Multi-robot collective path finding in dynamic environments

Carlos Astengo-Noguez, Gildardo Sanchez-Ante*,
José Ramón Calzada and Ricardo Sisnett-Hernández*
*Tecnológico de Monterrey, Campus Monterrey, *Campus Guadalajara*
México

1. Introduction

Planning the motion of a rigid or articulated object among static obstacles is known as the basic motion planning problem. In its simplest definition, it is a purely geometric problem that only considers avoiding collision with static obstacles, given that the robot is the only object that is able to move in the environment. This problem has been extensively studied in the last two decades, and we do know that the problem itself can be very challenging. For instance, it has been demonstrated that planning the motion of a set of polyhedral objects forming an articulated structure in an environment with static polyhedral obstacles is PSPACE-hard (Reif, 1979). That means that although some elegant complete algorithms exist –the most efficient complete and general algorithm for basic motion planning has a $O(2^n)$ complexity (Canny, 1988)-- their prohibitive computational cost has motivated the search for efficient algorithms that can run in significantly less time despite the fact that they are not complete.

In many practical problems the environments is more complex than in the basic motion planning problem. For instance, it might be the case that there are objects that are supposed to be moved by the robot, or the objects might not be rigid but deformable, or there could be objects moving in the environment whose trajectory might not be known in advance by the robot. All those cases can be considered as extensions of the basic motion planning problem. In this chapter we are interested in analyzing problems where several robots are moving in a shared workspace. This topic is raising interest in the community not only because many researchers argue that we have covered very well the simplest cases by designing efficient algorithms for them, but also because some applications are already pushing the envelop towards the automatic generation of behaviors for agents in dynamic and uncertain environments.

The chapter is structured as follows: first, we will introduce some concepts and notation commonly used in motion planning. Then, a description of some methods for motion planning in single-robot environments is offered. The last part of the chapter is about methods for multi-robot motion planning.

2. Concepts and Notation

A *configuration* of a robot is a list of parameters that uniquely defines the placement of the robot in its workspace (Lozano-Perez 1979; Lozano-Perez, 1983). For example, the configuration of a rigid body is its position and orientation. A configuration q is free if the object does not collide with the obstacles in the environment or with itself when placed at q. For any given robot, there are multiple ways of defining a configuration. The selected definition may affect the geometry of some sets, like the free space, but not their connectivity, which is what matters in our case. The set of all configurations forms the *configuration space C*. The C space is then the union of all the configurations of the robot, some of them free of collision (C_{free}) and some others in collision (C_{obs}). $C = C_{free} \cup C_{obst.}$ For a polygonal rigid body R that translates in a 2-D polygonal space, the configuration space can be explicitly computed by taking the Minkowsky difference of R and the obstacles. Intuitively, that means that we "grow" the obstacles by the shape of R and then R becomes a point in this space. This transformation implies that finding the path for a robot means constructing a one-dimensional curve in this space, regardless the fact that the robot itself can be located in a three-dimensional space and may have many degrees of freedom (dof). It is important to mention that the configuration space will have as many dimensions as degrees of freedom may have the robot.

There is a distinction between path planning and motion planning. A *path* is a continuous curve on the configuration space. It is represented by a continuous function that maps some path parameter, usually taken to be in the unit interval [0, 1], to a curve in C_{free} . The choice of unit interval is arbitrary; any parameterization would suffice. The solution to the path planning problem is a continuous function $c \in C^0$ such that
$c : [0, 1] \rightarrow C$ where $c(0) = q_{start}$, $c(1) = q_{target}$ and $c(s) \in C_{free}\ \forall\ s \in [0, 1]$.

When the path is parameterized by time *t*, then *c(t)* is a trajectory, and velocities and accelerations can be computed by taking the first and second derivatives with respect to time. This means that c should be at least twice-differentiable. Finding a feasible trajectory is called trajectory planning or motion planning.

Navigation is the problem of finding a collision-free motion for an agent-system from one configuration (or state) to another. The agent could be a videogame avatar, a mobile robot, or something else. Localization is the problem of using a map to interpret sensor data to determine the configuration of the agent. *Mapping* is the problem of exploring and sensing an unknown environment to construct a representation that is useful for navigation or localization. Localization and mapping can be combined.

3. Single-Robot Motion Planning Algorithms

Most of the work in motion planning has been done considering the case of a single robot moving in an environment populated with static obstacles. The problem can then be stated as finding a collision-free path from any given starting position to a goal or desired location for the robot. In some cases, a function to measure cost is introduced, so that the algorithm is able to search for the optimal path (i.e., minimum cost).

In general, we could classify the algorithms as complete or incomplete. On the one hand, a **complete** algorithm is one that either finds a solution or proves that it does not exist. Some authors call these algorithms *exact*. Algorithms under this classification are usually computationally expensive and, by consequence, impractical for many important instances of the problem. That is the case of the algorithm of Canny (Canny, 1987). Canny's algorithm is the most efficient general path planning algorithm known to date, with a time complexity of $O(2^n)$. The method involves powerful techniques from real algebraic geometry (Canny, 1987; Canny, 1988), but is nevertheless exponential in *n*.

On the other hand, **incomplete** algorithms are not able to offer such guarantee. When a complete algorithm is impractical, despite its elegance, what computer scientists would like to have is an algorithm that satisfies another notion of completeness, such as resolution completeness or probabilistic completeness. In the first case, we say that an algorithm is *resolution complete* if its accuracy arbitrarily improves when the resolution is increased, and it becomes an exact algorithm at the limit where the resolution approaches the continuum. Some of the cell decomposition methods are resolution complete, for instance (Zhu, 1990; Kondo, 1991). For the second case, we say that an algorithm is probabilistically complete if the probability of finding a solution (if one exists) approaches 1, as the running time increases. Algorithms such as those described in (Barraquand, 1990; Kavraki, 1996; Hsu, 1997) are probabilistically complete.

One of the main drawbacks of resolution complete methods is that the number of points required for the discretization of the configuration space grows exponentially in the number of degrees of freedom. Conversely, in a probabilistic complete method, we would need to guarantee an adequate coverage of the configuration space in order to have a high probability of finding a path whenever it exists.

Some well known examples of motion planning algorithms are:

Cell Decomposition: Based on the idea of decomposing the C_{free} space into convex regions (either regular or not) and using that discretization to build a representation of the connectivity of C_{free} (usually a graph), and then searching for the path in the graph. Cell decomposition is an approach that sounds simple at first sight, but whose complexity grows quickly with the number of degrees of freedom of the robot. Also, since it is conceived to work on the *C* space, it requires the computation of C_{free}, which may take very long time, depending on the dofs of the robot. It is an approach that, in general, can only be applied in very simple environments, making it unsuitable for real problems where obstacles may have complex geometries and the robot may have many degrees of freedom (Latombe, 1991).

