Robotics: Science and Systems I, MIT Press, Boston, MA, p. 233–241, 2005

Motion Planning in the Presence of Drift, Underactuation and Discrete System Changes

Andrew M. Ladd
Dept. of Computer Science
Rice Unversity
Houston, Texas, 77005
Email: aladd@cs.rice.edu

Lydia E. Kavraki
Dept. of Computer Science
Rice Unversity
Houston, Texas, 77005
Email: kavraki@cs.rice.edu

***Abstract*— Motion planning research has been successful in developing planning algorithms which are effective for solving problems with complicated geometric and kinematic constraints. Various applications in robotics and in other fields demand additional physical realism. Some progress has been made for non-holonomic systems. However systems with significant drift, underactuation and discrete system changes remain challenging for existing planning techniques particularly as the dimensionality of the state space increases. In this paper, we demonstrate a motion planning technique for the solution of problems with these challenging characteristics. Our approach uses sampling-based motion planning and subdivision methods. The problem that we solve is a game that was chosen to exemplify characteristics of dynamical systems that are difficult for planning. To our knowledge, this is first application of algorithmic motion planning to a problem of this type and complexity.**

I. Introduction

Motion planning algorithms are employed as a tool for reasoning about physical systems in diverse applications: object manipulation [1], assembly [2], prototyping of mechanical systems [3], autonomous robots [4], inspection and observation [5], humanoid robots [6], animation [7], virtual environments [8] and structural computational biology [9], [10].

Early algorithmic motion planning research focused on constructing collision-free paths in the presence of geometric constraints (workspace obstacles) and kinematic constraints (restrictions on the robot's motion). The earliest problems considered were polygonal robots in polygonal workspaces (Sofa Mover's Problem) and polyhedral robots in polyhedral workspaces (Piano Mover's Problem) [11]. Another important domain is path planning for 2-D and 3-D linkages [12]. Sampling-based planning algorithms such as the Probabilistic Roadmap Method (PRM) emerged as a powerful and effective approach for solving these kinds of planning problems [13], [14]. Applications using these techniques have been adapted to a large variety of systems: freely moving 2-D and 3-D robots [13], serial and parallel linkages [13], [15], object manipulation [16], humanoid robots [6], flexible objects [17] and proteins [9], [10].

In early planning research, the task of executing a computed path on a robot was viewed as a secondary problem. The path could be smoothed and scaled to satisfy dynamic constraints of the system and the resulting trajectory could be followed with an appropriate controller. In some applications, however, this leads to poor results since the converted motions tend to be of very low quality. In non-holonomic planning, motivated by applications for car-like robots, tractor-trailer robots and spacecraft, converting unconstrained paths into motions which satisfy dynamic constraints can introduce large numbers of brief and jerky motions [18]. In some cases, the resulting trajectories are impossible to follow on a physical platform [19]. Path planners which were restricted to generating motions satisfying the non-holonomic constraints were designed in order to address these difficulties. Non-holonomic variants of PRM have been formulated [20] and results for such systems have been achieved with the Rapidly Exploring Random Trees (RRT) family of planners [21] as well as the Expansive Spaces planner (EST) [22]. Non-holonomic motion planning applications in the sampling-based planning literature have been tested for car-like robots, tractor-trailer robots and other similar 2-D platforms [20], [21], [22]. In the context of 3-D examples, several different variations of free flying spacecraft have been examined [21].

In the sampling-based planning literature, there have been a few studies on generating paths for robots with second-order non-linear dynamics. The specific problem instances that appeared were the lane-change problem for a second-order car-like robot, the control of a spacecraft with omni-directional thrusters in a cage, second-order differential drive robots moving in a maze, and a second-order blimp-like robot moving around pillars [21], [23], [24].

This paper addresses the implementation of a planning method and its application to a motion planning benchmark with severe underactuation, significant drift, high dimensionality, discrete system changes that occur at boundary conditions and finally a system which is not reduceable to a system with lower order dynamics. This work has two concrete goals in the context of planning applications that demand a high degree of physical realism: the development of online motion planners that can provide stability and completeness guarantees and the development of offline motion planners that can be used interactively in prototyping as tools for feasibility and safety testing in complex environments. We see applications such as dynamic obstacle manipulation, part manipulation with force fields, pursuit-evasion problems and

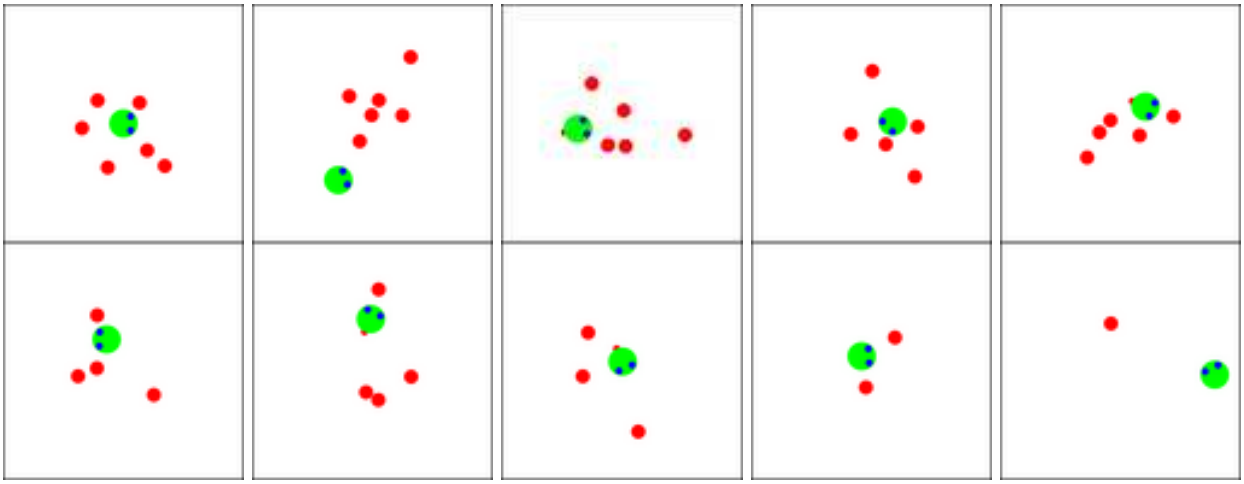

Fig. 1. Execution snapshots for a solution to the game of Koules with 6 Koules

hybrid system verification.

