Learning Sequences of Approximations for Hierarchical Motion Planning

Martim Brandão, Ioannis Havoutis
Oxford Robotics Institute, University of Oxford, UK
{martim, ioannis}@robots.ox.ac.uk

Abstract

The process of designing hierarchical motion planners typically involves problem-specific intuition and implementations. This process is sub-optimal both in terms of solution space (amount of possibilities for search-space approximations, choice of planner parameters, etc) and amount of human labour. In this paper we show that the design of hierarchical motion planners does not have to be manual. We present a method for parameterizing and then optimizing sequences of problem approximations used in hierarchical motion planning. We define these as a specific kind of graph with intermediate state-spaces and solutions as nodes, and costs and planner parameters as edge properties. These properties become a continuous optimization variable that changes the sequence and parameters of sub-planners in the hierarchy. Using Pareto-front estimation, our method automatically discovers multiple designs of optimal computation-time/motion-cost trade-offs. We evaluate the method on a set of legged robot motion planning problems where hand-designed hierarchies are abundant. Our method discovers sequences of problem approximations which achieve similar—though slightly higher—performance than the best human-designed hierarchies. The performance gain significantly increases on new problems, yielding 12x faster computation times and 10% higher success rates.

Introduction

Motion planning problems can be sometimes be too challenging to solve within acceptable time-frames, especially for high-dimensional robots and highly constrained environments. Hierarchical planning offers a good solution to reducing the time it takes to solve such problems. The strategy of hierarchical planning is to solve approximate versions of a problem and use those solutions to constrain or guide the solution of the original problem. A sequence of multiple approximations can even be used, which are iteratively refined in finer and finer approximations until a solution to the original problem is obtained.

One issue with hierarchical planning is the need to think about, and manually define, what these sequences of approximations should be. Approximations are often hand-designed using human intuition and ingenuity, which arguably makes the process of algorithm development for a new robot or environment time-consuming and ad-hoc. There are also so many possibilities for search-space approximations, orderings of approximations in multi-stage planners, planner parameters at each stage etc., that only a very small subset of the solutions might be tested by the designers in practice.

The main idea of this paper is that we can discover such sequences of approximations automatically. Our strategy is to optimize the approximation sequences on a training set of planning problems. This involves automatically extracting approximations for a given planning state-space, and then optimizing the sequence of those approximations and planner parameters on a training set of planning problems.

The work in this paper is tightly related to concepts of "multi-model planning" (Styler and Simmons 2017) and "quotient space sequences" (Orthey, Escande, and Yoshida 2018). Compared to these, our contributions are:

1. We learn the *sequencing* of problem approximations to use on a hierarchical motion planner, and obtain each problem approximation automatically.
2. We learn a separate computation time budget *for each approximation*, in order to allow the effective use of anytime planners and reduce overall computation time.
3. We obtain *a set of* sequences of approximations, each with a different trade-off of computation-speed and motion-quality. This allows a user to easily prioritize one or the other at run-time according to task requirements.

Our work has connections to hierarchical reinforcement learning, where the goal is to learn good hierarchies for control (Botvinick and Weinstein 2014; Frans et al. 2017), and to neural network architecture search, where the goal is to learn the number of layers, connections, etc. (Liu et al. 2017; Zoph and Le 2016). The high-level idea, common to these approaches and ours, is to parameterize hierarchical topologies and then search or optimize over those parameterizations in a training step.

Code related to this paper is available at:
https://github.com/ori-drs/hierarchical-planning-sequences

Related work

Hierarchies in motion planning

Many robot-specific hierarchical motion planning algorithms have been proposed in the literature in recent years. For example, (Chestnutt and Kuffner 2004; Mastalli et al. 2015) use 2D cost maps obtained from a circular approximation of a biped/quadruped robot as a heuristic when computing footstep plans with A* search. The work of (Montemerlo et al. 2008) also uses a A* heuristic obtained from approximations of a car-planning problem. (Wermelinger et al. 2016) uses a cascade of planners for a legged robot, which attempts to solve the motion planning problem with several robot model approximations in a sequence: first a circumscribed circle approximation, then an inscribed circle and finally a box model. (Park, Pan, and Manocha 2014) uses an incremental optimization strategy to motion planning where the set of joints and links considered is increased at each stage. More general hierarchical motion-planning algorithms include path-velocity decompositions (Pham et al. 2017) where a collision-free path in configuration space is planned first, and only then a velocity component is planned within that path. Furthermore, the decomposition-based motion planning of (Brock and Kavraki 2001; Brock and Yang 2005) uses a wave-front expansion of spheres to estimate a workspace towards which a sampling-based planner is biased. Similarly, the work of (Plaku 2013) uses heuristic costs on a discrete decomposition of the workspace as a guide to kinodynamic motion planning.

Across all these examples, common design choices in hierarchical planners are the use of workspaces (such as sphere-tunnels (Brock and Kavraki 2001) and cell-decompositions (Plaku 2013)) or the use of only part of the planning state-space (e.g. position and not time-components of motion) to guide or constrain motion on the full space. At a high-level, both involve discarding part of the state-space as an approximation and solving a planning problem in this lower-dimensional space using approximate robot models. In this paper we also consider hierarchies which are built this way. However, we allow an optimizer to find the best possible subset of the state-space dimensions to discard, and the order in which to do multiple approximate stages, during an offline training procedure.

Our work is closely related to that of (Styler and Simmons 2017), which considers multiple approximations of a problem in a sequence encoded by a single-source single-sink acyclic graph (ST-DAG). In our paper we also use an ST-DAG representation of approximation sequences. However, we augment the representation with per-edge costs and planner parameters (a computation time budget) which we then learn in order to obtain faster and more optimal motion plans. Additionally, (Styler and Simmons 2017) use transitive versions of the original graphs, which we show in this paper to perform poorly in terms of computation time. Another related effort is (Orthey, Escande, and Yoshida 2018), which solves motion planning problems through sampling on multiple quotient spaces. Our method also uses quotient spaces but again learns the most effective sequence to use at planning time—while (Orthey, Escande, and Yoshida 2018) samples from all approximate spaces at planning time.