Potential Fields: Khatib introduced this approach, in which the idea is to consider the point robot in the configuration space as a charged particle under the influence of an artificial potential field in a way that the particle is "pushed" away from the obstacles and "attracted" towards the goal. The vectorial sum of those forces defines the motion of the robot to a new location and then the potentials are computed again and the whole loop is repeated. The approach originally was intended to be used as an on-line process for mobile robots

equipped with sensors such as sonars. It has proven to be an approach useful for real robots moving on a plane, i.e., 2 or 3 degrees of freedom (Khatib, 1983).
The main drawback of this approach is the tendency it has to get trapped in local minima (since it relies on a greedy search algorithm). Also, unless the *C* space is pre-computed, it is not possible to guarantee optimal paths or even the completeness of the algorithm. In practice, the main problem is usually the definition of "good" functions to represent attractive and repulsive forces.

Roadmaps: In this approach, the idea is to represent the connectivity of the C_{free} space by a one-dimensional curve, called roadmap. Once this curve is constructed, it is used to search for paths connecting some given initial and goal configurations. There are different ways of constructing the roadmap, such as: visibility graphs, Voronoi Diagrams, Silhouettes, etc. The main problem with the roadmaps approach is that computing such curve in high-dimensional spaces is almost prohibitive in terms of computational time, making the approach not useful for environments involving many degrees of freedom.

More recently, a new family of methods have been proposed, based on the idea of sampling the configuration space instead of actually computing it. The approach relies on recent results for algorithms that can enable the computation of collision checks in very short amounts of time, such as object oriented bounding boxes (OBB) (ref), axis aligned bounding boxes (AABB), Spheres and other options (Lin, 1998).

The methods based on this idea are called Probabilistic Roadmaps (PRM), and we will describe them in more detail in the next section.

The PRM Planning Approach

Sampling-based motion planning is a well-known concept, (see, for instance (Donald, 1987)) that was originally used to deal with some difficulties encountered while implementing complete planners. PRM planning consists of sampling the configuration space at random and testing the sampled points, as well as connections between them, for collision.

The obstacle regions in configuration space make explicit the constraints on the possible motions of a robot. These constraints derive from the interaction between the geometric shapes of the robot and the obstacles in the workspace. Small and geometrically simple obstacles in the workspace may yield complex and large obstacle regions in the configuration space *C*.

There are two major issues in computing an explicit geometric representation of the obstacle regions in *C*. First, *C* has as many dimensions as the robot system has dofs. Second, the shape of C_{free} may be complex even when both the robot and the obstacles have simple shapes.

For those reasons, computing an explicit geometric representation of C_{free} is not possible in practice (even with much greater computational power than is available today). On the other hand, efficient collision-detection techniques have existed for some years, which can determine quickly whether an arbitrary robot configuration is collision-free, or not.

The existence of fast collision-checking techniques has led to the idea of probing the configuration space at random. One may pick many sampled configurations and test them for collision. Thus, the collision-free samples form a discrete approximation of the free space. This is the basic idea underlying probabilistic roadmaps (Kavraki, 1994).

There are two main classes of PRM planners: multi-query and single-query planners. Historically, multi-query planners were developed first (Kavraki, 1994; Kavraki, 1996; Svestka, 1997; Amato, 1998b}. Single-query planners are more recent (Hsu, 1997; Hsu, 2000; Kuffner, 2000; LaValle, 2001), but they have significant advantages over multi-query planners. Additionally, mixed planners have also been proposed in (Amato, 1998; Bohlin, 2000; Nielsen, 2000; Song, 2001). Basically, their goal is to distribute the time over a pre-computation and a query phase.

Multi-Query Planners

A multi-query PRM planner operates in two phases. It first pre-computes a probabilistic roadmap *R*. Then it uses *R* to answer an arbitrary number of queries, each defined by a pair of configurations. Each query must be made in the same robot-obstacle environment for which *R* was computed. The pre-computation of *R* may be rather expensive, but it is "amortized" over the several queries that are subsequently made (Kavraki, 1994; Kavraki, 1996). Processing one query is usually extremely fast.

Building the roadmap

A roadmap *R* is pre-computed by repeatedly sampling the configuration space *C* at random. Each sample is tested for collision, and the collision-free samples are retained as *milestones*. Then, the planner connects pairs of milestones that are not too far apart by simple paths and retains those which test collision-free as *local paths*. The milestones and local paths form a network over C_{free}, which is called the *probabilistic roadmap*. It is stored as an undirected graph, which usually has a large number of vertices (typically, several thousands to a few millions) (Kavraki, 1994; Kavraki, 1996).

More formally, the algorithm is as follows (*d* is the metric function in *C*):

```
BUILD_ROADMAP
        1 $R \leftarrow$empty graph
        2 Repeat until $s$ milestones have been generated
                3 Pick a configuration $q$ uniformly at random in $C$
                4 If $q$ is collision-free then add $q$ as a new vertex (milestone) of $R$
                5 For each pair of milestones $q$, $q'$ such that $d(q,q') \le \rho$
                6 If the line segment joining $q$ and $q'$ tests collision-free then
                  add it as an edge (local path) of $R$
        7 Return $R$
```

The algorithm consists of two independent loops. The first loop (lines 2-4) adds milestones to the roadmap. The parameter *s* defines the number of milestones to be generated. It is often called the *size* of the roadmap. The second loop (lines 5-7) establishes the local paths. Local paths are created only between milestones that are closer apart than some predefined distance ρ. Indeed, there are O(s^2) pairs of milestones and testing all segments would be too costly. Moreover, if two milestones are far apart, the segment joining them is unlikely to be collision-free, and, if it is collision-free, then the above algorithm is likely to connect the two milestones by a sequence of local paths through intermediate milestones. Figure 1 illustrates the process.

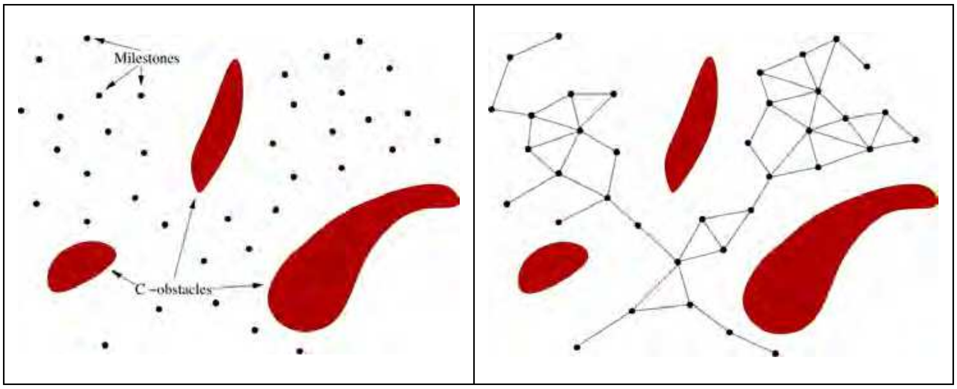

Fig. 1. Two steps in the computation of the roadmap. The image in the left shows the sampled milestones. The image on the right shows the graph built after calling the local planner.