A. Problem Characteristics

This paper details the generation of trajectories for a dynamical system which was chosen to have features which are challenging for motion planning techniques: namely drift, underactuation, discrete system changes and high dimensionality. Drift in a dynamical system occurs when the system cannot instantaneously stop. For example, second-order dynamical systems with bounded acceleration cannot instantaneously cancel a non-zero velocity. From a planning perspective, systems with drift are challenging since the shortest path cost between two states frequently disagrees with the metric. Underactuation occurs when the dimension of the control space is less than the dimension of the state space. Underactuation can occur as a result of non-holonomic constraints or other dynamic constraints. An underactuated system has instantaneously passive or coupled degrees-of-freedom. Analyzing the shape and dimensionality of the reachable space in the presence of underactuation and kinematic constraints can be quite challenging. Discrete system changes occur in hybrid dynamical systems and manifest as discontinuities in the dynamic constraints or in the state variables as the system evolves. The behavior of hybrid systems can be quite complex and difficult to analyze. Finally, high-dimensional motion planning is well known to be hard. This is particularly true for dynamical systems since state parameters typically interact in a complicated way.

The dynamical system that this paper deals with is based loosely on a Unix game called the game of Koules [25]. The game of Koules is a multi-agent second-order dynamical system where inter-agent collisions are resolved by elastic collision. The agents are discs which operate in the unit square. Elastic collisions cause discrete system changes by instaneously causing velocity discontinuities. One agent is a robot (ship) that has four different controls and the other agents (Koules) move according to a force-field function determined by the current state of the system. The ship wins the game by pushing or bouncing the Koules in the boundary of the workspace without touching the boundary itself. In Figure 1, we show some snapshots of a solution of the game of Koules with 6 Koules. The game of Koules is described in detail in Section II.

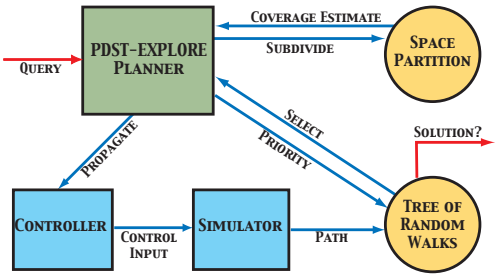

Fig. 2. Motion Planning Architecture

B. Planning Technique

The planning technique we employ to solve instances of the game of Koules is illustrated in Figure 2. The motion planning algorithm we employ is the Path-Directed Subdivision Tree exploration algorithm (`PDST-EXPLORE`) [24]. This paper continues our work on the `PDST-EXPLORE` algorithm and applies it to the game of Koules, which required additional and non-trivial adaptation. In fact, although it is not within the scope of this paper, obtaining an efficient and general implementation of `PDST-EXPLORE` was also interesting from a programming perspective.

The design of the planner used in this paper is shown in Figure 2. The general component is the `PDST-EXPLORE` block. The other blocks and data types need to be written specifically for each planning application. The *simulator* block represents the ground truth for the underlying system. As input, it takes states and raw controls and outputs a state one time step in the future. The simulator for the game of Koules is described in Section II. The *controller* block wraps the simulator by structuring the controls sent to the simulator. This controller is a motion generation model which creates trajectories that

have a nicer structure than a sequence of random controls. The controller that we use is described in Subsection III-B. The *tree of random walks* data structure stores a collection of path segments related by path branches. Representing paths to be compact and support fast interpolation is critical for good planner performance. The representation we use is summarized at the end of Subsection II-C. The search performed by the motion planner is guided by *coverage estimates* which we compute using a *space partition* data structure. Path segments are subdivided and assigned to cells in the subdivision. Coverage estimates are computed by computing cell volume, density and membership. The subdivision scheme we use is described in Subsection III-C.

C. Organization

Our version of the game of Koules is described in detail in Section II. Also in Section II, we describe the design and implementation of the simulator. Section III summarizes the `PDST-EXPLORE` planner, and the implementation of the local trajectory generator and the coverage estimation scheme. We also describe a high-level framework which uses `PDST-EXPLORE` as a subroutine to generate full solutions to game of Koules. Our experimental results are described in Section IV. In Section V, we discuss the experimental results and areas of future research.

II. Problem Description

Our version of the game of Koules takes place in a 2-D workspace, specifically a square. There are two types of robots inside the workspace: a single ship and the Koules. The ship is controlled by the user and the Koules follow independent trajectories. When a robot touches the boundary of the workspace, it is killed. The user loses the game if the ship is killed and the user wins the game if all of the Koules are killed. When two robots touch, an elastic collision occurs and the robots bounce away from each other. The ship is capable of four different actions that the user can control: to cruise, to turn left or right at a constant speed, or to apply a constant thrust in the direction of the ship's current heading. The Koules are attracted towards the center by a damped spring which makes them difficult to push towards the sides. The user can only influence the Koules by colliding with them.

