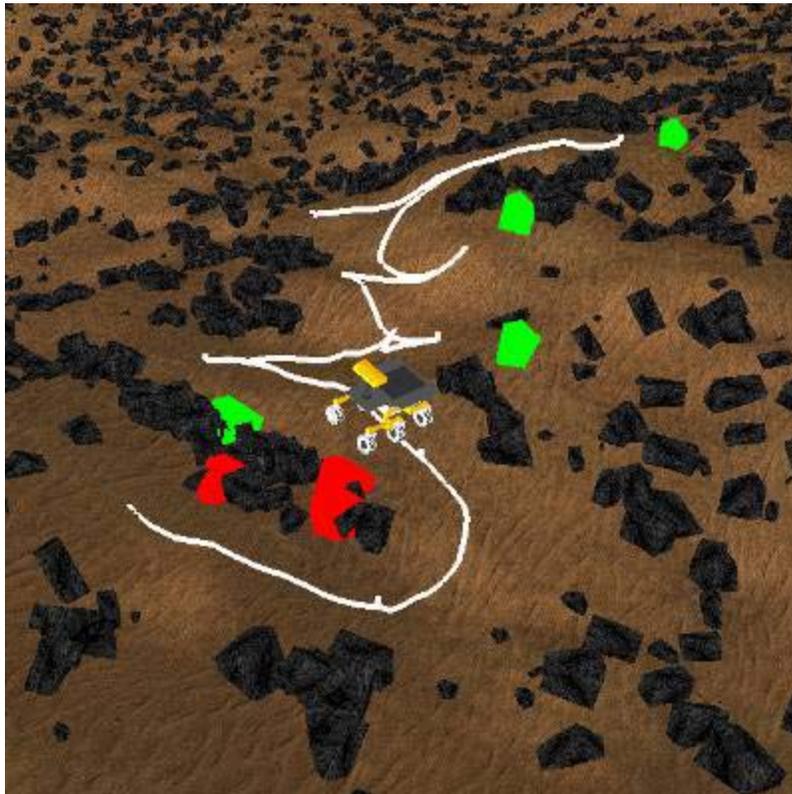


VERY ROUGH TERRAIN NONHOLONOMIC TRAJECTORY GENERATION AND MOTION PLANNING FOR PLANETARY ROVERS



LATTICE PLANNER DOCUMENTATION

CONTACT(S): **Mihail Pivtoraiko** (mihail@cs.cmu.edu)
 Thomas Howard (thoward@cs.cmu.edu)
 Ross Knepper (rak@cs.cmu.edu)
 Dr. Alonzo Kelly (alonzo@ri.cmu.edu)

REVISION DATE: **June 27th, 2008**

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Very Rough Terrain Regional Motion Planning
DOCUMENT TYPE Lattice Planner Schedule Document
AUTHOR(S) Mihail Pivtoraiko, Dr. Alonzo Kelly
LAST REVISION DATE June 27th, 2008

PROJECT OBJECTIVE

The Regional Motion Planner shall be implemented, validated using research rover prototypes and delivered to JPL. The planner shall be capable of computing motions that allow a rover to traverse very rough terrain environments and achieve distant goals. The delivered planner presents this capability, while featuring real-time computation, in the sense that it can execute live on-board a moving research prototype rover, as demonstrated at the JPL Mars Yard. The planner features pre-computing key data off-line, to allow fast on-line operation onboard the robot.

PROJECT SCHEDULE

Trajectory Generator	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16
	CY 2006	CY 2007			CY 2008			
Obstacle Avoidance	█							
Dynamic Edge Updates		█						
Sliding Mechanisms		█						
Pose Lattice Dstar			█					
Requirements and Concept Design			█					
Regional Planner Rewrite				█				
Regional Planner Testing					█			
CLARATy Integration						█		
Regional Planner Detailed Design, User Guide, and Test Documents						█		
Final Report							█	

PROJECT SCHEDULE DETAILS

Obstacle Avoidance (Q9): Develop a body-fixed lattice planner for avoiding obstacles.

Dynamic Edge Updates (Q10): Develop mechanism to update costs of edges with new perception information.

Sliding Mechanisms (Q10): Develop mechanism to update costs of edges as vehicle moves.

Pose Lattice Dstar (Q11) : Implement Dstar on the pose lattice.

Requirements and Concept Design (Q11): Produce a document describing requirements and concept design.

Regional Planner Rewrite (Q12): Rewrite the regional planner design and software for performance and robustness.

Regional Planner Software Testing (Q13): Test the rewritten trajectory generator software by measuring performance, convergence, memory requirements, etc....

CLARATy Integration (Q14): Integration the revised regional planner software into CLARATy and perform field experiments to validate the design.

Regional Planner Detailed Design, User Guide, and Test Documents (Q14): Develop and Deliver the CLARATy integration detailed design, user guide, and test documents.

Final Report (Q15): Final program report.

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Very Rough Terrain Regional Motion Planning
DOCUMENT TYPE Lattice Planner Requirements Document
AUTHOR(S) Mihail Pivtoraiko, Dr. Alonzo Kelly
LAST REVISION DATE June 27th, 2008

PROJECT OBJECTIVE

The Regional Motion Planner shall be implemented, validated using research rover prototypes and delivered to JPL. The planner shall be capable of computing motions that allow a rover to traverse very rough terrain environments and achieve distant goals. The delivered planner presents this capability, while featuring real-time computation, in the sense that it can execute live on-board a moving research prototype rover, as demonstrated at the JPL Mars Yard. The planner features pre-computing key data off-line, to allow fast on-line operation onboard the robot.

