A Practical Comparison of Motion Planning Techniques for Robotic Legs in Environments with Obstacles

Tristan B. Smith
Mission Critical Technologies, Inc.
NASA Ames Research Center
Moffett Field, CA 94035–1000
Email: tristan.b.smith@nasa.gov

Daniel Chavez-Clemente
Department of Aeronautics and Astronautics
Stanford University
Stanford, CA 94305–4035
Email: dchavez@stanford.edu

Abstract

ATHLETE is a large six-legged tele-operated robot. Each foot is a wheel; travel can be achieved by walking, rolling, or some combination of the two. Operators control ATHLETE by selecting parameterized commands from a command dictionary. While rolling can be done efficiently, any motion involving steps is cumbersome - each step can require multiple commands and take many minutes to complete. In this paper, we consider four different algorithms that generate a sequence of commands to take a step. We consider a baseline heuristic, a randomized motion planning algorithm, and two variants of A search. Results for a variety of terrains are presented, and we discuss the quantitative and qualitative tradeoffs between the approaches.*

1. Introduction

ATHLETE (All-Terrain Hex-Limbed Extra-Terrestrial Explorer) is a large six-legged robot developed at the Jet Propulsion Laboratory (see Figure 1). ATHLETE is a flexible platform designed to serve multiple roles during manned and unmanned missions to the moon, including transportation, construction and exploration. It is intended to be remotely operated from earth or by astronauts on the moon.

ATHLETE has six legs attached to a hexagonal chassis and is omni-directional. Each of ATHLETE's six legs has hip, knee, and ankle joints (each with 2 degrees of freedom), resulting in 36 degrees of kinematic freedom. At the end of each leg is a multi-purpose wheel. The wheel can be locked to serve as a foot or unlocked to roll; it can also be used to operate tools using a mechanical adaptor.

In the three years since prototype ATHLETE (1/2 scale) robots became operational, a wide array of capabilities have been demonstrated ([1], [4]). ATHLETE can roll on smooth terrain, combine walking with rolling to traverse uneven terrain and even climb ledges. It can manipulate tools, rappel down a steep slope, and coordinate with other robots.

Figure 1. ATHLETE on a hillside.

Operators use laptops to send parameterized commands to ATHLETE. Rolling can be commanded efficiently; a single command can direct the robot to travel long distances. However, any motion involving steps is cumbersome - a single step can require multiple commands and take many minutes to complete, with much of the time spent deciding how to proceed. For example, here is a sequence of commands that might be used to take a simple step over a rock:

- Raise foot 50 cm.
- Rotate hip joint 60 degrees.
- Rotate hip joint 10 degrees.
- Lower foot 50 cm.
- Lower foot 5 cm.

Notice that to produce this plan the operator must:

- Figure out a feasible sequence of commands. This can requires trial and error; even in relatively benign terrain, it is common for the robot to refuse operator commands due to physical constraints or safety limits it is forced to obey.
- Guess appropriate magnitudes for those moves. In the above example, it would have been faster to use a single Rotate of 70 degrees (and a single Lower of 35 cm), but this is only apparent in hindsight.

We are involved in work to make ATHLETE operation faster and more efficient by automatically suggesting sequences of commands to an operator; first to achieve individual steps, and subsequently to achieve multi-step

walking. The specific goal of this paper is to evaluate four diffent algorithms for generating a single-step sequence of commands.

Three of our four algorithms search *configuration space ("C-Space")*. Each dimension in configuration space represents the range of angles for one of ATHLETE's joints. A path through configuration space represents a sequence of moves (changes in joint angles) the robot can make to get from one configuration to another.

Our first algorithm only tries the straight line between the start and end configurations, our second is a standard randomized motion planning algorithm, and our third is an A* search through a discretization of configuration space.

Our fourth and final approach is A* search in *task space*, the three-dimensional Euclidean space in which the robot operates. Notice that in the example command sequence, the Rotate command moves the robot in configuration space, while the Raise/Lower commands are done in task space.

In Section 2, we explain the four algorithms used. Section 3 describes our experiments, while Sections 4 and 5 present results. Section 6 discusses related work, Section 7 suggests future directions, and we conclude in Section 8.

2. Algorithms

2.1. Preliminaries

The goal for each of our algorithms is to produce a sequence of commands to move an ATHLETE foot from one location to another. We assume that the position and orientation of the chassis remain fixed, and therefore that we can ignore the configuration of the other five legs.[1] This simplification means we are only concerned with the six-dimensional configuration space representing the joint angles shown in Figure 2.

