Traversing Environments Using Possibility Graphs with Multiple Action Types

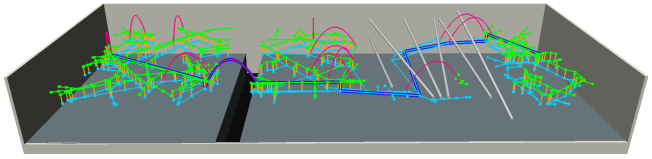
Michael X. Grey[1] C. Karen Liu[2] Aaron D. Ames[3]

***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. Multi-modal 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 continuous actions. To this end, this paper formulates and exploits the *Possibility Graph*—which uses high-level approximations of constraint manifolds to rapidly explore the "possibility" of actions—to utilize lower-level single-action motion planners more effectively. We show that the Possibility Graph can quickly find routes through several different challenging environments which require various combinations of actions in order to traverse.**

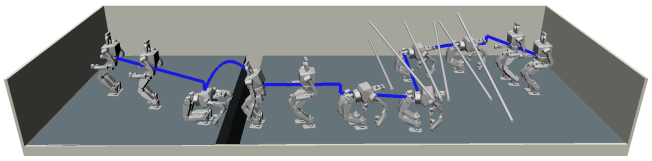
I. Introduction

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) [1] and Rapidly-exploring Random Tree (RRT) [2]. Standard PRM is well-suited for exploring a single *expansive* manifold, as defined in [3]. Constrained Bi-directional RRT (CBiRRT) [4] 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 discrete 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 both feet 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 [5] and Random-MMP [6] 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 methods for multi-modal planning are limited to quasi-static actions, which broadly eliminates their ability to utilize the dynamic capabilities of a robot system.



(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 showing the plan in action.

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.

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 Multi-modal PRM or Random-MMP do. They typically do this by approximating the problem of walking. In [7], [8] this is done using a 2D representation of obstacles and in [7]–[13] only a finite set of footstep parameters or action primitives are available to the planner. The two-stage method presented in [14] 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—while being limited in terms of the dynamics of the motion.

The use of optimization methods in the motion planning domain has been growing [15], [16], especially for walking motions. Nonlinear constrained optimization can

[1]Michael X. Grey is a doctoral candidate in the School of Interactive Computing, Georgia Institute of Technology, Atlanta, GA 30332 mxgrey@gatech.edu

[2]C. Karen Liu is an Associate Professor in the School of Interactive Computing, Georgia Institute of Technology, Atlanta, GA 30332 karenliu@cc.gatech.edu

[3]Aaron D. Ames is an Associate Professor in the School of Mechanical Engineering & the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 aaron.ames@me.gatech.edu

elegantly handle the mixed modes and hybrid dynamics [17] 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. Optimization methods also tend to be local, making them inappropriate for tackling problems that require a global search—especially for high dimensional humanoid robots.

The goal of this paper is to use a 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.

II. Possibility Graphs for Multiple Action Types

Possibility Graphs were introduced in [18] where they were used to guide Random-MMP searches through narrow passages in semi-unstructured environments. Possibility Graphs are effective at quickly exploring the environment at a high level to identify potential routes for the robot to follow. In this paper, we equip the Possibility Graph with multiple action types, allowing it to explore regions that would be unreachable with only one type of action. We augment the definition of the Possibility Graph from [18] with a set of available action types to get Def. 1:

Definition 1: A Possibility Graph is a tuple

$$PG = (\text{actions}, \mathscr{E}, \Gamma_{PG} = (V, E), Q_{Confirm}) \quad (1)$$

where,

- `actions` is the set of available action types,
- $\mathscr{E}$ is the *exploration space* for the graph (see Sec. II-B),
- Γ_{PG} is a graph consisting of vertices V and edges E,
- V is a set of vertices which are elements of $\mathscr{E}$,
- E is a set of directed edges that transition between vertices using actions,
- $Q_{Confirm}$ is a queue which manages confirmation jobs.

Example 1: In Fig. 2 we show a Possibility Graph for a toy problem. The `actions` available to the stick figure are walking forward, crawling forward, and transitioning between walking and crawling. Γ_{PG} can be seen in the lower panel: green edges and vertices belong to the walking action, blue edges and vertices belong to the crawling action, and orange edges are transitions where the robot is kneeling down or standing up.