PROJECT REQUIREMENTS

- | | |
|-----------------------------|--|
| FUNCTIONAL
REQUIREMENTS | <ul style="list-style-type: none"> • The lattice planner shall generate a motion plan between arbitrary boundary states in the predefined search space and return a trajectory in the form of a sequence of motion primitives. (R-1) • The lattice planner shall accommodate/incorporate several different search engines (e.g. A*, D*, RRT). (R-2) • The lattice planner shall produce a motion plan in an anytime manner. (R-3) • The lattice planner shall incorporate a facility to graduate the fidelity of a vehicle model as a function of distance from the start state. (R-4) • The search engine shall support heuristics (e.g. HLUt, function, or a different planner) in order to improve motion planning efficiency. (R-5) • The lattice planner shall be able to generate motion plans for search spaces fixed to the body or the world frame. (R-6) • It shall be possible to adapt the lattice planner to incorporate dynamic limitations of different vehicle models in the form of a motion template. (R-7) • The lattice planner shall be configured to plan long distance maneuvers provided a prior model of the environment or to continuously replan short actions based on perceptual information. (R-8) • The lattice planner's replanning mode shall remain convergent as the vehicle moves under the assumption of a static environment. (R-9) |
| PERFORMANCE
REQUIREMENTS | <ul style="list-style-type: none"> • The memory footprint of the lattice planner shall not exceed 300 MB. (R-10) • The lattice planner average runtime for plans over average distance 100 map cells shall not exceed 10 seconds. (R-11) • The lattice planner maximum runtime for plans over average distance 200 map cells shall not exceed 20 seconds. (R-12) • The heuristic lookup table generation program must generate a heuristic lookup table in 30 minutes. (R-13) • The template generation program must generate a motion template in 30 minutes. (R-14) • The lattice planner initialization time must not exceed 30 seconds. (R-15) |
| INTERFACE
REQUIREMENTS | <ul style="list-style-type: none"> • The lattice planner shall accommodate variable resolution of the state space. (R-16) • The lattice planner shall receive a uniformly sampled two-dimensional array of floating point values (representing the cost of occupying a state) for a cost map. (R-17) • The lattice planner shall receive a geometry representation of the vehicle footprint to convolve the cost map. (R-18) • The template generation program, used to generate the motion template for a specific vehicle model, shall be intuitive, semi-automated, and produce the motion template used by the lattice planner. (R-19) |
| RELIABILITY
REQUIREMENTS | <ul style="list-style-type: none"> • The lattice planner shall check the boundary states for invalid inputs. (R-20) • The lattice planner shall check the motion template for null sets or invalid edges. (R-21) • For those search engines which are deterministic (not random), the search engine will return failure when it becomes known that no path between the boundary state pair exists. (R-22) |

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers

TASK Very Rough Terrain Regional Motion Planning

DOCUMENT TYPE Lattice Planner Concept Design Document

AUTHOR Mihail Pivtoraiko, Dr. Alonzo Kelly

LAST REVISION DATE June 27th, 2008

INTRODUCTION: The regional motion planner is a deterministic nonholonomic motion planner. Depending on the search engine chosen, it may be complete or probabilistically complete and it may or may not be optimal. It is based on encoding vehicle maneuverability in a symmetric set of primitive motions which are repeated at all points in a discrete state space. The encoding of vehicle maneuverability directly in the state space means that all plans produced are intrinsically feasible (directly executable by the vehicle). The overall system consists of an offline component and an online component as shown in Figure 1.

Motion Template Generator: The motion template generator uses an externally supplied vehicle model and the trajectory generator developed as a companion task in this project. The goal of this module is to produce a set of primitive motions which span the set of feasible motions of the vehicle while being sufficiently small in number.

Motions can be visualized as curves embedded in the vehicle state space with a distinguished start and end state defined. Motion templates are computed by:

- Deciding on the parameterization of state space.
- Deciding on a regular discretization of that state space, including a state at the origin.
- Computing a preferred set of motions which connect the origin to some of the states in a neighborhood around it.

Once a motion template is specified, it is implicitly repeated at every node (state) in the discrete state space to create a graph encoding an infinite number of feasible motions which will be searched by the graph search engine.

Heuristic Lookup Table: This module exists because motion planners benefit significantly from heuristic search techniques, such techniques require good heuristics, and good heuristics for nonholonomic vehicles (like wheeled vehicles), are expensive to compute. A lookup table precomputes heuristics for a large number of cases and makes the results available for negligible computing cost. The process used to compute the heuristic lookup table (HLUT) is to simply present the entire planning system with a world devoid of obstacles and to generate plans from the origin to all points in a neighborhood around the original. The path lengths / costs of these paths are then stored in a manner which is indexed by the terminal state of each plan. The result is a mapping from terminal states to associated path costs which is computed in negligible time.

Graph Search: Many algorithms exist for searching graphs and the lattice planner design and implementation can accept any search engine that will find a sequence of motion primitives in the network that produces a path that starts at some designated start node and ends at a designated goal node. The A* algorithm is complete and optimal while not being particularly efficient in replanning, the D* algorithm is a variant of A* which is designed for efficient replanning and vehicle motion. The Rapidly Expanding Random Tree (RRT) algorithm trades optimality for runtime efficiency. All three of these options have been implemented for the lattice planner.

Edge Evaluation: In principle, the only part of the planner that needs to know about the external environment is the collision detection / edge evaluation module. Hence, while the motion template encodes maneuverability in directly useable form, the edge evaluator encodes the obstacles in the environment in directly useable form. Due to the intended use of the system in perceptive real-time replanning applications, this module is implemented by computing the swept area occupied by the vehicle envelope when a path is traversed. It computes the cost of this area as the cost of the path (edge).

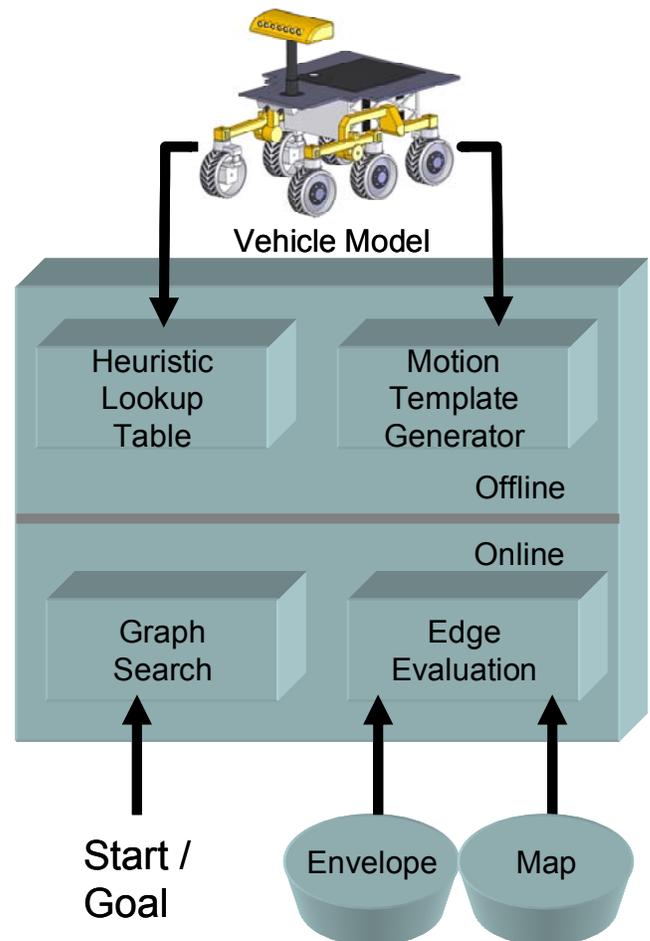


Figure 1: Lattice Planner System Architecture. The lattice planner consists of a graph search engine and a collision detection / edge evaluation module which is aware of obstacle and/or cost fields encoded in an externally supplied map. A vehicle model is also supplied externally. A motion template is generated offline in order to encode the maneuverability of the vehicle in a directly useable form. Also, a heuristic lookup table is optionally generated offline and it contains information about the shortest path between two states in the absence of obstacles.

