Traversing Environments Using Possibility Graphs for Humanoid Robots

Michael X. Grey, Aaron D. Ames, and C. Karen Liu

Georgia Institute of Technology, Atlanta GA 30332, USA

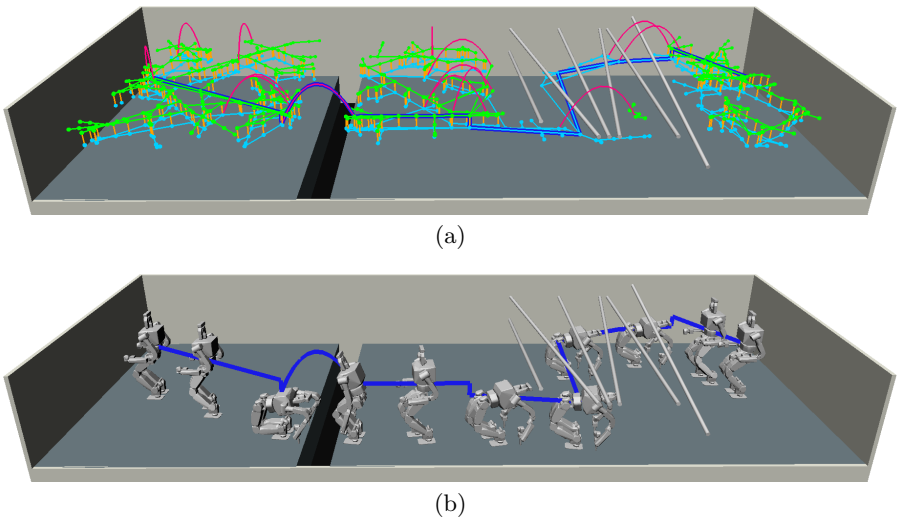
Abstract. Locomotion for legged robots poses considerable challenges when confronted by obstacles and adverse environments. Footstep planners are typically only designed for one mode of locomotion, but traversing unfavorable environments may require several forms of locomotion to be sequenced together, such as walking, crawling, and jumping. Multimodal motion planners can be used to address some of these problems, but existing implementations tend to be time-consuming and are limited to quasi-static actions. This paper presents a motion planning method to traverse complex environments using multiple categories of actions. We introduce the concept of the "Possibility Graph", which uses high-level approximations of constraint manifolds to rapidly explore the "possibility" of actions, thereby allowing lower-level single-action motion planners to be utilized more efficiently. We show that the Possibility Graph can quickly find paths through several different challenging environments which require various combinations of actions in order to traverse.

1 Introduction

Modern motion planning methods have proven effective at navigating geometric constraint manifolds within high dimensional configurations spaces. This capability is critical for robots to exhibit autonomy in complex real-world environments, because geometric constraints are frequently used to determine the *feasibility* of a physical action and hence are often used as "feasibility constraints" which must be satisfied or else the action is considered infeasible. Geometric constraints include requirements such as avoiding obstacles and placing end effectors in appropriate locations. Two common types of motion planners are Probabilistic Roadmaps (PRM) [15] and Rapidly-exploring Random Tree (RRT) [14]. Standard PRM is well-suited for exploring a single *expansive* manifold, as defined in [1]. Constrained Bi-directional RRT (CBiRRT) [2] can effectively handle constraint manifolds whose dimensionality is lower than the configuration space in which it exists.

Standard motion planning methods tend to struggle when a solution needs to traverse numerous topologically distinct constraint manifolds. This occurs most often in hybrid dynamic systems where the "mode" of the system alters its constraint manifold. For example, standing on the left foot is a different mode from standing on the right foot for a bipedal robot. The constraint manifolds of these two modes are different, and their intersection is narrow, resulting in a

low (sometimes zero) probability of randomly moving from one manifold to the other. Hauser et al. introduced the Multi-modal PRM [10] to address this problem. The primary bottleneck of this method is the combinatorial complexity of sampling and selecting modes, since each footstep taken by the robot represents a mode that must be explored. Additionally, existing implementations of the Multi-modal PRM are limited to quasi-static actions, which broadly eliminates its ability to utilize the dynamic capabilities of a robot system.



(a)

(b)

Fig. 1. The robot is tasked with traversing from the right side of a hallway to the left side. It must navigate underneath bars which are positioned at various angles, and then must jump across a gap. (a) The graph explored the space of the hallway until a solution was found. Green edges are walking actions, light blue edges are crawling actions, and fuchsia arcs are jumping actions. (b) Snapshots show the plan in action.

In contrast to motion planning methods, standard footstep planners are able to rapidly generate footsteps and walking trajectories without spending time exploring the constraint manifolds of combinatorial modes the way a Multi-modal PRM does. They typically do this by approximating the problem of walking. In [3, 4] this is done using a 2D representation of obstacles and in [3, 4, 5] only a finite set of footstep parameters or action primitives are available to the planner. The two-stage method presented in [6] uses a bounding cylinder to represent the collision geometry of the lower body. All of these estimations inherently limit the completeness of the methods. Moreover, these methods are all limited to a single category of action: bipedal walking.