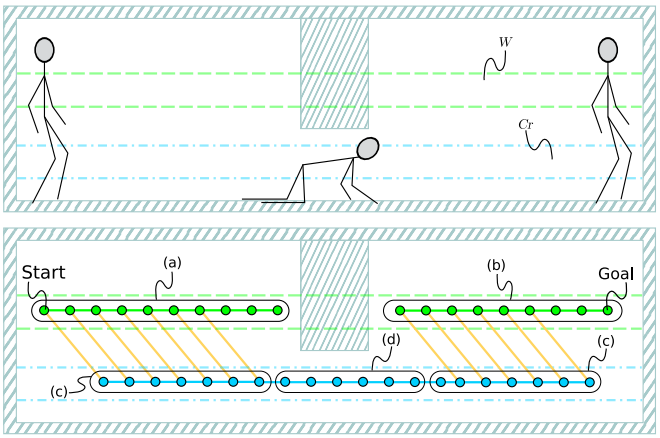


Fig. 2: Cartoon showing a simple 2D stick-figure example where the stick figure can walk or crawl forward. The graph's vertices represent the (x, z) values of a point fixed to the stick figure's chest. The upper region, marked by W in the top photo, is where walking is valid. The lower region, Cr, is where crawling is valid. (a) We extend from the start vertex towards a randomly sampled point in the center. [Alg. 4, line 7] (b) We extend from the goal vertex towards the last vertex that was created in the previous step. [Alg. 4, 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].

A. Sufficient vs. Necessary Conditions

The defining feature of the Possibility Graph is the decomposition of an action's feasibility constraints into "Sufficient Conditions" and "Necessary Conditions". If an element of the graph satisfies the action's sufficient conditions, we can label it with "Definitely Possible". If an element violates the necessary conditions, then that element is "Definitely Impossible", and we exclude it from the graph. Otherwise, the element is labeled as "Indeterminate", because the necessary conditions are designed to be lax—they do not fully evaluate whether an element is feasible.

The sufficient conditions and necessary conditions can each define an explorable manifold as described in [18]. As we build the Possibility Graph, the vertices of the graph will exist in the manifolds of these conditions instead of in the feasibility constraint manifold.

Example 2: In Fig. 2 we show the necessary condition manifolds of walking and crawling. The stick figure must be standing in order to walk, so any vertices that are to be used for a walking action must lie between the dotted green lines. The stick figure must be on the ground in order to crawl, so only vertices that are between the dotted blue lines can be used for crawling.

Algorithm 1: Finding a path using multiple action types

```
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_Confirm.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_Confirm,
                   actions) then
18                     return Γ_path;
19         t ← CurrentTime();
20     return null;
```

B. Exploration Space

A key advantage of the Possibility Graph is it allows us to reduce the dimensionality of the problem's search space. The Possibility Graph is constructed based on the sufficient and necessary conditions of the actions that it is equipped with; therefore, it only needs to search over the parameters which are used as input arguments by the sufficient and necessary conditions of the available action types. We call this the "Possibility Exploration Space" and denote it as $\mathscr{E}$. This space is defined as the union of all the parameters that act as input arguments to the sufficient and necessary conditions of all available action types. Exploring the union of these parameters provides us with a unified space in which to explore the possibilities of multiple actions side-by-side.

In the toy problem of Fig. 2, the walking, crawling, and transitioning conditions can be defined as functions of the (x, z) location of a point rigidly attached to the stick figure's chest. This makes $\mathscr{E} = \{x \in \mathbb{R}^2 \mid b_{\text{lower}} \leqslant x \leqslant b_{\text{upper}}\}$ where b_{lower} and b_{upper} are the lower and upper bounds of the environment. The parameter z allows us to determine whether the stick figure may walk, crawl, or transition from a given vertex. x lets us determine whether the stick figure would be in collision with the obstacle in the middle, and allows us to track whether we have progressed from the start to the goal. Later, when dealing with full 3D robots, we will use the SE(3) transform of a frame rigidly attached to the robot's pelvis for $\mathscr{E}$.

C. Exploring Possibilities

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. The procedure for finding solutions with the graph is described by Alg. 1, which is an augmented version of the `FindPath` algorithm of [18].

The important feature of this augmented multi-action version of the 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. 2.

