Why the Parti-game Algorithm Does not Work Satisfyingly for Manipulator Trajectory Generation

Martin Eldracher*, Rüdiger Merklein

Fakultät für Informatik, Technische Universität München, 80290 München, Germany
eldrache|merklein@informatik.tu-muenchen.de

Abstract

Trajectory generation for manipulators can be performed in a very efficient manner, if a model of the environment is available. A classical approach to construct a world model is the cell decomposition method. We describe in detail a recently published approach for planning in reinforcement-tasks, called Parti-game algorithm (Moore and Atkeson,1995), that also can be used for trajectory generation. Parti-game uses rectangular cells of varying size for modeling free-space in order to keep memory consumption and preprocessing time reasonable. We present experimental results for kinematic trajectory generation for manipulators in two example environments and a real world environment. We analyze the reasons for the disappointing behaviour of Parti-game even in examples still rather afar from real manipulators, opposite to results with other recent approaches using graph-based algorithms.

*This work has partly been funded by the German Ministry for Research and Technology (BMFT) grant no. 01 IN 102 B/0. The authors are solely responsible for the contents of this publication.

Contents

1 Introduction

For flexible automatic assembly, e.g. in small series production or single part production, a key requirement is automatic generation of kinematic trajectories of the assembly manipulator between arbitrary points in its working space. There are several classical algorithms to construct collision-free trajectories in static environments of small degree of freedom (dof) with limited computational resources (for details see e.g. Latombe,1991): the *road-map*-algorithm (Canny,1988), the *free-way*-method (Brooks,1983), or approaches with *potential fields* (e.g. Khatib,1986). Another well known basic approach are *cell-decomposition*-methods (e.g. Schwartz and Sharir,1983). Basically one distinguishes between exact cell decomposition and approximate cell decomposition.

In exact cell decomposition the manipulator's free-space $\mathcal{C}_{free}$ (i.e. that part of configuration-space $\mathcal{C}$ that is not occupied by obstacles) is divided into a set of non-overlapping cells. The neighborship-relations between all cells are stored in a graph. A trajectory between two configurations lying in certain cells is constructed using that graph. The final trajectory is chosen as some special case of a channel that is defined by the sequence of cell-boarders that must be crossed in order to get from start to goal. Exact cell decomposition is a complete algorithm, i.e. an existing solution is found. A first problem with exact cell decomposition is to choose the complexity of the cells. With large complex cells the planning inside the cell gets too complex; with small simple cells we run into computational problems due to the complexity of graph search techniques (in somehow practical environments, we have to deal with at least six-dimensional spaces). A second problem is the prerequisite of an exact geometric model of the manipulator and its environment in order to compute the cells. In reality, especially in changing environments, we usually do not know such an exact model.

In approximate cell decomposition we approximately model free-space $\mathcal{C}_{free}$ using differently sized cells of a prespecified simple geometric shape, e.g. hyper-cubes (Lozano-Peréz,1981). The decomposition is recursively refined for all cells that neither completely lie in free-space nor in space completely occupied by obstacles. Given minimum size of the cells, we also get a minimum amount of free-space around the trajectories (for trajectory generation cells that do not entirely lie in free-space at this minimum discretization are regarded as occupied completely by obstacles). Again we use a graph with the neighborship-relations between the cells to find a path between the cell around our start and the cell around our goal. One problem of approximate cell decomposition is that, as for exact cell-decomposition, we also need a precise geometric model of our environment in order to know where we still have to refine our decomposition. A second problem is that the approach is only resolution complete: With a relatively coarse minimum discretization we are prone not to find an existing path. With a fine minimum discretization we run into the problem of quickly having too much cells due to the exponentially growing number of cells.