Solving an instance of the game of Koules requires the generation of sequence of timed controls such that the ship survives and all of the Koules are killed. In the remainder of this section, we describe the implementation of our version of the game of Koules. In the next section, we describe the planner that we use to solve input instances of the game.

A. State Space and Controls

We begin by describing the state and control spaces. The state space for the game of Koules with n Koules is determined as follows:

$$Q_n = ([0,1]^2 \times S^1 \times \mathcal{R}^2) \times ([0,1]^2 \times \mathcal{R}^2)^n.$$

A state $q = (x_s, \theta, v_s, x_1, v_1, ..., x_n, v_n)$ determines the position, x_s, heading, θ, and velocity v_s of the ship together with the positions, $x_1, ..., x_n$ and velocities $v_1, ..., v_n$ of the Koules.

There are four distinct control inputs in the set of controls for the game of Koules, $U = \{u_0, u_L, u_R, u_1\}$, which correspond to cruise, u_0, turn left, u_L, turn right, u_R, and thrust, u_1.

An instance of the game of Koules consists of n, the number of Koules and an initial state $q_0 \in Q_n$. A partial solution to that instance is a path π of duration T such that at state $\pi(T)$, a Koule touches the boundary and no boundary collisions occur on the path before time T. A full solution is a sequence of paths $\pi_n, ..., \pi_1$ with durations $T_n, ..., T_1$ such that for all $i < n$, π_i is a partial solution to the instance $(i, \pi_{i+1}(T_{i+1}))$.

B. The Dynamic System

The game of Koules is a second-order dynamic system. The motion of the ship is determined by its state and the control input using the following equations:

$$\begin{bmatrix} \dot{x_s} \\ \dot{\theta} \\ \dot{v_s} \end{bmatrix} = \begin{bmatrix} v_s \\ v_\theta \\ R(\theta) \cdot [a\ 0]^T \end{bmatrix} \tag{1}$$

where v_θ is the turning speed, $R(\theta)$ is the rotation matrix in $SO(2)$ determined by θ and a is the thrust. The turning speed, v_θ, and thrust, a are determined as functions of the current control input $u \in U$,

u	$v_\theta(u)$	$a(u)$
u_0	0	0
u_L	$v_\theta^{\max}$	0
u_R	$-v_\theta^{\max}$	0
u_1	0	$a^{\max}$

.

The motion of each Koule is determined by its state and the position of the ship using the following damped spring equation:

$$\begin{bmatrix} \dot{x_i} \\ \dot{v_i} \end{bmatrix} = \begin{bmatrix} v_i \\ (o - x_i) \cdot \lambda_c - v_i \cdot h \end{bmatrix} \tag{2}$$

where o is the center of the workspace, λ_c is spring constant attracting towards the center and h is a friction parameter.

In the simulator, control inputs are applied over a fixed timestep Δt and numerically integrated with a fourth-order Runge-Kutta-Nystrom method [26].

C. Rules for Elastic Collisions

During each time-step of the simulator must simulate the system to generate the state that results from applying the current control, $u \in U$, to the initial state. This is a two-step process: first, a numerical integration of the equations of motion and followed by a discrete event simulation to resolve any collisions.

At the beginning of the time-step, the system is in state q^0. The result of integrating the control u for time Δt is a new state, q^f. However, although q^0 is collision-free, it is possible that collisions between robots or between the robots and the

boundary of the workspace occur along the path between q^0 and q^f. In order to calculate collisions and the results of the induced velocity changes, a locally linear approximation is used and first-order motions are simulated with a discrete event simulator. To begin with, a new initial state,

$$q^+ = (x_s^+, \theta^+, v_s^+, x_1^+, v_1^+, ..., x_n^+, v_n^+),$$

is constructed from q^0 and q^f as follows: $x_s^+ = x_s^0$, $\theta^+ = \theta^0$, $x_i^+ = x_i^0$, $v_s^+ = (x_s^f - x_s^0)/\Delta t$ and $v_i^+ = (x_i^f - x_i^s)/\Delta t$.

All robots are then assumed to begin at q^+ and to move along the lines determined by their velocities during the discrete event simulation. If there are no collisions, after time Δt has elapsed, the system will reach a state with the same positions as state q^f and with the velocities of state q^+. The velocities are constant along the time step and are approximately correct with error linearly proportion to Δt.

The events in the discrete event simulation occur when a pair of robots collide or when a robot touches the boundary. The ship has radius r_s and mass m_s. Each Koule has radius r_k and mass m_k.

Pairwise collisions occur when the distance between two robots is equal to the sum of their radii. This is predicted by the solution of the appropriate quadratic equation. It is best to use iterative root polishing to avoid simulation errors caused by near singular states. Collisions with the boundary are determined by solving linear equations. Inter-robot collisions are resolved by applying the 1-D elastic collision formula and boundary collisions end the simulation.

The minimum amount of information required to store a path is the initial state q^0 and a sequence of timed control inputs: $0 = t_0, u_1, t_1, ..., t_{m-1}, u_m, t_m$ where the input u_i is applied from time t_{i-1} to time t_i and $u_i \neq u_{i+1}$. In order to reconstruct the state, q^t, at time t the integrator and discrete event simulator must be run. Our implementation stores key frames at times were collisions occurred and with a certain minimum density to reduce the amount of integration that needs to be done during interpolation while maintaining a compact representation for path data.