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Very Rough Terrain Regional Motion Planning
DOCUMENT TYPE Lattice Planner Detailed Design Document
AUTHOR Mihail Pivtoraiko, Dr. Alonzo Kelly
LAST REVISION DATE June 27th, 2008

1. Introduction

The Regional Motion Planner system is composed of the planning program, executed onboard the moving robot ("on-line planner"), and a number of utilities that prepare the information needed for the planner ("off-line tools"). In this document, we will describe the design of both parts of the system. First we will cover the design of the on-line planner and discuss the data that it needs to operate. Further, we will explain how the off-line tools provide an important part of that data. We will also describe how the on-line planner integrates with other subsystems of the rover and how it receives the rest of the necessary data from the rover. Further detail on the design of each of the components of the regional planing system can be found in related publications [4] [5] [3].

2. On-line Planner

Overall, the on-line planning program implements a graph search algorithm. While any graph search algorithm can be used in this design, we chose D* Lite [2] for this delivery. Similar to most other graph search implementations, the delivered design is divided into two parts: a data structure for representing the graph and an implementation of the search algorithm itself. While the second part, the search algorithm, follows its standard implementation as described in [2], there is no universal solution to data structure design. This implementation is geared specifically for fast execution onboard rovers with computational constraints. A custom design of the graph data structure was done specifically for this purpose. In the following subsection, we will orient the discussion by describing the graph search in general and then focus its aspects that are particularly relevant for the Regional Planner. We will proceed with a detailed exposition of the components that comprise our implementation of the search data structure.

2.1 Graph Search

A graph is a structure that consists of *vertices* that represent datapoints. In the context of motion planning, the data we wish to represent is vehicle state, including its initial state before motion and its desired final state. Hence, it is convenient to assign to the vertices of the graph the meaning of points in the state space of the robot. The vertices of the graph are connected with *edges* that establish relationships between the vertices. In motion planning, the edges can be understood as the motions of the robot that take it from one state to another state. For example, moving forward 1 foot may be represented as two vertices (robot state before and after motion, where the latter would reflect having moved forward) and an edge between them, denoting the command the robot executed to move forward. Similarly, we can come up with edges that describe any motion the robot can execute. In general, there are infinitely many such motions, which would be infeasible to represent and reason about using today's computers [9]. The best anyone can do is find approximate solutions, and with a careful design of the graph data structure, such solutions are quite good. Finding a graph structure that works well for planetary rovers is one of the contributions of our project.

Suppose we assume a graph structure, where the vertices are arranged at regular intervals to each other, in a grid pattern. The edges between the vertices are also distributed in a regular fashion, where each vertex is connected to exactly 8 of its nearest neighbors (e.g. using Euclidean distance metric), as shown in Figure 2. Such a structure would serve well to illustrate the mechanism of grid search; the actual delivered implementation operates similarly, and the primary difference is in the actual arrangement of vertices and edges.

Now that we have this graph structure, we pick two vertices in it that correspond to initial and final states of the robot. Now finding a motion plan that takes the robot from initial to the final state is equivalent to finding a path in this graph, i.e. a sequence of the edges that connects the chosen vertices. As we alluded to earlier, now that the graph structure is in place, we can rely on standard graph search algorithms to find this path. Moreover, under appropriate conditions, it would be possible to find the *shortest* path in this graph.

Suppose we wish to find the path from the initial vertex in the lower-left part of the Figure 3a to the vertex in the upper-right corner of the figure. One way to search for the path would be to treat the initial vertex as the root of a tree, and all its 8 neighbor vertices as successors. Each of these successors in turn has 8 successors, similar to Figure 2. The process of finding and examining the immediate successors of a graph vertex is called *node expansion*. In this manner, a tree structure arises, which can be searched using *breadth-first search* or other algorithms. One popular algorithm for this search is A^* , a variety of *best-first search* that uses a *heuristic*

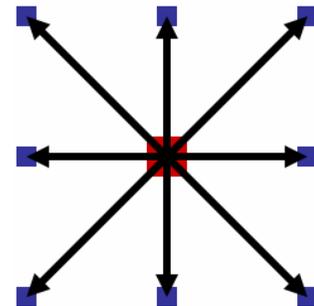


Figure 2. An example of a simple graph, where vertices are arranged in a regular grid-like structure. Each vertex is connected to 8 nearest neighbors using edges (arrows in the figure).

to guide the search in the promising direction towards the goal. The heuristic is essentially a "best guess" of how far each vertex is from the goal. It provides a mechanism to postpone expanding extraneous vertices that are, for example, located in the opposite direction to the goal, thereby speeding up the search. In this example we will use Euclidean distance as the heuristic. Figure 3a demonstrates heuristic estimates of each of the successor vertices, found by the node expansion presented in Figure 2.