We can represent the location of the foot as either:

- A six-tuple in configuration space, c_i, or
- A three-tuple in task space, xyz_i.

In addition, we have functions, *TO-TSPACE*(c_i, leg_j) and *TO-CSPACE*(xyz_i, leg_j), that convert between the two spaces via the forward or inverse kinematics of the leg. Note that, while one location for the foot in task space, xyz_i, could correspond to many different configurations, our implementation of *TO-CSPACE*(xyz_i, leg_j) is one-to-one and always computes the same c_i for a given xyz_i.

Finally, we have a function *COLLISION-FREE*(c_i, c_{i+1}) that determines whether the straight line in configuration space between c_i and c_{i+1} is free of collisions; the leg must not collide with itself, other parts of the robot, or the terrain.

As problem input, we assume:

1. Although it might be necessary in tight space to adjust other legs or the chassis in order to reach a goal, we consider such motions part of multi-step walking and do not include them here.

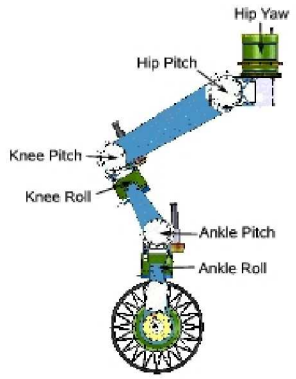

Figure 2. The joints on an ATHLETE leg.

- Terrain data. For our experiments, it is auto-generated; in reality, it is acquired with ATHLETE's 15 on-board cameras.
- The leg, leg_i, to move.
- Current position. This includes the location and orientation of the chassis, and all six joint-angles for each leg. We assume that this represents a valid and stable position on the terrain, and that ATHLETE will remain stable when leg_i is lifted.
- A goal position in task space, xyz_{goal}, for leg_i.

Given this data, we compute start and goal configurations, c_{start} and c_{goal}. We get c_{start} by lifting leg_i 10 cm above its current position, and c_{goal} is a configuration 10 cm above xyz_{goal}. We include these 10 cm buffers because a weight-bearing leg must be raised by about this much before it is truly free of the ground, due to the way the chassis will sag and the tire will expand as the leg is lifted.

The goal for each algorithm is to produce a path $(c_{start}, ..., c_{goal})$ through configuration space such that each edge (c_i, c_{i+1}) is collision free. Ultimately, the solution path is converted to a sequence of low-level commands, which move leg_i to c_{goal}. In the next four subsections, we outline our different approaches to generate the path.

2.2. Straight line approach

Our baseline algorithm, $SMPL$,[2] simply calls *COLLISION-FREE*(c_{start}, c_{goal}). If the straight line between c_{start} and c_{goal} has no collisions, it is returned as the solution path. If not, the algorithm fails.

2.3. SBL

Our second approach is a Single-query Bi-directional planner with Lazy collision checking (SBL), and is outlined in Algorithm 1.

2. We use the abbreviation $SMPL$ as shorthand for "Simple".

Algorithm 1

function SBL(c_{start}, c_{goal})

1: $T1.root = c_{start}$
2: $T2.root = c_{goal}$
3: **while** not timed out **do**
4: Execute EXPAND-TREE
5: $\tau \leftarrow$ CONNECT-TREES
6: **if** τ is not empty **then**
7: Return *success*
8: **end if**
9: **end while**
10: Return *failure*

SBL is a sampling-based motion planning technique. The search for feasible paths is conducted by sampling configurations between the start and goal, and verifying if (a) they are feasible, and (b) they can be connected without collisions.

The algorithm proceeds by growing two C-Space trees T_1 and T_2 rooted at c_{start} and c_{goal} toward each other. On every iteration one of the trees is selected at random with probability 0.5, and a new milestone m_{new} is added to it (*EXPAND-TREE* step). The planner then checks if a connection can be established between the trees (*CONNECT-TREES* step), and if so it generates a candidate path τ from c_{start} to c_{goal}. This path includes a segment called a bridge, connecting m_{new} to m, the nearest milestone in the opposite tree. If τ is found to be collision-free, success is returned. Otherwise iterations continue until time-out, at which point failure is returned. This means that either no path exists, or SBL was unable to find one.