The use of optimization methods in the motion planning domain has been growing [16, 17], especially for walking motions. Nonlinear constrained optimization can elegantly handle the mixed modes and hybrid dynamics [18] required

for walking and crawling. However, they tend to be tailored for generating single behaviors (e.g. a walking behavior that consists of a single- and double-support phase). This is insufficient for traversing a complex environment where a sequence of different types of behaviors is needed. Optimizations methods also tend to be local, making them inappropriate for tackling problems that require a global search.

The goal of this paper is to introduce a new high-level motion planning method, named the *Possibility Graph*, that can leverage the speed and efficiency of standard footstep planners with the completeness of randomized motion planning methods and the dynamics capabilities of optimization-based methods. The Possibility Graph is general enough to handle arbitrary categories of actions instead of being limited to only walking or stepping primitives. The role of the Possibility Graph is to quickly explore what actions might be possible throughout an environment. Different action types are compactly interlaced with each other within the graph, allowing a solution to utilize any action types in any order. Once a potential route is discovered, lower-level planning methods are used to confirm whether the route is truly feasible. This allows the lower-level (and computationally intensive) planners to focus their efforts on queries which are likely to achieve a solution. These queries can be performed in parallel, ensuring that the overall planning effort does not get bottlenecked by any single challenging step.

The three categories of actions used in this paper are walking, crawling, and a standing long jump. Figure 1 shows these three actions being utilized in a hallway example. We test the Possibility Graph on various scenarios. In some scenarios, multiple action categories may be required to reach the goal. We show that the Possibility Graph works reliably on the order of seconds. Complete solutions tend to generate at faster than 100x real time. This makes Possibility Graphs suitable for online planning. They could also be incorporated into higher level task planners [9] which require numerous high-speed queries.

2 Possibility Graph

The governing logical principles behind the Possibility Graph have a theoretical grounding in Possibility Theory [7], but the concepts are intuitive enough that a knowledge of Possibility Theory is not necessary to proceed. It is enough to understand that the *possibility* of any given action instance can be labelled with "impossible", "possible", or "indeterminate" depending on whether the instance satisfies the necessary or sufficient conditions that are assigned to it. The motivation for using a Possibility Graph is two-fold:

1. We can design necessary and/or sufficient conditions that can be checked quickly, making expansion of the graph very efficient.
2. Even though different actions may have constraint manifolds with different dimensionalities, we can design their necessary and/or sufficient conditions to share a common set of parameters, allowing for a single unified graph which combines all actions.

Motion planning methods ordinarily operate by constructing graphs or trees which fully exist within the feasibility constraint manifold of the action they are performing. Remaining within this manifold is a reasonable requirement to place on the graph, because any vertices or edges which step outside of the manifold are, by definition, invalid—which may mean it is physically impossible, or simply harmful to the robot or its surroundings. Unfortunately, for a humanoid robot to remain on the constraint manifold, expensive calls to whole body inverse kinematics solvers (for more on whole body IK, see [11, 12, 13]) must be performed. This results in a critical bottleneck if a broad area needs to be explored before finding a solution. By exploring the *possibility* of actions first, we can broadly avoid expensive whole body inverse kinematics queries and easily slide through transition spaces which would otherwise be narrow.

2.1 Simplifying the Manifold: Sufficient vs. Necessary Conditions

To construct the Possibility Graph, we must first design sufficient and/or necessary conditions for the feasibility constraint manifold of each action. The two motivations which were mentioned earlier imply that the conditions we create should satisfy two criteria: (1) They should be quick to test, and (2) they should be low dimensional, using as few parameters as is reasonable.

Suppose we have a 2D constraint manifold, C, which exists in 3D space, as shown in Fig. 2(a). Supposing we can directly compute the z-value of the manifold given valid x and y values, it makes sense to project this manifold down onto the xy-plane. We can call the projection C_P. This accomplishes our goal of lower dimensionality.

Even with a flattened-out projection, identifying which points are inside or outside of the manifold may still be costly or difficult, because the boundaries of the projection may be functions that are expensive to compute or hard to fully define. However, suppose a box, circle, or some other simple shape can be fit within the projection such that it is *guaranteed* that every point within the simplified shape also lies within the manifold projection. Such a shape would be a suitable representation of the sufficient condition manifold, C_S, for the constraint manifold C. Mathematically, this means $C_S \subseteq C_P$. Similarly, if a simple shape, C_N, could be fit *around* the projection C_P such that $C_P \subseteq C_N$, then C_N would qualify as the necessary condition manifold. The specific designs of necessary and sufficient manifolds for this paper are discussed in Sec. 3.

2.2 Explore Possibilities by Expanding the Graph

The purpose of the Possibility Graph is to find a feasible action sequence to get from a start point to at least one goal point. We define the contents of the Possibility Graph in Def. 1. The procedure for finding solutions with the graph is described by Alg. 1. The graph is initialized with a set of subgraphs, each subgraph consisting solely of either the start point or a goal point. All the graph's vertices are elements of the "possibility exploration space", $\mathscr{E}$. The Possibility Graph finds paths through the exploration space by querying each

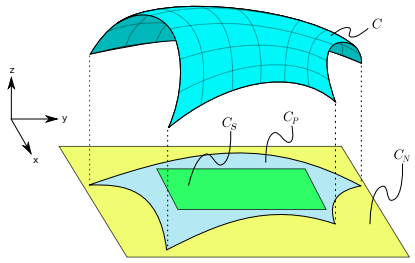


Fig. 2. Visual depiction of an abstract constraint manifold, C and its projection. The manifold is projected, C_P, from 3D space onto a plane. "Sufficient" C_S and "Necessary" C_N boundaries are fitted within and around the projection of the manifold.