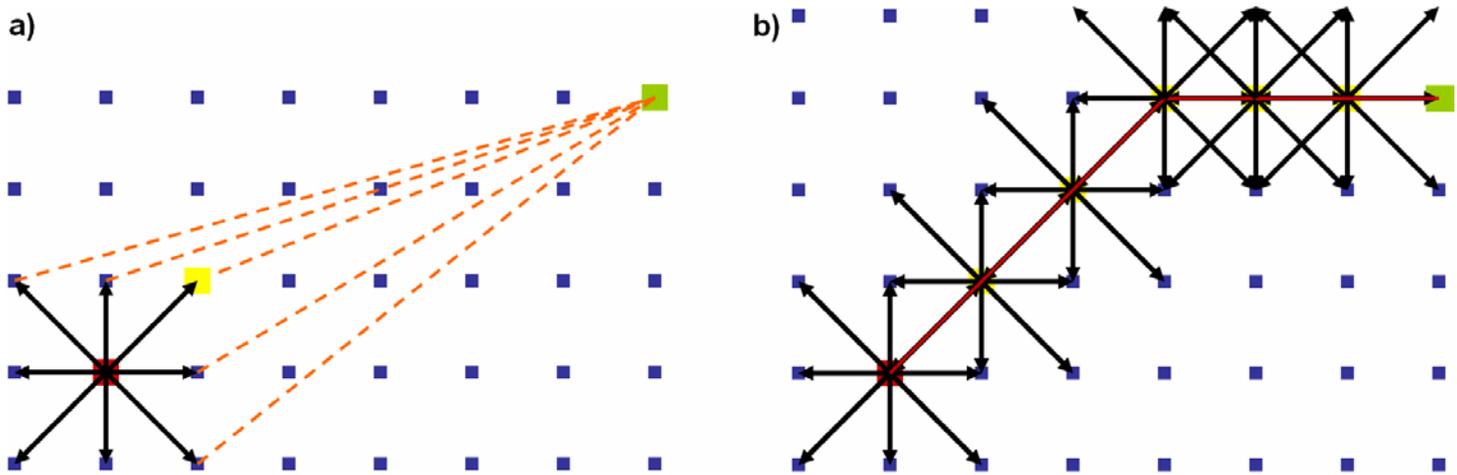


Figure 3. Finding a shortest path in the simple graph example using A* search algorithm. a) A heuristic is used to find the most "promising" vertex to continue the search with (next vertex to expand). b) The search proceeds to expand vertices until it reaches the goal vertex, thereby completing the search.

2.2 Heuristics

Even though Euclidean distance worked well for our simple example above, it has a number of disadvantages in the context of differentially constrained planning. In particular, the differential constraints of real rovers result in motions that are different than simple straight line motions. Hence Euclidean (linear) distance becomes a poor estimate of the motion ability of the rover. The Euclidean distance still satisfies the *admissibility* property of a heuristic, namely that its estimate of cost of a vertex must be less than or equal to the true cost, to ensure finding the shortest path. However, its estimate of the cost can be so much less than the true cost that the value of the heuristic is notably diminished. There are better alternatives for heuristics in this setting. In particular, the result by Reeds and Shepp [8] makes it possible to compute better path length estimates analytically. Also, lookup tables have been used successfully to precompute the obstacle-free traversal cost between vertices and use it during search [4] [1]. All these methods are supported by the Regional Planner system and can be utilized via simple configuration.

2.3 Data Structures

The graph search is a core part of the on-line planner in the delivered Regional Motion Planner system. The graph search we utilize is conceptually identical to the simplified example in the previous section, it differs only in detail. In particular, we utilize D* Lite search algorithm to enable efficient replanning via caching and reusing previous computation. Also, it allows flexibility in the connectivity of edges in the graph. Any connectivity is supported, as required by the application. This flexibility is an important benefit of the delivered system. It is available due to the two data structures that were utilized to represent the graph in the implementation of the on-line planner: the *hash table* and the *priority queue*.

2.4 Hash Table

The hash table is the most widely used data structure in the delivered implementation. A hash table is a structure that stores data points. It is essentially an array (or a vector), where each element is a linked list (or a pointer to it). Data points are stored as elements of these linked lists, as shown in Figure 4. Thanks to the linked lists, the hash table can store more data points than there are elements in its array. A *hash function* is used to generate an integer key, given the information in the data point. The best way of generating keys is highly dependent on the application, yet the goal is make sure the data are distributed evenly

throughout the array, in other words that the length of all the linked lists is roughly the same. The hash function allows random-access to the data: given the data point, its hash key is computed, which immediately specifies which linked list the data point must lie in (if it exists in the hash table). Thus, using a good hash function, data lookups in the hash table can be significantly more efficient than in an array or vector (which do not allow random access and require linear search to find the data).

There are several uses for hash tables, listed below, in the on-line planner of the Regional Motion Planner system. Note that all these uses share the same and unique implementation of the hash table.

- In order to allow efficient replanning, it is necessary to cache and potentially reuse the computation of cost for each graph vertex encountered during the search. We use the hash table to store these vertices. When necessary, we look them up in the hash table to review the results of previous computations on them.
- To be able to repair the plan in the event of a change of the cost of an edge (perhaps due to a perception update), it is important to be able to look up whether a particular vertex has another vertex as a parent. To enable this, we attach a hash table that stores pointers to parent vertices to every vertex in the graph. Each such hash table is typically much smaller than the one in the previous item, which stores the entire graph.
- It is helpful to be able to figure out whether a particular vertex has been expanded or not, so we place all vertices that have not yet been expanded into a hash table.

The hash table is a flexible and scalable data structure. By utilizing it for most of data management in the on-line planner, we inherit the flexibility and scalability in our implementation. Even though the hash table has some overhead for its operation, its benefits significantly over-weigh the costs. The properties it provides are extremely useful to the Regional Planner System, especially due to its reusability and extensibility to future rovers.

2.5 Priority Queue

Besides knowing whether a particular vertex has been expanded or not, it is also necessary to know which of the non-expanded vertices has *lowest* cost. This is important in the search algorithm we utilize, as such vertex would have to be expanded on the next iteration of the algorithm. In order to be able to find such a vertex, we provide a priority queue. As the name suggests, it is a queue data structure, a sequence of data points, with an ability to track priority (synonymous with cost in this context) of each vertex. This data structure always keeps the lowest cost vertex at the head of the queue, so queries for lowest cost are constant time. There is indeed a cost to this benefit, in particular the additional computation required for ensuring the ordering of items in the queue. However, thanks to utilizing the *heap* structure to implement the priority queue, this computation is more efficient than sorting an array. Only one element of the heap needs to be sorted, namely the lowest-priority one. Other elements in it do not need to be sorted at all.

Thus, the priority queue is implemented as a dual hash table – heap data structure. The former allows fast lookups of containment of vertices, and the latter provides least-cost item in constant time. There is indeed extra overhead for keeping the two component data structures in sync, however it is nearly trivial, since addition and removal of elements in both component structures can be done as an atomic operation, avoiding discrepancies in them.