The *EXPAND-TREE* step proceeds as follows: from the selected tree *T*, an existing milestone is selected at random with probability $\pi(m)$, which is inversely proportional to the density of milestones of *T* near *m*. Then, a collision-free configuration is randomly selected within an adaptively-chosen distance of *m*, and is added to *T* as the new milestone m_{new}. This selection strategy distributes the exploration around areas reachable from the root configurations, and at the same time prevents over-sampling. It should be noted that only m_{new} is checked for collisions at this stage, not the segment connecting it to *m*. Segment checks are postponed until they are absolutely necessary in the *CONNECT-TREES* step. This lazy approach has the effect of reducing the total number of expensive collision checks.

The *CONNECT-TREES* step of SBL is executed when the distance between m_{new} and m is smaller than or equal to the distance threshold. At this point the candidate path τ is checked for collisions to a resolution ε by a *TEST-PATH* routine, and τ is returned as the motion plan if it is collision-free; otherwise, iterations continue.

For further details on SBL the reader is referred to the original paper by Sanchez and Latombe [2].

2.4. A* Search in Configuration Space

Our third approach, CFG uses A* search [5] through the six-dimensional configuration space for leg_i. We discretize each dimension into increments of r radians, and search over the resulting grid.

Algorithm 2

function ASTAR($start$, $goal$)

1: $start.g = 0$
2: $start.h =$ DISTANCE($start, goal$)
3: $start.parent =$ NULL
4: $queue$.ADD($start$)
5: **while** n = GET-BEST-NODE($queue$) and n not NULL and not timed out **do**
6: **if** n is goal **then**
7: Return *success*
8: **end if**
9: $succs$ = GET-SUCCESSORS(n)
10: **if** n near goal **then**
11: $succs$.ADD(goal)
12: **end if**
13: **for all** s in $succs$ **do**
14: $s.g = n.g +$ DISTANCE(n, s)
15: $s.h =$ DISTANCE(s, goal)
16: $s.parent = n$
17: $queue$.ADD(s)
18: **end for**
19: **end while**
20: Return *failure*

Algorithm 2 outlines our approach.[3] A queue of nodes is initialized with a node representing the start configuration. Each node n in $queue$ stores:

- $n.g$,the distance travelled to get there,
- $n.h$, an optimistic estimate of the distance to the goal, and
- $n.parent$, the node from which n was generated.

At each step, the function *GET-BEST-NODE* returns the node n in $queue$ for which $n.g + n.h$ is lowest.[4] Then, n is expanded; *GET-SUCCESSORS* returns the twelve grid nodes (obtained by moving left or right along each of the six dimensions) adjacent to n, which are then added to $queue$ with appropriate g and h values.[5]

3. We used a slightly modified version of the C++ implementation written by Justin Heyes-Jones: http://www.geocities.com/jheyesjones/astar.html

4. This order in which nodes are explored distinguishes A* search from other graph-search algorithms, and ensures that the resulting solution will be optimal.

5. In traditional A* search, *GET-SUCCESSORS* will only return a successor s if *COLLISION-FREE*(n, s) passes. However, we have implemented a lazy version of A* described further in Section 2.6.

When *success* is returned, the solution can easily be extracted because each node stores its *parent*.

2.5. A* Search in Task Space

Finally, our fourth approach, TSK, uses A* search but over a discretized grid in three-dimensional task space. Each point xyz_i represents a position of the foot (which then has a corresponding point, *TO-CSPACE*(xyz_i), in configuration space). Algorithm 2 is still used but *GET-SUCCESSORS*(n) returns the six grid nodes in task space adjacent to n, and *DISTANCE*(n_i, n_j) computes three-dimensional Euclidean distance rather than distance in configuration space. The function *COLLISION-FREE* still checks the line between each pair of nodes in configuration space since the final commands to the robot will be configuration space moves.

Task space search is probably the most intuitive approach, as one can picture the wheel moving through the three-dimensional grid. In addition, the smaller branching factor (6 instead of 12) means the search space is exponentially smaller than that of CFG, which allows a much finer granularity to be used for the grid.[6]

The smaller search space is also a potential disadvantage to this approach. Recall that function *TO-CSPACE*(xyz_i, leg_j) is one-to-one, even though xyz_i could map to multiple configurations. Recall also that to check an edge (xyz_i, xyz_j) in task space, we still check for collisions in configuration space, using *COLLISION-FREE(TO-CSPACE*(xyz_i),*TO-CSPACE*(xyz_j)). This test might fail even if there exist other valid configurations for xyz_i and xyz_j for which the edge is collision-free. Therefore, there is the risk that this approach, even with very fine resolution, will fail to find solutions that do exist. In effect, we're searching only a portion of the configuration space searched by our other approaches.