III. Planner Design

As described in Subsection I-B, the planner architecture contains several components: the PDST-EXPLORE planner, the controller, the simulator, the tree of random walks and the space partition. In this section, we describe each part in detail.

The PDST-EXPLORE planner, the tree of random walks and the overall control flow in Figure 2 are described Subsection III-A. In Subsection III-B, we decribe the controller that we used generate trajectories for the game of Koules. The simulator was described in the previous section. The space partition that we used is described in Subsection III-C.

A single execution of PDST-EXPLORE is used to find a feasible path that kills one Koule. Multiple executions of the planner can be used to construct a sequence of paths that combine to be solution for given initial state, i.e. a feasible path for which the ship kills all of the Koules. However, it is possible that no solution is reachable from the final state of a partial solution generated by one execution of PDST-EXPLORE. To handle this possibility, we apply a task planner, which is described in Subsection III-D.

A. The PDST-EXPLORE Algorithm

The intuition behind the PDST-EXPLORE algorithm is quite simple: the space of random walks is searched to optimize coverage of the state space. PDST-EXPLORE is presented in Algorithm 1. Beginning from an initial state, $q^0 \in Q$, a tree of reachable states is constructed incrementally. The parameter N_{iter} is the maximum number of iterations that PDST-EXPLORE will run for.

A sample for PDST-EXPLORE is taken to be a collision-free feasible path in the state space, i.e., a path which is collision-free and obeys the motion constraints. The set of samples generated during a run of PDST-EXPLORE is referred to as P. The union of the samples form a tree with q^0 as the root and pairwise intersections at the branch states. In this way every $\pi \in P$ is associated with a path, $\texttt{path}(\pi)$, beginning at q^0 and with π as its suffix (line 8).

Coverage optimization is effected in PDST-EXPLORE by maintaining an incrementally refined subdivision of Q. The subdivision, S is a set of cells (subsets of Q) which partitions Q. A probability measure, μ, is used to quantify the volume of each cell. When a cell is subdivided it split into two cells. The set S is updated by removing the subdivided cell and adding the new cells. The subdivision scheme and measure are referred to as the coverage estimation scheme. This component of the planner is designed for the application and the one used in this paper is described in Subsection III-C. The subdivision operation occurs on line 12 of Algorithm 1.

At the end of each iteration of PDST-EXPLORE on line 13, an invariant relating P and S is enforced. Every $\pi \in P$ must have the property that π is contained in a unique cell of S. This is implemented by subdividing paths into a set of segments if they cross multiple cells.

A new sample is created from an existing sample by invoking the random propagation primitive, PROPAGATE. When a sample is propagated from a path $\gamma \in P$, the result of $\texttt{PROPAGATE}(\gamma)$ is a new path which branches from γ. The PROPAGATE operation is used on line 7. Iterated calls to PROPAGATE determine a random walk and PDST-EXPLORE can be thought of as constructing a tree of random walks. The implementation of PROPAGATE is described in Subsection III-B.

At each iteration of PDST-EXPLORE a sample $\gamma \in P$ is selected on line 6 and $\texttt{PROPAGATE}(\gamma)$ is invoked to obtain a new sample on line 7. The cell which contained γ is eventually subdivided on line 13. The crux of PDST-EXPLORE is the selection method which is designed to balance a completeness guarantee with a search that greedily covers the space. Each sample $\pi \in P$ is associated with a score, $\texttt{score}(\pi)$. The selected sample is the sample with the smallest score and is therefore deterministic. Each sample $\pi \in P$ is assigned a priority, $\texttt{priority}(\pi)$. The score for that sample is calculated

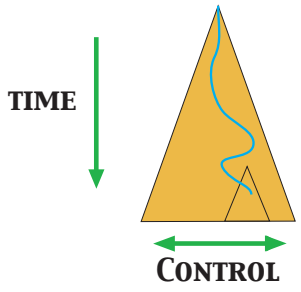

Fig. 3. Covering Control-Space

by $\texttt{score}(\pi) = \texttt{priority}(\pi)/\mu(C)$, where C is the unique cell containing π.

Algorithm 1 PDST-EXPLORE(q^0, N_{iter})

1: Let π^0 be the 0 duration path consisting of the state q^0.
2: Set the sample set $P := \{\pi^0\}$.
3: Set the subdivision $S := \{Q\}$ (the trivial subdivision).
4: Set $\texttt{priority}(\pi^0) := 0$.
5: **for** i ranges from 1 to N_{iter} **do**
6: Let γ be the sample such that $\texttt{score}(\gamma)$ is minimized.
7: Let $\pi^i := \texttt{PROPAGATE}(\gamma)$.
8: **if** π^i is a solution **then return** $\texttt{path}(\pi^i)$.
9: Add the new sample π^i to P.
10: Set $\texttt{priority}(\pi^i) := i$.
11: Set $\texttt{priority}(\gamma) := 2 \cdot \texttt{priority}(\gamma) + 1$.
12: Update S by subdividing the cell that contained γ.
13: Update P such that each sample lies in a unique cell.
14: **end for**

The data structure that stores the current subdivision S in Algorithm 1 is a binary space partition. The only operations performed on it are stabbing queries which run in time proportional to the depth of the tree. The selection on line 6 is implemented using a binary heap. The time cost for using the heap is logarithmic in the number of entries. All other lookup and referencing operations necessary for implementing Algorithm 1 are done with hash sets which run in constant time.

A sufficient condition for probabilistic completeness of a motion planner operating on a dynamical system is illustrated in Figure 3. Every subtree of positive measure in the control input-time tree must eventually be touched by the planner. The priority scheme of PDST-EXPLORE is designed to ensure this while permitting greedy coverage of the state space.

