Fast Planning Over Roadmaps via Selective Densification
Brad Saund and Dmitry Berenson

Abstract—We propose the Selective Densification method for fast motion planning through configuration space. We create a sequence of roadmaps by iteratively adding configurations. We organize these roadmaps into layers and add edges between identical configurations between layers. We find a path using best-first search, guided by our proposed estimate of remaining planning time. This estimate prefers to expand nodes closer to the goal and nodes on sparser layers.

We present proofs of the path quality and maximum depth of nodes expanded using our proposed graph and heuristic. We also present experiments comparing Selective Densification to bidirectional RRT-connect, as well as many graph search approaches. In difficult environments that require exploration on the dense layers we find Selective Densification finds solutions faster than all other approaches.

Index Terms—Motion and Path Planning

I. INTRODUCTION

We examine the motion planning problem, the task of finding a valid path through continuous configuration space from a start to a goal. Probabilistically complete [1] and asymptotically optimal [2] algorithms are known, but to be practical algorithms must be fast. The challenge for a search algorithm is to explore regions that will likely lead to a path, while not performing excessive checking of configurations that do not lead to the goal.

A common planning approach is to precompute a (probabilistic) roadmap (PRM), which is a graph where vertices represent configurations and edges represent motions. Planning can then be reduced to connecting the query start and goal to the graph and finding a path through the graph. In many robotics problems the obstacles are not known a priori, thus the validity of edges must be checked online. The computation time of this approach is dominated by node expansions of a graph-search algorithm, and collision checks for edges. Lazy edge evaluation has proved effective in robotics, with algorithms like LazyPRM [3] and the generalization Lazy Shorted Path (LazySP) [4] that minimize the number of collision checks.

A challenge when precomputing a roadmap is choosing the appropriate density of nodes in configuration space. Too few nodes and edges can result in a roadmap with no solutions.

Increasing the number of nodes in the roadmap increases the computation cost by increasing both the number of node expansions and number of collision-checks during a search. For many problems dense roadmaps are needed only in some regions while sparse roadmaps perform better in other regions with more free space. We desire a search method that is able to shift between graphs of different densities, achieving fast performance by searching sparser graphs in most regions and only densifying when required.

Our key contribution is the Selective Densification method for guiding the density of nodes and edges explored using a heuristic of remaining planning time. Specifically, this paper contributes:

- The Layered Graph: A sequence of roadmaps of increasing density combined into a single graph (Fig. 1)
- The Selective Densification Heuristic: An estimate of both remaining execution cost and planning cost used to guide best-first search
- Bidirectional search in LazySP with search direction during an iteration chosen by minimum planning time

We present proofs of the solution quality and maximum depth searched. To evaluate our method in practice, we perform motion planning experiments on a robotic manipulator arm in simulation and benchmark against existing methods. We find that in environments where paths exist on low-density graphs Selective Densification performs similarly to common
methods. In the most challenging environment tested, where paths exist only on high-density graphs, we find Selective Densification performs 4x faster than the next best approach.

II. RELATED WORK

A. Sampling-based C-space planning

In many robotics domains a graph is constructed to perform search over a continuous space. Sampling-based motion planning consists of building a graph with an embedding into C-space and then searching the graph for a valid (collision-free) path from a start to a goal.

While approaches such as RRT [1] build a tree online, other approaches such as PRM [5] can precompute a roadmap and perform online collision checking of edges. Both RRT and PRM seek feasible solutions in continuous space without explicit regard to path quality. Early asymptotically-optimal planners such as RRT* [2] find minimum cost solutions in the limit as \( t \to \infty \), with improvements such as BIT* [6] seeking faster convergence. Interestingly, in practice, repeated RRT with shortcut smoothing tends to outperform RRT* and variants both in time and path quality [7] [8]. Given these results, our approach focuses on finding solutions quickly and relies on the shortcut smoothing post-processing to reduce cost.

A’ [9] can be used to search roadmaps, however the edge collision-check is typically the most expensive operation in robotics applications, thus algorithms such as LazyPRM [3] and the later generalization LazySP [4] that minimize the number of edge evaluations typically run faster. In fact, LazySP is optimal w.r.t. the number of edge checks [10] and later work balances time on edge checking vs. expansion [11]. Using a precomputed roadmap offers the advantage that in static scenes edge validity is constant, so at most one collision check is required across multiple queries. However even in a single query setting using a precomputed roadmap offers two distinct advantages over a RRT: determinism and the ability to precompute environment-independent properties of edges. We employ both forms by generating a graph in the robot configuration space and precomputing the swept volumes occupied by the robot for some edges. This is done without knowledge of the specific environment and prior to query thus we consider this precomputation not time sensitive.