Hierarchies in other fields of study

Also relevant work is that on hierarchical mechanisms of human decision-making (Balaguer et al. 2016; Huys et al. 2015) and the intersection of human action control with reinforcement learning (Botvinick and Weinstein 2014). These studies show not only that humans plan in hierarchical ways, but they try and investigate which hierarchies are actually used in practice. (Solway et al. 2014) interestingly defines a methodology based on Bayesian model selection for the computation of optimal hierarchies given a planning task, and shows how humans naturally discover and use such hierarchies when solving the same tasks. In practice these hierarchies involve planning on lower-dimensional spaces of "options" or sub-goals, which are nodes of high centrality (e.g. doors and other bottlenecks in the planning space).

Computational work in hierarchical reinforcement learning has also introduced new methods to discover sub-goals and other hierarchical representations which involve planning over a smaller set of graph nodes (McGovern and Barto 2001) before finding paths between these nodes. More recent work learns hierarchies as general motor-primitives that are used to control the motion of complex ant and human-like agents, and which are selected by "master" actions of a lower-dimensional space on shorter planning horizons (Frans et al. 2017). These examples involve more complex abstractions of state-spaces than those considered in this paper—where we build approximate state-spaces by discarding part of the full-space dimensions. However, such reinforcement learning work is also in its infancy compared to the search-based planning methods we consider, which are straightforwardly applicable to real robots.

Architecture search and optimization of hierarchies

Another body of literature relevant to this paper is that of architecture search in neural networks. This is the process of optimizing the topology of neural networks, such as the number of layers, their connections and even activation functions. Early work includes that of (Maniezzo 1994) which uses evolutionary algorithms to optimize the topology and weight distribution of neural networks. More recently, architecture search has been applied to deep neural networks using evolution (Liu et al. 2017), reinforcement learning (Zoph and Le 2016) and continuous optimization (Liu, Simonyan, and Yang 2018).

New developments in evolutionary methods (Deb et al. 2002) and the availability of open-source implementations of many of these algorithms (Fortin et al. 2012) further motivates the use of evolution for architecture search, and in our case the discovery of complex sequences-of-approximations for hierarchical motion planning. In this paper we focus on multi-objective optimization since motion planning algorithms will usually have to trade-off total execution time and final motion cost. Our proposal here uses recent work in Pareto-curve estimation using evolutionary methods (Deb et al. 2002; Zitzler, Laumanns, and Thiele 2001) to provide a human decision-maker with a set of possible sequences-of-approximations with optimal time-cost trade-offs. Examples of the use of Pareto-optimality in motion planning include the work of (Lavin 2015) and (Choudhury, Dellin, and Srinivasa 2016), although they focus on obtaining Pareto-optimal motion and we focus on obtaining *algorithms* that optimally trade-off motion cost and computation time.

Problem

We consider the motion planning problem of obtaining a sequence of feasible states in a state-space of interest S. We further consider the planning algorithm to be hierarchical, which in this paper means it solves a sequence of increasingly complex approximate problems until finding a solution in S. Multiple combinations for the sequencing exist, so the planner actually tries to solve a problem with a sequence of approximations $\mathcal{S}_1$ first, but in case of failure it switches to sequence $\mathcal{S}_2$, etc. until all available sequences of approximations are exhausted. The sequence $\mathcal{S}_k = \{\hat{S}^k_1, ..., \hat{S}^k_{P_k}\}$ is of increasingly larger subspaces, i.e. $\hat{S}^k_1 \subset \hat{S}^k_2 \subset ... \subset \hat{S}^k_{P_k}$, and the last state-space $\hat{S}^k_{P_k} = S$. When solving an approximate problem on subspace $\hat{S}^k_i$, the planner uses the solution obtained on $\hat{S}^k_{i-1}$ to guide the search. We assume the motion planner used on each (approximate) problem to be anytime and optimal. Therefore, the planner may use a separate computation time budget for each $\hat{S}^k_i$. Our goal is to automatically obtain the sequences $\mathcal{S}_1, \mathcal{S}_2, ...$ in a way that leads to optimal (total) computation times, and optimal cost of the solution in S.

Method

Representing hierarchical planning as a graph

Our method relies on a two-terminal directed acyclic graph (ST-DAG) which encodes the sequencing of approximations. See Figure 1 for a visual explanation. Each node V_i in this graph G represents an approximation (i.e. a subspace $\hat{S}_i$ where i is the index of the node here) and each edge E_{ij} represents a pair of successive approximations. A subspace is further associated with a state-validity model M_i (e.g. bounds and a collision geometry to use for checking state validity). We parametrize the order in which the approximations are used by assigning costs C_{ij} to each edge in the graph. Solving the least-cost sequence of edges from the source to the sink node on this graph provides the sequence of approximations $\mathcal{S}_1$. If any of the approximate problems is not solved successfully, then the cost of the associated edge is changed to ∞ and the least-cost sequence of edges will reveal $\mathcal{S}_2$. This can be done repeatedly until the sink node is reached (i.e. a solution in S is found), or until all sequences are exhausted. In addition to assigning costs to each edge, we also assign a computation-time budget to each edge. This represents the maximum amount of time that can be spent solving an approximate problem, and it will allow to make effective use of anytime planners.

