;
return $\mathcal{T}$;

The function `PopBestInQueue`$(\cdot)$ pops the best element off the front of a queue given its ordering. A goal bias may be implemented in SORRT* by adding a small probability of sampling the goal instead of removing the best sample from the queue. This algorithm is publicly available in OMPL.

SORRT* can be viewed as a simplified version of BIT* that searches the RGG by considering only the best-possible edge to each vertex. Attempting to connect each sample once avoids the computational cost of repeated connection attempts to infeasible samples but makes the search dependent on the sampling order. High-utility samples (e.g., samples near the optimum) may be underutilized depending on the state of the tree when they are found. This can become problematic if these samples have a low sampling probability (e.g., samples in narrow passages). Making multiple connections attempts per sample and retaining samples for multiple batches allows BIT* to exploit these useful samples more than algorithms such as SORRT*. As seen in Section 5.5, this results in different performance especially in high state dimensions.

5.5 Experiments

The benefits of ordering the search of continuous planning problems are demonstrated on simulated problems in $\mathbb{R}^2$, $\mathbb{R}^8$, and $\mathbb{R}^{16}$ (Section 5.5.1) and one and two-armed problems for HERB (Section 5.5.2) using OMPL[3]. BIT* is compared to RRT, RRT-Connect, RRT*, FMT*, Informed RRT*, and SORRT*.

All planners used the same tuning parameters and configurations where possible. Planning time was limited to 3 seconds, 150 seconds, and 300 seconds in $\mathbb{R}^2$, $\mathbb{R}^8$, and $\mathbb{R}^{16}$ and 5 seconds and 600 seconds for HERB ($\mathbb{R}^7$ and $\mathbb{R}^{14}$), respectively. RRT-style planners used a goal-sampling bias of 5% and a maximum edge length of $\eta = 0.3$, 0.9, and 1.7 on the abstract problems ($\mathbb{R}^2$, $\mathbb{R}^8$, and $\mathbb{R}^{16}$) and 0.7 and 1.3 on HERB, respectively. These values were selected experimentally to reduce the time required to find an initial solution on simple training problems.

The RRT* planners, FMT*, and BIT* all used a connection radius equal to 2 times their theoretical minimum (e.g., $r_{\text{RRT}^*} = 2r^*_{\text{RRT}^*}$) and approximated the Lebesgue measure of the free space with the measure of the entire planning problem. The RRT* planners also used the ordered rewiring technique presented in Perez et al. (2011). Informed RRT*, SORRT*, and BIT* used the L^2 norm as estimates of cost-to-come and cost-to-go, direct informed sampling, and delayed pruning the graph until solution cost changed by more than 5%. SORRT* and BIT* both used $m = 100$ samples per batch and BIT* used an approximately sorted queue. BIT* also thresholded its large initial connection radius by using the same radius for both the first and second batches.

[3]The experiments were run on a laptop with 16 GB of RAM and an Intel i7-4810MQ processor. The abstract experiments were run in Ubuntu 12.04 (64-bit) with Boost 1.58, while the HERB experiments were run in Ubuntu 14.04 (64-bit).

5.5.1 Simulated Planning Problems

The algorithms were tested on simulated problems in $\mathbb{R}^2$, $\mathbb{R}^8$, and $\mathbb{R}^{16}$ on worlds consisting of many different homotopy classes (Section 5.5.1.1) and randomly generated obstacles (Section 5.5.1.2). The planners were tested with 100 different pseudo-random seeds on each problem and state dimension. The solution cost of each planner was recorded every 1 millisecond by a separate thread and the median was calculated from the 100 trials by interpolating each trial at a period of 1 millisecond. The absence of a solution was considered an infinite cost for the purpose of calculating the median.