B. Densification Strategies

A challenge when constructing roadmaps is determining the number of vertices and edges needed. Selecting too few may yield a roadmap without a feasible solution or only a costly solution. Selecting too many yields a roadmap that is computationally expensive to store and search. Approaches such as SPARS [12], bridge sampling [13], and others therefore build explicit graphs online using specific vertex sampling and edge-connection methods that are aware of obstacles to limit the graph size. We pre-compute a much larger graph than these methods generate, but bias search towards the sparser portions.

Other approaches use multi-resolution roadmaps, connecting a low resolution roadmap to high resolution roadmaps in certain regions specified by heuristics [14] [15]. Planning with Adaptive Dimensionality uses a user-specified projection into a lower dimensional space, and reverts to the full dimensional space where the projection is inadequate [16] [17] [18]. For example, a projection may map a car to a 2D point or a robotic arm to the end effector location. This approach will only construct two graphs of different densities and relies on a user-defined projection that must obey certain properties.

Yet another approach is batching, where a fixed roadmap is searched completely before densifying [19], [20]. This has been used to first find solutions quickly on sparse layers then find better [21], and asymptotically optimal [22] solutions by searching denser layers. A drawback is the expense of the search of an entire layer before densifying, which can be especially large if no path exists. As suggested in the future work of LEMUR [23] it would be desirable to further densely promising regions before exploring the entire batch. Our work pursues this approach of selectively densifying certain regions by balancing the expected path length and computational cost at different batch levels.

C. Heuristic Graph Search

Finding a path in a roadmap requires searching a graph. While A’ finds optimal paths using an admissible heuristic [9], weighted A’ inflates the heuristic [24] achieving bounded sub-optimality (both with [25] and without [26] using a closed list). Methods such a ARA* [27], ANA* [28] and numerous others observe that an inflated heuristic and/or a greedy search tend to find solutions faster, trading path cost for planning time, although there is no such guarantee and in some cases this can slow down search [29]. A’ [24], BUGSY [30] and LEMUR [23] make this tradeoff explicitly by including various estimates of “planning-cost-to-go” in the search heuristic. We extend this idea to our Layered Graph, on each layer increasing the estimated future planning cost, and therefore inflation factor.

III. PROBLEM DEFINITION AND NOTATION

Let \( C \) be a configuration space with free space \( C_{free} \) and C-space obstacles \( C_{obs} = C \setminus C_{free} \). \( C_{obs} \) are represented implicitly via a collision-check function that given a \( q \in C \) tests whether \( q \in C_{free} \). A path \( \xi_{s \rightarrow g} \) from \( q_s \) to \( q_g \) is feasible if \( \xi_{s \rightarrow g} \subset C_{free} \). In practice we relax feasibility by assuming it is sufficient to collision-check a densely discretized \( \xi_{s \rightarrow g} \).

The execution cost of the straight-line motion between configurations is given by \( c_x : C \times C \to \mathbb{R}_{\geq 0} \), which for this work we assume is given by a distance: \( c_x(q_1, q_2) = ||q_1 - q_2|| \). The execution cost of a feasible discretized path \( \xi_{s \rightarrow g} = [q_1, ..., q_n] \) is given by

\[
c_x(\xi_{s \rightarrow g}) = \sum_{i=1}^{n-1} c_x(q_i, q_{i+1}) \tag{1}
\]

Furthermore, a planner incurs a cost \( c_p \) in computing a feasible path \( \xi \). We consider the case where \( c_p \) is the time spent to return \( \xi \) once \( q_s, q_g \) and \( C_{obs} \) are provided. As the primary objective we seek planners that find a feasible path as quickly as possible.
Define a graph \( G(V, E) \) with vertices \( V \) and edges \( E \). A roadmap is an embedding of \( G \) into \( C \) such that vertices map to configurations \( q \in C \) and edges map to C-space paths \( \xi_e \subset C \) connecting vertices. Let an \( r \)-disk graph be a graph with edges connecting exactly those vertices \( v_1, v_2 \) for which the corresponding configuration \( |q_1 - q_2| < r \).