The basic idea of Parti-game (Moore and Atkeson,1995) is to construct a cell decomposition of the whole configuration-space (opposite to the above standard methods, that only divide free-space) not in advance prior to any planning, but on the fly during planning a trajectory. In Parti-game the cell decomposition (called *partitioning*) is independent of the obstacle-free-space–distribution, but is based on the actual experience during planing. Therefore cells can contain free-space and obstacles but nevertheless are used for planning. So Parti-game does not need an a priori geometric model of the environment (e.g. Glavina,1990; Baginski,1996b), but the information whether cells have to be further repartitioned is extracted from (simulated) sensor information. Parti-game is similar to approximate cell-decomposition with three respects: First, configuration-space is partitioned with different granularity, saving computational and memory resources. Second, all cells have the form of hyper-rectangles. Furthermore these hyper-rectangles all are axis-aligned. Third, with the cell-decomposition a neighborship-relation graph is constructed, that can be used to plan a trajectory. Opposite to approximate cell decomposition this graph is also used to decide whether a further refinement of the decomposition is necessary, while for approximate cell decomposition this is done using the exact information of the geometric model of the environment.

2 The Parti-game Algorithm

Parti-game is designed to solve reinforcement problems by learning from delayed rewards in multi-dimensional, continuous state spaces with unknown system dynamics and control laws. Parti-game learns a controller from a start partition to a goal partition(Moore and Atkeson,1995). Opposite to other approaches that recursively partition state-space (e.g. Chapman and Kaelbing,1991; Dayan and Hinton,1993) Parti-game uses ideas from game-theory and a database *DB* of all previous experiences. Parti-game rests upon two prerequisites. First, the system is always able to compute its configuration (or state). In manipulator environments this is easily possible using a sensor in each joint. Using this configuration information and the partitioning (cell decomposition), it is also cheap to compute in which partition (cell) the manipulator is. Second, Parti-game uses a greedy controller that tries to directly get from the current configuration to a desired configuration (even if this often will result in a failure). Such a controller usually is included in the fundamental, low-level manipulator control, using straight lines in configuration-space.

2.1 Basic Description of Parti-game

Parti-game is conservative assuming always to start from the most difficult configuration if heading from one partition to another (worst case assumption). Working on a certain partitioning, Parti-game uses the greedy controller to aim at the center of that neighboring-partition that has the least discrete distance to the goal partition with respect to the worst case assumption (this information is taken from the neighborhood-relation between the partitions and the database *DB* of all experiences that Parti-game stores). As long as Parti-game reaches the desired partition this step 'go for the next partition' is iterated until the goal-partition is reached. If the desired partition is missed due to collisions with obstacles, or if some other partition is entered, we update the database *DB* storing our newly encountered experience as the triplet (*current-partition*, *aim-partition*, *reached-partition*). Now we update our discrete distances to the goal with respect to the worst case assumption. We proceed trying to arrive at the goal-position from the reached-partition, as long as we do not end up in a partition with an infinite discrete distance to the goal partition. In that case we refine the partitioning for all partitions that have infinite distances but neighbors with finite distances (this may be the case, because due to the worst case assumption an infinite distance only indicates that we had the experience that we missed to reach each neighbor at least one time; starting from different configurations might possibly yield different results). These neighbors are split as well. After the partitioning refinement we store the new neighborship-relation and recompute the worst case discrete distances for all partitions. Then we try to get from start to goal in a next *run*.

Seen from game theory, the database *DB* can be interpreted as if we try to get from the current-partition to the aim-partition (in order to reach the goal partition as fast as possible). However, some adversary sees our choice of the aim-partition and is allowed to set our position in the current-partition in order to prevent us to reach this aim-partition. We have to refine the partitioning as long as the adversary is able to hinder us to reach our aim-partitions.

2.2 Detailed Description with Examples

2.2.1 Initialization

Parti-game is initialized with two partitions. The first partition covers the whole state space and will be split during the planning process. The second partition is the goal partition. The goal partition is usually rather small, since all configurations in the goal partition are accepted as goal configuration. The goal partition will never be split. Furthermore the goal partition is the only partition that may overlap with one or several other partitions.

2.2.2 Greedy Controller

Given a partitioning, the current configuration in the current partition and an aim-partition, the greedy controller tries to reach the center of the aim-partition from the current configuration on a straight line (in configuration-space). The greedy controller stops, if a different than the desired partition is entered, if there is a collision with an obstacle, or if the aim-partition is entered and half the distance from the entry point to the partition's center is covered (see Figure 1, p.5). Furthermore the greedy controller returns the partition where the controller stopped (this result is used to build up the database *DB*, see paragraph 2.2.3).