2.6 System Integration

Our implementation of the on-line planner is designed to be easy to interface with other components of the robot, including the perception and motor control systems. There are two types of inputs to the planner: live information about the rover and its environment, and the data precomputed by the off-line tools. The former is received from the rover's subsystems continuously as it moves. It primarily consists of new perception information, incorporated into updates to the cost map of the environment, and rover's estimates of its own state. The precomputed off-line information is supplied via data files (either stored onboard the rover or made available via the network). It primarily consists of the *control set*, a specification of how vertices in the graph are connected. The control set is computed by the off-line tools and is dependent on the mobility model of the vehicle.

Since the delivered implementation of the Regional Planner system can operate both live on the rover and in simulation, for the purpose of discussing its structure, we can safely decouple it from the specifics of the rover interfacing. A diagram in Figure 5

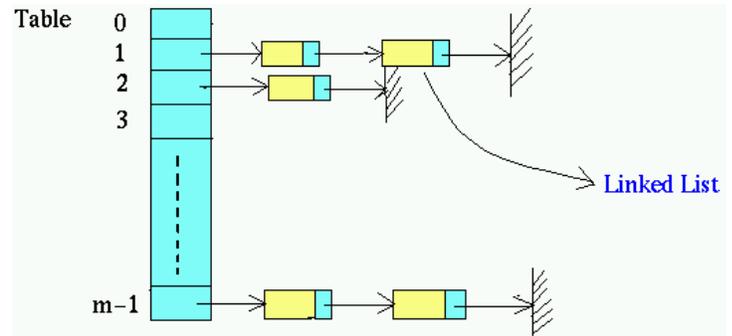


Figure 4. A simple hash table: an array, where each element is a link list, containing the data points. Image credit: www.cs.mcgill.ca/~cs251/OldCourses/1997/topic12

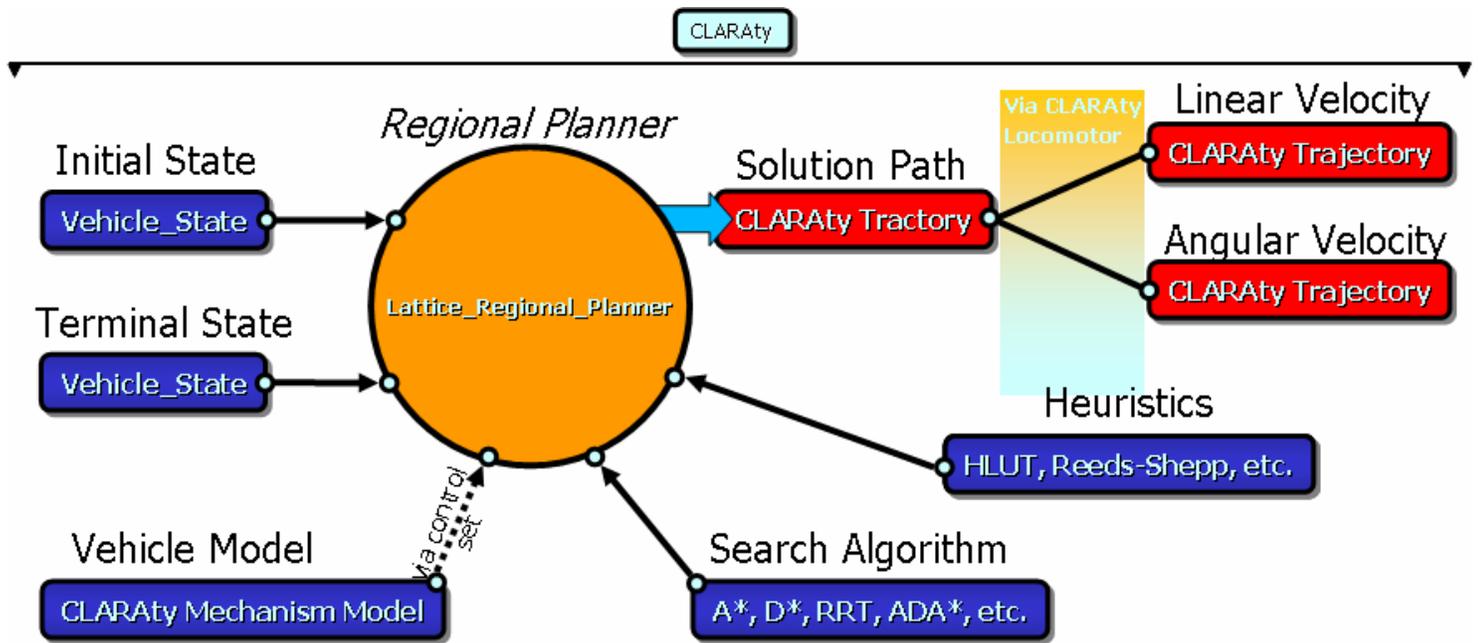


Figure 5. The concept design of the on-line planner system.

illustrates the components that comprise the on-line planner. The Figure lists the components we have already discussed. The two inputs on the left of the Figure are the terminal state constraints: initial and final states for this planning problem. The planner must find a trajectory that drives the rover from the specified initial state to the specified final state. The third input counter-clockwise is the control set, derived from a model of the rover's mobility. The following component is the search algorithm, introduced in Section 2.1, followed by the choice of a heuristic, described in Section 2.2.

All these components are necessary to enable the Regional Planner, executing onboard the rover, to find the solution path to the planning problem at hand. This path is the output of the system. By virtue of the design of the Regional Planner, no post-processing of the solution path is required; it can be handed directly to the rover for execution. Linear and angular velocities of the rover body are computed directly from this solution trajectory, and further converted to steer angles and drive velocities of wheel of the rover. Thus, the Regional Planner is capable of driving a rover autonomously, while avoiding obstacles and maximizing rover's maneuverability, as specified in the provided rover model.

3. Off-Line Tools

The off-line tools are a very important part of the Regional Planner system, as they perform the bulk of the processing required to enable the satisfaction of the differential constraints of the robot, as specified by its system model. Thanks to the availability of the pre-computed information from the off-line tools, the delivered planner can readily use it to speed up the on-line, onboard motion planning. There are two principal kinds of pre-computed information: a control set that is a representation of the robot's maneuverability (kinematics) model, and a set of path swaths that represents its geometric model, in particular the cells of the robot's cost map that the robot will cover as it moves along the motions in the control set.