Querying the roadmap

A query is defined by two configurations, q_i and and q_g. To answer the query, the planner first attempts to connect each of these configurations to a milestone of *R* by a local path. If the two connections succeed, then the planner searches *R* for a sequence of local paths connecting q_i and q_g. Such a sequence, if one is found, constitutes a free path for the robot.

The algorithm is as follows:

```
QUERY_ROADMAP($q_i$,$q_g$)
    1 Repeat for all $q \in R$ such that $d(q, q_i) \leq \rho$
        2 If the line segment joining $q_i$ and $q$ tests collision-free then
          connect $q_i$ to $q$ and exit loop
    3 If $q_i$ have not been connected to a milestone of $R$ then return failure
    4 Repeat for all $q \in R$ such that $d(q, q_g) \leq \rho$
        5 If the line segment joining $q_g$ and $q$ tests collision-free then
          connect $q_g$ to $q$ and exit loop
    6 If $q_g$ have not been connected to a milestone of $R$ then return failure
    7 Search $R$ for a path connecting $q_i$ to $q_g$
    8 If a path has been found, then return this path, else return no path
```

The algorithm returns *failure* if it fails to connect q_i or q_g to the roadmap. It returns *no path* if it fails to find a path connecting q_i to q_g after they have been successfully connected to the roadmap. The possible interpretations of these two outcomes will be discussed below.

Probabilistic completeness

There are two cases where QUERY_ROADMAP does not return a path: (1) if it fails to connect q_i or q_g to the roadmap, and (2) if it fails to find a path in the roadmap. Clearly, it is desirable that the first case happens as rarely as possible. In the second case, the algorithm's output is *no path*. This output may be correct, as it is possible that q_i and q_g belong to two distinct components of the free space C_{free}. But it may also be incorrect: q_i or q_g may belong to the same component of C_{free}, but the roadmap R may have more than one component lying in it. It is desirable that the planner rarely returns "*no path*" incorrectly.

Single-Query Planners

Multi-query PRM planners are appropriate when the pre-computation of a roadmap can be amortized over a rather large number of queries performed in the same environment. However, in practice, the number of queries in a given environment is rather small, as one or several objects are often moved, deleted, or added between two queries. Single-query PRM planners are a better solution in those cases.

A single-query PRM planner computes a new probabilistic roadmap for each query (Hsu, 1997; Hsu, 2000; Kuffner, 2000). While multi-query planners must use a sampling strategy that covers well the whole free space, in order to later successfully deal with any query, a single-query planner applies a more focused strategy aimed at exploring the smallest portion of free space needed to find a solution path. More precisely, it takes advantage that it knows the two query configurations to explore restricted subsets of the components of C_{free} that are reachable from these configurations. This is done either by growing one tree of milestones rooted at one query configuration, until a connection is found with the other query configuration (*single-directional* search), or by growing two trees concurrently, respectively rooted at one of the two query configurations, until a connection is found between the two trees (*bi-directional* search) (Hsu, 2000). In both cases, milestones are iteratively added to the roadmap. Each new milestone m' is selected in a neighbourhood of a milestone m already installed in a tree and is connected to m by a local path (hence, m' becomes a child of m). Bi-directional planners are usually more efficient than single-directional ones (Amato, 1998; Hsu, 1998; Hsu, 1999; Hsu, 2000). Fig. 2 shows an example of a single-query planner in process.

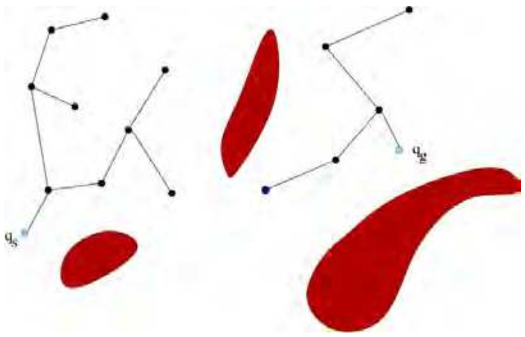

Fig. 2. A single-query, bidirectional planner. Two trees are grown rooted at the start and goal configurations.

A single-query planner is allowed to generate a maximal number *s* of milestones. Either it outputs a free path between the query configurations, or it indicates that it has not found a path after generating *s* milestones. The second output occurs when the two query configurations lie in two distinct connected components of C_{free}. It may also occur when a solution path exists, but the planner did not find one because *s* was set too small.

In (Hsu, 2000), it is shown that if C_{free} is expansive, then the probability that a slightly idealized version of a single-query PRM planner fails to find a path when one exists goes to 0 exponentially in *s*. This property defines the probabilistic completeness of a single-query PRM planner. The proof in (Hsu, 2000) requires that the milestones generated by the planner eventually ``diffuse'' through the components of *F* reachable from the query configurations.

Intermediate Planners

There are several planners which posses characteristics of both multi-query and single-query planners. Below we briefly sketch some of them.

In (Bohlin, 2000) a Lazy PRM is described. The algorithm is similar to the original PRM (Kavraki, 1994) in the sense that the aim is to find the shortest path in a roadmap constructed by randomly distributed configurations. Nevertheless, in this approach, instead of constructing a roadmap of feasible paths, a roadmap of paths *assumed* to be feasible is build. The idea is to lazily evaluate the feasibility of the roadmap as planning queries are processed. In other words, a number of uniformly distributed points form nodes in a roadmap, and the connections between nodes being sufficiently close form the edges on the roadmap. Neither nodes nor edges are validated until a possible solution path is found. At that moment, both edges and nodes are checked for collision. If a collision is found, the corresponding node/edge is removed and the search process is re-started.

In (Nielsen, 2000) a Fuzzy PRM planner is presented. In such approach, the process is started in the query phase, and if the roadmap does not contain a possible solution path, it enters to the learning phase, adding milestones and edges. The milestones are always collision-free configurations, while in the case of the edges, they are annotated by a probability. This probability is an estimate of the chance that the edge is actually feasible. The query phase is split into three steps: update, search and upgrade. The update step adds nodes to the graph, starting with the query ones. In the search step, the most probable path is found from start to goal configurations on the graph. The upgrade step is used to do the actual verification of the path. As a result of the application of this step, the probabilities on the edges are upgraded.