2.6. Optimization 1: Lazy A* Search

For our domain, the computationally expensive piece of each A* implementation is the *COLLISION-FREE* function. This is different than typical A* domains, where the computation of g and/or h are most expensive. Therefore, we have implemented a lazy version of Algorithm 2 that changes two aspects of typical A* search:

1) In typical A*, *GET-SUCCESSORS* only returns a neighbor s if *COLLISION-FREE*(n, s) succeeds. Our lazy A* returns all neighbors, and therefore avoids calling *COLLISION-FREE* when a node is added.
2) As a result unreachable nodes are included in *queue*. Therefore, *GET-BEST-NODE*(*queue*), instead of simply returning the top node n in *queue*, must call *COLLISION-FREE*($n.parent, n$); if this succeeds n can be returned; if it fails, n is discarded, the next node in *queue* is considered, and so on.

This approach means there will be nodes n in *queue* that cannot be expanded because the path from n's parent has collisions. However, the same point in space with a different parent might expand successfully. Therefore, unlike traditional A*, we may need to add a single point in space to *queue* multiple times; we can only avoid adding a point if the same point has been both added *and* successfully expanded.

As a result, it is not at all obvious that this lazy approach is a good idea. On the one hand, we avoid checking edges to nodes that never end up getting expanded. On the other hand, each point in space could have multiple copies in *queue*, making the maintenance (especially sorting) of *queue* more difficult. In the worst case, where A* expands all nodes in *queue* before finding a solution, this overhead certainly makes the lazy approach more expensive.

For the experiments we describe in Section 3, the lazy version of A* is an improvement. In configuration space, between 1.3 and 9.7 (depending on the terrain) times more nodes are added on average, while search times are reduced by 43 to 82 percent on average. Similarly, in task space, between 1.2 and 4.8 times more nodes are added, while search times are reduce by 25 to 70 percent. Nonetheless, lazy A* is not always better; for 4 of the 958 instances considered, the lazy A* version of CFG times out (and therefore fails) even though the standard implementation succeeds.

2.7. Optimization 2: Path Smoothing

Because SBL is a random algorithm, and returns the first valid path found, the result can be a very inefficient and odd-looking step. For results to be acceptable to human operators a post-processing algorithm to smooth the resulting path was developed. We follow a smoothing approach similar to the one proposed in [6]. Our algorithm does the following:

1) Expand *path* into a graph by joining every pair (c_i, c_j) of vertices for which *COLLISION-FREE* (c_i, c_j) succeeds.
2) Run Dijktstra search on this graph to find the shortest path from c_{start} to c_{goal}. This has the effect of cutting off unnecessary corners in the original path.
3) Add vertices to *path* by bisecting each edge.
4) Repeat steps 1 through 3 until the improvement made in a given iteration is less than 10%.

We use smoothing to improve the paths returned by our A* algorithms as well; although they return optimal paths along the discretized grids, there are usually shorter paths that cut corners and pass diagonally through the grid. The smoothing algorithm in configuration space is outlined in Figure 3.

6. For example, doubling the granularity increases the search space size by a factor of 8 for task space, but by a factor of 64 for configuration space.

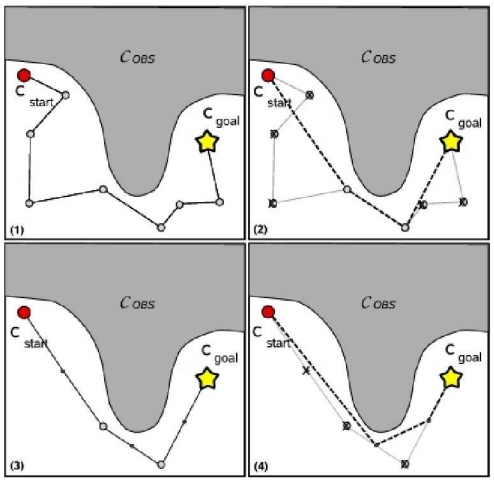

Figure 3. Path smoothing in a hypothetical 2D C-space: c_{start} and c_{goal} are separated by a C-obstacle. (1) Motion plan with N_1 nodes before smoothing; (2) shortest path found using Dijkstra's algorithm, with $N_2 \leq N_1$ nodes; (3) the simplified path is bisected, adding $N_2 - 1$ nodes; (4) Dijkstra's algorithm is re-run.