In more detail, we use the pseudo-code in Algorithm 1 to solve a hierarchical motion planning problem. The algorithm traverses the graph starting at the source node. At each iteration it obtains the next node to visit by computing the lowest-cost sequence from source to sink using the Dijkstra algorithm. The first unvisited node along this sequence represents the next approximate problem to solve. Thus, when we visit a node V_j by an edge E_{ij} we obtain the state-space $\hat{S}_j$, state-validity model M_j, computation-time budget B_{ij} and previous solution T_i (obtained on a previous plan in subspace $\hat{S}_i$). We project the start and goal states of the original problem into $\hat{S}_j$, and use an off-the-shelf motion planner on $\hat{S}_j$ to solve the approximate problem—using M_j as a state-validity checker and T_i as a solution guide (the guiding method is described later). If a solution is not found within the time budget B_{ij}, we assign infinite cost to C_{ij} to avoid the future use of this approximation. If a solution is found, we store it as T_j inside node V_j. When a solution to the original problem is obtained (i.e. the sink node is reached), or all possible paths to reach the sink node are exhausted, the algorithm discards changes to graph edge costs and returns the final trajectory or a failure code.

Algorithm 1 Motion planning with a sequence of approximations encoded by a graph G

```
Require: graph $G(V, E, C, B)$; start state $s$ and goal state
    $g$ of the original motion planning problem
 1: $E_{ij}$ ← GetNextApproximation($G$)        // next edge in G
 2: $T_i$ ← GetGuideTrajectory($V_i$)           // stored in $V_i$
 3: $M_j$ ← GetStateValidityFunction($V_j$)     // stored in $V_j$
 4: $\hat{S}_j$ ← GetSubSpace($V_j$)            // stored in $V_j$
 5: $\hat{s}$ ← ProjectToSubSpace($s$, $\hat{S}_j$)
 6: $\hat{g}$ ← ProjectToSubSpace($g$, $\hat{S}_j$)
 7: $B_{ij}$ ← GetCompTimeBudget($E_{ij}$)      // stored in $E_{ij}$
 8: $T_j$ ← SolveMotionPlanning($\hat{s}$, $\hat{g}$, $\hat{S}_j$, $M_j$, $T_i$, $B_{ij}$)
 9: if $T_j$ not empty then
10:     if IsSinkNode($V_j$,$G$) then
11:         return: $T_j$
12:     SetGuideTrajectory($V_j$, $T_j$)
13: else
14:     $C_{ij} \leftarrow \infty$
15: if SourceToSinkPathExists($G$) then
16:     Go to 1
17: else
18:     return: failure
```

Obtaining the graph structure (V, E)

We assume the state-space of interest S is a Cartesian product of factors

$$S = S_1 \times S_2 \times ... \times S_N, \qquad (1)$$

where each factor S_k is a state-space such as a position in $\mathbb{R}^3$, an orientation $SO(2)$, a set of joint angles, a mode, etc. We then obtain a list of lower-dimensional subspaces by computing all N-combinations of S_k including the empty set (i.e. we compute the power set of $S_1, ..., S_N$ and apply the Cartesian product to each combination). Each combination will be a new state-space $\hat{S}_i$ that will lack zero or more

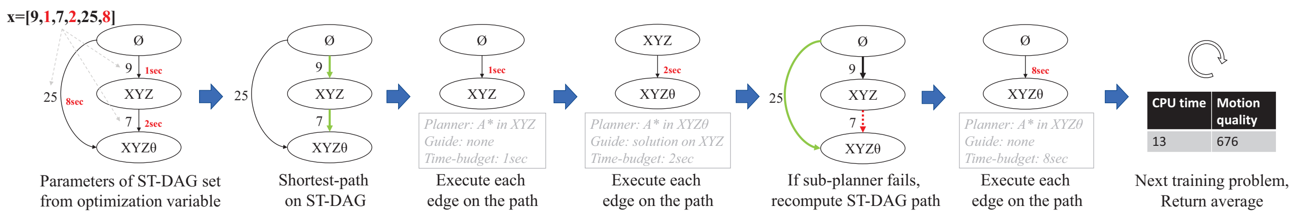


Figure 1: How a parameterized hierarchy is evaluated within the optimizer. In this example the edge costs are such that the minimum cost to the sink of the graph is through the sequence ∅–XYZ–XYZθ: an initial plan in 3D position space only, followed by a plan in the space of interest which uses the previous solution as a guiding heuristic cost-to-goal. In case the plan fails, the next-lowest cost to the sink is ∅–XYZθ: a direct plan on the full space, allowing a larger computation time budget.

of the factors in S, and so $|\hat{S}_i| \leq |S|, \forall i$.

To build the structure of graph G, we add an edge E_{ij} between nodes i and j if i is a subspace of j (i.e. all the state-space factors of i are present in j). The sink of the graph represents the original problem, associated with state-space S. The source of the graph is a dummy node with an edge to all other nodes, thus allowing the learned sequence of approximations to start from any of the available state-spaces.

Optimizing the sequence of approximations (C,B)

The graph $G(V, E, C, B)$ imposes a sequencing of approximations to use in hierarchical motion planning. With this representation in place, our objective is to learn the values of the edge-costs C_{ij} and time-budgets B_{ij} that lead to the most effective sequences-of-approximations.

To optimize (C, B) we use evolutionary optimization. In short, this consists of picking random values for (C, B), running Algorithm 1 on the training-set motion planning problems with each value of (C, B), scoring the results, obtaining new values for (C, B) with an evolutionary strategy (Deb et al. 2002), and repeating the process. Since we are interested both in good solutions to the original problem, and in finding them quickly, we score each (C, B) based on these two criteria:

- f_{cost}: average, over the training set, of the quality of the motion planning solutions found by Algorithm 1. This is given by the problem-specific cost of the motion returned by Algorithm 1 (i.e. this is a motion planning cost such as distance or energy, not the graph-edge costs C).
- f_{time}: average, over the training set, of the time spent on Algorithm 1.

Since f_{cost} can depend on the length of the motion, we set it equal to the cost of the motion planning solution, normalized by straight-line distance between start and goal states.

We use the NSGA-II algorithm (Deb et al. 2002) for the evolutionary optimization strategy, which estimates the Pareto-front of the two objectives, i.e. a curve of optimal trade-off between f_{cost} and f_{time}. As we will see, this generates multiple solutions to the sequence-of-approximations, which can be picked at run-time according to the time and optimality requirements of the task at hand.