In (Song, 2001) a "Customizable" PRM planner is described. In the learning step a coarse roadmap is constructed by performing only approximate validation of nodes and edges. In the query step, the roadmap is validated and refined only in the area of interest for the query. Moreover, it is ``customized'' in accordance with any specified query preferences (e.g., maintaining certain clearance from the obstacles).

In (Vallejo, 2001) an adaptive framework for single-query (or *single-shot*) planning is presented. In this approach, two trees are constructed, rooted on the start and goal configurations. In each iteration, it is attempted to generate a path that connects both query configurations. To do this, all potential query pairs with one configuration in each tree, and all the algorithms in the bank are evaluated, and it is selected the query pair and algorithm combination that is most likely to make a connection. The approach assumes that several planners are available.

4. Multiple-Robot Motion Planning Algorithms

There are two established approaches to multi-robot motion planning: centralized and decoupled (Latombe, 1991). So far, the prevalent approach has been decoupled planning. In most cases, centralized planning has been beyond the practical capabilities of existing planning techniques, as it requires searching configuration spaces with many dimensions. Instead, decoupled planning breaks the original planning problem into several more tractable sub-problems. Despite the fact that decoupled planning is known to be inherently incomplete – that is, it is not guaranteed to find a solution whenever one exists – it has been assumed that the loss of completeness is relatively small in most practical cases and worth the gain in computational time.

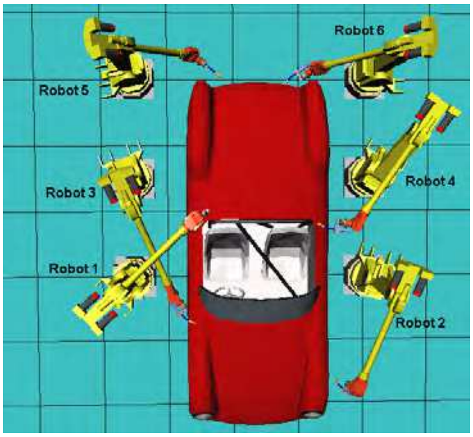

Fig. 3. Model of six-robot spot-welding station.

Centralized planning

It consists of considering all the robots involved in the problem as if they were forming a single multi-limb robot, by encoding all their dofs in a single "composite" configuration space *C* and searching that space for a collision-free path between the initial composite configuration and the goal one. In such case, the "total" configuration space is given by the combination of the configuration spaces of all the robots: $C = C_1 \times C_2 \times ... \times C_p$, where *p* is the number of robots and C_i is the configuration space of the *i*-th robot ($i \in [1,p]$). Thus, the number of dimensions of *C* is equal to the total number of dofs of the robots. In the example of Fig. 3, where each robot has 6 dofs, the composite configuration space has 36 dimensions.

Let τ: $s \in [0,1] \leftarrow \tau(s) \in F$ be a path in the free subset *F* of *C*. The projection τ_i of τ into the subspace C_i is the path to be followed by the *i*-th robot. For each $s \in [0,1]$, $\tau(s)$ is of the form $(\tau_1(s), \tau_2(s), ... , \tau_p(s))$, which describes the configurations of the *p* robots at a single point along the path τ. Hence, a collision-free path in *F*, if one exists, not only describes the individual path to be followed by each robot, but also how the robots are to be coordinated.

In principle, any sufficiently general path-planning algorithm can be used to implement centralized planning. This only requires applying this algorithm to the composite space *C*. However, in the past, centralized planning has not been considered practical because it usually leads to searching large-dimensional configuration spaces that are beyond the practical capabilities of existing planning techniques. Most proposed centralized planners have been based on ad-hoc and incomplete heuristics, for example potential field techniques, which are too unreliable to be widely useful (Tournassoud, 1986; Barraquand, 1990; Barraquand, 1991; Barraquand, 1992). Complete centralized planning algorithms have only been proposed for very simple robotic systems, e.g., the coordination of discs among polygonal obstacles (Schwartz, 1983). The complexity of the algorithm described in that work is $O(n^3)$ for two discs, and $O(n^{13})$ for three discs.

Centralized approaches have the advantage that, at least in theory, they allow for complete planners.

Decoupled planning

This is a two-phase approach. In the first phase, a collision-free path is generated for each robot by considering only the obstacles in the environment and ignoring the other robots. In the second phase, called *velocity tuning*, the relative velocities of the robots along their respective paths are selected to avoid collision among them (Kant86, Odonnell89, Alami98,Aronov99}.

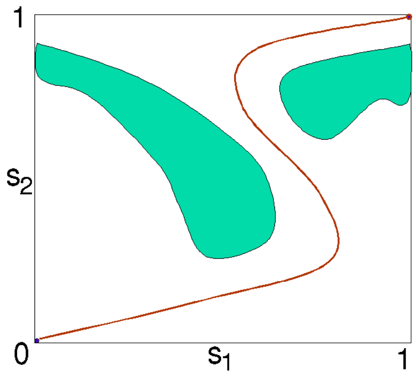

Fig. 4. Coordination space for two robots.

Velocity tuning consists of searching a coordination space. Consider two robots, and let τ_1 and τ_2 be the two paths (one for each robot) generated in the first phase of decoupled planning. By forcing the robots to move along these paths, we actually reduce the number of dofs of each robot to 1, hence the dimension of their composite configuration space -- now called the *coordination space* -- to 2 (O'Donnell, 1989).

Let each path τ_i (i = 1,2) be parameterized by some $s_i \in [0,1]$. The set $P = [0,1] \times [0,1]$ represents the coordination space of the two robots (see Fig. 4). Each point $(s_1,s_2) \in P$ defines a placement of the two robots at their respective configurations $\tau_1(s_1)$ and $\tau_2(s_2)$. This point is collision-free if at this placement the two robots do not collide with each other. (Collisions with obstacles in the environment were already taken care of during the generation of τ_1 and τ_2). A path joining the point (0,0) -- where both robots are at their respective initial configurations -- to the point (1,1) -- where they are at their goal configurations -- in the collision-free subset of *P* defines a valid coordination of the two robots along τ_1 and τ_2; it determines the relative velocities of the robots along their respective paths. Note that this path may not be monotonic along any of the dimensions of *P*. If it is not non-monotonic along s_i (i = 1 or 2), then for a while the *i*-th robot will move backward along τ_i. Such motion may be required to provide maneuvering space to the second robot. An unfortunate choice of τ_1 and τ_2 in the first phase of decoupled planning may lead the points (0,0) and (1,1) to lie in two distinct connected component of the free subset of *P*.