5.5.1.1 Worlds with Many Homotopy Classes

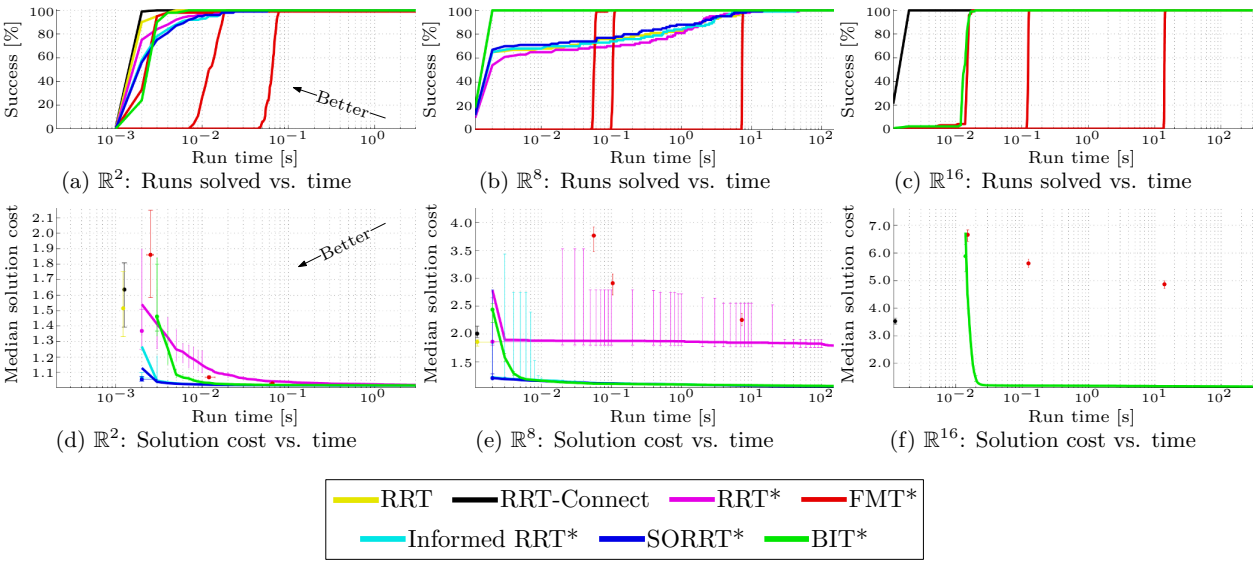
The algorithms were tested on problems with many homotopy classes in $\mathbb{R}^2$, $\mathbb{R}^8$, and $\mathbb{R}^{16}$ (Fig. 4.10b). The worlds consisted of a (hyper)cube of width $l = 4$ with the start and goal located at $[-0.5, 0, \ldots, 0]^T$ and $[0.5, 0, \ldots, 0]^T$, respectively. The problem domain was filled with a regular pattern of axis-aligned (hyper)rectangular obstacles with a width such that the start and goal were 5 'columns' apart.

The results are presented in Fig. 5.5 with the percent of trials solved and the median solution cost plotted versus run time. These results demonstrates the advantages and disadvantages of attempting multiple connections per sample. In low dimensions, the main challenge of this planning problem is avoiding obstacles and BIT* can be outperformed by planners that attempt only a single connection per sample (e.g., Informed RRT*). In high dimensions, the exponential increase in planning problem measure (i.e., the curse of dimensionality) makes navigating towards the goal an equal challenge to avoiding obstacles and BIT* outperforms other planners, even SORRT*.

This effect is visible in the relative performance of planners across state dimension. In $\mathbb{R}^2$, BIT* performs roughly the same as Informed RRT* and SORRT*. It is faster to solve 100% of the planning trials (Fig. 5.5a) but improves the median solution cost slower than the two RRT*-based algorithms (Fig. 5.5d). In $\mathbb{R}^8$, BIT* drastically outperforms Informed RRT* and SORRT* in solving all planning trials (Fig. 5.5b) but is slightly outperformed by those two planners in terms of median solution cost (Fig. 5.5e). In $\mathbb{R}^{16}$, the difference is stark and BIT* is the only anytime almost-surely asymptotically optimal planner to always solve the planning problem in the given 300 seconds.