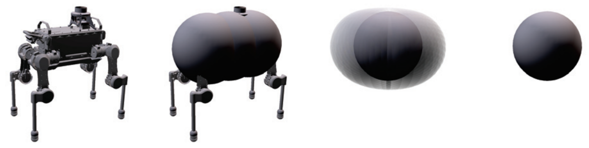


Figure 2: Obtaining the collision-checking geometry of a subspace XYZ which discards yaw rotations from the full space XYZθ. Left to right: the robot, the collision geometry in the original state-space, geometries of densely sampled states (varying yaw), intersection of all geometries. The collision-checking geometry used for this subspace is a sphere.

Automatically obtaining state validity functions

To obtain state-validity functions automatically for any subspace we assume the original function $M : S \rightarrow \{0, 1\}$ consists only of state-space bounds and geometric collision checking. The function takes on value 1 when the state is valid and 0 otherwise. For any given subspace $\hat{S}_i$ we automatically obtain the state validity function $\hat{M}_i$ in the following way.

- The bounds of $\hat{S}_i$ are taken directly from S since all of the state-space factors of $\hat{S}_i$ are also present in S.
- The collision-checking geometry is obtained by computing an admissible approximation. First, we obtain K samples $s_1, ..., s_K$, where each $s_k = (s_{k1}, s_{k2}) \in S$, $s_{k1} = (0, ..., 0) \in \hat{S}_i$ and $s_{k2} = \text{UniformSample}(S/\hat{S}_i)$. In other words, we randomly sample the dimensions which are not part of the subspace $\hat{S}_i$. Then we intersect all the geometries $\mathcal{G}(s_1) \cap ... \cap \mathcal{G}(s_k)$ to obtain the collision geometry for subspace $\hat{S}_i$. See Figure 2 for an illustration.

Using previous solutions to guide motion planning

The way we use lower-dimensional plans to guide the search on higher-dimensional plans is through a heuristic cost-to-goal and follows the work of (Brandao et al. 2015). The heuristic cost-to-goal of a state s is $h = \underline{c}.d(s, g)$, where $d(.)$ is a distance function, g is the goal state, and $\underline{c}$ is a lower bound on the cost-per-distance. We obtain this lower bound by pre-computing $\underline{c} = \min_{j,k} c(s_j, s_k)/d(s_j, s_k)$

where s_j, s_k are neighbor robot states and $c(.)$ is the state-transition cost function.

In the absence of a trajectory guide (i.e. for nodes V_i that are children of the source node of G), we use the Euclidean distance $d(s,g) = ||s-g||$. In other cases where a trajectory guide T_i does exist, we compute an improved estimate of the distance-to-goal from T_i. In particular, we compute the distance-to-goal of a state s as the sum of 1) the Euclidean distance from s to T_i (after projecting s to $\hat{S}_i$), and 2) the total Euclidean distance traveled along the guide-trajectory from the projection of s until the goal g.

Summary

To conclude, we solve motion planning problems with a learned sequence of problem approximations by:

1. Building a graph structure V, E where each node represents a different combination of state-space factors (see above section "Obtaining the graph structure").
2. Optimizing edge costs C and planner time-budgets B in a training set of motion planning problems (see above section "Optimizing sequences of approximations").
3. Picking one of the Pareto-optimal values of (C, B) obtained above, according to the desired computation-time/path-optimality trade-off.
4. Using Algorithm 1 with the above values of V, E, C, B to solve new motion planning problems.

Experimental evaluation

Implementation and setup details

In all our experiments we use ARA* (Likhachev, Gordon, and Thrun 2003) from the SBPL library (Likhachev 2010) as our anytime optimal motion planner. We use a 2.5D representation of the environment for collision and state-cost computations, as implemented in the GridMap library (Fankhauser and Hutter 2016). For the optimization we use the Pareto-front estimation algorithm NSGA-II (Deb et al. 2002) as implemented in the DEAP Python library (Fortin et al. 2012). We make use of its integration with the SCOOP Python library (Hold-Geoffroy, Gagnon, and Parizeau 2014) for parallelization. For fast training and prototyping we used a high performance computing system with 64 cores.

Hierarchical planning for ANYmal

Our first experiment is for motion planning of a legged robot, ANYmal, within an industrial site with obstacles (walls) and stairs, as shown in Fig. 5. We obtained the map through laser scans on-site.

The goal is to find the optimal sequences-of-approximations for obtaining trajectories in the state-space

$$S = \mathbb{R}^3 \times SO(2) \times SO(2) \times \{0, 1, 2\}, \qquad (2)$$

which corresponds to planning an XYZ position for the base, a pitch and a yaw angle, and a locomotion mode. The modes represent different controllers which can be used on the robot and are specialized for specific terrain types. The choice of locomotion mode influences state-costs: mode 1 is preferred on flat ground, mode 0 on slightly rough terrain, and mode 2 on stairs and very rough terrain. The actual definition of the cost functions is the same as the work by (Brandao, Fallon, and Havoutis 2019). The robot model used for collision-checking is shown in Figure 2.

We optimized (C, B) on a set of 6 random start-goal configurations on the environment shown in Figure 5. The poses were obtained by uniform sampling of position and yaw coordinates within an area of the map rich in walls, stairs and narrow passages. We compare our algorithm to several baselines: 1) a non-hierarchical planner (planning directly in S); 2) common serial-hierarchy planners using a single intermediate subspace solution to guide planning (we consider an XYZ sub-plan (Chestnutt and Kuffner 2004), an XYZ-yaw plan and an XYZ-mode plan); and 3) the hierarchical representation of (Styler and Simmons 2017) which consists of a fixed-sequencing version of our method (breadth-first-search) and a transitive version of the graph to reduce the number of edges—we will call this method "BFStransitive". The graph representation of our hierarchy and the baselines is shown in Fig. 3. Training took roughly 5 hours for our method and 2 for the baselines on a 64-core machine, with 20 generations of 20 individuals and 100 children.