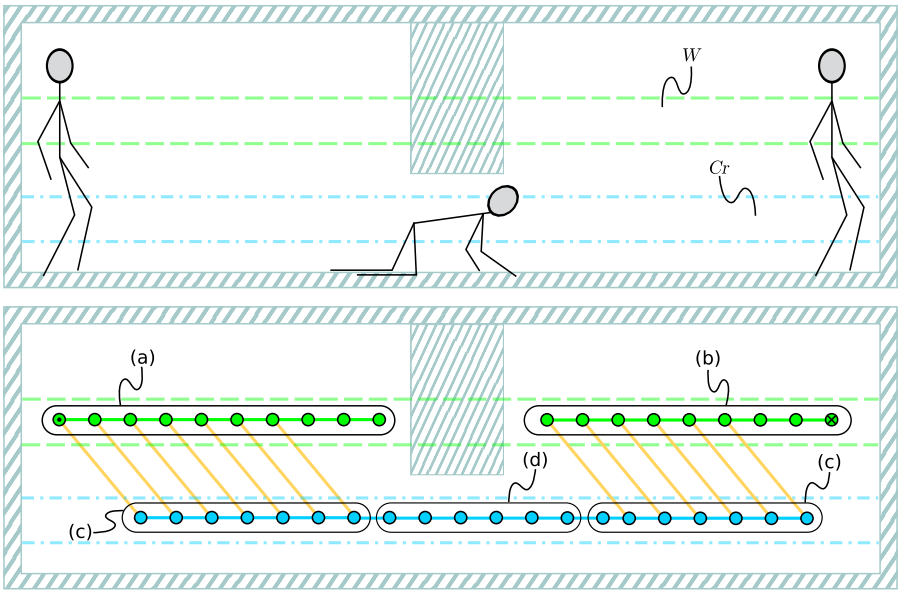


Fig. 3. Cartoon showing a simple 2D stick-figure example where the stick figure can walk or crawl forward. The graph's vertices represent the (x, z) values of a point fixed to the stick figure's chest. The upper region, marked by W in the top photo, is where walking is valid. The lower region, Cr, is where crawling is valid. (a) We extend from the start vertex towards a randomly sampled point in the center. [Alg. 3, line 7] (b) We extend from the goal vertex towards the last vertex that was created in the previous step. [Alg. 3, line 13] (c) For each new walking vertex, we create a crawling vertex and connect it to the walking vertex using a transition edge [Alg. 1, line 8]. For some of the walking vertices, a transition into crawling is not viable due to obstacles. (d) We now extend the crawling subgraphs towards a point that was sampled near the center of the room, and the subgraphs manage to connect [Alg. 1, line 10].

available action to expand the graph in randomized directions within $\mathscr{E}$ (see line 10 of Alg. 1). If a query meets at least the necessary conditions of the action, then it will be appended to the graph.

Definition 1. A Possibility Graph is a tuple
$PG = (\texttt{actions}, \mathscr{E}, \Gamma_{PG}, Q_{Confirmation})$, where

- `actions` is a set of actions (defined in Table 1),
- $\mathscr{E}$ is a space consisting of the union of all the parameters used by the necessary and sufficient conditions of each `Action`,
- Γ_{PG} is a directed graph whose vertices are elements of $\mathscr{E}$,
- $Q_{Confirmation}$ is a queue which manages confirmation jobs (see Table 1).

An important feature of our algorithm is the exploration of transitions between various action categories. Each time vertices are added for one action, the other actions will be queried to see if they can transition from it (Alg. 1 , Line 8). This allows different actions to be interlaced with each other within the graph. Each action keeps track of its own exploration by storing a set of subgraphs, Γ_a, consisting of its own vertices and edges. At the same time, the Possibility Graph maintains the "master" graph, Γ_{PG}, which combines all the subgraphs of all the different action types. The algorithm is illustrated by a toy example in Fig. 3.

Over time, the Possibility Graph will consist of vertices and edges from various actions interlaced with each other. Some elements of the graph will satisfy the sufficient conditions of their respective actions, but some will only satisfy the necessary conditions. Once the graph contains a path from the start vertex to a goal vertex, we need to inspect the vertices and edges of that path to confirm whether all the path elements are truly feasible. This process is shown in the `ConfirmPath` function of Alg. 1. Actions are responsible for spawning "confirmation jobs" which are low-level planning routines whose job is to verify whether or not an edge in the possibility graph is truly feasible. These routines are loaded into the Confirmation Queue, $Q_{Confirmation}$. The Confirmation Queue will rotate between running each job to ensure that easy ones are finished promptly while difficult ones do not halt the overall confirmation progress. These jobs are executed on threads which run parallel to the graph expansion and each other. This allows the planner to search for alternative potential solutions when certain edges are difficult to confirm.