B. Design of the Controller

Let γ be a path segment of duration T. The operation $\texttt{PROPAGATE}(\gamma)$ creates a path segment π branching from γ. There are many possible choices for the PROPAGATE operation and the performance of the PDST-EXPLORE planner depends on this choice. We have observed that the following design principles are good choices: an iterated sequence of calls to PROPAGATE should be able to approximate any given path with some non-zero probability and a short sequence of iterated calls should extend into the local space around the initial segment. These principles were taken into the design and testing of the trajectory generation scheme which was used in the planner described in this paper. We now present PROPAGATE in Algorithm 2.

Algorithm 2 PROPAGATE(π)

1: Generate uniformly at random $t \in [0, |\pi|]$.
2: Let $q^0 := \pi(t)$.
3: Let x_s^0 be the ship's position at q^0.
4: Generate $x \in [0,1]^2$ uniformly and at random.
5: Generate $v_s^{\text{mag}} \in [v_s^{\min}, v_s^{\max}]$.
6: Set $v_s^{\text{targ}} := v_s^{\text{mag}} \frac{x - x_s^0}{||x - x_s^0||}$.
7: **for** i ranges from 0 to $N_{\max}$ **do**
8: Let v_s be the ship velocity of state q^i.
9: Let θ be the ship direction of state q^i.
10: Let $v := v_s^{\text{targ}} - v_s$.
11: Let θ^{targ} be the direction of vector v.
12: Let $\Delta\theta := \theta^{\text{targ}} - \theta$.
13: **if** $|v| < \delta$ **then** $u = u_0$.
14: **else if** $|\Delta\theta| < \epsilon$ **then** $u = u_1$.
15: **else if** $\Delta\theta > 0$ **then** $u = u_L$.
16: **else** $u = u_R$.
17: **end if**
18: Let $q^{i+1} := \texttt{SIMULATE}(q^i, u)$.
19: **if** q^i is a terminal state **then return** the path $\{q^0, ..., q^i\}$.
20: **end for**
21: **return** $\emptyset$.

Algorithm 2 incrementally constructs a path by running a controller with the simulator. The operation $\texttt{SIMULATE}(q^i, u)$ is the result of running the simulator described in Section II to compute the state that results from applying control u for time Δt from state q^i. The controller is designed to change the ship's velocity into a given target velocity. The target velocity has a random magnitude. Its direction is towards a randomly and uniformly chosen point in the workspace (unit square) from the ship's position at initial state of the new path. The initial state is chosen randomly from the states along the path being branched, π. The controller runs until the ship or a koule collides with the boundary or until $N_{\max}$ iterations have occurred.

In lines 4, 5 and 6 of Algorithm 2, the target velocity is computed. Notice the biased sampling that occurs as a function of the ship's current position. When the ship is close to the boundary of the workspace, the target velocity will tend to move away from the boundary. The target velocity is sampled this way to reduce the probability that the ship will collide with the boundary at the beginning of the path.

Algorithm 2 has several external parameters: the maximum number of iterations, $N_{\max}$, the minimum and maximum velocity magnitudes, $v^{\min}$ and $v^{\max}$ respectively, and the switching bounds for the controller, δ and ϵ. Choosing $\delta = \frac{a^{\max} \cdot \Delta t}{2}$ and $\epsilon = \frac{v_\theta^{\max} \cdot \Delta t}{2}$ guarantees stability.

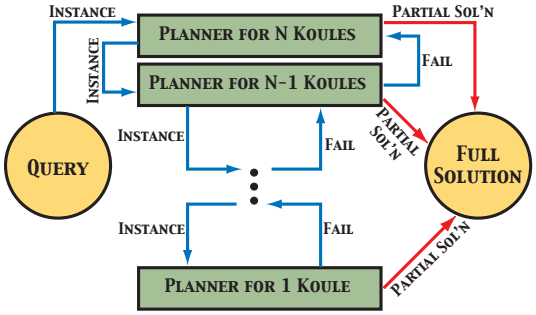

Fig. 4. Task Planner

C. Coverage Estimation

The subdivision scheme used in our implementation was relatively unsophisticated. Initial tests determined that subdividing the velocity dimensions led to poor performance. Consequently, the scheme we employed only worked on the position dimensions: $x_s, \theta, x_1, ..., x_n$. The variables were subdivided in that order and we employed uniform splits. In an example with n koules, the coverage space is $3 + 2n$-dimensional and the state space is $5 + 4n$-dimensional. The measure μ is uniform probability measure on $\mathcal{R}^2 \times S^1 \times \mathcal{R}^{2n}$.

D. Task Planning Algorithm

The PDST-EXPLORE planner creates partial solutions (recall of Subsection II-A). In order to construct a full solution, a sequence of partial solutions must be generated. It is possible that the endpoint of a partial solution may leave the system in a state from which no further solution exists. Therefore the full solution planner needs a backtracking mechanism. The method presented as Algorithm 3 is very simple but was quite effective for the purposes of the game of Koules. The method proceeds recursively: PDST-EXPLORE is invoked to find a solution and if one is found then Algorithm 3 runs on the end state of the solution path. If repeated invocations of PDST-EXPLORE fail to find a solution or if the recursive calls fail, then the recursion stack pops one level and another attempt is made. The operation of the full solution planner is depicted in Figure 4.