The final Pareto-front of the cost-time objectives is shown in Fig. 4. The figure shows that our sequences of approximations obtained similar motion costs and computation times to the best-performing serial-hierarchies. Our method can also decrease computation times considerably compared to a transitive version of the graph with fixed sequencing (Styler and Simmons 2017). The no-hierarchy baseline could not solve all problems within the time budget and so its Pareto-front is not shown in the figure (it would be a cost/time average over the "easy" problems only, hence lower cost but not comparable). The Pareto front shows a clear gain in motion-cost when computation times are increased to around 20s, even though the trade-off becomes quite expensive after this point.

Analysing the particular solutions of the Pareto-set shows that each point in Fig. 4 corresponds to a different sequencing of the approximations. This can be seen by tracing the actual executions of the planner in different problems, as shown in Fig. 5 for the extremes of the Pareto set, i.e. the lowest-cost solution and lowest-computation-time solution. The figure shows that the speed-focused sequence first attempts to quickly find solutions directly in the full-space, which provides fast-and-low-cost solutions in no-obstacle problems ("fast hierarchy" graphs). On failure, the planner will use an XYZ-fullspace hierarchy. This is therefore an extension to the "XYZ-hierarchy" baseline and explains the improvement in total computation time. The cost-focused sequence, on the other hand, uses planning on lower-dimensional spaces which include the mode component of the state. These are easier to compute than full-space plans but provide better estimates of cost than position-only guides. The figure also shows the final motion that corresponds to the graph executions, and shows that cost-specialized sequences are qualitatively smoother on turns than fast-hierarchies. The length of use of each mode is also slightly different: there is one more section

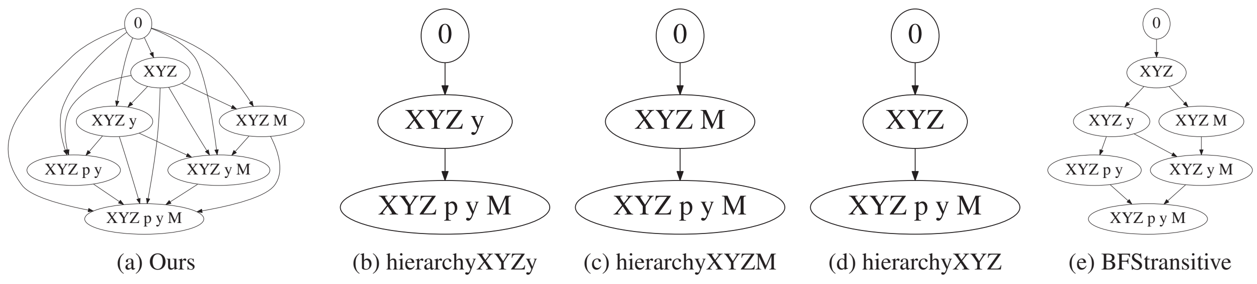


Figure 3: The different hierarchies we evaluate on the ANYmal experiment. Note that only "ours" optimizes the sequencing of approximations (induced by per-edge costs).

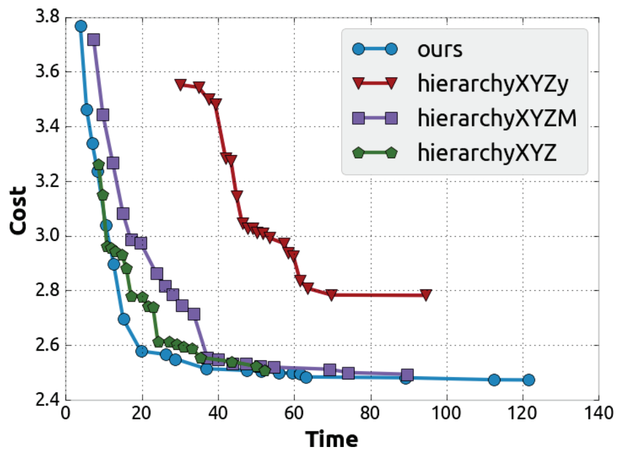


Figure 4: Pareto fronts of total-computation-time and final-motion-cost (ANYmal experiment).

Table 1: Performance on test set (ANYmal experiment)

Hierarchy	Success rate*	Avg. time (s)*	Avg. cost	Worst-case success rate
ours	1.00	11.46	3.43	0.90
XYZy	1.00	94.50	2.86	0.90
XYZM	1.00	69.23	2.74	0.90
XYZ	1.00	33.15	3.00	0.90
fullspace	0.80	13.33	1.97	0.70
BFStransitive	1.00	25.03	2.73	0.90

*best of Pareto set (by success and time in that order)

where mode 2 is used close to the start of problem 1, because proximity to obstacles is better tackled by this mode, and the same mode is less used towards the goal because of the absence of obstacles.