If there are $p > 2$ robots, one may coordinate all the robots by generating a collision-free path in the *p*-dimensional space *P* where the *i*-th axis encodes the parameter s_i of the path of the *i*-th robot, from the point (0,...,0) to the point (1,...,1). We term this approach to velocity tuning *global coordination*. An alternative, *pairwise coordination*, consists of planning *p*-1 paths in a series of *p*-1 two-dimensional spaces $P_2,\dots,P_p$. The axes of P_2 encode the parameters s_1 and s_2 along the paths of the 1st and 2nd robots, and a collision-free path $\tau_{1,2}$: $s_{1,2} \in [0,1] \rightarrow \tau_{1,2}(s_{1,2}) \in P_2$ defines a valid coordination of these two robots. One axis of P_3 encodes the parameter s_3 along the path of the 3rd robot, while the other axis encodes the parameter $s_{1,2}$ along the coordinated path of the 1st and 2nd robots. Hence, each point in P_3 determines a placement of the first three robots, and a collision-free path in P_3 defines a valid coordination of these robots.

Decoupled planning leads to searching lower-dimensional spaces than centralized planning. But, it is inherently incomplete, even if the core planning algorithms used in the first and second phases are complete. Velocity tuning may fail because the paths generated in the first planning phase cannot be coordinated without collision between robots, while this coordination would have been possible if other paths had been selected. A decoupled planner based on global coordination is less incomplete than one based on pairwise coordination, since a specific path selected in the path space P_i may result into a space P_{i+1} with no collision-free path between (0,...,0) and (1,...,1). Nevertheless, in the past, pairwise coordination has been more widely used than global coordination, since it only requires planning in two-dimensional spaces. In theory, when velocity tuning fails, the planner could backtrack and generate new robot paths. But this option has rarely been used. Indeed, it is difficult to extract from a failure the information that can be used to generate new paths. Moreover, backtracking quickly increases the planner's running time.

An alternative to velocity tuning, called *prioritized planning*, is proposed in (Erdmann, 1986). It consists of processing the robots in some predefined order and planning the path of each robot by treating the robots whose paths have already been planned as moving obstacles of known trajectories. A problem with this approach is finding a good way of defining the priorities for the robots, as this assignment affects the likelihood of finding the solution.

In (Sánchez, 2002), we describe the use of the SBL planner to implement both the centralized and the decoupled approaches. Interestingly, SBL can be invoked at each stage of decoupled planning, not only to plan individual paths of robots, but also to coordinate these paths (velocity tuning). We give experimental results obtained for the model of a 6-robot spot-welding station shown in Fig. 3, comparing the relative performance and reliability of centralized planning, decoupled planning with global coordination, and decoupled planning with pairwise coordination (these terms will be precisely defined below). The results reveal that, in the context of multi-robot spot welding, which requires rather tight robot coordination, decoupled planning is too un-reliable to be practical. This is an important observation, since it invalidates the assumption that the loss of completeness in adopting decoupled planning is not very significant in practice and indicates that centralized planning is a more desirable approach. By no means, however, does this imply that decoupled planning is useless. First, it may be reasonably reliable for other applications where interactions among robots are less constraining. Second, there are distributed-robot systems where centralized planning is not possible because no robot or processor knows the global state of the system or the goals of all robots. Finally, even in cases where decoupled planning is possible but unreliable, a decoupled planner may still have some utility if it receives interactive hints from a human user.

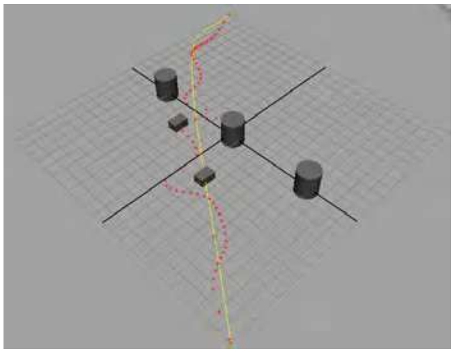
Fig. 5. Snapshot of a path obtained in Maya™ using SBL Planner

We have recently programmed SBL inside Autodesk Maya software, using Python. In Fig 5. it is shown the path obtained by SBL (red) and the optimized path (yellow). In general, computing such paths take less than a second for uncluttered 3D environments and a robot with 2-3 dofs.

5. Multiple-Robot Planning as Multi-Agent Systems

A somewhat different way of dealing with the coordination of multiple robots is based on the idea of the robots forming a multi-agent system. A multi-agent system (MAS) is a system composed of multiple interacting intelligent agents. Multi-agent systems can be used to solve problems which are difficult or impossible for an individual agent to solve.
The agents in a multi-agent system have several important characteristics (Shoham, 2008):

- *Autonomy*: the agents are at least partially autonomous.
- *Local views*: no agent has a full global view of the system, or the system is too complex for an agent to make practical use of such knowledge.

* *Decentralization*: there is no designated controlling agent (or the system is effectively reduced to a monolithic system).

Typically multi-agent systems research refers to software agents. However, the agents in a multi-agent system could equally well be robots, humans or human teams. A multi-agent system may contain combined human-agent teams.

Multi-agent systems can manifest self-organization and complex behaviors even when the individual strategies of all their agents are simple.

Agents are assumed to operate in a planar (R^2) or three dimensional (R^3) environment or (vectorial) space, called workspace *W*. This workspace will often contain obstacles; let WO_i be the i-th obstacle. Motion planning, however, does not usually occur in the workspace. Instead, it occurs in the configuration space (also called C-space).

The game industry is demanding algorithms that allow multiple agents to plan for non-colliding routes on congested-dynamical environments (Pottinger, 1999), such problems also appear in flock traffic navigation (FTN) based on negotiation (Astengo, 2007).

FTN uses A* for path planning and, only at intersections, flocks use the Dresner and Stone reservation algorithm (Dresner, 2005). We intend to improve this algorithm using cooperative strategies.

Cooperative Pathfiding

In cooperative pathfinding each agent is assumed to have full knowledge of all other agents and their planned routes (Silver, 2005). The complementary problem is called "non-cooperative pathfinding", where the agents have no knowledge of each other's plans and must predict their future movements. There is also another approach called "antagonist pathfinding" where agents try to reach their own goals while preventing other agents from reaching theirs.

FTN based on negotiation uses a decoupled approach called local repair A*: each agent searches for a route to the destination using A*, ignoring all other agents except for its current neighbors. It is in this neighborhood that negotiation takes place and the flock is created. The agents then follow their route (according to the bone-structure) until a collision is imminent.