An edge \( e \) is collision-free if and only if the corresponding path \( \xi_e \subset C_{free} \). As such, a feasible path through a graph from \( v_s \) to \( v_g \) consists of collision-free edges and induces a feasible C-space path \( \xi \) from \( q_s \) to \( q_g \). Each edge has an associated cost, and the cost of the full path is the sum of the edge costs.

For use in lazy edge evaluation, let \( \hat{G} \) denote the graph \( G \) initialized with the collision state of all edges labeled as "unknown".

### IV. Approach: Selective Densification

Our approach consists of finding a path on a graph with an embedding into C-space. The core innovation is the combination of specific graph structure with a heuristically guided search that balances execution cost and expected remaining planning cost. The graph has connected layers of different densities allowing the search to traverse to denser layers to navigate through tight spaces with precision and then return to sparser layers that are searched more quickly. The graph is searched using \( A^* \) with an inadmissible heuristic estimating both path cost and planning cost. As the true remaining planning cost is unknown, it is estimated via a heuristic that grows as the layer depth increases.

#### A. Graph Structure

Consider a sequence of unique configurations \( Q = (q_1, q_2, \ldots) \), and a strictly increasing sequence of positive integers \( (n_1 < n_2 < \ldots n_D) \). A layer \( L_i \) is an \( r \)-disk graph constructed from the first \( n_i \) configurations of \( Q \) and connection radius \( r_i \). Denote a vertex of \( L_i \) with \( v_{ij} \) where \( j \leq n_i \). For each pair of adjacent layers \( (L_i, L_{i+1}) \) define the inter-layer edges as \( E_{i+1} = \{e(v_{ij} \leftarrow v_{ij+1}) \forall j \leq n_i \} \). Note that \( v_{ij} \) and \( v_{ij+1} \) represent the same configuration \( q_j \), thus the inter-layer edges are zero-cost edges that change layers without changing configurations. The Layered Graph is defined as \( G = (\bigcup_i L_i) \cup (\bigcup_i E_{i+1}) \).

We define \( v.q \) as the configuration associated with vertex \( v \) and \( v.d \) as the layer number \( i \) where \( v \in L_i \).

Figure 1 visualizes two Layered Graphs in a 2D C-space, with the vertical dimension representing layer depth. Vertical edges are therefore the zero-cost edges connecting vertices in adjacent layers representing the same configuration.

Although this graph can be precomputed, a query may consist of a \( q_s \) and \( q_g \) that are not in \( G \), thus during a query vertices for \( q_s \) and \( q_g \) are added to each layer, edges within each \( L_i \) are added determined by the connection radius \( r_i \), and inter-layer edges are added for the vertices corresponding to \( q_s \) and \( q_g \).

#### B. Utility Guided Graph Search

To answer a query \((q_s, q_g)\) first the corresponding graph nodes \((v_s, v_g)\) are found (or added), a best-first \((A^*)\) search with lazy edge evaluation is performed over the graph yielding a path through \( \hat{G} \), and the induced C-space path is returned. Lazy edge evaluation attempts to reduce the number of edge collision-checks, but may require many calls to \( A^* \). This approach has been called LazyPRM [3] or LAZYSP [4].

Algorithms 1 or 2 and 3 show the outer loops that perform the lazy edge evaluation. Line 2 of Algorithm 1 performs a best-first search over \( G \), which is graph \( G \) with the optimistic assumption that unreviewed edges are collision-free. Edges along this optimistic path are collision-checked in order by Algorithm 3. The result of a collision check is stored so that future iterations of \( A^* \) may not traverse invalid edges.

Algorithm 2 implements bidirectional LAZYSP as bidirectional search tends to find solutions faster than unidirectional variants. Bidirectional LAZYSP alternates a unidirectional \( A^* \) search at each iteration reusing the results of collision checks and does not require the algorithmic machinery of bidirectional variants of \( A^* \) to maintain guarantees. A common implementation of bidirectional search alternates search direction at each iterations. We introduce this variant that balances the total time spent searching in each direction. If the search time per iteration is independent of direction, our version is equivalent to alternating direction after each iteration. However in practice we observe the \( A^* \) time is not negligible and can have a huge dependence on direction. In such cases we empirically observe significantly better performance when balancing total time instead of iterations.