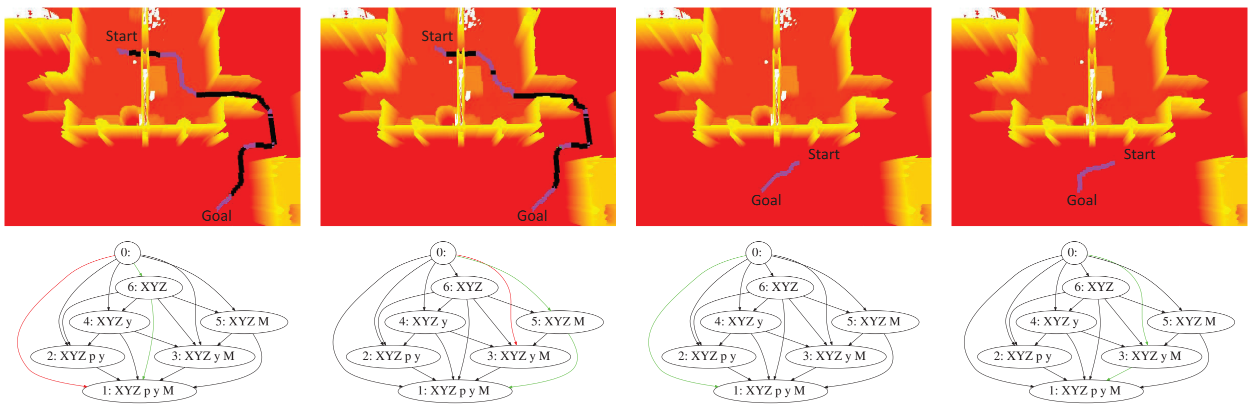
Finally, we solved a new set of 10 random problems (generated from a different random seed) using all the optimized sequences-of-approximations (all points of all Pareto-fronts). We compare the best performing solution of each in Table 1, where performance was obtained by sorting all results of a sequence by best success rate and then by best computation time. The table shows that our optimized sequences-of-approximations obtain considerably faster results of 3x to 9x for those with 100% success rate. This shows a better generalization and transfer power than other methods as the gap of performance grew larger compared to the training set. The table also shows the worst-case success rates (minimum over all solutions of the Pareto set) which are the same for all hierarchies (90%).

Comparison with a cascade-planner

In another experiment we compare the performance of our optimized sequences-of-approximations to that of carefully and manually designed hierarchies. We use the example of the cascade planner proposed for the StarlETH robot in (Wermelinger et al. 2016). In that paper, the authors propose a hierarchical planner that plans motion in a sequence: first using a circumscribed-cylinder model, then an inscribed-cylinder model, and finally a bounding-box model of the robot. If one planner on the sequence fails to find a solution within a time budget, the next planner is used. We mimic this behavior using the collision checking geometries of Fig. 6 and by fixing the ST-DAG edge costs so that the planning order just described is fulfilled.

We set the motion planning state-transition cost to Euclidean distance. Our comparison is generous since we optimize the computation time budgets given to each of the plans—even though this is done manually in (Wermelinger et al. 2016) and in other approaches. We compare fully optimized sequences-of-approximations (optimizing per-edge time budgets and costs) to several baselines: 1) "cascadeOptTime" has fixed costs but optimizes a single computation budget which is applied to all edges, and 2) "cascadeOptTime2" has fixed costs but optimizes per-state-space computation budgets (one variable which applies to all XYZ spaces and another to XYZ-yaw). The latter is the strategy used in (Wermelinger et al. 2016). We also evaluate the performance of more traditional methods: 3) directly planning on the full space, 4) serial-hierarchy with an intermediate sub-plan on XYZ using an inscribed-cylinder model. The optimization vectors were of size 1, 2, 1 and 2 respectively for each baseline, and 10 for our hierarchy. Training took roughly 2.5 hours for each hierarchy on a 64-core machine, with 20 generations of 20 individuals and 100 children.

We show the results of the optimization in Fig. 7. The figure shows that our optimized sequences-of-approximations achieve similar total computational times but slightly lower final motion costs (by 2%) than the baseline hierarchies,



(a) Problem 1, fast-sequence (b) Problem 1, cost-sequence (c) Problem 3, fast-sequence (d) Problem 3, cost-sequence

Figure 5: The execution of our hierarchical planner in two problems and at two cost/time trade-offs (ANYmal). Red graph edges indicate failed plans, green are successful. Fast-sequence and cost-sequence are the two extremes of the Pareto set. Robot trajectories on top row are black for mode 2 (rough terrain preference) and purple for mode 1 (flat ground preference).

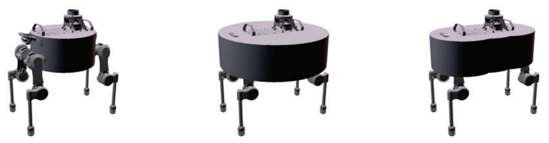


Figure 6: Collision-checking geometries used in the cascade-planner experiment. Left to right: inscribed-cylinder model used in XYZ space, circumscribed-cylinder model used in XYZ space, full model for XYZ-yaw. The full robot model is shown underneath for clarity.

even if these were partly optimized too. More importantly, the hierarchical designs of the baselines are the result of a careful manual design and tuning process (Chestnutt and Kuffner 2004; Wermelinger et al. 2016), which in our approach is done automatically—both the decision of computation time budgets at each planning stage and the sequencing of the approximations used themselves.

Analysing the particular solutions of the Pareto-set shows that each point along the front in Fig. 7 corresponds to a different sequencing of approximations. This can be seen by tracing the actual executions of the planner in different problems, as shown in Fig. 8 for the extremes of the Pareto set. The figure shows that the speed-focused sequence first attempts to quickly find solutions directly in the full-space. On failure, a plan with a circumscribed cylinder model (marked as XYZ*) is attempted before an inscribed cylinder one (marked as XYZ). This is therefore slightly different from the proposal of (Wermelinger et al. 2016) where planning in full-space is only attempted if all others have failed. The cost-focused sequence-of-approximations exhibits a different strategy: it first attempts a quick (sub-optimal) plan with a circumscribed cylinder model which is then refined for a long time. On failure it attempts planning in full space, and only then with an inscribed-cylinder-then-full-model hierarchy. The order enables quick-and-low-cost solutions for problems with few obstacles but still allows low-cost solutions to be found in complex problems with narrow passages (requiring the inscribed cylinder model).