5.5.1.2 Random Worlds

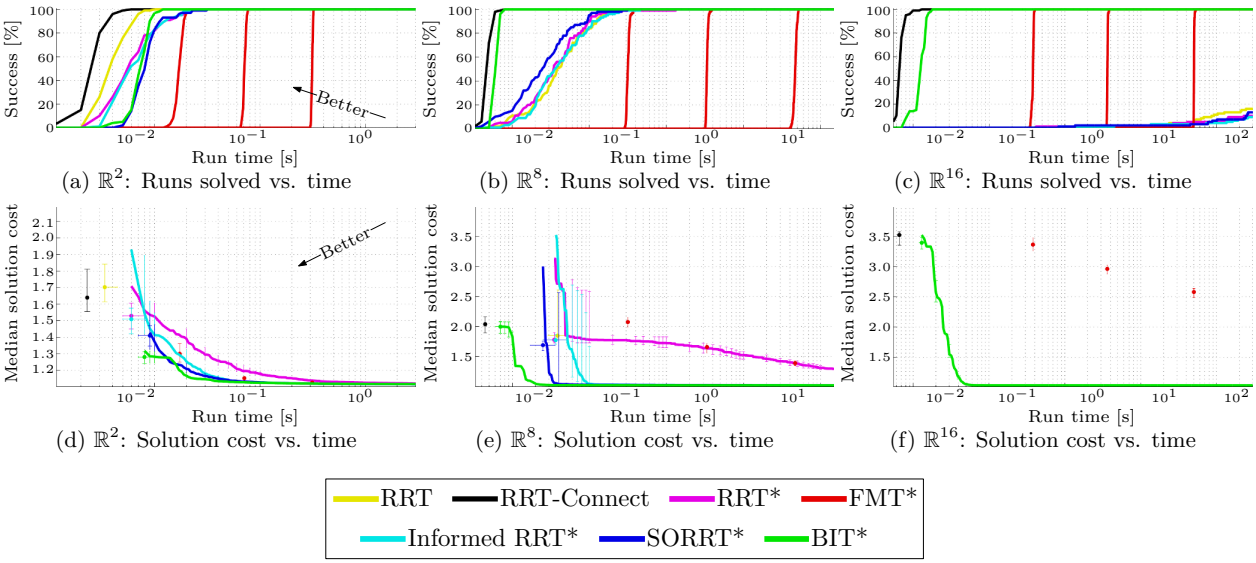
The planners were tested on randomly generated problems in $\mathbb{R}^2$, $\mathbb{R}^8$, and $\mathbb{R}^{16}$. The worlds consisted of a (hyper)cube of width $l = 2$ populated with approximately 75 random axis-aligned (hyper)rectangular obstacles that obstruct at most one third of the environment.



(a) $\mathbb{R}^2$: Runs solved vs. time (b) $\mathbb{R}^8$: Runs solved vs. time (c) $\mathbb{R}^{16}$: Runs solved vs. time

(d) $\mathbb{R}^2$: Solution cost vs. time (e) $\mathbb{R}^8$: Solution cost vs. time (f) $\mathbb{R}^{16}$: Solution cost vs. time

Figure 5.5: Planner performance versus time for the problem illustrated in Fig. 4.10b. Each planner was run 100 different times in $\mathbb{R}^2$, $\mathbb{R}^8$, and $\mathbb{R}^{16}$ with $l = 4$ and run times limited to 3, 150, and 300 seconds, respectively. The percentage of trials solved is plotted versus run time for each planner and presented in (a)–(c). The median path length is plotted versus run time for each planner and presented in (d)–(f), with unsuccessful trials assigned infinite cost. The error bars denote a nonparamentric 99% confidence interval on the median. The results show that BIT* increasingly outperforms other almost-surely asymptotically optimal planners as state dimension increases. Note the low success rates of RRT* planners in high dimensions.