2.2.3 Content, Construction and Use of Database

Given some partitioning, the database *DB* with all experiences from prior planning, basis for the further planning and partitioning, is initialized as if each neighbor can be reached from any configuration inside the partition (best case assumption due to lack of information). I.e. an entry (*current-partition*, *aim-partition*, *reached-partition*) $= (i, j, j)$ is assumed to be in the database *DB* for all partitions i and all their neighbors $j \in N(i)$. In our example (see Figure 2, p.5) we would have the following initial database: $DB = \{(1,2,2),(1,3,3),(2,1,1),(2,3,3),(2,4,4),(3,1,1),(3,2,2),(3,4,4),(4,2,2),(4,3,3),(4,G,G)\}$. However, since these entries are always assumed to be in the database *DB*, we do not need to store them. Whenever the greedy controller returns a result, the database *DB* is scanned whether this result already is stored; if not the corresponding triplet is entered into the database *DB*. For these experiences we use the worst case assumption, i.e. if an aim-partition could not be reached from one configuration, we assume that this aim-partition can not be reached from any configuration inside the current partition (reflected in the discrete distances for the shortest path J_{WC}, see paragraph 2.2.4).

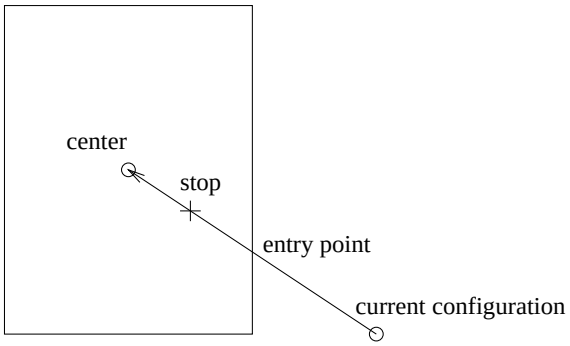

Figure 1: *Example for behaviour of the greedy controller (see text for explanation).*

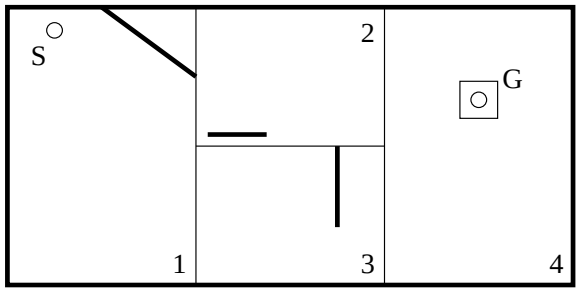

Figure 2: *Example environment with start configuration S, goal configuration G, four partitions (numbered), the goal partition (square around goal) and some obstacles (thick black lines).*

Given some database *DB* Parti-game decides which partition should be tried to be entered standing in configuration S in partition i (the goal configuration is implicitly contained in the discrete distances to the goal partition J_{WC}, see paragraph 2.2.4). Of course we choose that neighbor $j \in N(i)$ with the lowest discrete distance $J_{WC}(j)$, i.e. that neighbor that provides us with the shortest path to the goal partition.

2.2.4 Diffusion Algorithm for Discrete Distance to Goal Partition

Starting from the goal partition a diffusion algorithm is run to compute worst case shortest path step counters J_{WC}, i.e. the discrete distance in number of partitions to be entered until the goal partition is reached. In detail this computation runs as follows: First for all partitions i we set $J_{WC}(i) = \infty$. Then the goal partition is labeled with $J_{WC}(\text{goal partition}) = 0$. We proceed computing the discrete distances following the neighborhood relation of already computed partitions. For a not yet computed partition

i (with at least one already computed neighbor), we first compute the worst discrete distances for all neighbors $j \in N(i)$ of i. To compute the worst discrete distances we use the entries in the database *DB*. Based on the set $O(i,j)$ of all outcomes k when trying to go from partition i to partition j, i.e. $O(i,j) = \{k|(i,j,k) \text{ in database } DB\}$, we choose that k with highest discrete distance $J_{WC}(k)$. Then we set $J_{WC}(i)$ to the minimum of the neighbors' worst discrete distances.