#### Algorithm 1 Selective Densification Search:

```python
1: while True do
2: \( \xi \leftarrow A^* (\hat{G}, v_s, v_g) \)
3: if CheckEdges(\( \xi \)) then
4: return \( \xi \)
```

#### Algorithm 2 Bidirectional Selective Densification Search:

```python
1: \( t_{forward} \leftarrow 0 \)
2: \( t_{backward} \leftarrow 0 \)
3: while True do
4: if \( t_{forward} \leq t_{backward} \) then
5: \( \xi \leftarrow A^* (\hat{G}, v_s, v_g) \)
6: \( t_{forward} \leftarrow t_{forward} + \text{timeOf(Line 5)} \)
7: else
8: \( \xi \leftarrow A^* (\hat{G}, v_g, v_s).\text{reverse()} \)
9: \( t_{backward} \leftarrow t_{backward} + \text{timeOf(Line 8)} \)
10: if CheckEdges(\( \xi \)) then
11: return \( \xi \)
```

#### Algorithm 3 CheckEdges(\( \xi \)):

```python
1: for \( e \) in \( \xi \) do
2: if \( e \) is not valid then \( \triangleright \) collision check
3: mark \( e \) as invalid
4: return False
5: mark edge as valid
6: return True
```
Algorithm 4 $A^*$ ($\hat{G}, v_s, v_g$)

1: $open \leftarrow \{v_s\}$, $closed \leftarrow \emptyset$, $\hat{g}(v_s) \leftarrow \infty$, $\hat{g}(v) = 0$
2: while $open$ is not empty do
3:    $v \leftarrow$ open.pop_lowest_fvalue()
4:    insert $v$ into $closed$
5: if $v$ is $v_g$ then
6:    return ReconstructPath($v_s, v_g$)
7: for edge $e$ and successor $v'$ of $v$ do
8: if $e$ is invalid then skip
9: if $\hat{g}(v') > \hat{g}(v) + c_e(e)$ then
10:   $\hat{g}(v') = \hat{g}(v) + c_e(e)$
11: if $v' \notin closed$ then
12:   $\hat{f}(v') = \hat{g}(v') + \hat{h}_SD(v')$
13: insert $v'$ into $open$
14: return No Path Exists

Algorithm 4 is $A^*$ with invalid edges removed, and using the inadmissible heuristic $\hat{h}_SD$. The functions $g^*(v), h^*(v)$, and $f^*(v)$ for a node are the optimal cost-to-come, cost-to-go, and cost from the start to goal through $v$ respectively. $\hat{g}, \hat{h}_SD,$ and $\hat{f}$ are estimates of these quantities.

C. Heuristics

The heuristic $\hat{h}_SD$ contains both a heuristic estimate of the execution cost-to-go ($\hat{h}_x$) and the planning time-to-go ($\hat{h}_p$). We choose $\hat{h}_x$ as the euclidean distance to the goal. $\hat{h}_x$ is consistent (and therefore admissible) and the cost is only achieved if there is a straight line path from $v$ to the goal. $\hat{h}_p$ is proportional to distance $||v.q - g_q||$, the number of nodes in the layer $n_{v,d}$, and a tunable constant $w_t$.

$$\hat{h}_x(v) = ||v.q - g_q||$$

(2)

$$\hat{h}_p(v) = w_t n_{v,d} ||v.q - g_q||$$

(3)

$$\hat{h}_SD(v) = \hat{h}_x(v) + \hat{h}_p(v)$$

(4)

$$= \hat{h}_x(v)(1 + w_t n_{v,d})$$

(5)

We now discuss our choice of $\hat{h}_p$, with formal analysis provided in the next section. First consider that $\hat{h}_p$ is proportional to $||v.q - g_q||$. Assuming a maximum edge length, doubling the distance to the goal will double the lower bound on the number of node expansions required (approximately, due to the discretization of nodes). Mathematically, proportionality to $||v.q - g_q||$ causes $\hat{h}_p$ to appear as an inflation of $\hat{h}_x$, which has empirically shown to reduce planning times [24] [29].

Next, we discuss $\hat{h}_p(v) \propto n_{v,d}$. Consider search over a traditional roadmap composed of a single layer. We expect search time to increase for a denser roadmap. If an oracle guides a search to only expand nodes on the optimal path, we might expect search time to be inversely proportional to the maximal edge. However, consider the case of misleading cul-de-sacs, as shown in Fig. 3, where a best-first search must expand nodes in the entire volume of the cul-de-sac, thus the total time is proportional to the total number of nodes. We intentionally tailor $\hat{h}_p$ to these environments.