Algorithm 3 SOLVE$(n, q^n, N_{\text{iter}}, N_{\text{attempts}})$

1: **for** i ranges from 1 to N_{attempts} **do**
2: Let $\pi^n :=$ PDST-EXPLORE(q^n, N_{iter}).
3: **if** $\pi^n = \emptyset$ **then continue**.
4: **if** $n = 1$ **then return** π^1.
5: Let q^{n-1} be the endpoint of π^n.
6: Let $\pi^{n-1} :=$ SOLVE$(n-1, q^{n-1}, N_{\text{iter}}, N_{\text{attempts}})$.
7: **if** $\pi^{n-1} \neq \emptyset$ **then return** $\pi^n \circ \pi^{n-1}$.
8: **end for**
9: **return** $\emptyset$.

IV. Experiments

Two different kinds of experiments were run to establish evidence for our claims: partial solutions and full solutions.

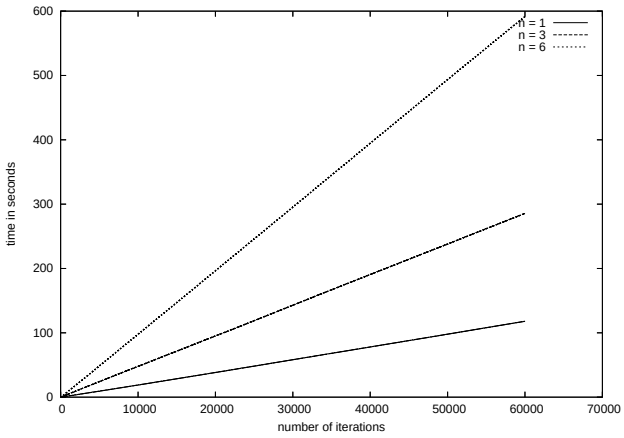

Fig. 5. Average time spent versus number of iterations for $1, 3$ and 6 Koules

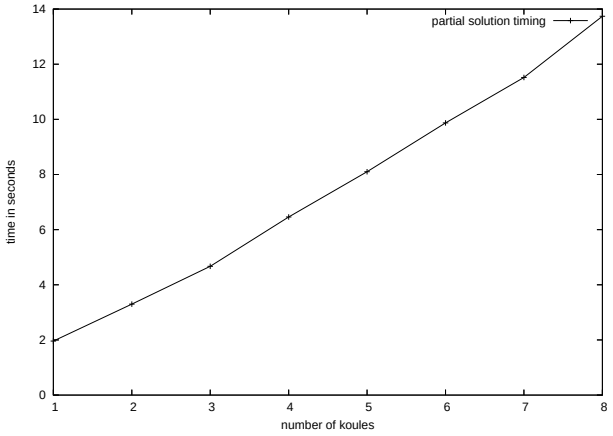

Fig. 6. Average time spent per 1000 iterations versus number of Koules

The partial solution experiments were run for various numbers of Koules. They use PDST-EXPLORE to search for paths that eliminate a Koule. The planner is allowed to continue after finding a solution and may generate many solutions. The full solution experiments were also run for various numbers of Koules and uses Algorithm 3 to construct a sequence of partial solutions each, in turn, generated with PDST-EXPLORE.

The experiments were conducted on a cluster of 16 dual AMD 1900MPs with 1 GB of RAM running Debian unstable with the 2.4.18 Linux kernel. The code is written in C/C++/fluid and uses the FLTK, GLUT, OpenGl and S-Lang packages. Throughout the experiments, the following parameters were used: $v_\theta^{\max} = \pi$, $a^{\max} = 1$, $\lambda_c = 4$, $h = 0.05$, $m_s = 0.75$, $m_k = 0.5$, $r_s = 0.03$, $r_k = 0.015$ and $\Delta t = 0.005$. These parameters were set to create a challenging motion planning task and were tuned by using an interactive interface to the game. With these parameters, we found that human players in our research group were not able to solve examples with more than a few Koules.

A. Partial Solutions

In this set of experiments, we measure the cost per iteration of PDST-EXPLORE during partial solutions. Each run was for

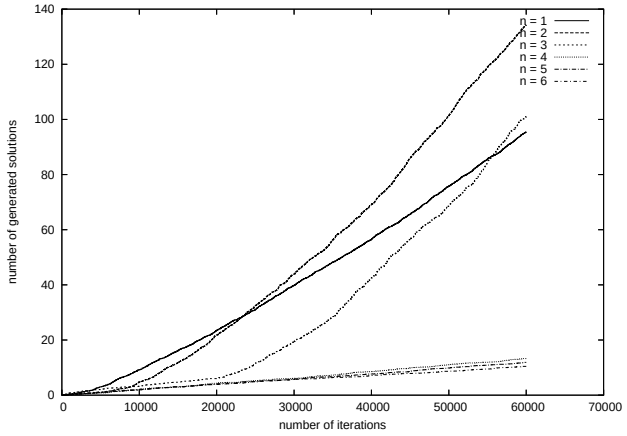

Fig. 7. Average number of solutions generated versus number of iterations

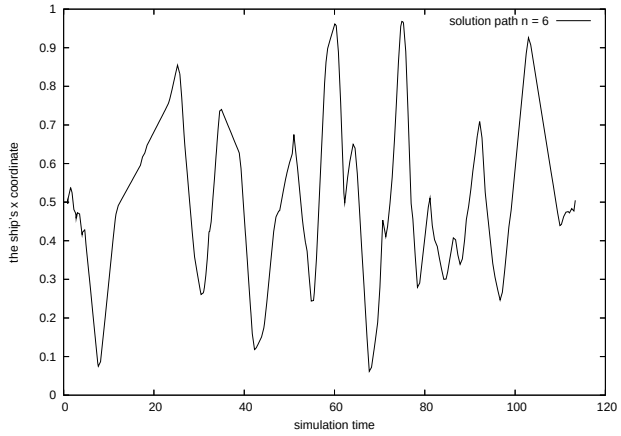

Fig. 8. The trajectory taken the ship's x-coordinate during a full solution of a problem with 6 Koules