2.3 Extending Action Subgraphs

For the Possibility Graph to explore actions, we need to fully define each action type. Table 1 lays out the implementation-dependent functions which must be engineered for each action type. The functions in that table enable the `GrowTowards` and `PerformTransitions` functions to work. `PerformTransitions` is described in Alg. 2. It simply pulls vertices from other actions out of a queue and attempts to create transitions from those actions to itself. `GrowTowards` serves two primary roles: (1) expand the graph in new directions, and (2) connect together disjoint subgraphs. The nature of how an action grows will depend

Algorithm 1: Finding a path by exploring possibilities

```
1  Function FindPath(start, goals, actions)
2      Γ_PG.V ← {start, goals};
3      Initialize each action graph with the start and goal vertices;
4      Q_Confirmation.launchThreads();
5      t ← 0;
6      while t < t_max do
7          for a in actions do
8              {V_new, E_new} ← a.PerformTransitions();
9              p_target ← RandomSample();
10             {V_new, E_new}.append(a.GrowTowards(p_target));
11             for all a_other not a in actions do
12                 a_other.Q_Transition.insert(V_new);
13             {Γ_PG.V, Γ_PG.E}.append({V_new, E_new});
14         for g in goals do
15             if Connected(start, g) then
16                 Γ_path ← ShortestPath(start, g);
17                 if ConfirmPath(Γ_PG, Γ_path, Q_Confirmation, actions) then
18                     return Γ_path;
19         t ← CurrentTime();
20     return null;
21 Function ConfirmPath(Γ_PG, Γ_path, Q_Confirmation, actions)
22     pathConfirmed ← true;
23     for edge in Γ_path.E do
24         edgeConfirmed ← false;
25         for a in actions do
26             if a.C_S(edge) then
27                 edgeConfirmed ← true;
28             else if a.C_N(edge) then
29                 Γ_PG.remove(edge);
30                 Q_Confirmation.insert(a.SpawnConfirmationJob(edge));
31         if not edgeConfirmed then
32             pathConfirmed ← false;
33     return pathConfirmed;
```

on what kind of action it is. For this paper, we have two methods of expanding an action, one for holonomic actions and the other for nonholonomic.

Holonomic actions are expanded using Alg. 3. When we describe an action as "holonomic" in this context, we mean that its sufficient/necessary condition manifold is holonomic. Even if the full feasibility constraint manifold of the action is nonholonomic, it can be treated as holonomic by the Possibility Graph if its necessary/sufficient condition manifold is simplified to be holonomic within

Table 1. Definition of an Action

All action types	
`ExtendTowards`(v_0, v_1):	Create a vertex by moving towards v_1 from v_0 via this action.
$C_N(x)$:	Return **true** if x meets the action's necessary conditions, otherwise return **false**. x may be a vertex or an edge.
$C_S(x)$:	Return **true** if x meets the action's sufficient conditions, otherwise return **false**. x may be a vertex or an edge.
`TransitionFrom`(v):	Attempt to return a path that goes from v into the necessary condition manifold of this action.
`SpawnConfirmationJob`(e):	Return a routine (called a confirmation job) which can examine edge e to ascertain whether it is truly feasible.
Holonomic action types	
`Project`(v):	Attempt to return a point on the necessary condition manifold which is close to v.
Nonholonomic action types	
`ReverseExtend`(v_0, v_1):	Create a vertex which can arrive at v_0 from the direction of v_1 via this action.
`FindLaunchPoint`(v, v_1):	Return a point, v_0, close to v which can be used in a call to `ExtendTowards`(v_0, v_1)
`FindLandingPoint`(v, v_1):	Return a point, v_0, close to v which can be used in a call to `ReverseExtend`(v_0, v_1)

Algorithm 2: Utilizing the Transition Queue

```
1 Function Action::PerformTransitions()
2     {V_new, E_new} ← {new VertexQueue, new EdgeQueue};
3     i ← 0;
4     while i < MaxTransitionsPerCycle do
5         v ← PopRandom(Q_Transitions);
6         {V_new, E_new}.append(TransitionFrom(v));
7         i ← i + 1;
8     Γ_a.append({V_new, E_new});
9     return {V_new, E_new};
```

the exploration space, $\mathscr{E}$. Alg. 3 shows how the possibilities of holonomic actions are expanded. Importantly, holonomic actions always try to connect disjoint subgraphs together. This procedure is very similar to the growth of a CBiRRT [2], except that it accommodates numerous directional subgraphs. To avoid having subgraphs needlessly cross over each other, we only extend two at a time: The subgraph who has the vertex closest to the random target is extended towards the target up to some point v_0 [Alg. 3, line 7] (at which point it cannot extend any further); then the second closest subgraph attempts to connect to v_0 [Alg.

3, line 13]. However, if the first subgraph was goal-connected, then the second subgraph must not be (i.e. we skip over the next closest subgraph until we reach one which is not goal-connected), because connecting together two goal-connected subgraphs cannot help in finding a solution.

Nonholonomic actions are expanded in a more complex way than holonomic actions, as shown in Alg. 4. Nonholonomic actions generally cannot move directly towards a goal, so they need to "line themselves up" first. We do this by identifying a launch point which is reachable from an existing point on the graph [Alg. 4, line 10]. The launch point should be chosen such that it allows the action to land as close to the randomly generated target as possible, so long as the launch point is still reachable from the existing graph. Since nonholonomic actions are also generally direction-dependent, we do the reverse for goal-connected subgraphs [Alg. 4, line 17]: Pick a landing point which can connect to an existing goal-connected vertex such that it has a viable launch point coming from the direction of the target. Section 3.2 describes this for the jump action.