3.1 Control Set Generation

The control set has a very important role in the Regional Planner system. As suggested above, it informs the planner how different states of the rover are connected. Indirectly, the control set is a representation of the planner's knowledge of how the rover can move. Thus, it is important to make this representation as accurate as possible, while maintaining the complexity of the planning within required limits. The algorithmic details of developing good control sets have been extensively described in the publications that resulted during the development of the Regional Planning system, in particular [5] [7] [6]. The delivered system provides the implementation of the tools required to generate the control sets provided the model of the vehicle, and its usage is further described in the User Guide document.

3.2 Path Swaths computation

The set of path swaths is simple to compute, yet also very important for the operation of the motion planner. In principle, the procedure for computing this data product involves selecting each element of the control set and simulating the vehicle's motion under its command. Any desired coordinates of the resulting swath are simply recorded in a data file, organized as a database indexed by the control set elements for fast lookup during on-line planning.

4. References

- [1] R. Knepper and A. Kelly. High performance state lattice planning using heuristic look-up tables. In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 2006.
- [2] S. Koenig and M. Likhachev. D* Lite. In Proceedings of the AAAI Conference of Artificial Intelligence (AAAI), 2002.
- [3] M. Pivtoraiko, T. Howard, I. Nenas, and A. Kelly. Field experiments in rover navigation via model-based trajectory generation and nonholonomic motion planning in state lattices. In Proc. of the Int. Symp. on Artificial Intelligence, Robotics and Automation in Space, 2008.
- [4] M. Pivtoraiko and A. Kelly. Constrained motion planning in discrete state spaces. In Proc. of the Int. Conf. on Field and Service Robotics, 2005.
- [5] M. Pivtoraiko and A. Kelly. Generating near minimal spanning control sets for constrained motion planning in discrete state spaces. In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 2005.
- [6] M. Pivtoraiko and A. Kelly. Differentially constrained motion replanning using state lattices with graduated fidelity. In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 2008.
- [7] M. Pivtoraiko, R. Knepper, and A. Kelly. Optimal, smooth, nonholonomic mobile robot motion planning in state lattices. Technical Report CMU-RI-TR-07-15, Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 2007.
- [8] J. A. Reeds and L. A. Shepp. Optimal paths for a car that goes both forwards and backwards. Pacific Journal of Mathematics, 145(2):367–393, 1990.
- [9] J. Reif. Complexity of the mover’s problem and generalizations. In Proc. of IEEE Symposium on Foundations of Computer Science, pages 421–427, 1979.

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Very Rough Terrain Regional Motion Planning
DOCUMENT TYPE Lattice Planner API Specification and User Guide Document
AUTHOR Mihail Pivtoraiko, Dr. Alonzo Kelly
LAST REVISION DATE June 27th, 2008

INTRODUCTION

The Regional Motion Planner system has been delivered as a software module integrated into the Coupled-Layer Architecture for Robotic Autonomy (CLARAty) at JPL. This document describes the components of the delivered module and illustrates how these components come together to implement motion planning using state lattices. We will also describe the principles of applying the delivered motion planner to real-time operation onboard research prototype rovers.

REQUIRED FILES

The following software files are necessary to use the Regional Motion Planner system:

Curve.h	the specification of the base class of the curve that represents motions considered by the planner
IntegratedCurve.h	the derived class that represents a curve obtained by integrating a control-space trajectory
PolynomialCurve.h	the derived class for the curve, parameterized as a polynomial
Discretization.h	encapsulates the details of state discretization
State.h	the specification of the base class of a discrete value of robot state
NHState.h	the derived class that includes state coordinates relevant for non-holonomic planning
DStarState.h	the derived class that provides additional information helpful for implementing D* variants
DL_NHState.h	the derived class that is specifically geared for D* Lite implementation
Template.h	the base class that specifies the structure of a control set (template) used during planning
PolynomialTemplate.h	the derived class of a control set designed to support the curves parameterized as polynomials
Heuristic.h	the base class that provides the structure for all heuristics used in the planner
EuclideanHeuristic.h	the derived class that implements the Euclidean distance heuristic
ReedsSheppHeuristic.h	the derived class for the Reeds-Shepp heuristic
LUTHeuristic.h	the derived class that implements the Look-Up Table heuristic
Hash.h	the class for performing hash-key computation
HashTable.h	the class that implements a generic hash table
World.h	the base class for defining a generic representation of the environment of the robot
FileWorld.h	the derived class that supports reading environment information from a file
PriorityQueue.h	the base class that implements a priority queue
DL_PriorityQueue.h	the derived class that introduces D* Lite specific extensions to the priority queue
Graphics.h	the utility class that provides the interface for the debugging graphics library
Movable_Polygon2D.h	the class that performs the geometry computations related to movable polygons
Planner.h	the base class that specifies the interfacing for a number of different planning algorithms
DL_Planner.h	the derived class that provides the functionality specific to the D* Lite planning algorithm
liblattice_planner.so	the compiled motion planner library

REGIONAL PLANNER API SPECIFICATION

The Regional Planner API consists of two distinct parts: the programmer's API of the Lattice Planner, the "on-line" planner that executes onboard the vehicle, and the "off-line" tools that pre-compute the data used by the on-line planner.

Lattice On-line Planner API

The Lattice Planner API is composed of the following functions. The configuration of the planner is performed via the arguments to these functions, listed and explained below function names.

Template::loadTextFile(char *control_set_filename, char *swath_filename, bool verbatim)

loads the control set data from file

control_set_filename the name of the file where the control set data are stored

swath_filename the name of the file where the path swath data are stored

verbatim a flag that indicates whether the software should take the controls in the control set as-is (true) or interpret them as rotated to the I quadrant (false)

Movable Polygon2D::Movable Polygon2D(int shape, double size_x, double size_y)

constructs the movable polygon

shape an integer that indicates one of the predefined shapes of the polygon
 size_x the size of this region along the x-axis
 size_y the size of this region along the y-axis

FileWorld::FileWorld(char *world_file_name, int repeat_type)

constructs the class that reads environment information from a file

world_file name the name of the file that specifies the environment information
 repeat_type the integer that specifies the type of repetition of the information in the world file

DL_Planner::DL_Planner(Heuristic* heuristic, World* world, vector <PolynomialTemplate*> c_sets,
 vector <Movable Polygon2D> regions)

constructs the class that implements D* Lite search algorithm

heuristic a base-class pointer to the chosen heuristic for the planner
 world a base-class pointer to the environment information
 c_sets a vector of control sets for each fidelity region in the representation of the planning problem
 regions a vector of movable polygons, specifying the size and location of each fidelity region