Algorithm 2: Confirm a path

```
1  Function ConfirmPath(Γ_PG, Γ_path, Q_Confirm, actions)
2      pathConfirmed ← true;
3      for edge in Γ_path.E do
4          edgeConfirmed ← false;
5          for a in actions do
6              if a.C_S(edge) then
7                  edgeConfirmed ← true;
8              else if a.C_N(edge) then
9                  Γ_PG.remove(edge);
10                 Q_Confirm.insert(
                   a.SpawnConfirmationJob(edge) );
11         if not edgeConfirmed then
12             pathConfirmed ← false;
13     return pathConfirmed;
```

Algorithm 3: 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};
```

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. 2. Each action type is 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, $Q_{Confirm}$. 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.

III. Action Specifications

For the Possibility Graph to explore actions, we need to fully define each action type. Table I 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. 3. 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 on what kind of action it is. In this paper we use three actions: walk, crawl, and standing long jump. These are divided into two categories: holonomic and nonholonomic.

Holonomic actions are expanded using Alg. 4. When we describe an action as "holonomic" in this context, we mean that its sufficient/necessary condition manifold allows edges to branch into any direction at any time, just like the dynamics of a holonomic system. Even if the true physical dynamics of the action are nonholonomic, it can be treated as holonomic by the Possibility Graph if its necessary/sufficient condition manifold is simplified to behave holonomically within the exploration space, $\mathscr{E}$. Alg. 4 shows how the possibilities of holonomic actions are expanded. This is a modified version of the `GrowTowards` function from [18] where the action type now keeps track of which vertices and edges it constructs by storing them in subgraphs.

Nonholonomic actions are expanded in a more complex way than holonomic actions, as shown in Alg. 5. Nonholonomic actions generally cannot move directly towards a target, 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. 5, 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. 5, 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 III-B describes this for the jump action.

A. Walk and Crawl

The walking and crawling actions are formulated very similarly to each other. These are the same conditions used for walking by [18]. Sufficient conditions for walking and crawling are holonomic, and include these simplifications:

1) The swept geometries in Fig. 3 must not be in collision with the environment.
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. 3.
3) The root must be in the "nominal" orientation of the action (upright for walking and pitched backwards 80° for crawling).

The necessary condition is easier to satisfy: We use only the collision geometry seen in Fig. 3c, because all other bodies depend on joint parameters which are not included in $\mathscr{E}$.

Algorithm 4: Growing the graph for a holonomic action

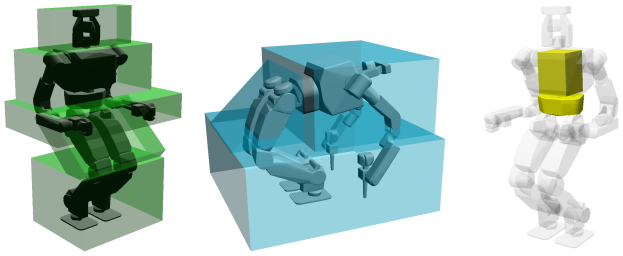
```
1  Function Holonomic::GrowTowards(p_target)
2      Q_closest ← new SortedVertexQueue;
3      for g in Γ_a.SubGraphs do
4          v ← g.FindClosestVertexTo(p_target);
5          Q_closest.insert(dist(v, p_target), v);
6      v_0 ← Q_closest.pop_front();
7      {V_new, E_new} ← Connect(v_0, p_target);
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};
```

B. 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 4 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:

Algorithm 5: Growing the graph for a nonholonomic action