3 Action Implementations

In this paper, we implement three action types to serve as a proof of concept. Two are holonomic and one is nonholonomic. They include walking, crawling, and a standing long jump. We use a model of the DRC-HUBO1 robot, because its kinematic structure is designed to accommodate crawling. The scenarios in which we apply these actions will be discussed in Sec. 4. For the exploration space of the Possibility Graph, $\mathscr{E}$, we use the SE(3) coordinates of a reference frame attached to the robot's pelvis.

3.1 Walk and Crawl

The walking and crawling actions are formulated very similarly to each other. Sufficient conditions for walking and crawling are holonomic, and include these simplifications:

1. We use a swept collision geometry, similar to [9]. The geometries can be seen in Fig. 4. These geometries must not be in collision with the environment when given a point in $\mathscr{E}$.
2. Each point that defines the robot's support polygon must be touching flat ground when the robot is in a "nominal" walk/crawl configuration. The nominal configurations can be seen in Fig. 4.
3. The root must be in the "nominal" orientation of the action (upright for walking and pitched backwards 80° for crawling).

The necessary conditions are significantly easier to satisfy:

1. We use only the collision geometry of the pelvis, because all other bodies depend on joint parameters which are not included in $\mathscr{E}$.
2. At least one foot must be able to reach some ground surface.

Algorithm 3: Growing the graph for a holonomic action

```
1  Function HolonomicAction::GrowTowards(p_target)
2      Q_closest ← new SortedVertexQueue;
3      for g in Γ_a.SubGraphs do
4          v ← g.FindClosestVertexTo(p_target);
5          Q_closest.insert(dist(v, p_target), v);
6      v_0 ← Q_closest.pop_front();
7      {V_new, E_new} ← Connect(v_0, p_target);
8      p_target ← V_new.back();
9      if UpstreamFromGoal(v_0) then
10         while UpstreamFromGoal(Q_closest.front() do
11             Q_closest.pop_front();
12     v_1 ← Q_closest.pop_front();
13     {V_new, E_new}.append(Connect(v_1, p_target));
14     Γ_a.append({V_new, E_new});
15     return {V_new, E_new};
16 Function HolonomicAction::Connect(v_start, p_target)
17     {V_new, E_new} ← {new VertexQueue, new EdgeQueue};
18     v_last ← v_start;
19     v ← ExtendTowards(v_start, p_target);
20     v_p ← Project(v);
21     while C_N(v_p) and v ≠ p_target do
22         edge ← Edge(v_last, v_p);
23         if not C_N(edge) then
24             break;
25         {V_new, E_new}.append({v_p, edge});
26         v_last ← v_p;
27         v ← ExtendTowards(v, p_target);
28         v_p ← Project(v);
29     return {V_new, E_new};
```

The `ExtendTowards`(v_0, v_1) function simply applies an SE(3) transform which translates and rotates v_0 to bring it closer to v_1. Changes in rotation should be weighted less than changes in translation in order to have sensible differences between steps. The `Project` function for these action templates adjusts the height and orientation of the SE(3) input so that it matches the nominal configuration of the action. Translation in x/y and rotation along z are unaffected. The `TransitionFrom` function moves between these actions by generating a simple motion that goes from one nominal configuration to the other. If an edge only meets the necessary conditions of the action, then another planning method (such as Multi-modal PRM) must be generated by the `SpawnConfirmationJob` function. On the other hand, when the sufficient conditions are satisfied, the final motion for these actions is easily determined by placing footsteps along

Algorithm 4: Growing the graph for a nonholonomic action

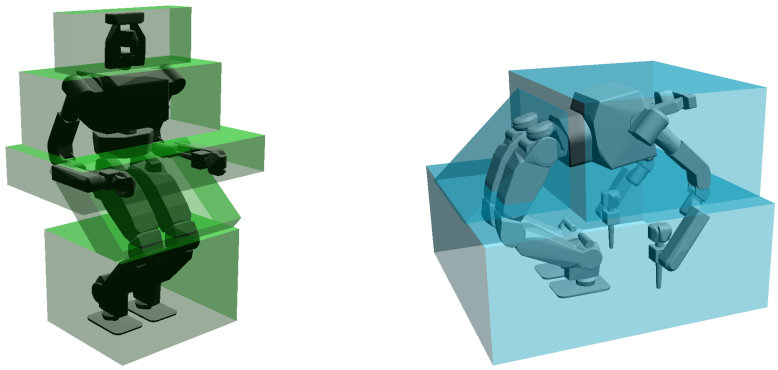
```
1  Function NonholonomicAction::GrowTowards(p_target)
2      {V_new, E_new} ← {new VertexQueue, new EdgeQueue};
3      Q_closest ← new SortedVertexQueue;
4      for v in Γ_a.V do
5          Q_closest.insert(dist(v, p_target), v);
6      Useful ← new BooleanArray(Γ_a.V.size(), true);
7      for v in Q_closest do
8          if not Useful[v] then continue ;
9          if not UpstreamFromGoal(v) then
10             v_launch ← FindLaunchPoint(v, p_target);
11             v_landing ← ExtendTowards(v_launch, p_target);
12             edge ← Edge(v_launch, v_landing);
13             if C_N(edge) then
14                 {V_new, E_new}.append({v_launch, v_landing, edge});
15                 RecursivelySetUpstreamVerticesToFalse(v, Useful);
16         if not DownstreamFromStart(v) then
17             v_landing ← FindLandingPoint(v, p_target);
18             v_launch ← ReverseExtend(v_landing, p_target);
19             edge ← Edge(v_launch, v_landing);
20             if C_N(edge) then
21                 {V_new, E_new}.append{v_launch, v_landing, edge};
22                 RecursivelySetDownstreamVerticesToFalse(v, Useful);
23     Γ_a.append({V_new, E_new});
24     return {V_new, E_new};
```

the specified route through SE(3) and then generating a whole body motion to follow those footsteps. Our sufficient conditions guarantee that it will be possible to generate and follow those footsteps.