3. Experimental Setup

To compare algorithms, we generated four different types of terrain. For each, we hand-picked a set, L, of representative points on the left side of the leg, and a similar set, R, on the right. We then consider each possible pair, $(l_i \in L, r_j \in R)$, and try stepping both from l_i to r_j and vice versa, resulting in $|L| \cdot |R| \cdot 2$ problem instances for each terrain. The four terrains, three of which are shown in Figure 4, are:

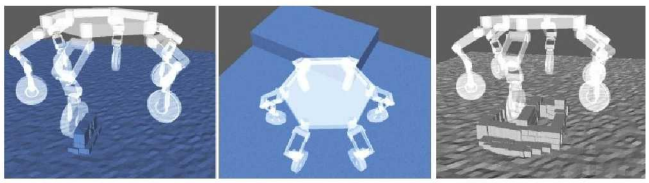

Figure 4. The *Bump*, *Step*, and *Well* terrains used in our experiments.

- *Flat*: Completely flat terrain ($|L| = |R| = 11$). This serves as a baseline.
- *Bump*: Terrain with a 40 cm bump between L and R ($|L| = |R| = 14$). This is probably the most realistic terrain; stepping with ATHLETE is most likely to be done over rocks in the lunar landscape.
- *Step*: Terrain with a 50 cm ledge; L is at the top of the ledge, and R is at the base ($|L| = 15$, $|R| = 6$).
- *Well*: Terrain with two wheel-sized wells surrounded by raised terrain, where L is in one well and R the other ($|L| = 8$, $|R| = 9$). This terrain was our attempt to generate a difficult example that was quite different than the other terrains.

This results in a total of 958 problem instances. We configure our algorithms as follows:

- Search fails if a solution is not found within 5 minutes.[7]
- We run each A* approach with two granularities. $CFG(1.0)$ and $CFG(0.33)$ use 1 radian and 0.333 radians,[8] respectively, while $TSK(0.2)$ and $TSK(0.1)$ use 0.20 m and 0.10 m, respectively. Roughly speaking, the coarser granularity was intended to make the search time comparable to SBL while the finer granularity allows better answers to be found, but more slowly.
- We attempt to reach the goal node from a search node in A* (see line 11 in Algorithm 2) if the distance to the goal is less than 2 radians in configuration space, and 40 cm in task space.
- Since each run of SBL produces a different result, we average the SBL results over 10 runs for each problem instance.

4. Experimental Results

Figure 5 shows the fraction of problem instances solved by each approach. As expected, $SMPL$ often fails and is not really a viable approach. Besides $SMPL$, there are very few failures. SBL and $TSK(0.1)$ succeed on every instance. $TSK(0.2)$ fails on 13 of the *Bump* instances. $CFG(1.0)$ cannot solve 23 of the 144 *Well* instances because no solution exists using the coarse grid, while $CFG(0.33)$ times out on 4 of the 180 *Step* instances.

Figure 5. Success ratios.

7. This seems like a long time to wait for a solution to a single step; that it has been acceptable in practice points out how time-consuming the stepping process currently is.

8. Note that the high dimensionality of configuration space forces us to use very coarse granularities in this space; 1 radian is almost 60 degrees.

Figure 6 shows average runtimes while Figure 7 shows standard deviations. Running times are small on average, with $SMPL$ obviously the fastest, and the finer granularity searches taking generally the longest. One notable exception is the case of the $Well$, where $CFG(1.0)$ is the slowest algorithm. This is almost certainly caused by an increase in the number of collision checks required to find a sequence of large C-Space swings that can maneuver within the constrained space of the well. As evidenced by Figure 7, CFG clearly suffers from the higher dimensionality of the search space which causes a higher variance in runtimes; it is the only algorithm with any instances that exceed five seconds. For example for the $Bump$ instances, $TSK(0.1)$ averages 1.17 seconds per instance, but never takes longer than 3.5 seconds. On the other hand, $CFG(0.33)$ averages 4.07 seconds per instance, but 7 of the 450 instances take longer than 30 seconds. For the $Well$, the same curious behavior is seen to influence $CFG(1.0)$ for the reasons mentioned above.