Finally, consider the constant $w_t$. Theoretically $w_t$ can appear as a weighting between a user’s preference between planning time and execution time. Clearly if a user is agnostic about planning time $w_t$ should be set to 0, as then $\hat{h}_SD$ reduces to $\hat{h}_x$, and thus $A^*$ will yield the optimal path on $\hat{G}$. On the other hand if the goal is simply to find a feasible path as quickly as possible, setting $w_t$ very high yields a greedy search over each layer, which empirically finds solutions quickly [28], [29]. However, the optimal $w_t$ might also depend on factors such as the expected path length, past planning experience, expected size of cul-de-sacs, and expected percentage of freespaces in $C$. In practice we observe that regardless of the value of $w_t$, shortcut smoothing tends to yield paths with similar cost.

Fig. 2 illustrates (unidirectional) Selective Densification search applied to a toy example. Because $\hat{h}_p \propto n_{v,d}$, $A^*$ tends to explore nodes in sparser layers first. The search first explores the sparsest edges in the top layer but neither is valid. The search progresses to nodes on denser layers closest to the goal, and then pops back up after navigating the narrow region. Although the graph extends deep, these edges are never explored, due to the higher planning heuristic of nodes on denser layers.

As an aside, suppose Selective Densification finds a path well before a robot can begin execution. An anytime method that converges to the optimal path could be created through the following modifications. Store the cost of the best path found so far $c_g(\xi_{Best})$ after Algorithm 2. Add the check $\hat{g}(v') < c_g(\xi_{Best})$ to Algorithm 4 Line 11 to prevent expansion of nodes that could not possibly lead to better paths. Repeat Algorithm 2 until the user requests a result, or until no nodes are on the open list. One could imagine a variety of schemes to decrease $w_t$, but choosing a particular scheme is beyond the scope of this work.

V. Analysis

In this section we prove bounds on the solution cost and search depth of Selective Densification. Because $\hat{G}$ contains nodes with different estimated planning times connected by zero-cost (vertical) edges, $\hat{h}_SD$ is not simply an inflated
consistent heuristic, however we build up similar guarantees relying on consistency across each layer of $G$.

For this section define $g_i^*$ as $g^*$ over the single-Layered Graph $L_i$. References to $A^*$ use the proposed heuristic Eq. (5), which is written as a per-layer inflation factor $\hat{h}_{SD}(v) = \epsilon_i \hat{h}_x(v)$ with $\epsilon_i = (1 + w_i n_i)$.

First, consider the $A^*$ search over a single layer ($G = L_i$).

Theorem V.1. Any $v$ expanded by $A^* (L_i, v_s, v_g)$ satisfies $\hat{g}(v) \leq \epsilon_i g^*(v)$.

Proof. On a single layer $A^*$ reduces to weighted-A* with $\hat{h}_{SD}$ equivalent to the consistent heuristic $\hat{h}_x$ inflated by $\epsilon_i$, thus we can directly apply the proof presented in [25], Theorem 10.

Next we show this per-layer bound holds on the full graph.

Theorem V.2. Consider an optimal path $\xi_i = (v_1^i = v_1, ..., v_k = v_k^i)$ on $L_i$. For every $v_j \in \xi_i$ expanded by $A^*$ over $G$ it holds $\hat{g}(v_j) \leq \epsilon_i g^*_i (v_j)$

Proof. Each iteration of $A^*$ expands the node with the lowest $\hat{f}$, thus it is sufficient to show that if a node $v_z \in \xi_i$ is on the open list with $\hat{g}(v_z) > \epsilon_i g^*_i (v_z)$ then there must be another node on the open list with lower $\hat{f}$-value. We show this by induction, by showing that if at some iteration Theorem V.2 holds for all $v \in \xi_i$ expanded so far implies it holds for the next iteration. The base case trivially holds, as initially no nodes have been expanded.

Consider some iteration of $A^*$ and assume Theorem V.2 holds for all $v \in \xi_i$ expanded so far. Consider some node $v_z \in \xi_i$ with $\hat{g}(v_z) > \epsilon_i g^*_i (v_z)$, thus with $\hat{f}(v_z) > \epsilon_i (g^*_i (v_z) + \hat{h}_x(v_z))$. Note that if no such $v_z$ exists, Theorem V.2 trivially holds for the next iteration. We show that $v_z$ will not be chosen for expansion.