3.2 Standing Long Jump

A standing long jump is a forward jump which begins from standing in place and launches forward without taking any steps. Figure 5 shows an example of a jumping trajectory. We use a standing long jump in this paper for simplicity; future work will include long jumps that take running starts, which can achieve considerably greater range. We provide necessary conditions for the standing long jump but not sufficient conditions. The necessary condition manifold is nonholonomic, and contains the following:

1. The vertex that begins the jump must be a valid walk vertex.
2. The vertex that finishes the jump must be a valid crawl vertex.
3. There must be at least one collision-free parabola through $\mathscr{E}$ from the beginning vertex to the finishing vertex. The parabola must follow a feasible jump arc according to the physical limitations of the robot.



(a) (b)

Fig. 4. The nominal configurations used for (a) walking and (b) crawling, with their swept geometries surrounding them.

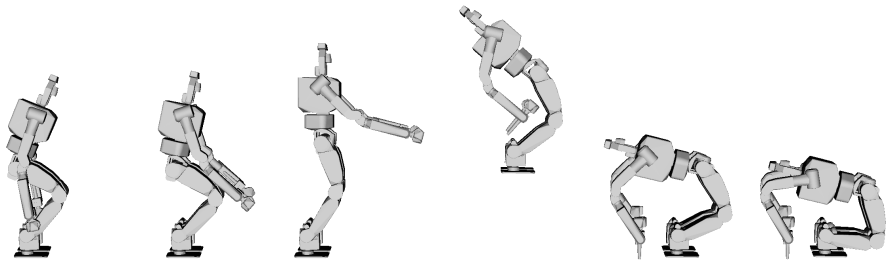


Fig. 5. An example standing long jump trajectory. The robot begins from a standing configuration, swings its arms, and jumps forward. It plans out its angular momentum so that it is able to land in a crawling configuration. After hitting the ground, it absorbs some of the impact by letting its joints behave elastically.

The `TransitionFrom` function for the jump action is trivial, because it always begins from valid walking configurations and ends in valid crawling configurations, therefore the transition function does nothing. The `ExtendTowards`(v_0, v_1) function performs a forward jump from v_0 to v_1. If v_1 is too far to reach from v_0, then it performs the furthest allowable jump. The `ReverseExtend`(v_0, v_1) function instead performs a jump which lands at v_0 and begins as close to v_1 as the robot's physical limitations allow. The `FindLaunchingPoint`(v, v_1) function returns a point, v_0, whose translation is the same as v but whose orientation has the robot facing v_1; this allows the `ExtendTowards`(v_0, v_1) function to bring the robot closer to v_1. Conversely, the `FindLandingPoint`(v, v_1) function returns a point, v_0, whose translation is the same as v but which is facing *away* from v_1; this allows the robot to jump towards v from the direction of v_1 using `ReverseExtend`(v_0, v_1).

The `SpawnConfirmationJob` function of the jump action is a basic collocation optimization on a boundary value problem. The boundary value constraints are (1) zero initial velocity, (2) a take-off configuration and velocity which will allow the robot to reach its jump target. The objective function of the optimization problem attempts to minimize the accelerations during take-off. While generating the trajectory, we also check that the joint and contact forces required to achieve the trajectory are physically feasible. Trajectories which fail this test are discarded. Once the jump is generated, we can check for collisions along its trajectory. If the jump was successfully generated (i.e. the jumping motion is physically feasible) and is collision-free, then its "possibility" status is changed from "indeterminate" to "possible", and it can be used in a final solution.

4 Experimental Results

We run performance tests on three scenarios (one of which has three versions). Each performance test is the result of 50 trials. The Possibility Graph is a randomized planner, so the time required for the same trial can vary between runs. We put a 60 second time limit on the planner; if a solution is not found within 60 seconds, we consider it a failed run.

Three Routes scenario is shown in Fig. 6. There are three potential routes that the robot might take to get from the start to the goal. We have three different versions of this scenario, and each version has progressively stronger requirements for what actions are needed by the solution, allowing us to compare the performance impact caused by specific action sequences being required.
Hallway scenario was shown in Fig. 1. The robot must crawl underneath some bars and then jump across a gap to get from the start on the right side to the goal on the left.
Double Jump scenario is shown in Fig. 7. The robot must jump twice to get from the right side to the left.