DL_Planner::init(State* init_state, State* final_state)

initializes the terminal states of the robot

init_state initial state of the rover
 final_state final state where rover needs to get to

DL_Planner::compute shortest path()

performs graph search to find the planning solution

DL_Planner::move robot(State *new_state)

informs the planner that the robot has moved to a new state, so it can replan

new_state the new state of the rover

DL_Planner::update map(int x, int y)

informs the planner that map cell, indexed by the provided arguments, has changed cost

x the x-coordinate of the map cell that changed cost
 y the y-coordinate of the map cell that changed cost

Off-line Tools

There are two principal offline tools: for computing the control set and the path swaths associated with the control set. Both tools are command-line utilities that are compiled and run as separate programs. They can be run on any computing system and do not have to be run onboard the rover. Both programs indirectly use the vehicle model via the trajectory generator, described in the accompanying documentation of the trajectory generator.

generate_control_set control_set_filename

this command-line utility generates the control set; it directly uses the trajectory generator, which encapsulates the details of the vehicle model

control_set_filename the name of the file where the control set data are stored

compute_template_swaths control_set_filename verbatim_flag

this command-line utility computes the swaths of the motions, contained in the control set

control_set_filename the name of the file where the control set data are stored

verbatim_flag this flag indicates whether the software should take the controls in the control set as-is (true) or interpret them as rotated to the I quadrant (false)

EXAMPLE PROGRAM

The following code listing demonstrates how the Regional Planner system can be used, either on a desktop or onboard a rover.

```

#include <sys/time.h> // gettimeofday
#include "EuclideanHeuristic.h"
#include "ReedsSheppHeuristic.h"
#include "ManhattanHeuristic.h"
#include "FileWorld.h"
#include "DL_Planner.h"

#define VEH_REGION_SIZE 31

using namespace std;

int main(int argc, char **argv)
{
    // parse input params
    if(argc < 12) {
        printf("ERROR: expecting a complete goal specification (8 numbers) and num of replans\n");
        exit(-1);
    }
    int xi = atoi(argv[1]);
    int yi = atoi(argv[2]);
    int ti = atoi(argv[3]);
    int phi_i = atoi(argv[4]);
    int ki = atoi(argv[5]);
    int xf = atoi(argv[6]);
    int yf = atoi(argv[7]);
    int tf = atoi(argv[8]);
    int phi_f = atoi(argv[9]);
    int kf = atoi(argv[10]);

    // boundary states:
    DL_NHState start(xi, yi, ti, phi_i, ki);
    DL_NHState final(xf, yf, tf, phi_f, kf);

    // define the templates:
    vector<PolynomialTemplate*> template_list;

    PolynomialTemplate grid_tmpl;
    // the order of inserting templates matters: root template should be idx 0,
    // and so on in order of precedence
    grid_tmpl.loadTextFile("data/template_8grid.dat", "data/path_swaths_8grid.dat", true);
    template_list.push_back(&grid_tmpl);

    PolynomialTemplate lattice_tmpl;
    lattice_tmpl.loadTextFile("template_r8.dat", "path_swaths_r8.dat");
    template_list.push_back(&lattice_tmpl);

    // define the topology regions:
    vector<Movable_Polygon2D> region_list;
    Movable_Polygon2D vehicle_region(Movable_Polygon2D::RECTANGLE,
                                     VEH_REGION_SIZE, VEH_REGION_SIZE);
    region_list.push_back(vehicle_region);

    // heuristics, choose one:
    ManhattanHeuristic mh;
    EuclideanHeuristic eh;
    ReedsSheppHeuristic rsh;

    // worlds:
    FileWorld world("data/rover_perception.pgm", FileWorld::RepeatNone);

    DL_Planner dl_planner(&rsh, &world, template_list, region_list);
    // initialize the planner:
    dl_planner.init(&start, &final);
    // compute the path:
    dl_planner.compute_shortest_path();

    return 0;
}

```

The presented API was designed to make the Regional Planner relatively straightforward to run. The above program can be used as-is to test the planner, and it contains some OS-specific code (compatible with UNIX and VxWorks). The key portions of the API code that have been described above are illustrated in this program. The desired initial and final state are specified in the format $(x, y, \theta, \varphi, \kappa)$, where, in addition to (x, y) coordinates, θ is heading, φ is the crab angle (set to 0 if not using crabbing), and κ is curvature (similarly set to 0, if wheels are desired to be straight at either terminal state). Further, the control sets ("templates") for each of the fidelity regions are combined into a STL vector. Similarly, the sizes, shapes and locations of each fidelity region are specified as movable regions, also combined in an STL vector. These two vectors, along with a choice of the heuristic and the representation of the environment (via the `World` class and its derived classes), are passed to the planner in its constructor. Afterwards, after relaying the terminal states to the planner via the `init` function, the planner is ready to compute the solution. In the event of a perception update and change of the cost map, the planner must be informed of each map cell that changed cost, via the function `update_map`. In the event that the rover moves since the previous plan was found, its new state must be relayed to the planner via the `move_robot` function. Afterwards, the `compute_shortest_path` is once again called, and, if necessary, the rover motion is replanned efficiently.

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers

TASK Very Rough Terrain Regional Motion Planning

DOCUMENT TYPE Lattice Planner Test Plan Document

AUTHOR Mihail Pivtoraiko, Dr. Alonzo Kelly

LAST REVISION DATE June 27th, 2008

INTRODUCTION: The Regional Planner shall be tested to determine if the functional, performance, interface, and reliability requirements have been met. This document describes the testing that was undertaken to certify the algorithm. % and trace which requirements each test validates.