```
Function Nonholonomic::GrowTowards(p_target)
    {V_new, E_new} ← {new VertexQueue, new EdgeQueue};
    Q_closest ← new SortedVertexQueue;
    for v in Γ_a.V do
        Q_closest.insert(dist(v, p_target), v);
    Useful ← new BooleanArray(Γ_a.V.size(), true);
    for v in Q_closest do
        if not Useful[v] then continue ;
        if not UpstreamFromGoal(v) then
            v_launch ← FindLaunchPoint(v, p_target);
            v_landing ← ExtendTowards(v_launch, p_target);
            edge ← Edge(v_launch, v_landing);
            if C_N(edge) then
                {V_new, E_new}.append({v_launch, v_landing, edge});
                SetUpstreamVerticesToFalse(v, Useful);
        if not DownstreamFromStart(v) then
            v_landing ← FindLandingPoint(v, p_target);
            v_launch ← ReverseExtend(v_landing, p_target);
            edge ← Edge(v_launch, v_landing);
            if C_N(edge) then
                {V_new, E_new}.append{v_launch, v_landing, edge};
                SetDownstreamVerticesToFalse(v, Useful);
    Γ_a.append({V_new, E_new});
    return {V_new, E_new};
```



(a) (b) (c)

Fig. 3: The nominal configurations used for (a) walking and (b) crawling, with their swept geometries surrounding them. (c) shows the collision geometry for the necessary conditions in yellow.

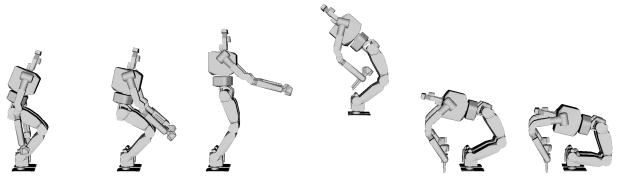


Fig. 4: An example standing long jump trajectory. The robot begins from a standing configuration, swings its arms, and jumps forward. It plans out its angular momentum so that it is able to land in a crawling configuration. After hitting the ground, it absorbs some of the impact by letting its joints behave elastically.

TABLE I: 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)

1) The jump must begin from a valid walk vertex.
2) The jump must finish at 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.

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 jump towards 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, and (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.

IV. Experiments

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. 5. The robot must jump twice to get from the right side to the left.

In Table II 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 or of 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.

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 or 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.

V. Conclusions

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 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 [19], [20] 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 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.

TABLE II: 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. "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	Success
Three Routes (a)	1	0.088 (0.048)	8.47 (0.81)	100%
Three Routes (a)	2	0.134 (0.076)	8.75 (0.91)	100%
Three Routes (a)	3	0.484 (0.450)	7.52 (1.86)	100%
Three Routes (b)	2	0.152 (0.112)	9.23 (1.09)	100%
Three Routes (b)	3	0.561 (0.502)	7.59 (2.30)	100%
Three Routes (c)	3	1.210 (0.218)	5.73 (1.79)	100%
Hallway	3	3.67 (11.52)	8.29 (0.84)	96%
Double Jump	3	1.48 (0.34)	4.32 (0.28)	100%

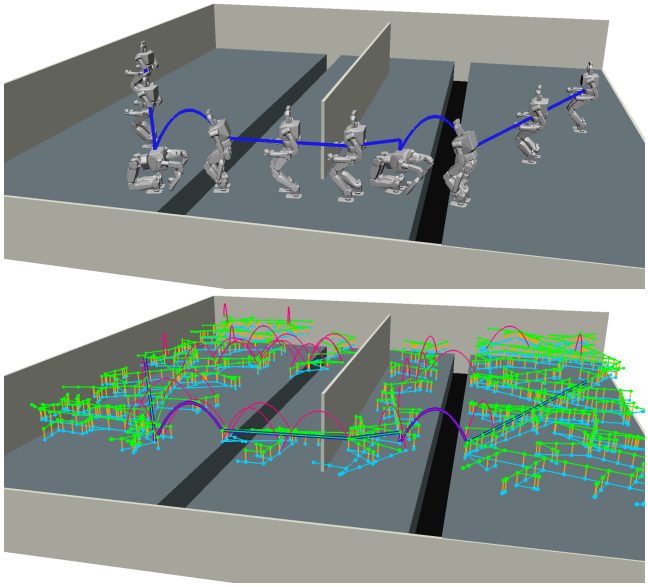
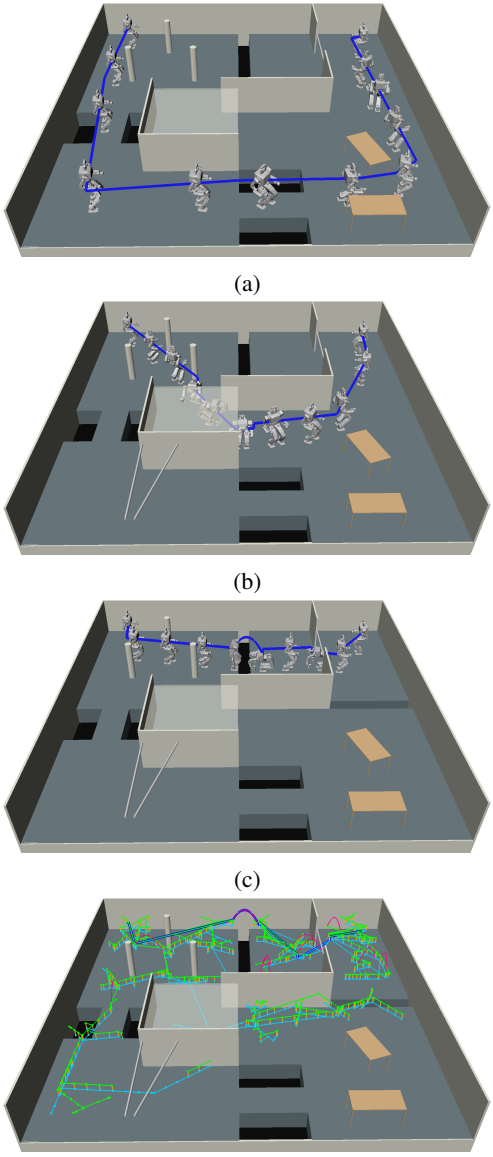


Fig. 5: The "double jump" scenario. The robot must jump across two gaps and navigate around a wall in the middle to get from the right side to the left side.



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

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.

Acknowledgments

This work was supported by DARPA grant D15AP00006.

References

[1] L. E. Kavraki, M. N. Kolountzakis, and J.-C. Latombe, "Analysis of probabilistic roadmaps for path planning," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 1, pp. 166–171, 1998.

[2] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, vol. 2. IEEE, 2000, pp. 995–1001.