(a) $\mathbb{R}^2$: Runs solved vs. time (b) $\mathbb{R}^8$: Runs solved vs. time (c) $\mathbb{R}^{16}$: Runs solved vs. time

(d) $\mathbb{R}^2$: Solution cost vs. time (e) $\mathbb{R}^8$: Solution cost vs. time (f) $\mathbb{R}^{16}$: Solution cost vs. time

Figure 5.6: Planner performance versus time for a randomly generated problem. Each planner was run 100 different times in $\mathbb{R}^2$, $\mathbb{R}^8$, and $\mathbb{R}^{16}$ with $l = 2$ and run times limited to 3, 150, and 300 seconds, respectively. The percentage of trials solved is plotted versus run time for each planner and presented in (a)–(c). The median path length is plotted versus run time for each planner and presented in (d)–(f), with unsuccessful trials assigned infinite cost. The error bars denote a nonparamentric 99% confidence interval on the median. The results show that BIT* increasingly outperforms other almost-surely asymptotically optimal planners as state dimension increases. Note the low success rates of RRT* planners in high dimensions.

For each state dimension, 10 different random worlds were generated and the planners were tested on each with 100 different pseudo-random seeds. The true optima for these problems are different and unknown and there is no meaningful way to compare the results across problems. Results from a representative problem are instead presented in Fig 5.6 with the percent of trials solved and the median solution cost plotted versus computational time.

These experiments show that BIT* generally finds better solutions faster than other sampling-based optimal planners and RRT on these types of problems regardless of the state dimension. It has a higher likelihood of having found a solution at a given computational time (Fig. 5.6a–c), and converges faster towards the optimum (Fig. 5.6d–f), with the relative improvement increasing with state dimension. The only tested planner that found solutions faster than BIT* was RRT-Connect, a nonanytime planner that cannot converge to the optimum.

5.5.2 Path Planning for HERB

It is difficult to capture the challenges of actual high-dimensional planning in abstract worlds. Two planning problems inspired by manipulation scenarios were created for HERB, a 14-DOF mobile manipulation platform (Srinivasa et al., 2012).

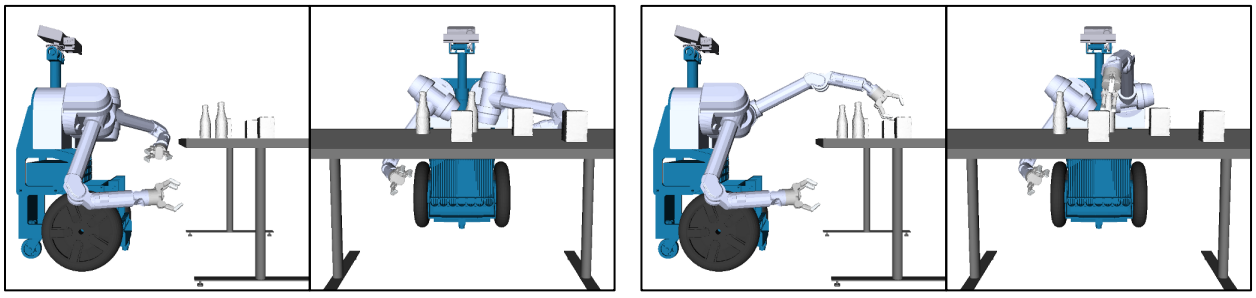
Start and goal poses were chosen for one arm (7 DOFs, Section 5.5.2.1) and two arms (14 DOFs, Section 5.5.2.2) to define planning problems with the objective of minimizing path length through configuration space. They were used to compare RRT, RRT-Connect, Informed RRT*, SORRT*, FMT*, and BIT*. The planners were run on each problem 50 times while recording success rate, initial solution time and cost, and final cost. Trials that did not find a solution were considered to have taken infinite time and have infinite path length, respectively, for the purpose of calculating medians. The number of FMT* samples for both problems was chosen to use the majority of the available computational time.

5.5.2.1 A One-Armed Planning Problem

A planning problem was defined for HERB's left arm around a cluttered table (Fig. 5.7). The arm starts folded at the elbow and held at approximately the level of the table (Figs. 5.7a and 5.7b) and must be moved into position to grasp a box (Figs. 5.7c and 5.7d). The planners were given 5 seconds of computational time to solve this 7-DOF problem with the objective of minimizing path length in configuration space. FMT* used $m = 30$ samples.