It is clear that collisions will happen at the intersections, so there are, with this approach, two possible solutions:

1. The Dresner and Stone reservation method (Dresner, 2004; 2005; 2006; 2007).
2. The Zelinsky (Zelinsky, 1992) brute force algorithm: whenever an agent or a flock is about to move into an occupied position it instead recalculates the remainder of its route.

The implementation of each method depends on the information and the time that the agent or flock has at that particular moment. The Zelinsky algorithm usually suffers from cycles and other severe breakdowns in scenarios where bottlenecks are present (Pottinger, 1999; Zelinksy, 2992).

The Dresner and Stone reservation model was developed for individual agents that can accelerate or decelerate according to the reservation agenda. In simulations performed it was shown that, at the beginning, it works properly, but as time moves on the agents are eventually stopped.

Cooperative Pathfinding with A*

The task is decoupled into a series of single agent searches. The individual searches are performed in three-dimensional space-time and takes the planned routes of other agents into account. A wait move is included in the agent's action set to enable it to remain stationary. After each agent's route is calculated the states along the route are marked into a reservation table. Entries in the reservation table are considered impassable and are avoided during searches by subsequent agents.

The reservation table represents the agents' shared knowledge about each other's planned routes. The reservation table is a three-dimensional grid: two spatial dimensions and one time dimension. Each cell that is intersected by the agent's planned route is marked as impassable for precisely the duration of the intersection, thus preventing any other agent from planning a colliding route. Only a small proportion of grid locations will be touched, and so the grid can be efficiently implemented as a hash table (Silver, 2005).

New Cooperative Pathfinding Algorithm

Using the Cooperative A* a D* algorithms as a starting point, we can propose a new algorithm that can use similar techniques to plan paths in a dynamic environment in which several agents exist.

As in Cooperative A*, a reservation table is used to store the agents' planned paths and the time at which they will occupy certain regions in space. Assuming a three-dimensional space, we need a four-dimensional table that allows us to reference a specific point in space at a certain point in time. Any dynamic elements of the environment and their movement need to be included in the table as well.

Agents rely on a visibility index to determine how far ahead in time they can detect potential collisions with other agents or objects. In a fully cooperative environment where all agents have complete access to the planning information of other agents, this index is equal to the total amount of time required for all agents to move through their planned paths.

Once the visibility index has been established, all agents will be added to a priority queue, where their priority will be the delay the agent has suffered due to adjustments made to its path in order to avoid collisions. At first, all agents start with a priority value of zero.

During execution, each agent will be removed from the queue and then, assuming an initial planned path that was the result of an A* algorithm, will attempt to make a reservation in the table that covers the different points in space-time that correspond to the path of the agent within the visibility index previously established.

If the reservation is successful, then we update the agent's position according to its planned path and velocity. If not, then we need to roll back any entries made to the table within the current reservation for this particular agent and detect at which point in time the collision would have occurred.

Once, and if, we have set the collision time, we can use it to calculate a lower velocity that would allow us to avoid it. Agents can optionally set a speed threshold which would prompt the agent to calculate a new path, using A*, from its current position to its desired position in case the new speed would fall below it. In any case, the delay caused by the modifications is calculated and then stored. This is the value that determines the agent's priority when making a reservation, which ensures that agents do not receive preferential treatment.

```
COOPERATIVE_PF()
Until all agents have reached their goals do
    For each agent in priority queue do
        Remove agent from queue
            For t=0 to t=v
                Calculate position occupied at t according to current path
                Attempt to make reservation
                if reservation not successful
                    Undo previous reservations
                     set collision Time
                     set collision Agent
                     break;
                end
                if collision Time = -1 do
                    update agent position according to its velocity and path
                else
                     calculate speed decrease required to avoid collision
                     if speed after decrease < speed threshold
                         calculate new path from goal to current position

             calculate time difference between modified time and speed and
             original time and speed.

            Use that value to re-insert the agent into priority queue
            Store it in agent.total-delay
        end
    Insert all agents into queue using their total delays as priority values.
END COOPERATIVE_PF
```

Once all agents have been removed from the priority queue, they are inserted back in if they haven't reached their goals yet.

New Cooperative Pathfinding Algorithm Applied to Flock Traffic Navigation

We will explain the classic FTN based on Negotiation algorithm and then propose an improvement to avoid collisions at intersections.

Flock Traffic Navigation

Flock Traffic Navigation (FTN) based on negotiation is a new approach for solving traffic congestion problems in big cities (Astengo, 2007). In FTN, vehicles can navigate automatically in groups called flocks allowing the coordination of intersections at the flock level, instead of at the individual vehicle level, making it simpler and far safer. To handle flock formation, coordination mechanisms are issued from multi-agent systems.

The mechanism to negotiate (Astengo, 2006) starts with an agent who wants to reach its destination. The agent knows an a priori estimation of the travel time that takes to reach its goal from its actual position if he travels alone. In order to win a speed bonus (social bonus) he must agree with other agents (neighbors) to travel together at least for a while. The point in which the two agents agree to get together is called the meeting point and the point where the two agents will separate is called the splitting point. Together they form the so-called "bone" structure diagram.

Individual reasoning plays the main role in this approach. Each agent must compare its a priori travel time estimation versus the new travel time estimation based on the bone-diagram and the social bonus and then make a rational decision. Decision will be made according to whether they are in Nash equilibrium (there is no incentive for either of them to choose another neighbor agent over the agreed one) or if they are in a Pareto Set (Wooldridge, 2002).

If both agents are in Nash equilibrium, they can travel together as partners and can be benefited with the social bonus. In this moment a new virtual agent is created in order to negotiate with future candidates. Agents in a Pareto Set can be added to this “bone” diagram if their addition benefits them without affecting the original partners negatively. Simulations indicate that flock navigation of autonomous vehicles could substantially save time to users and let traffic flow faster (Astengo, 2007).
Until now, Agents make their own path planning according to A* and only at intersections use the Dresner-Stone Algorithm.

A Collision Detection Flock Traffic Navigation Algorithm

FTN based on Negotiation now are decoupled in two main parts:

```
OFFLINE Algorithm
  For each Agent do
   plan using A* and find
        an optimal path traveling  and
        an arrival time estimation.

REAL Time Algorithm
  For each spirit flock do
     A reservation δ-time units forward according to its path
     If reservation = TRUE
     Follow previous calculated A*-path
     else
     Conflict Module
```

The critical issue is the conflict Module that can be changed according to the social rules. Here we present a conflict module according to the rules in (Astengo 2006).

```
Conflict Module
Rules
  Priority 1: Larger Flock goes first
              Then tile is marked as obstacle.
  Priority 2: If Size of Flocks are equal
               compare delay-table
               delayed Flock goes first
               Then tile is marked as obstacle
 Priority 3: If none previous priority is accomplished
              Use Stone-Dresner Algorithm
```

Application to Continuous Domains

When applied to computer games, path finding usually takes place on top of one of two representations: A grid structure that wholly describes the traversable game world or a waypoint graph that samples the continuous space over which game agents can move.