60000 iterations and worked on a randomly generated problem instance. The data was merged and averaged from 80 runs, but for these results there was very little variations. In Figure 2, we see the total time in seconds versus the iteration counter. Although running N iterations of Algorithm 1 is guaranteed to take at least time proportional to $N \log N$, the timing plots are very close to linear. This is explained by observing that most of the runtime is spent in the simulator. The additional cost of the PDST-EXPLORE algorithm is a slight super-linear cost due to the binary space partition stab operations and the binary heap make nearly no impact on the scale of a few hundred thousand iterations. The growth in the cost of iterations is shown in Figure 3. The super-linear trend is due to the increased number of inter-robot collisions.

An important question that must be asked about Algorithm 1 is: how well does PDST-EXPLORE perform as coverage estimates become coarser due to the dimensionality increase? One way to examine this is to look at the number of solutions a run of PDST-EXPLORE generates as a function of the number of iterations. When the space becomes well covered then the rate solutions are generated frequently. Before good coverage is achieved, the solution rate will be much less. In Figure 4, we show the average solution count for partial solutions with $n = 1, ..., 6$ Koules. The sharp drop-off that occurs when moving from $n = 3$ to $n = 4$ suggests the coverage estimator begins to fail when moving from 9 to 11-dimensional space.

B. Full Solutions

Algorithm 3 is used for generating full solutions for instances of the game of Koules by repeatedly invoking PDST-EXPLORE. For each trial, we generated a random problem instance and then ran Algorithm 3. In our tests, $N_{\text{attempts}} = 1$ and $N_{\text{iter}} = 40000$ were used.

The computed paths were quite complicated, with durations of several hundred thousand simulator steps and thousands of manuevers. In Figure 5, we see an example of a computed solution for an instance with 6 Koules. The figure shows the path by the ship's x-coordinate. Qualitatively, the paths tended to look quite good. The random trajectory generation did tend

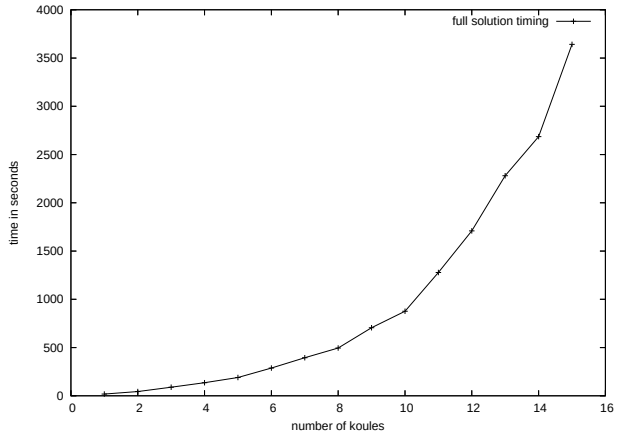

Fig. 9. Timing results for full solutions averaged over 90 trials

to produce occasional path sections where the ship coasted away from the Koules, however the usual mode was that the ship would seperate a Koule from the pack and systematically bounce it into the wall using three or four hits, while avoiding the walls and the other Koules.

In Figure 6, we present the time used by the planner to solve instances of various complexity. The number of backtracks in Algorithm 3 grew at slightly higher rate than linear with the number of Koules. This is due to PDST-EXPLORE failing to find solutions more frequently as n increases. The amount of time used grows fairly quickly with the number of Koules. This is expected to be worse than quadratic since the number of invocations of Algorithm 1 grows linearly and the cost per iteration is super-linear in the number of Koules. Experiments with up to 20 Koules were conducted and solutions were produced in less than 3 hours. The runtime began to grow very quickly around $n = 18$ due to memory paging. When $n = 15$, the state space is 65 dimensional and when $n = 20$, the state space is 85 dimensional.

C. Additional Experiments

At the end of Subsection III-B we discussed the motivation behind the design of Algorithm 2. The direction of the target velocity vector is set using the procedure on lines 4 and 6. We replaced this procedure with choosing the direction of the target velocity uniformly and randomly. We then ran full solutions trial with 3, 6 and 9 Koules and observed a severe peformance degradation. Sample bias in trajectory generation and the kinds of paths being generated are extremely important to determine the performance of PDST-EXPLORE. Biased trajectory generation helps the planner reduce the time spent searching.

In Section I, we mentioned that the difficulty of the game can be varied by modifying the physical parameters. The most important parameters for varying difficulty are relative masses of the Koules and the ship, the ship's thrust, $a^{\max}$ and the spring constant for the Koules λ_c. To our suprise, reducing the value $v_\theta^{\max}$ by a factor less than 4 did not seem to affect the solution times which is interesting as human players seem to be extremely sensitive to this parameter.

V. Discussion and Future Work

In this paper, we develop planning techniques that can handle motion planning for dynamical systems with high-dimensionality, drift, underactuation and discrete system changes. We demonstrate the robustness and efficiency of our planner by applying it to the game of Koules. To our knowledge, this is first application of algorithmic motion planning techniques to a problem of this type and complexity.