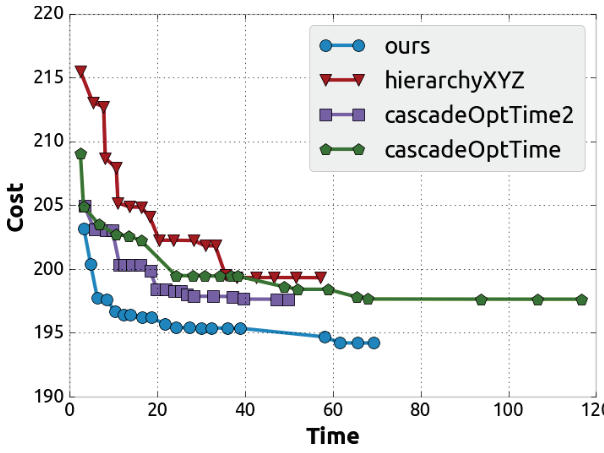


Figure 7: Pareto fronts of total-computation-time and final-motion-cost (cascade-planner exp.)

Finally, we solved a new set of 10 random problems (generated from a different random seed) using all the optimized sequences-of-approximations. We compare the best performing solution of each sequence-of-approximations in Table 2, where performance was obtained by sorting all results of a sequence by best success rate and then by best computation time. The table shows our method is 1.5x to 12x faster than other hierarchies. For those methods with similar computation times to ours, they obtain higher motion costs than ours. The table also shows worst-case success rates, which are 100% for our planner and 80 to 90% for other hierarchies.

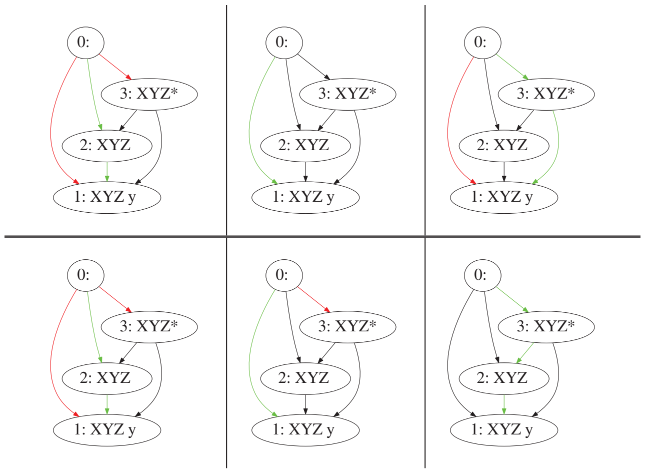


Figure 8: Execution in several planning problems, at different cost/time trade-offs (cascade-planner experiment). Each column is a different problem. Top row: fast-sequence, bottom row: cost-sequence. Red edges indicate failed plans, green are successful. XYZ* uses a circumscribed cylinder robot model, XYZ uses an inscribed cylinder.

Table 2: Performance on test set (cascade-planner exp.)

Hierarchy	Success rate*	Avg. time (s)*	Avg. cost	Worst-case success rate
ours	**1.00**	**4.72**	208.48	**1.00**
XYZ	1.00	57.06	**196.04**	0.90
cascadeOptTime2	1.00	17.04	209.64	0.90
cascadeOptTime	1.00	7.41	211.44	0.80
fullspace	0.70	28.60	112.51	0.60

*best of Pareto set (by success and time in that order)

Conclusion and discussion

We proposed a method to automate and optimize the process of designing sequences of problem approximations for hierarchical motion planners. We use single-source single-sink graphs to represent sequences of approximations, and each approximation consists of an automatically obtained subspace and state validity function. Subspaces are obtained by discarding parts of the factored state-space. We find sequences of approximations through evolutionary optimization of per-graph-edge parameters: a cost and a computation time budget. We use a Pareto-front estimation method to find multiple alternative parameters, with optimal computation-time and motion-quality trade-offs.

Our experiments showed that our method achieves similar, though slightly better, performance than the best-performing serial-hierarchies and also other carefully hand-designed robot-specific hierarchies. The performance gap increases on test problems, i.e. new problems not present during the learning stage, where we obtain solutions up to 12x faster. This supports that our method is more robust and generalizes better than existing state-of-the-art hierarchy designs. The intuition behind this result is that ST-DAG representations of sequences-of-approximations are more flexible, having more pathways to obtain a solution in the state-space of interest when compared to traditional serial-hierarchies (i.e. with a single path from source to sink).

The take-home-message is that hand-designed motion planning hierarchies, because they are narrow in terms of the number of considered sequences of approximations and planner parameters, can lead to sub-optimal planners that do not transfer well to new problems. Our baselines were quite generous and assumed the computation-time allowed at each stage of traditional planning hierarchies to be optimized, but in practice these parameters are usually manually obtained by a few tests on a few problems of interest. Our parameterization allows us to automate not only this process but the process of hierarchy design itself.

Regarding limitations, our method as presented here only considers state-space approximations obtained by discarding parts of a factored state-space. Ideally, and as part of future work, the state-space approximations should be general state-space projections automatically discovered. It is still unclear how this could be best achieved, but we believe this paper brings us a step closer to this goal. Additionally, the edge-cost component of our hierarchy parameterization is non-differentiable, which is also why we use an evolutionary method to solve it. Other parameterizations could lead to better results, so the question of which parameterization is best suited for hierarchy-optimization is still open. Finally, during the training stage we update optimization variables based only on the quality of the motion obtained at the state-space of interest. However, this means that variations in computation-time-budgets or edge-costs along pathways of the ST-DAG that do not get executed on the training set will not be reflected in the computation time or quality of motion. Currently we tackle this problem by adding sufficient motion planning training problems. Alternatively, the optimization method could aggregate the computation times and motion costs over all possible planning sequences in the graph, for example weighted by edge cost—similarly to what is done in neural networks (Liu, Simonyan, and Yang 2018). This could lead to better generalization and would allow the use of gradient-based optimization methods.

Acknowledgments

This work was supported by the UKRI and EPSRC ORCA Hub [EP/R026173/1]. We would also like to thank Jonathan Gammell for his feedback on a previous version of the paper.

References

Balaguer, J.; Spiers, H.; Hassabis, D.; and Summerfield, C. 2016. Neural mechanisms of hierarchical planning in a virtual subway network. *Neuron* 90(4):893 – 903.