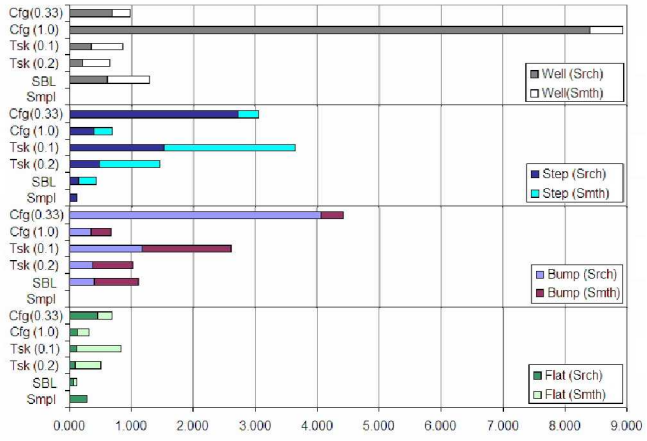

Figure 6. Runtimes, split into search and smoothing times.

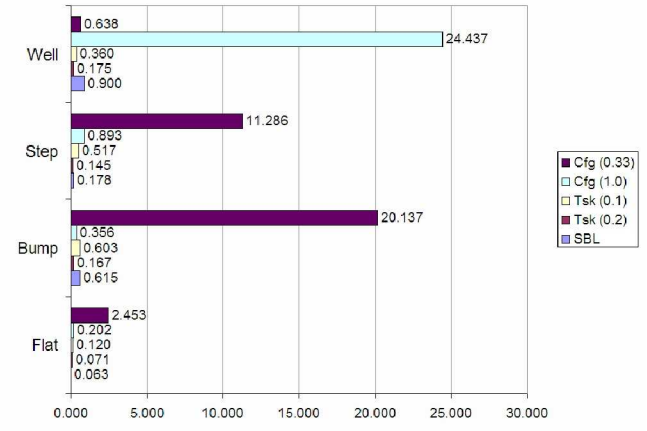

Figure 7. Runtime standard deviations.

Figure 8 shows the length of the resulting configuration space paths, before and after smoothing.[9] Surprisingly, although $CFG(0.33)$ is best, and TSK is worst, before smoothing, those results are almost completely negated by smoothing, with SBL doing best for three of the four terrains. Smoothing also helps TSK become competitive on this metric, although it still does poorly on the $Bump$ terrain.

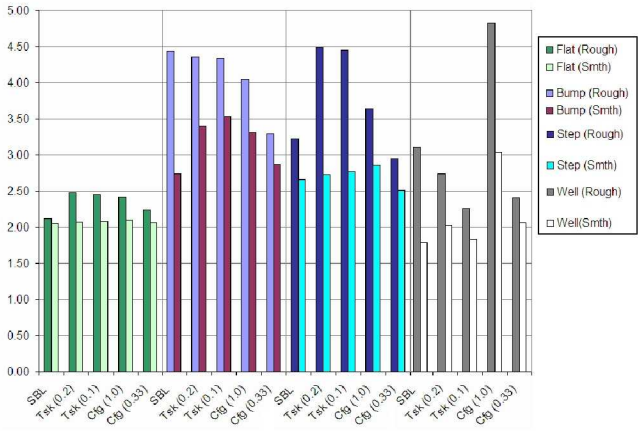

Figure 8. Configuration space distances.

Figure 9 shows the average distances in task space for each approach. Here, $TSK(0.1)$ is the clear winner, outperforming all other algorithms on all data sets, before and after smoothing. Finally, Figure 10 shows the maximum values for each terrain, confirming that SBL and $CFG(1.0)$ occasionally produced very long paths, even after smoothing.

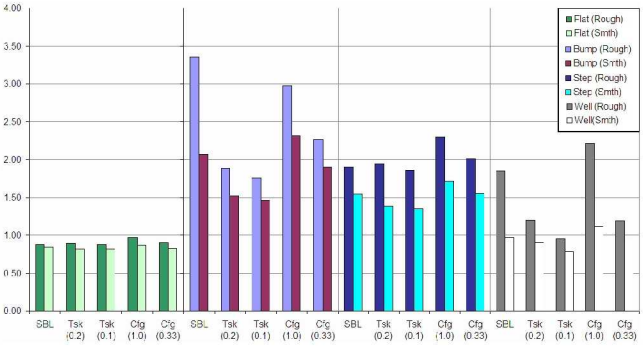

Figure 9. Task space distances.

5. Operator Preferences

Originally, SBL without smoothing was demonstrated to ATHLETE operators. It was immediately clear that the

9. For the rest of the results presented here, we exclude $SMPL$; because it only succeeds on the easiest instances, results for that algorithm are skewed.