[3] D. Hsu, J.-C. Latombe, and R. Motwani, "Path planning in expansive configuration spaces," in *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, vol. 3. IEEE, 1997, pp. 2719–2726.

[4] D. Berenson, S. Srinivasa, D. Ferguson, and J. Kuffner, "Manipulation planning on constraint manifolds," in *IEEE International Conference on Robotics and Automation (ICRA '09)*, May 2009, pp. 625–632.

[5] K. Hauser and J.-C. Latombe, "Multi-modal motion planning in non-expansive spaces," *The International Journal of Robotics Research*, 2009.

[6] K. Hauser and V. Ng-Thow-Hing, "Randomized multi-modal motion planning for a humanoid robot manipulation task," *The International Journal of Robotics Research*, vol. 30, no. 6, pp. 678–698, 2011.

[7] J. Garimort, A. Hornung, and M. Bennewitz, "Humanoid navigation with dynamic footstep plans," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 3982–3987.

[8] A. Hornung, A. Dornbush, M. Likhachev, and M. Bennewitz, "Anytime search-based footstep planning with suboptimality bounds," in *2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*. IEEE, 2012, pp. 674–679.

[9] S. Candido, Y.-T. Kim, and S. Hutchinson, "An improved hierarchical motion planner for humanoid robots," in *Humanoids 2008-8th IEEE-RAS International Conference on Humanoid Robots*. IEEE, 2008, pp. 654–661.

[10] J. J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue, "Footstep planning among obstacles for biped robots," in *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, vol. 1. IEEE, 2001, pp. 500–505.

[11] J. Kuffner, S. Kagami, K. Nishiwaki, M. Inaba, and H. Inoue, "Online footstep planning for humanoid robots," in *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, vol. 1. IEEE, 2003, pp. 932–937.

[12] N. Perrin, O. Stasse, L. Baudouin, F. Lamiraux, and E. Yoshida, "Fast humanoid robot collision-free footstep planning using swept volume approximations," *IEEE Transactions on Robotics*, vol. 28, no. 2, pp. 427–439, 2012.

[13] J. Chestnutt, J. Kuffner, K. Nishiwaki, and S. Kagami, "Planning biped navigation strategies in complex environments," in *IEEE Int. Conf. Hum. Rob., Munich, Germany*, 2003.

[14] J. Pettré, J.-P. Laumond, and T. Siméon, "A 2-stages locomotion planner for digital actors," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association, 2003, pp. 258–264.

[15] M. Zucker, N. Ratliff, A. D. Dragan, M. Pivtoraiko, M. Klingensmith, C. M. Dellin, J. A. Bagnell, and S. S. Srinivasa, "Chomp: Covariant hamiltonian optimization for motion planning," *The International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1164–1193, 2013.

[16] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Permenter, T. Koolen, P. Marion, and R. Tedrake, "Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot," *Autonomous Robots*, vol. 40, no. 3, pp. 429–455, 2016.

[17] A. Hereid, E. A. Cousineau, C. M. Hubicki, and A. D. Ames, "3d dynamic walking with underactuated humanoid robots: A direct collocation framework for optimizing hybrid zero dynamics," in *IEEE International Conference on Robotics and Automation (ICRA). IEEE*, 2016.

[18] M. X. Grey, A. D. Ames, and C. K. Liu, "Footstep and motion planning in semi-unstructured environments using possibility graphs," in *ICRA'17. IEEE International Conference on Robotics and Automation, 2017 (Submitted)*. IEEE, 2017, available at http://www.prism.gatech.edu/~mlooby3/icra-walk.pdf.

[19] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Backward-forward search for manipulation planning," in *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*. IEEE, 2015, pp. 6366–6373.

[20] M. X. Grey, C. R. Garrett, C. K. Liu, A. Ames, and A. L. Thomaz, "Humanoid manipulation planning using backward-forward search," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2016. IROS 2016*, 2016.