Botvinick, M., and Weinstein, A. 2014. Model-based hierarchical reinforcement learning and human action control. *Philosophical Transactions of the Royal Society B: Biological Sciences* 369(1655):20130480.

Brandao, M.; Hashimoto, K.; Santos-Victor, J.; and Takanishi, A. 2015. Optimizing energy consumption and preventing slips at the footstep planning level. In *15th IEEE-RAS International Conference on Humanoid Robots*, 1–7.

Brandao, M.; Fallon, M.; and Havoutis, I. 2019. Multi-controller multi-objective locomotion planning for legged robots. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Brock, O., and Kavraki, L. E. 2001. Decomposition-based motion planning: a framework for real-time motion planning in high-dimensional configuration spaces. In *2001 IEEE International Conference on Robotics and Automation*, volume 2, 1469–1474 vol.2.

Brock, O., and Yang, Y. 2005. Efficient motion planning based on disassembly. In *Robotics: Science and Systems*.

Chestnutt, J., and Kuffner, J. J. 2004. A tiered planning strategy for biped navigation. In *4th IEEE/RAS International Conference on Humanoid Robots*, volume 1, 422–436 Vol. 1.

Choudhury, S.; Dellin, C. M.; and Srinivasa, S. S. 2016. Pareto-optimal search over configuration space beliefs for anytime motion planning. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, 3742–3749. IEEE.

Deb, K.; Pratap, A.; Agarwal, S.; and Meyarivan, T. 2002. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation* 6(2):182–197.

Fankhauser, P., and Hutter, M. 2016. A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation. In Koubaa, A., ed., *Robot Operating System (ROS) – The Complete Reference (Volume 1)*. Springer. chapter 5.

Fortin, F.-A.; De Rainville, F.-M.; Gardner, M.-A.; Parizeau, M.; and Gagné, C. 2012. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research* 13:2171–2175.

Frans, K.; Ho, J.; Chen, X.; Abbeel, P.; and Schulman, J. 2017. Meta learning shared hierarchies. *arXiv preprint arXiv:1710.09767*.

Hold-Geoffroy, Y.; Gagnon, O.; and Parizeau, M. 2014. Once you scoop, no need to fork. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, 60. ACM.

Huys, Q. J. M.; Lally, N.; Faulkner, P.; Eshel, N.; Seifritz, E.; Gershman, S. J.; Dayan, P.; and Roiser, J. P. 2015. Interplay of approximate planning strategies. *Proceedings of the National Academy of Sciences* 112(10):3098–3103.

Lavin, A. 2015. A pareto optimal d* search algorithm for multiobjective path planning. *arXiv preprint arXiv:1511.00787*.

Likhachev, M.; Gordon, G. J.; and Thrun, S. 2003. Ara*: Anytime a* with provable bounds on sub-optimality. In *Advances in Neural Information Processing Systems*, 767–774.

Likhachev, M. 2010. Search-based planning library.

Liu, H.; Simonyan, K.; Vinyals, O.; Fernando, C.; and Kavukcuoglu, K. 2017. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*.

Liu, H.; Simonyan, K.; and Yang, Y. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*.

Maniezzo, V. 1994. Genetic evolution of the topology and weight distribution of neural networks. *IEEE Transactions on Neural Networks* 5(1):39–53.

Mastalli, C.; Havoutis, I.; Winkler, A. W.; Caldwell, D. G.; and Semini, C. 2015. On-line and on-board planning and perception for quadrupedal locomotion. In *IEEE International Conference on Technologies for Practical Robot Applications (TePRA)*.

McGovern, A., and Barto, A. G. 2001. Automatic discovery of subgoals in reinforcement learning using diverse density. In *ICML*.

Montemerlo, M.; Becker, J.; Bhat, S.; Dahlkamp, H.; Dolgov, D.; Ettinger, S.; Haehnel, D.; Hilden, T.; Hoffmann, G.; Huhnke, B.; Johnston, D.; Klumpp, S.; Langer, D.; Levandowski, A.; Levinson, J.; Marcil, J.; Orenstein, D.; Paefgen, J.; Penny, I.; Petrovskaya, A.; Pflueger, M.; Stanek, G.; Stavens, D.; Vogt, A.; and Thrun, S. 2008. Junior: The stanford entry in the urban challenge. *Journal of Field Robotics* 25(9):569–597.

Orthey, A.; Escande, A.; and Yoshida, E. 2018. Quotient-space motion planning. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 8089–8096. IEEE.

Park, C.; Pan, J.; and Manocha, D. 2014. High-dof robots in dynamic environments using incremental trajectory optimization. *International Journal of Humanoid Robotics* 11(02):1441001.

Pham, Q.-C.; Caron, S.; Lertkultanon, P.; and Nakamura, Y. 2017. Admissible velocity propagation: Beyond quasi-static path planning for high-dimensional robots. *The International Journal of Robotics Research* 36(1):44–67.

Plaku, E. 2013. Robot motion planning with dynamics as hybrid search. In *AAAI Conference on Artificial Intelligence*.

Solway, A.; Diuk, C.; Córdova, N.; Yee, D.; Barto, A. G.; Niv, Y.; and Botvinick, M. M. 2014. Optimal behavioral hierarchy. *PLOS Computational Biology* 10(8):1–10.

Styler, B. M. K., and Simmons, R. 2017. Plan-time multi-model switching for motion planning. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*.

Wermelinger, M.; Fankhauser, P.; Diethelm, R.; Krüsi, P.; Siegwart, R.; and Hutter, M. 2016. Navigation planning for legged robots in challenging terrain. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, 1184–1189. IEEE.

Zitzler, E.; Laumanns, M.; and Thiele, L. 2001. Spea2: Improving the strength pareto evolutionary algorithm. *TIK-report* 103.

Zoph, B., and Le, Q. V. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.