A major issue that arises in planning for dynamical systems is that memory efficiency is a very important concern. The number of iterations that we were able to perform for a partial solution was memory bounded. Once the number of states we needed to represent the tree exceeded the size of the core, performance degraded significantly. Since the cost per iteration of PDST-EXPLORE does not grow quickly, storage usage becomes the bottleneck. Using a path sample representation and by making these representations as compact as possible was essential to solve the larger examples, however handling applications with additional state complexity that may result from increases in physical realism, algorithmic changes may be necessary to further reduce storage requirements during planning. These issues are almost certainly present regardless of the planning algorithm that is employed.

Apart from attacking the storage problem, we would like to study techniques for building and using multi-query structures, improving the accuracy and utility of coverage estimates and generating paths under optimality constraints.

Acknowledgment

Work on this paper has been partially supported by NSF 0308237, NSF 0205671, NSF EIA 0216467, an FCAR fellowship to A. Ladd and a Sloan Fellowship to L. Kavraki. We also thank AMD for donating hardware.

References

[1] R. Alami, J.-P. Laumond, and T. Siméon, "Two manipulation planning algorithms," in *Algorithmic Foundations of Robotics*, K. Goldberg, D. Halperin, J.-C. Latombe, and R. Wilson, Eds. A.K. Peters, 1995.

[2] R. H. Wilson and J. C. Latombe, "Geometric reasoning about mechanical assembly," *Artificial Intelligence*, vol. 71, pp. 371–396, 1995.

[3] H. Chang and T. Y. Li, "Assembly maintainability study with motion planning," in *Proc. IEEE Int. Conf. on Rob. and Autom.*, 1995, pp. 1012–1019.

[4] W. Burgard, A. B. Cremers, D. Fox, D. Hahnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun, "The interactive museum tour-guide robot," in *Proc. Am. Assoc. Artif. Intell.*, 1998, pp. 11–18.

[5] E. Acar, H. Choset, Y. Zhang, and M. Schervish, "Path planning for robotic demining: Robust sensor-based coverage of unstructured environments and probabilistic methods," *IJRR*, vol. 22, no. 7-8, pp. 441–466, 2003.

[6] J. J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue, "Motion planning for humanoid robots," in *Proc. 20th Int'l Symp. Robotics Research*, 2003.

[7] D. Hsu, "Randomized single-query motion planning in expansive spaces," Ph.D. dissertation, Department of Computer Science, Stanford University, 2000.

[8] D. Nieuwenhuisen and M. H. Overmars, "Motion planning for camera movements," in *ICRA*, 2004.

[9] N. M. Amato, K. A. Dill, and G. Song, "Using motion planning to map protein folding landscapes and analyze folding kinetics of known native structures," in *RECOMB*, April 2002, pp. 2–11.

[10] M. S. Apaydin, D. L. Brutlag, C. Guestrin, D. Hsu, and J. C. Latombe, "Stochastic roadmap simulation: An efficient representation and algorithm for analyzing molecular motion," in *RECOMB*, April 2002, pp. 12–21.

[11] J. Latombe, *Robot Motion Planning*. Boston, MA: Kluwer, 1991.

[12] N. M. Amato, O. B. Bayazit, L. Dale, C. Jones, and D. Vallejo, "OBPRM: An obstacle-based PRM for 3D workspaces," in *WAFR*, P. Agarwal, L. Kavraki, and M. Mason, Eds., 1998, pp. 156–168.

[13] L. E. Kavraki, P. Švestka, J. C. L. e, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *TRA*, vol. 12, no. 4, pp. 566–580, June 1996.

[14] R. Geraerts and M. Overmars, "A comparitive study of probabilistic roadmap planners," in *Algorithmic Foundations of Robotics V*, J.-D. Boissonnat, J. Burdick, K. Goldberg, and S. Hutchinson, Eds. Springer Verlag, 2003, pp. 43–57.

[15] J. Cortés and T. Siméon, "Sampling-based motion planning under kinematic loop closure constraints," in *Proc. of Workshop on Algorithmic Foundations of Robotics*, 2004.

[16] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani, "Manipulation planning with probabilistic roadmaps," *IJRR*, no. 23, pp. 729–746, 2003.

[17] O. B. Bayazit, J.-M. Lien, and N. M. Amato, "Probabilistic roadmap motion planning for deformable objects," in *ICRA*, 2002.

[18] J.-P. Laumond, "Feasible trajectories for mobile robots with kinematic and environment constraints," in *The International Conference on Intelligent Autonomous Systems*. Amsterdam, The Netherlands: Elsevier Science Publishers, B.V., 1986, pp. 346–354.

[19] F. Lamiraux and J.-P. Laumond, "Smooth motion planning for car-like vehicles," in *ICRA*, 2001.

[20] P. Švestka and M. H. Overmars, "Coordinated motion planning for multiple car-like robots using probabilistic roadmaps," in *IEEE Int. Conf. Robot. & Autom.*, 1995, pp. 1631–1636.

[21] S. M. LaValle and J. J. Kuffner, "Randomized kinodynamic planning," *IJRR*, vol. 5, pp. 348–400, May 2001.

[22] D. Hsu, R. Kindel, J. C. Latombe, and S. Rock, "Randomized kinodynamic motion planning with moving obstacles," *IJRR*, pp. 233–255, 2001.

[23] P. Cheng and S. LaValle, "Resolution-complete rapidly-exploring random trees," in *ICRA*, 2002, pp. 267–272.

[24] A. Ladd and L. Kavraki, "Fast exploration for robots with dynamics," in *WAFR*, 2004.

[25] J. Hubicka, "Koules homepage," www.ucw.cz/~hubicka/koules/English/koules.html, 1995.

[26] M. Abramowitz and I. Stegun, Eds., *Handbook of Mathematical Functions*. National Bureau of Standards, Dover, 1974.