In Table 2 we see that the time required to solve a problem scales up with the number of actions being used (comparing the values in the **Graph** column of rows 1–3 and 4–5). For every action that is utilized by the planner, more exploration needs to be performed, which tends to increase the runtime. Not only does the action's space get explored, but also the transitions between the actions need to be explored. However, this cost is additive, not multiplicative, so the overall costliness will be related to the sum (not product) of the costliness of the individual actions. Jumping exploration is considerably more expensive than walking or crawling exploration. To curb this, we can modify Alg. 1, line 10 so that there is some probability of skipping the jumping expansion each cycle. In the results of Table 2, we use a 90% chance of skipping.

We can also see that the time required to solve a problem scales up with the number of actions *required* by the environment to get a solution (comparing the **Graph** values of row 2 to 4 and of row 3 to 5). This is not surprising since

requiring certain actions can be viewed as tightening the constraints on the solution, and tighter constraints tend to take longer to solve with randomized search.

Table 2. Time performance results, tested on an Intel® Xeon® Processor E3-1290 v2 (8M Cache, 3.70 GHz) with 16GB of RAM. N_a is the number of action types that were provided to the planner. "Graph" is the time it took to generate a solved graph. "Motion" is the time it took to generate the physical motions for the solution. γ is the "Real-Time Ratio", i.e. the time it would take to execute the plan divided by how long the whole plan (graph+motion) took to generate. "Success Rate" is how many times the planner succeeded (instead of timing out). All times are given in seconds. Each result is the average of 50 runs; the standard deviation is given in parentheses.

Scenario	N_a	**Graph**	**Motion**	γ	**Success**
Three Routes (a)	1	0.088 (0.048)	8.47 (0.81)	143.2 (3.5)	100%
Three Routes (a)	2	0.134 (0.076)	8.75 (0.91)	143.6 (2.6)	100%
Three Routes (a)	3	0.484 (0.450)	7.52 (1.86)	136.8 (10.0)	100%
Three Routes (b)	2	0.152 (0.112)	9.23 (1.09)	142.5 (3.2)	100%
Three Routes (b)	3	0.561 (0.502)	7.59 (2.30)	134.0 (11.1)	100%
Three Routes (c)	3	1.210 (0.218)	5.73 (1.79)	121.2 (7.03)	100%
Hallway	3	3.67 (11.52)	8.29 (0.84)	127.2 (6.96)	96%
Double Jump	3	1.48 (0.34)	4.32 (0.28)	113.3 (6.24)	100%

Limitations Our experiments only used sufficient conditions for walking and crawling; any actions which violate the sufficient conditions for walking and crawling are ignored. Unfortunately, this eliminates the probabilistic completeness of the implementation. A more complete approach would consider the necessary conditions of walking and crawling, and then employ Multi-modal PRM [10] to examine walking/crawling segments where only the necessary conditions are met. This would allow the robot to step over small obstructions and squeeze through narrow passages between obstacles. These considerations will be the topic of future work.

5 Conclusion

We presented performance results of multi-action traversal plans being generated for the DRC-HUBO1 platform in complex environments. The complexity of the environments is derived from the fact that they require a variety of different action types to be interlaced in the correct sequence in order to navigate from the start to the goal. Three action types were used to traverse these environments: walking, crawling, and the standing long jump. The time required to fully generate the motion plans was less than 1/100th of the time that the motions require for physical execution. This makes the Possibility Graph a promising option for

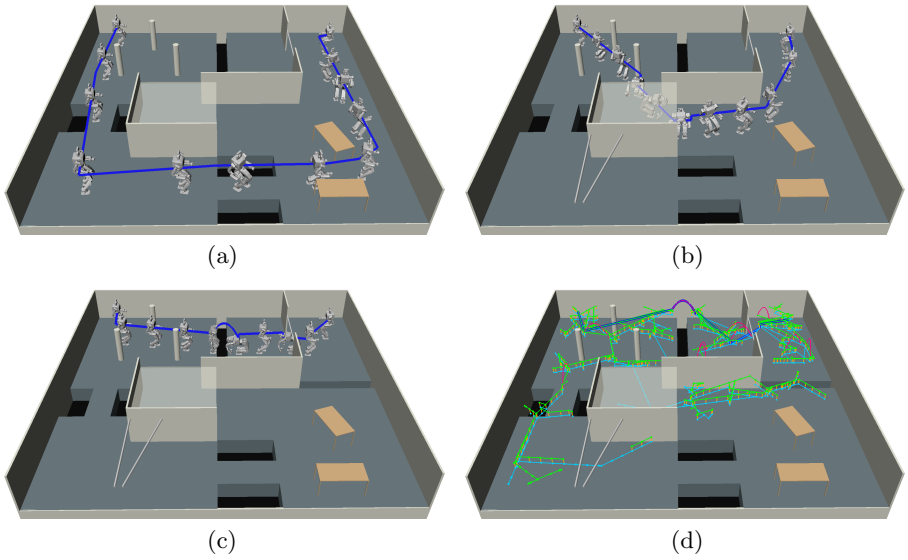


Fig. 6. The three versions of the "Three routes" scenario. The robot must get from the back left corner to the back right corner. (a) A route exists that allows the robot to walk all the way to the goal. (b) Some bars were added to the walking route, so the robot must crawl at least once to reach the goal. (c) A gap was added at the end of the crawling routes, so the robot must jump at least once to reach the goal. (d) A grid that shows the map being explored.