- REGRESSION TESTS
- Test the lattice planner with a large number of queries on benign terrain with few and rare obstacles and compare solved trajectories to stored solutions. The controls parameterization and vehicle model shall be the default control parameterization (5th order polynomial in curvature, constant unity velocity, constant zero direction) and the default vehicle model, respectively.
 - (Functional Requirement R-1) Test the lattice planner and report % successful solutions.
 - (Functional Requirement R-2) Test the lattice planner ability to execute different search algorithms.
 - (Functional Requirement R-3) Test the lattice planner capacity to operate in an anytime manner.
 - (Functional Requirement R-4) Test the lattice planner ability to vary the fidelity of representation of its search space.
 - (Functional Requirement R-5) Test the ability of the search algorithm to utilize heuristics in order to improve motion planning efficiency.
 - (Functional Requirement R-6) Test that the search spaces fixed to the body or the world frame can be utilized with the lattice planner.
 - (Functional Requirement R-7) Test that it is possible to adapt the lattice planner to incorporate dynamic limitations of different vehicle models in the form of a motion template.
 - (Functional Requirement R-8) Test that the planner is configured to plan long distance maneuvers provided a prior model of the environment or to continuously replan short actions based on perceptual information.
 - (Functional Requirement R-9) Test that the lattice planner's replanning mode remains convergent as the vehicle moves under the assumption of a static environment.
 - (Functional Requirement R-10) Test that the memory footprint of the lattice planner does not exceed 300 MB.
 - (Functional Requirement R-11) Test that the lattice planner average runtime for plans over average distance 100 map cells does not exceed 10 seconds.
 - (Functional Requirement R-12) Test that the lattice planner maximum runtime for plans over average distance 200 map cells does not exceed 20 seconds.
 - (Functional Requirement R-13) Test that the heuristic lookup table generation program generates a heuristic lookup table in 30 minutes.
 - (Functional Requirement R-14) Test that the template generation program generates a motion template in 30 minutes.
 - (Functional Requirement R-15) Test that the lattice planner initialization time does not exceed 30 seconds.
 - Test the lattice planner with a large number of queries on very rough terrain with dense obstacles and compare solved trajectories to stored solutions. The controls parameterization and vehicle model shall be the default control parameterization (5th order polynomial in curvature, constant unity velocity, constant zero direction) and the default vehicle model, respectively.
 - (Functional Requirement R-1) Test the lattice planner and report % successful solutions.
 - (Functional Requirement R-2) Test the lattice planner ability to execute different search algorithms.
 - (Functional Requirement R-3) Test the lattice planner capacity to operate in an anytime manner.
 - (Functional Requirement R-4) Test the lattice planner ability to vary the fidelity of representation of its search space.
 - (Functional Requirement R-5) Test the ability of the search algorithm to utilize heuristics in order to improve motion planning efficiency.
 - (Functional Requirement R-6) Test that the search spaces fixed to the body or the world frame can be utilized with the lattice planner.
 - (Functional Requirement R-7) Test that it is possible to adapt the lattice planner to incorporate dynamic limitations of different vehicle models in the form of a motion template.
 - (Functional Requirement R-8) Test that the planner is configured to plan long distance maneuvers

provided a prior model of the environment or to continuously replan short actions based on perceptual information.

- (Functional Requirement R-9) Test that the lattice planner's replanning mode remains convergent as the vehicle moves under the assumption of a static environment.
- (Functional Requirement R-10) Test that the memory footprint of the lattice planner does not exceed 300 MB.
- (Functional Requirement R-11) Test that the lattice planner average runtime for plans over average distance 100 map cells does not exceed 10 seconds.
- (Functional Requirement R-12) Test that the lattice planner maximum runtime for plans over average distance 200 map cells does not exceed 20 seconds.
- (Functional Requirement R-13) Test that the heuristic lookup table generation program generates a heuristic lookup table in 30 minutes.
- (Functional Requirement R-14) Test that the template generation program generates a motion template in 30 minutes.
- (Functional Requirement R-15) Test that the lattice planner initialization time does not exceed 30 seconds.

INTERFACE TESTS

- Test the interface of the lattice planner for compatibility with the representation of the robot environment.
 - (Interface Requirement R-16) Test that the lattice planner accommodates variable resolution of the state space.
 - (Interface Requirement R-17) Test that the lattice planner receives a uniformly sampled two-dimensional array of floating point values (representing the cost of occupying a state) for a cost map.
- Test the interface of the lattice planner for compatibility with the representation of the robot's mobility model.
 - (Interface Requirement R-18) Test that the lattice planner receives a geometry representation of the vehicle footprint to convolve the cost map.
 - (Interface Requirement R-19) Test that the template generation program, used to generate the motion template for a specific vehicle model, is intuitive, semi-automated, and produces the motion template used by the lattice planner.

RELIABILITY TESTS

- Provide the lattice planner with unreasonable (NaN, inf, etc.) state constraints to test for proper error handling
 - (Interface Requirement R-20) Test that the lattice planner shall check the boundary states for invalid inputs.
- Provide the lattice planner with an improper control set to test for proper error handling
 - (Interface Requirement R-21) Test that the lattice planner checks the motion template for null sets or invalid edges.
- Provide the lattice planner with an infeasible motion planning problem to detect non-convergence or divergence of the algorithm (exceeding maximum number of iterations)
 - (Interface Requirement R-22) Test that the lattice planner, when configured with a deterministic search algorithm, will return failure when it becomes known that no path between the boundary state pair exists.

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Very Rough Terrain Regional Motion Planning
DOCUMENT TYPE Lattice Planner Test Results Document
AUTHOR Mihail Pivtoraiko, Dr. Alonzo Kelly
LAST REVISION DATE June 27th, 2008

This document details the results of software testing the implementation of the Very Rough Terrain Regional Motion Planner, as delivered to the Mars Technology Program. The table below provides the results for the tests outlined in the Test Plan Document.

	Benign Terrain		Rough Terrain	
	Omniscient Perception	Limited Perception	Omniscient Perception	Limited Perception
Test R-1	100%	100%	100%	100%
Test R-2	PASSED	PASSED	PASSED	PASSED
Test R-3	PASSED	PASSED	PASSED	PASSED
Test R-4	PASSED	PASSED	PASSED	PASSED
Test R-5	PASSED	PASSED	PASSED	PASSED
Test R-6	PASSED	PASSED	PASSED	PASSED
Test R-7	PASSED	PASSED	PASSED	PASSED
Test R-8	PASSED	PASSED	PASSED	PASSED
Test R-9	PASSED	PASSED	PASSED	PASSED
Test R-10	PASSED	PASSED	PASSED	PASSED
Test R-11	PASSED	PASSED	PASSED	PASSED
Test R-12	PASSED	PASSED	PASSED	PASSED
Test R-15	PASSED	PASSED	PASSED	PASSED
Test R-16	PASSED	PASSED	PASSED	PASSED
Test R-17	PASSED	PASSED	PASSED	PASSED
Test R-18	PASSED	PASSED	PASSED	PASSED
Test R-20	PASSED	PASSED	PASSED	PASSED
Test R-21	PASSED	PASSED	PASSED	PASSED
Test R-22	PASSED	PASSED	PASSED	PASSED
Test R-13	PASSED			
Test R-14	PASSED			
Test R-19	PASSED			