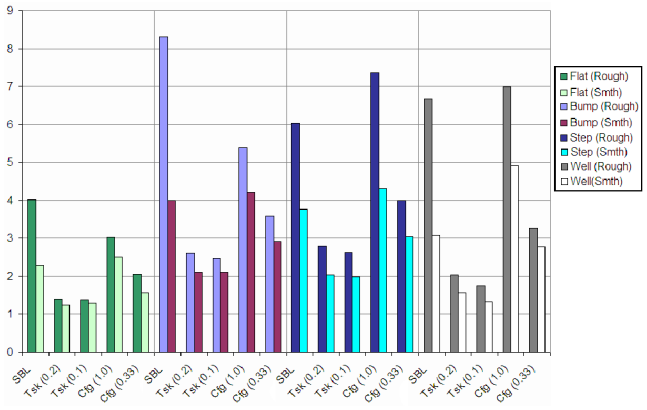

Figure 10. Maximum distances in task space for each terrain.

results were unacceptable; as suggested by Figure 10. Large moves would be made to traverse small distances and steps would look awkward and unnatural. Smoothing was quickly implemented, resulting in an algorithm that operators accept.

However, a certain amount of hesitance exists when using SBL. One reason is that it still produces large, awkward steps occasionally. An equally important reason is the lack of predictability in the randomized approach. ATHLETE is large, weighs almost a ton, and is very expensive; safety of the robot and those around it is the primary concern, especially when it is operated in a remote location. For good reason, operators are more comfortable with positions and movements they have seen before. If there are two equally-efficient path to get from a to b, operators would like software that returns the same path each time using a straightforward algorithm; this allows them to become familiar and comfortable with the software and the decisions it makes.

These concerns motivated our research. CFG was developed first, and demonstrated to ATHLETE operators, using the finer granularity (0.33). This approach was well-received, although operators have not yet had enough experience with it to determine if the occasional long running time is of concern. In simulation, the coarse granularity CFG was found to produce larger swings than desirable when stepping over some medium-sized obstacles. This prompted the use of the finer-granularity version on the robot, since no post-planning smoothing had been applied to CFG at the time.

Given the results of this work, we suspect that $TSK(0.1)$ will utimately be the approach chosen. It was able to solve all instances and ran quickly. Most importantly, it produced the shortest paths in task space. We suspect this is the most important metric for gauging human preferences; because the goal is to get a foot from a to b, we believe that a path that minimizes the movement of that foot will be preferred by operators.

6. Related Work

The A* algorithm has seen widespread use in the field of robotics, and is explained in detail in a number of publications, e.g. [5]. Several variants and derivatives have been developed, and an excellent overview of these is provided in [7].

For problems with high-dimensional configuration space sampling-based techniques have become increasingly popular. The techniques can be roughly divided in two groups: single-query and multiple-query algorithms. Again, a number of these techniques exist, with [8], [9] providing good summaries and tips regarding applicability to specific problems.

Practical comparisons of planning techniques are rare. Of special interest is [10], which compares A* with RRT-Connect as applied to the problem of motion planning for reconfigurable robots. RRT-Connect is a bi-directional technique like SBL, and their findings indicate a similar advantage in processing speed of RRT over A*. The quality of the RRT solutions, while poor, are also improved via smoothing.

Finally, for a very complete and accessible open-source library of motion planning algorithms we refer the reader to the OOPSMP project at Rice University [11].

7. Future Work

The most important next step is to allow ATHLETE operators to experiment with the different approaches, especially TSK, and determine the benefits and drawbacks to each, from their perspective.

In addition, all four of our approaches are simple, and there are ways that they could be tuned to produce better results. For example, if A* search perfoms line 11 of Algorithm 2 for every expanded node, we have observed better results for most runs, but slower runtimes for the most difficult instances. An alternative would be to try c_{goal} with some small probability.

Notice that we treat all dimensions in configuration space equally. In reality, a move of 1 radian at the hip joint swings the entire leg around, while the same magnitude rotation at the wheel joint leaves most of the leg in place. Therefore, we should individually scale each dimension of configuration space to reflect these differences. This scaling would be used for measuring configuration space distance, and would also push CFG towards 'small' moves, since the algorithm minimizes this distance.