The first variant can usually be seen in Real-Time Strategy Games (RTS) and bi-dimensional role-playing games (RPG). The cooperative path finding algorithm in dynamic environments is a perfect fit for these representations and allows game agents to react realistically to the presence of other agents and unforeseen obstacles. Other genres, however, require the simulation of a continuous space updated in fixed time-steps.

Applying a grid-like structure to such spaces can be prohibitively expensive. The algorithm can be modified to work in continuous domains by replacing the reservation table with an analysis of world geometry.

The algorithm consists of an update function, responsible for advancing the state of each agent by a single time-step. The agents are stored in a priority queue that uses each agent's total delay as its key. The function calculates the time elapsed between the last and the current call and uses this value to update the agents. The agents are updated by first performing a collision-detection test between the Minkowsky sum of the agent's path and an assigned bounding volume and each of the Minkowsky sums of the other agents' predicted paths and their bounding volumes. If a collision is detected, the agent will try to adjust its speed to avoid the intersection point at the intersection time. If this value falls under a specified threshold, the agent will, instead, attempt to calculate a different path.

The predicted paths are calculated using only the other agent's current position, orientation, and speed. These predictions are also limited by the forecasting index, which indicates how far in time are agents willing to predict.

This approach allows the path-planning operation to be distributed across frame updates and the individual update steps for each agent cause no side-effects, making them trivially parallelizable. The algorithm can be further optimized by pruning the collision-detection search by using spatial partitioning schemes such as kd-Trees and by limiting the path used to calculate the Minkowsky sums with the forecasting index.

The update function is presented below:

```
INPUTS: agents[0…N], forecastIdx[0…N], agentPositions[0…N], agentSpeeds[0…N], speedBound,
agentOrientations[0…N], agentDelays[0…N], agentPaths[0…N], boundingVolumes[0…N]

  WHILE agents NOT EMPTY
      a <- agents[0]
      removeFromQueue(a)
      currentPath <- minkowski(agentPaths[a], boundingVolumes[a], forecastIdx[a])
          FOR o IN 0…N
              otherPath <- minkowski(predictPath(agentPositions[o], agentSpeeds[o],
                            agentOrientations[o], forecastIdx[a]), boundingVolumes[o], forecastIdx[a])
              IF intersects(currentPath, otherPath) THEN
                  slowdown <- calculateSlowdown(agentPaths[a], agentSpeeds[a], intersectionPoint,
                  intersectionTime)
                  IF slowdown < speedBound THEN
                      IF isOtherPathAvailable(a) THEN
                          previousPath <- agentPaths[a]
                          agentPaths[a] <- calculateNewPath(a)
                          delay <- calculatePathDelay(agentPaths[a], previousPath)
                          agentDelays[a] <- agentDelays[a] + delay
                          addToQueue(a, agentDelays[a])
                  ELSE
                      delay <- calculateSpeedDelay(agentSpeeds[a], slowdown)
                      agentSpeeds[a] <- slowdown
                      agentDelays[a] <- agentDelays[a] + delay
                      addToQueue(a, agentDelays[a])
                  END
              ELSE
                  agentPositions[a] <- updatePosition(agentPaths[a], agentSpeeds[a])
              END
          END
      END
      agents <- buildPriorityQueue(agentDelays)
```

5. Concluding Remarks and future work

Pathfinding is a critical element of AI in many modern applications like multiple mobile robots, game industry and flock traffic navigation based on negotiation (FTN).

We develop a new algorithm capable of planning paths for multiple agents on partially known and changing environments inspired by cooperative A* and D*.

From a distributed approach (Decoupled) our collective pathfinding in dynamic environments algorithm decomposes the task of individual plan into weakly-dependent problems for each agent. Each agent can search greedily for a path according to its destination, given the current state of all other agents. Then based on a space-time search space each agent attempt to make a reservation on (x,y,t,δ) where x,y are in the Euclidean space, t is a time measure and δ is a forward planning-vision measure (forecasting index).

Because these kinds of algorithms are problem dependent we developed a modification of our collective pathfinding in dynamic environments algorithm in the FTN context. Taking care that in FTN the main issue is that it is based on negotiation a conflict-solver module that has the social rules within.

Evidently in FTN we will not in general obtain a globally optimal path from the individual agent perspective but it is a at least a better plan compared with traveling alone (the worst scenario is if an agent can't find Nash or Pareto partners in the whole path so, it becomes a 1-individual flock).

It was shown that the algorithm can, with relatively few modifications, work on continuous domains updated in fixed time-steps, such as those used by most 3D computer games. The shift from using a reservation table to analyzing world geometry allows the work to be cleanly distributed amongst the agents. This creates a clear separation of concerns and the lack of side-effects makes it trivially parallelizable.

6. References

R. Alami and F. Ingrand and S. Qutub (1998), A Scheme for Coordinating Multi-Robot Planning Activities and Plans Execution, *Proc. 13th European Conf. on Artificial Intelligence ECAI 98*, pp. 617-621.

N. M. Amato, C. Jones & D. Vallejo. (1998), An Adaptive Framework for "Single Shot" Motion Planning, *Technical Report 98-025*, Texas A&M University.

N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones & D. Vallejo. (1998), OBPRM: An Obstacle-Based PRM for 3D Workspaces, Ed. P. K. Agarwal et al., *Robotics: The Algorithmic Perspective*, pp. 155-168.

B. Aronov, M. de Berg, A. F. Van der Stappen, P. Svestka & J. Vleugels. (1999), Motion Planning for Multiple Robots, *Discrete and Computational Geometry*, Vol. 22, pp. 505-525.

C. Astengo-Noguez and G. Sánchez-Ante G. (2007), Collective Methods on Flock Traffic Navigation Based on Negotiation. *Proc. 6th Mexican International Conference on Artificial Intelligence*, Springer.

C. Astengo-Noguez and R. Brena (2006) Flock Traffic Navigation Based on Negotiation. *Proc. of 3rd International Conference on Autonomous Robots and Agents (ICARA 2006)* . Palmerston North, New Zealand, pp. 381-384.

J. Barraquand & J. C. Latombe. (1990), A Monte-Carlo Algorithm for Path Planning with Many Degrees of Freedom, *Proc. IEEE Int. Conf. Rob. & Autom.*, pp. 1712-1717.