The percentage of trials that successfully found a solution (Fig. 5.8a), the median time and cost of the initial solution (Figs. 5.8b and 5.8c) and the final cost (Fig. 5.8d) were plotted for each planner. Infinite values are not plotted.

The results show BIT* finds solutions more often than other almost-sure asymptotically optimal planners on this problem (Fig. 5.8a) and also finds better solutions faster than all tested algorithms, including RRT-Connect (Figs. 5.8b–5.8d). Fig. 5.9 presents a composite photograph of HERB executing a path found by BIT* for a similar problem.



(a) (b) (c) (d)

Figure 5.7: A one-armed motion planning problem for HERB in $\mathbb{R}^7$. Starting at a position level with the table, (a) and (b), HERB's left arm must be moved in preparation for grasping a box on the far side of the table, (c) and (d).

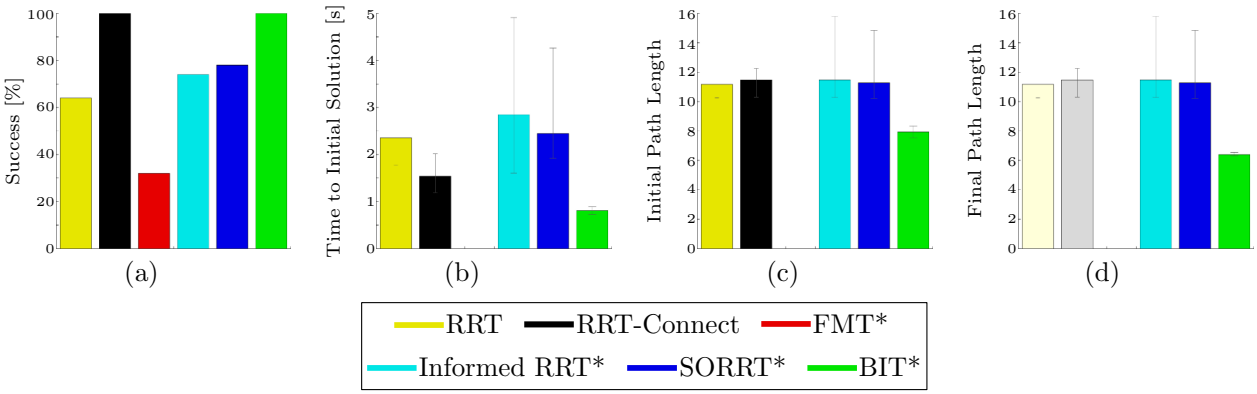


Figure 5.8: Results from 50, 5 second trials on the one-armed HERB planning problem shown in Fig. 5.7. The percent of solutions solved, (a), the median time to an initial solution, (b), the median initial path length, (c), and the median final path length, (d), are presented with 99% confidence intervals for each planner. Unsuccessful trials were assigned infinite time and cost. The inability of nonanytime planners (e.g., RRT, RRT-Connect, and FMT*) to use the remaining available time to improve their initial solution is denoted with diminished colour in (d), where present. BIT* is the only almost-surely asymptotically optimal planner to solve all 50 trials and does so in a time comparable to RRT-Connect. It also finds significantly lower cost paths then all the other planners.