$$J_{WC}(i) = \begin{cases} 0 & \text{if } i \text{ is the goal partition} \\ 1 + \min_{j \in N(i)} \left(\max_{k \in O(i,j)} \left(J_{WC}(k) \right) \right) & \text{otherwise} \end{cases} \quad (1)$$

2.2.5 Refining the Partitioning

Each time we store some result in the database *DB* (see paragraph 2.2.3) we also run again the diffusion algorithm for J_{WC} (see paragraph 2.2.4). Since all successful trials are (implicitly) already included in the initial database, we only will take in failures. This implies that the a re-computation of the discrete distances J_{WC} using the minimax-principle of formula 1 (p.6) only can worsen them. All partitions i that end up with $J_{WC}(j) = \infty$ are called losers. A partition i will be a loser, if *DB* contains the entry (i,j,i) for all neighbors $j \in N(i)$ (possibly besides other entries). Now we will split all losing partitions that have non-losing neighbors, as well as these neighbors. We split each of these hyper-rectangular partitions along their longest axis into two smaller partitions. We recompute the neighborship relation for all partitions. Furthermore we delete all entries in the database that contain one of the just split partitions as current-partition, aim-partition or reached-partition.

The selection of partitions to split is motivated by the following considerations. Since we assume that there is a continuous trajectory from start S to goal G (see Figure 2, p.5) we must have missed to find a hole to pass through somewhere at the borderline between losing and non-losing partitions. Since there is no objection that the obstacles are equally spread over winners and losers, it makes sense also to split winners. Using a finer discretization at the winner-loser-border, we have a better chance to find a hole. On the other hand it would not make sense to also split losers that only have losing neighbors, since this would not help to solve the trajectory planing (still other loser will block us independent of the discretization) but increase the number of partitions. Furthermore with this re-partitioning strategy we do not end up with neighbors of strongly differing size, which automatically causes a lot of failures if a very small partition should be entered from a huge one.

2.2.6 Simple Example

Now lets look at a toy example to clarify the many interlocking parts of the Parti-game algorithm described. We start with the partitioning in Figure 2 (p.5). The database *DB* is empty, since our initial assumptions of (i,j,j) for all partitions i and $j \in N(i)$ do not have to be stored. The initial discrete distances are computed as $J_{WC}(4) = 1$, $J_{WC}(3) = 2$, $J_{WC}(2) = 2$, and $J_{WC}(1) = 3$. Without any prior knowledge we will try to directly enter partition 2 from partition 1 (dotted arrow in Figure 3, p.7). However, we will end up still in partition 1 due to a collision with an obstacle after performing a part of that trajectory (arrow from S to the obstacle in Figure 3, p.7). Therefore we insert the triplet $(1,2,1)$ into the database *DB*. Furthermore we recompute the discrete distances to $J_{WC}(4) = 1$, $J_{WC}(3) = 2$, $J_{WC}(2) = 2$, and $J_{WC}(1) = 3$. From here we will proceed to try to reach partition 3 (dotted arrow). Since we again hit an obstacle we end up in partition 2 (arrow from first to second obstacle). Therefore we enter the triplet $(1,3,2)$ into *DB* and recompute again $J_{WC}(4) = 1$, $J_{WC}(3) = 2$, $J_{WC}(2) = 2$, and $J_{WC}(1) = 3$. Now we reach partition 4 (arrow from second obstacle) and from there the goal partition (arrow to G). Our first run is completed.