Case 1: A node before $v_z$ along $\xi_i$ has been expanded. There then must be a $v_{m-1} \in \xi_i$ before $v_z$ along $\xi_i$ with successor $v_m \in \xi_i$ on the open list. By assumption $\hat{g}(v_{m-1}) \leq \epsilon_i g^*_i (v_{m-1})$. It follows that $\hat{g}(v_m) \leq \epsilon_i g^*_i (v_{m-1}) + c_x(v_{m-1}, v_m)$

$$\hat{g}(v_m) \leq \epsilon_i g^*_i (v_{m-1}) + c_x(v_{m-1}, v_m)$$

(6)

$$\leq \epsilon_i g^*_i (v_{m-1}) + c_x(v_{m-1}, v_m)$$

(7)

$$\leq \epsilon_i g^*_i (v_m)$$

(8)

We show $v_m$ has a lower $\hat{f}$-value than $v_z$ so the assumption will also hold in the next iteration.

For every node along $\xi_i$ it holds that

$$g^*_i (v_j) + \hat{h}_x(v_j) \leq g^*_i (v_{j+1}) + \hat{h}_x(v_{j+1})$$

(9)

$$= g^*_i (v_{j+1}) + \hat{h}_x(v_{j+1})$$

(10)

Eq. (9) is obtained using the consistency of $\hat{h}_x$ over $L_i$. Eq. (10) uses $g^*_i (v_{m-1}) + c_x(v_{m-1}, v_m) = g^*_i (v_{m+1})$ along the optimal path.

This is then used to related the $\hat{f}$-values of $v_m$ and $v_z$:

$$\hat{f}(v_m) = \hat{g}(v_m) + \epsilon_i \hat{h}_x(v_m)$$

(11)

$$\leq \epsilon_i (g^*_i (v_m) + \hat{h}_x(v_m))$$

(12)

$$\leq \epsilon_i (g^*_i (v_z) + \hat{h}_x(v_z))$$

(13)

$$< \hat{f}(v_z)$$

(14)

Thus $v_z$ will not be expanded in this iteration.

Case 2: No node before $v_z$ along $\xi_i$ has been expanded yet. Thus there must be some $v_i^j$ with $i' \leq i$ on the open list. That is, the open list will contain either the start node, or a node on a denser layer representing the same configuration as the start node. It is straightforward to see $f(v_i^j) < f(v_z)$ for every $i' \leq i$. Applying the same analysis from Case 1: $\hat{f}(v_i^j) < \hat{f}(v_z)$

Corollary 1. Any $v_i^j \in L_i$ expanded by $A^*$ on $G$ satisfies $\hat{g}(v) \leq \epsilon_i g^*_i (v)$.

Proof. Theorem V.2 directly proves this by setting $v_i^j$ as the end of $\xi_i$. If there is no feasible $\xi_i$ then $g^*_i (v) = \infty$ and this trivially holds.

As this bound on $\hat{g}$ holds for any node, it must hold for the goal node $v_g$. This bound on $\hat{g}$ therefore also applies to the path returned by $A^*$.

Corollary 2. Consider an optimal path $\xi_i^*$ on a $L_i$. Selective Densification over $G$ returns a path $\xi_{SD}$ with execution cost $c_x(\xi_{SD}) \leq c_x(\xi_{SD}) + 1 \cdot c_x(\xi_{SD})$ with $\epsilon_i = (1 + w_i n_i)$.

Proof. Selective Densification consists of a LazySP-style search (Algorithm 2) using $A^*$. $\hat{A}^*$ is performed over $\hat{G}$, the copy of $G$ that is optimistic about unevaluated edges. Due to this optimism, $\xi_{i}^*$ will always be valid on $\hat{G}$.

$A^*$ returns $\xi$, a path over $\hat{G}$ with edges that may not have been collision checked yet. On the final iteration Selective Densification validates all edges in $\hat{G}$, returning $\xi_{SD}$.

$$c_x(\xi_{SD}) \leq c_x(\xi_{SD}) = c_x(v_g) \leq \epsilon_i g^*_i (v_g) \leq \epsilon_i c_x(\xi_{SD})$$

(15)

This demonstrates that adding additional layers to $G$ can only improve the bound of the execution cost of the path.
returned by A*. In particular the larger inflation factor of the dense layers does not worsen the bound from the lower inflation factor of a sparse layer.