Our results suggest that combining our algorithms into a hybrid approach would yield better results than any individual approach. For example, the data here suggests we should simply try $SMPL$ before another algorithm in case it finds an answer. Even better would be to give each approach a

small amount of time (5 seconds, perhaps), and use the best result.

In this work, we assume the chassis is already in an appropriate position for the step. In practice, a chassis shift might be necessary to perform a step, either to redistribute weight, or to raise (or lower) the chassis in order for the leg to reach the goal. We could expand any of our approaches to include search over possible chassis positions.

Ideally, we should also include consideration of dynamics and related safety concerns directly into search. Such considerations will be especially important when ATHLETE is carrying heavy cargo.

Finally, some very interesting work remains to be done on the subject of "natural-looking" steps. This is a challenging problem largely because it is difficult to establish and objective measure of the visual appeal of a given motion. While it has been suggested that minimal-energy paths might look more natural, we have yet to explore this through experiments. The argument for energy minimization is bio-inspired, but given the mechanical differences between its limbs and those of living organisms, the question of how closely a robot like ATHLETE can mimic natural behaviors remains unanswered.

8. Conclusion

We have outlined four different algorithms for taking a step with ATHLETE. Three approaches, $SMPL$, SBL, and CFG, search in configuration space while TSK searches in task space. We tried each algorithm on a total of 958 problem instances spread across 4 types of terrain, and then considered the advantages and disadvantages of each.

As expected $SMPL$ is extremely fast, but untenable due to its high failure rate. SBL also runs quickly, but produces a wide range of path lengths in both configuration and task spaces; these variances are only partially muted by the post-SBL smoothing.

CFG produces short paths in configuration space, but suffers from the high dimensionality of its search space; the fine-grained version can run for minutes on difficult instances, while the faster version is too coarse to get good results.

TSK results are most consistent; the finer grained version is the only approach other than SBL to solve all instances, runtimes are comparable to SBL, and configuration space distances are only slightly worse than other methods. TSK consistently get the shortest task space distance, arguably the most important metric.

The motivation of this work was to explore alternative to SBL because operators are uncomfortable with the randomness, and occasional bad results, of that approach. CFG has already been well recieved by operators, despite its drawbacks. Due to our experimental results, we suspect that TSK will becomed the approach of choice.

Acknowledgment

The authors would like to thank Vytas Sunspiral for his early interest in the research, David Smith and Javier Barreiro for their input, and David Mittman at JPL for supporting our time with ATHLETE. Some of the SBL libraries were kindly provided by Kris Hauser at Stanford University.

References

[1] Brian H. Wilcox et al., ATHLETE: A cargo handling and manipulation robot for the moon, *Journal of Field Robotics*, 24(5), pp. 421–434, 2007.

[2] Gildardo Snchez and Jean-Claude Latombe, A Single-Query Bi-Directional Probabilistic Roadmap Planner with Lazy Collision Checking. R.A. Jarvis and A. Zemlinsky (Eds): *Robotics Research*, STAR 6, pp. 403–407, 2003.

[3] Kris Hauser et. al., Motion Planning for a Six-Legged Lunar Robot. *Proceedings of the Workshop on the Algorithmic Foundations of Robotics*, 2006.

[4] Matthew Heverly and Jaret Matthews, A Wheel-on-limb Rover for Lunar Operation, iSAIRAS08, 2008.

[5] Stuart Russell and Peter Norvig, Artificial Intelligence, A Modern Approach (2nd. Ed), pp. 97–101, 2003.

[6] Jayesh N. Amin et. al., A Fast and Efficient Approach to Path Planning for Unmanned Vehicles, *Proceedings of the AIAA Guidance and Control Conference and Exhibit*, Keystone, CO, August 2006.

[7] Dave Ferguson et al., A Guide to Heuristic-based Path Planning, *AAAI Journal*, 2005.

[8] Roland Geraerts and Mark H. Overmars, A Comparative Study of Probabilistic Roadmap Planners, *Algorithmic Foundations of Robotics V*, STAR 7, pp. 43-57, 2004.

[9] Stefano Carpin, Randomized Motion Planning – A Tutorial, *International Journal of Robotics and Automation*, Vol 21-3, 2006.

[10] David Brandt, Comparison of A* and RRT-Connect Motion Planning Techniques for Self-Reconfiguration Planning, *Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Beijing, China, Oct 9-15, 2006.

[11] Erion Plaku et. al., OOPS for Planning, An Online, Open-source Programming System. *IEEE International Conference on Robotics and Automation (ICRA)*, 2007.