J. Barraquand & J. C. Latombe. (1991), Robot Motion Planning: A Distributed Representation Approach, *Int. J. of Robotics Research*, Vol. 10, Nr. 6, pp. 628-649.

J. Barraquand, B. Langlois & J. C. Latombe. (1992), Numerical Potential Field Techniques for Robot Path Planning, *IEEE Trans. on Systems, Man and Cybernetics*, Vol. 22, Nr. 2, pp. 224-241.

R. Bohlin & L. E. Kavraki. (2000), Path planning using lazy PRM, *Proc. IEEE Int. Conf. Rob. & Autom.*, pp. 521-528.

J. F. Canny & J. Reif. (1987), New Lower Bound Techniques for Robot Motion Planning Problems}, *Proc. 28th Symp. Foundations of Computer Science*, pp. 49-60.

J. F. Canny, (1988), *The Complexity of Robot Motion Planning,* MIT Press, Cambridge, MA.

B.R. Donald. (1987), A Search Algorithm for Motion Planning with Six Degrees of Freedom, *Artificial Intelligence,* Vol. 31, Nr. 3, pp. 295-353.

K. Dresner and P. Stone (2004), Multiagent Traffic Management: A Reservation-Based Intersection Control Mechanism. *Proc. Third International Joint Conference On Autonomous Agents and Multiagent Systems*, pp. 530-537.

K. Dresner and P. Stone (2005), Multiagent Traffic Management: An Improved Intersection Control Mechanism. *Proc. Fourth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 05).* pp. 471-477.

K. Dresner and P. Stone (2006), Human-Usable and Emergency Vehicle–Aware Control Policies for Autonomous Intersection. Hakodate, Japan. *Proc. Fourth Workshop on Agents in Traffic and Transportation (ATT 06).* pp. 17-25.

K. Dresner and P. Stone (2007) Sharing the Road: Autonomous Vehicles Meet Human Drivers. Sharing the Road: Autonomous Vehicles Meet Human Drivers. Hyderabad, India. *Proc. Twentieth International Joint Conference on Artificial Intelligence (IJCAI 07)*. pp. 1263-1268.

D. Hsu, J. C. Latombe & R. Motwani. (1997), Path Planning in Expansive Configuration Spaces, Proc. *IEEE Int. Conf. Rob. & Autom.*, pp. 2719-2726.

D. Hsu. (2000), Randomized Single-Query Motion Planning in Expansive Spaces, *PhD Thesis*, Stanford University, Stanford, CA, USA.

L. E. Kavraki. (1994), *Random Networks in Configuration Space for Fast Path Planning*, PhD Thesis, Stanford University, Stanford, CA, USA

L. E. Kavraki, P. Svestka, J. C. Latombe & M. H. Overmars. (1996), Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces*, IEEE Trans. on Robotics and Automation*, Vol. 12, Nr. 4, pp. 566-580

K. Kondo. (1991), Motion Planning with six degrees of freedom by multistrategic, bidirectional heuristic free space enumeration, *IEEE Trans. on Robotics and Automation*, Vol. 7, Nr. 3, pp. 267-277.

J. J. Kuffner, Jr., & S. M. LaValle. (2000), RRT-Connect: An Efficient Approach to Single-Query Path Planning, *Proc. IEEE Int. Conf. Rob. & Autom.*

J. C. Latombe. (1991), *Robot Motion Planning*, Kluwer Academic Publishers, Boston, MA.

S. M. LaValle & J.J. Kuffner, Jr. (2001), Randomized Kinodynamic Planning, Int. J. on Robotics Research, Vol. 20, Nr. 5, pp. 378-400.

M. C. Lin & S. Gottschalk. (1998) Collision Detection Between Geometric Models: A Survey, *IMA Conference on Mathematics of Surfaces*.

T. Lozano-Perez & M. A. Wesley (1979), An algorithm for planning collision-free paths among polyhedral obstacles, *Communications of the ACM*, Vol. 22, Nr. 10, pp. 560-570.

T. Lozano-Perez. (1983), Spatial Planning: A configuration Space Approach, *IEEE Trans. on Computers*, Vol. 32, Nr. 2, pp. 108-120.

C. Nielsen & L. Kavraki. (2000), A Two level Fuzzy PRM for Manipulation Planning, *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*.

P. A. O'Donnell & T. Lozano-Perez. (1989), Deadlock-Free and Collision-Free Coordination of Two Robot Manipulators, *Proc. IEEE Int. Conf. Rob. & Autom.*, pp. 484-489.

D. C. Pottinger (1999), Coordinated Unit Movement . *Game Developer Magazine*, Vol. 3.

J.H. Reif (1979), Complexity of the Mover's Problem and Generalizations, *Proc. 20th Symp. on the Foundations of Computer Science*, pp. 421-427.

G. Sánchez-Ante & J. C. Latombe. (2002), On Delaying Collision-Checking in PRM Planning -- Application to Multi-Robot Coordination, *Int. J. of Robotics Research*.

J. T. Schwartz & M. Sharir (1983), On the Piano Movers' Problem I: The Case of a Two-Dimensional Rigid Polygonal Body Moving Amidst Polygonal Barriers, *Communications Pure and Applied Mathematics*, Vol. 36, pp. 345-398.

Yoav Shoham and Kevin Leyton-Brown. (2008), *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*, Cambridge University Press.

D. Silver (2005), Cooperative Pathfinding, *American Association for Artificial Intelligence*, 2005.

G. Song, S. Miller & N. Amato. (2000), Customizing PRM Roadmaps at Query Time, *Proc. IEEE Int. Conf. Rob. & Autom*.

P. Svestka. (1997), *Robot Motion Planning using Probabilistic Roadmaps*, Ph.D. Thesis, Utrecht University, The Netherland.

P. Tournassoud. (1986), A strategy for obstacle avoidance and its applications to multi-robot systems, *Proc. IEEE Int. Conf. Rob. & Autom.*, pp. 1224-1229.

D. Vallejo, I. Remmler & N. M. Amato. (2001), An Adaptive Framework for "Single Shot" Motion Planning: A Self-Tuning System for Rigid and Articulated Robots, *Proc. IEEE Int. Conf. Rob. & Autom.*

M. Wooldridge (2002), *Introduction to Multiagent Systems*. John Wiley and Sons.

A. A. Zelinsky (1992), Mobile Robot Exploration Algorithm, *IEEE Transactions on Robotics and Automation*, Vol. 8.

D. Zhu & J. C. Latombe. (1990), Constraint reformulation in a hierarchical path planner, *Proc. IEEE Int. Conf. Rob. & Autom.*, pp. 1918-1923.