**Corollary 3.** With \( w_1 = 0 \) Selective Densification returns the optimal path.

**Proof.** \( w_1 = 0 \iff \epsilon = 1 \iff c_x(\xi) = c_x(\xi^*) \)

Next we show that when a path exists the \( \hat{h}_{SD} \) heuristic bounds the deepest layer of a search. We do this by showing that for a sufficiently deep layer \( \hat{h}_{SD} \) will be larger than any \( \hat{f} \) on the open list. To make this claim, we first define clearance. A path \( \xi \) has clearance \( \delta \) if and only if for every point \( p \) on \( \xi \) the configuration-space ball of radius \( \delta \) centered at \( p \) contains only valid configurations. We assume there exists a path with clearance \( \delta \).

**Theorem V.3.** Consider a Layered Graph \( \hat{G} \) that contains a feasible solution \( \hat{\xi}_i \) with clearance \( \delta \) on \( L_i \). The deepest node expanded by \( A^* \) is at most on \( L_j \) for a \( j \) with \( \epsilon_j \geq \epsilon_i \cdot c_x(\xi_i)/\delta \).

**Proof.** From Theorem V.2

\[
e_{\epsilon} \cdot c_x(\xi_i) \geq \epsilon_i \cdot c_x(\xi_j) \geq \hat{f}(v) \forall v \in \xi_i
\]

Define \( v_k \) as the first node expanded on layer \( L_{j+1} \).

Consider two cases:

If \( ||v_k - v_g|| \geq \delta \) then

\[
\hat{f}(v_k) \geq h_{SD}(v_k) \geq \epsilon_{j+1} ||v_k - v_g|| \geq \epsilon_{j+1} \delta \geq \epsilon \delta
\]

Consider a layer \( j \) sufficiently deeper than layer \( i \) such that \( \epsilon_j \geq \epsilon_i \cdot c_x(\xi_i)/\delta \). Then Eq. (20) yields \( \hat{f}(v_k) \geq \epsilon_j \cdot c_x(\xi_j) \).

Thus by Eq. (16) all nodes in \( \xi_i \) would be expanded before any node on layer \( L_{j+1} \).

Otherwise consider \( ||v_k - v_g|| < \delta \). Since \( v_k \) is the first node expanded on layer \( L_{j+1} \) the predecessor to \( v_k \) therefore must represent the same configuration on layer \( L_j \). However, with \( \delta < r_j \) then \( v_g \) is on the open list and will be expanded before \( v_k \). Since \( v_g \) has clearance \( \delta \) the edge would be valid. Selective Densification would then terminate without ever expanding \( v_k \).
Results are shown in Fig. 4. The deepest layer on which a collision check was performed using SD was 8 (Table), 9 (Bookshelf), and 15 (Slot).

We find our proposed method (SD and SD-pre) finds a feasible solution to our scenarios in a time comparable to (Table) or faster than (Slot) all other algorithms tested. The path length of the initial solution found by SD (and other algorithms) was substantially suboptimal, however a few seconds of shortcut smoothing dramatically reduced the path length in all trials for all methods. This suggests that in practice although SD finds paths far better than the bound provided in Theorem V.2, the solution found is far from optimal and will be sensitive to the quality of the smoother.

In Fig. 4 we set \( w_t \) (which governs the inflation of each layer of SD) to 1.0, causing a near-greedy search on deeper layers. We performed a sweep over \( w_t \) for a single \( G \), with results for the Table Scenario shown in Fig. 5. As expected, we found that smaller \( w_t \) result in slower search with shorter paths after the graph search.

We find that the largest component of planning time is the collision checking of edges, as expected. However, we find repeated A* search time is significant. As this LazySP approach repeatedly searches similar graphs, we attempted to reuse information from previous A* iterations using generalized LPA* [33] but found this approach slower.

Overall we observed precomputing the swept volume of edges significantly reduces collision checking time of valid edges, however only a modest decrease is seen when checking invalid edges, as we observe in our Scenarios that edges are invalidated typically after checking only a few configurations. Overall we observed precomputation yields a modest improvement in overall performance.

Finally, we compare a baseline bidirectional LazySP search to our proposal of balancing the time searching each direction (Algorithm 2). In the Table and Bookshelf Scenarios both approaches perform similarly. In the Slot Scenario the forward search expands far more nodes and edges than the reverse search. By balancing search time our proposed approach performs fewer forward search iterations before finding a solution using the reverse search (Table I).

VII. CONCLUSIONS AND FUTURE WORK

We proposed and implemented Selective Densification, a motion planning method that searches a graph composed of