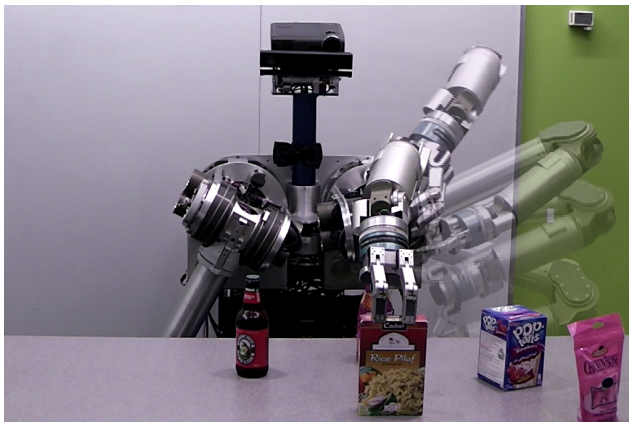
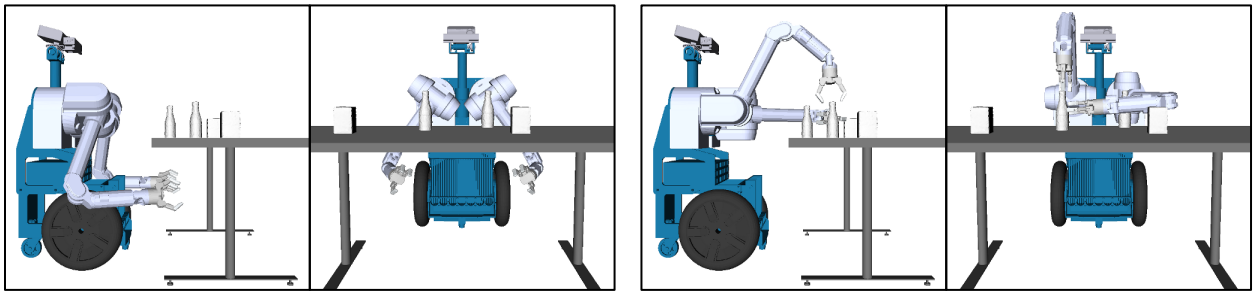


Figure 5.9: A composite figure of HERB executing a path found by BIT* on a one-armed planning problem similar to Fig. 5.7.



(a) (b) (c) (d)

Figure 5.10: A two-armed motion planning problem for HERB in $\mathbb{R}^{14}$. Starting under the table, (a) and (b), HERB's arms must be moved in preparation for opening a bottle, (c) and (d).

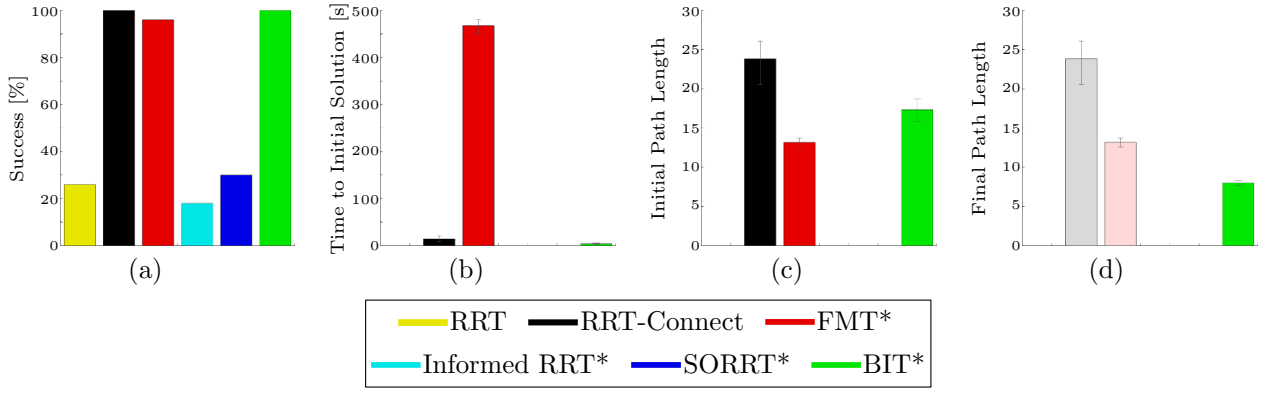


Figure 5.11: Results from 50, 600 second trials on the two-armed HERB planning problem shown in Fig. 5.10. The percent of solutions solved, (a), the median time to an initial solution, (b), the median initial path length, (c), and the median final path length, (d), are presented with 99% confidence intervals for each planner. Unsuccessful trials were assigned infinite time and cost. The inability of nonanytime planners (e.g., RRT, RRT-Connect, and FMT*) to use the remaining available time to improve their initial solution is denoted with diminished colour in (d), when present. BIT* is the only anytime planner to solve all 50 trials and does so in a time comparable to RRT-Connect. It focuses the search to the informed set once a solution is found and the resulting increasingly dense RGG allows it to find lower cost paths then all the other planners.