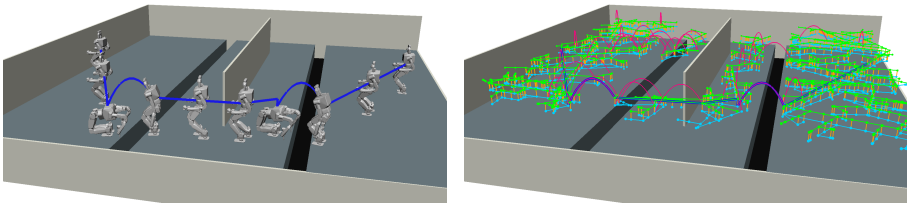


Fig. 7. The "double jump" scenario. The robot must jump across two gaps and navigate around a wall in the middle to get from the right side to the left side.

online use. Moreover, the time required to guarantee that a solution exists is even smaller, which suggests that the Possibility Graph would be an effective tool for higher-level task planners such as the Hybrid Backward-Forward planner [8, 9] which only needs to know whether a query is solvable.

The theoretical framework of the Possibility Graph can extend beyond the applications seen here. Future work will incorporate Multi-modal PRM to achieve probabilistic completeness in the quasi-static domain. We will also incorporate highly dynamic actions, e.g. running jumps, using nonlinear constrained optimization. This will open the door to fast, global, dynamic planning for high dimensional systems.

References

[1] Hsu, D., Latombe, J.-C.: Path planning in expansive configuration spaces. IEEE Int. Conf. on Rob. and Aut. (ICRA), vol. 3, 2719–2726 (1997)

[2] Berenson, D., Srinivasa, S. S., Kuffner, J.: Task space regions: A framework for pose-constrained manipulation planning. The Int. J. of Rob. Res. (2011)

[3] Garimort, J., Hornung, A., Bennewitz, M.: Humanoid navigation with dynamic footstep plans. IEEE Int. Conf. on Rob. and Aut. (ICRA) 3982–3987 (2011)

[4] Hornung, A., Dornbush, A., Likhachev, M., Bennewitz, M.: Anytime search-based footstep planning with suboptimality bounds. 12th IEEE-RAS Int. Conf. on Humanoid Rob. (Humanoids 2012) 674–679 (2012)

[5] Candido, S., Kim, Y. T., Hutchinson, S.: An improved hierarchical motion planner for humanoid robots. 8th IEEE-RAS Int. Conf. on Humanoid Rob. 654–661 (2008)

[6] Pettré, J., Laumond, J. P., Simon, T.: A 2-stages locomotion planner for digital actors. Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation 258–264 (2003)

[7] Dubois, D., Prade, H.: Possibility Theory. Meyers, R. A. (ed) Computational Complexity: Theory, Techniques, and Applications 2240–2252 (2012)

[8] Garrett, C. R., Lozano-Pérez, T., Kaelbling, L. P.: Backward-forward search for manipulation planning. IEEE/RSJ Int. Conf. on Intl. Rob. and Sys. 6366–6373 (2015)

[9] Grey, M. X., Garrett, C. R., Liu, C. K., Ames, A., Thomaz, A. L.: Humanoid Manipulation Planning using Backward-Forward Search. IEEE/RSJ Int. Conf. on Intl. Rob. and Sys. (2016) (to appear)

[10] Hauser, K., Latombe, J. C.: Multi-modal motion planning in non-expansive spaces. The Int. J. of Rob. Res. (2009)

[11] Sentis, L., Khatib, O.: A whole-body control framework for humanoids operating in human environments. Proceedings IEEE Int. Conf. on Rob. and Aut. 2641–2648 (2006)

[12] Sugihara, T., Nakamura, Y.: Whole-body cooperative balancing of humanoid robot using COG Jacobian. IEEE/RSJ Int. Conf. on Intl. Rob. and Sys., vol. 3, 2575–2580 (2002)

[13] Gienger, M., Janssen, H., Goerick, C.: Task-oriented whole body motion for humanoid robots. IEEE-RAS Int. Conf. on Humanoid Rob. 238–244 (2005)

[14] Kuffner, J. J., LaValle, S. M.: RRT-connect: An efficient approach to single-query path planning. IEEE Int. Conf. on Rob. and Aut., vol. 2, 995–1001 (2000)

[15] Kavraki, L. E., Kolountzakis, M. N., Latombe, J.-C.: Analysis of Probabilistic Roadmaps for Path Planning. IEEE Trans. on Rob. and Aut., vol. 14, no. 1, 166–171, (1998)

[16] Zucker, M., Ratliff, N., Dragan, A.D., Pivtoraiko, M., Klingensmith, M., Dellin, C.M., Bagnell, J.A., Srinivasa, S.S.: CHOMP: Covariant Hamiltonian optimization for motion planning. The Int. J. of Rob. Res., 32(9-10), 1164–1193 (2013)

[17] Kuindersma, S., Deits, R., Fallon, M., Valenzuela, A., Dai, H., Permenter, F., Koolen, T., Marion, R., Tedrake, R.: Optimization-based locomotion planning, estimation, and control design for the Atlas humanoid robot. Autonomous Robots, 40(3), 429–455 (2016)

[18] Hereid, A., Cousineau, E.A., Hubicki, C.M., Ames, A.D.: 3D Dynamic Walking with Underactuated Humanoid Robots: A Direct Collocation Framework for Optimizing Hybrid Zero Dynamics. IEEE Trans. on Rob. and Aut. (2016)