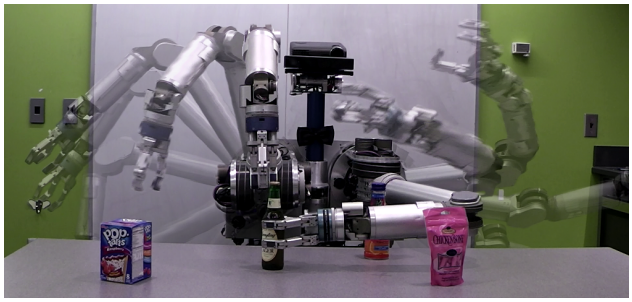


Figure 5.12: A composite figure of HERB executing a path found by BIT* on a two-armed planning problem similar to Fig. 5.10.

5.5.2.2 A Two-Armed Planning Problem

A second planning problem was defined for both of HERB's arms moving around a cluttered table (Fig. 5.10). The arms start at a neutral position with their forearms extended under the table (Figs. 5.10a and 5.10b) and must be moved into position to open a bottle (Figs. 5.10c and 5.10d). The planners were given 600 seconds of computational time to solve this 14-DOF problem with the objective of minimizing path length in configuration space. FMT* used $m = 3000$ samples.

The percentage of trials that successfully found a solution (Fig. 5.11a), the median time and cost of the initial solution (Figs. 5.11b and 5.11c) and the final cost (Fig. 5.11d) were plotted for each planner. Infinite values are not plotted.

The results show that even when more computational time is available, BIT* still finds solutions more often than other almost-surely asymptotically optimal planners (Fig. 5.11a) and also finds initial solutions faster than all tested algorithms, including RRT-Connect (Fig. 5.11b). As a nonanytime algorithm, FMT* is tuned to use the majority of the available time and finds a better initial solution (Fig. 5.11c) but as BIT* is able to improve its solution it still finds a better final path after approximately the same amount of time (Fig. 5.11d). Fig. 5.12 presents a composite photograph of HERB executing a path found by BIT* for a similar problem.

5.6 Discussion

Most anytime sampling-based planners, such as RRT*, solve planning problems by simultaneously searching the entire problem domain. This is inefficient and can become prohibitively expensive in large planning problems or high state dimensions. Informed graph-based search techniques, such as A*, avoid this unnecessary computational effort by ordering their search by potential solution quality. This only considers states when they could provide the best solution and avoids the computational costs of searching large regions of the problem domain. This solves planning problems by searching the informed set that will be defined by their eventual solution (the *future informed set*).

Existing methods to include heuristic ordering in sampling-based planning are insufficient (Section 5.1). They either partially apply heuristics (e.g., RRT#), sacrifice anytime resolution (e.g., RA* and FMT*), or waste computational effort on states that are unnecessary to find the solution by ordering the search on metrics other than solution cost and expanding outside the future informed set (e.g., SBA*).

BIT* is an informed anytime sampling-based planner that is ordered only by potential solution quality (Algs. 5.1–5.3; Section 5.2). It applies informed graph-based search techniques (i.e., LPA*) directly to continuous planning problems without requiring *a priori* approxi-

mations of the search domain. Instead, it alternately builds and searches a sampling-based approximation (i.e., a RGG) that increases in accuracy indefinitely with additional computational time. This avoids the computational costs associated with both choosing an insufficient approximation (graph-based search) and performing an unordered search (sampling-based planning). It is both probabilistically complete and almost-surely asymptotically optimal (Theorems 5.1 and 5.2) and can be stopped at any time if a suitable solution is found.

BIT* improves its approximation by adding batches of new samples into the existing RGG and uses incremental search techniques to update its search efficiently by reusing information. This is accomplished by using heuristics in all aspects of the search. Estimates of solution cost are used to order and focus the search, estimates of cost-to-come are used to account for future graph improvements, and estimates of edge costs are used to avoid unnecessary collision checks and boundary-value problems. Delayed edge cost calculations can also be found in *lazy* versions of both graph-based searches and sampling-based planners (Bohlin and Kavraki, 2000; Branicky et al., 2001; Cohen et al., 2014b; Hauser, 2015; Helmert, 2006).

A brief set of extensions to BIT* are presented Algs. 5.4–5.5 (Section 5.4). These include prioritizing an initial solution, avoiding the need to define *a priori* search limits in unbounded problems, and avoiding unreachable areas of the problem domain. These ideas also motivate the development of SORRT* as an extension of batch-ordered search to the algorithmic simplicity of RRT* (Alg. 5.6).

The benefits of BIT* are demonstrated experimentally on abstract planning problems and simulated experiments for HERB (Section 5.5). The results highlight the advantages of both using an ordered search and considering multiple connections per sample. BIT* is more likely to have found a solution at any given time on the tested problems and generally finds better solutions faster than the other almost-surely asymptotically optimal planners, especially in high state dimensions.

The experiments also highlight the relative sensitivity of existing anytime planners to tuning parameters. The performance of RRT-style planners depends heavily on the maximum edge length, η, and achieving the best performance requires tuning it for the problem size, dimension, and even obstacle characteristics. Alternatively, the performance of BIT* on the tested problems did not vary strongly with different choices for its main tuning parameter, batch size.

Avoiding unnecessary edge evaluations allows BIT* to spend more computational effort on edges. It is satisfying to note that this already being exploited in the literature. Xie et al. (2015) show that a two-point BVP solver can be used to calculate edges for BIT* for problems with differential constraints. They find that doing so is competitive to state-of-the-art optimal sampling-based techniques that are explicitly designed to avoid solving two-point BVPs. Choudhury et al. (2016) show that a path optimizer (i.e., CHOMP; Zucker et al., 2013) can be used on potential edges in BIT*. This provides a method to exploit local problem information (i.e., cost gradients) to propose higher-quality edges and improve performance.

BIT* has also been used in the literature as a part of multistage planning algorithms. Lan et al. (2016) use it in their two-stage online planning algorithm for micro UAVs. They convert the collision-free path found by BIT* into a dynamically feasible path by solving a series of two-point BVPs. This allows them to avoid the cost of considering differential constraints during the initial search.

This chapter shows the benefits of unifying informed graph-based search and sampling-based planning. Using incremental search techniques to efficiently search an increasingly dense RGG allows BIT* to outperform existing anytime almost-surely asymptotically optimal planners. These results will hopefully motivate further research into combining graph-based search and sampling-based planning. Of particular interest would be probabilistic statements about search efficiency analogous to the formal statements for A*. There is also a clear opportunity to consider different sampling-based approximations and to use more advanced graph-based-search techniques, such as ARA* and MHA*, to further accelerate the search performance of BIT*.

BIT* is described as using LPA* to efficiently search an incrementally built (i.e., changing) RGG embedded in a continuous planning problem. It is important to note a key difference in how these two algorithms reuse information. When LPA* updates the cost-to-come of a vertex it reconsiders the cost-to-come of all possibly descendent vertices. This can be prohibitively expensive in large graphs and the results of RRT* demonstrate that this propagation is unnecessary for a planner to almost-surely converge asymptotically to the optimum. By not propagating these changes, BIT* is more similar to TLPA*.

In summary, this chapter presents the following specific contributions:

- Reviews existing methods to combine informed search and sampling-based representations and shows that they either provide incomplete/inefficient ordering or sacrifice anytime performance (Section 5.1).
- Develops BIT* as a unification of informed graph-based search and sampling-based planning that is almost-surely asymptotically optimal (Algs. 5.1–5.3).
- Presents extensions to BIT* (Section 5.4), including JIT sampling to allow for the direct search of unbounded problems (Alg. 5.5) and SORRT* (Alg. 5.6) to apply ordered search concepts directly to RRT*.
- Demonstrates experimentally the benefits of using ordered search to combat the curse of dimensionality (Section 5.5).