Though we already found a trajectory, i.e. the trajectory generation task is solved, we could end up with a partitioning and corresponding information in database *DB* that does not allow us to directly find a trajectory from S to G in the next run. We furthermore can do better with respect to shorter trajectories and constant solutions for the same task, if we perform some additional runs. Using the information stored in *DB* we do a second run from S to G (Figure 4, p.7). Our first step from partition 1 to partition 3 is successful (we do not try to get to partition 2, since we still have stored the failure from

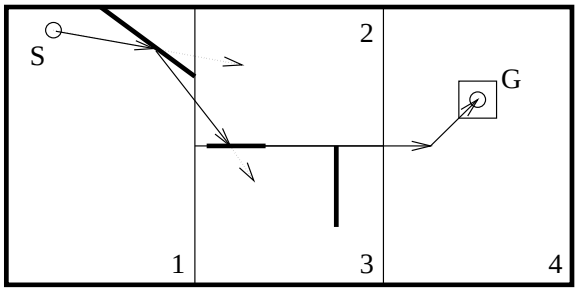

Figure 3: *First run in the environment described in Figure 2 (p.5).*

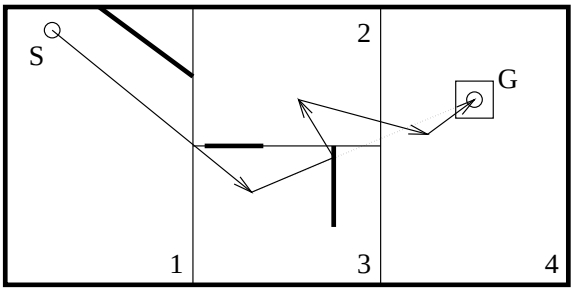

Figure 4: *Second run in the environment described in Figure 2 (p.5).*

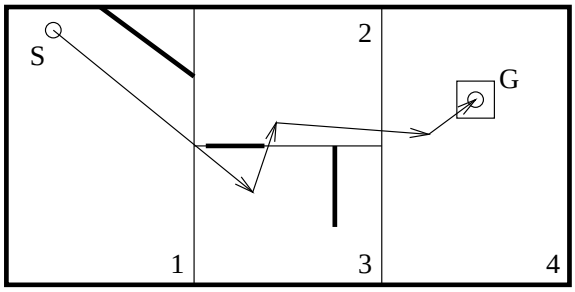

Figure 5: *Third run in the environment described in Figure 2 (p.5).*

the first run). If due to some fourth obstacle this step would have yielded the entry $(1, 3, 1)$ partition 1 would have been a loser (compare paragraph 2.2.4) and we would have had to refine the partitioning. Though our initial success we have now to include the triplet $(3, 4, 3)$ into database *DB*, since we collide with the third obstacle. We recompute the discrete distances to $J_{WC}(4) = 1$, $J_{WC}(3) = 3$, $J_{WC}(2) = 2$, and $J_{WC}(1) = 4$. Our next three steps from partition 3 to 2, partition 2 to 4, and partition 4 to goal G are successful. Now starting again for a third run we successfully perform the steps from partition 1 to 3, from partition 3 to 2 from partition 2 to 4, and from partition 4 to goal G (Figure 5, p.7).

2.2.7 Important Implementation Details

Parti-game, as described in Moore (1994), does not specify explicitly what the function $A(i)$, the number of possible actions to take in a partition i should denote (in our application such actions are to go to a certain partition using the greedy controller). From these actions we always choose that one that promises to reach a partition with the lowest value J_{WC}. The most general interpretation for $A(i)$ as 'aim at one of the centers of all neighbors $j \in N(i)$' can lead to situations, where Parti-game gets stuck in the sense that it stops before an obstacle. An example is given in Figure 6 (p.8): We see the configuration-space of a simple example manipulator (compare Section 4) and an example partitioning. Each partition is labeled with its discrete distance J_{WC}. Within this situation initially the partition around start S is given the value $J_{WC} = 2$. After a failed attempt to reach the center of the goal partition (very small, covered by the point for goal G), this value is changed to $J_{WC} = 3$. However, choosing $A(i)$ to head for all neighbors' centers, the partition around goal G will remain the partition with the lowest value for J_{WC} and the system will try to go to this partition again and again.

Therefore we use the database *DB* and choose $A(i)$ as the set of actions 'head for all neighbors $j \in N(i)$, that seem to be reachable from partition i taking into account the information stored in *DB* (and the worst case assumption)'. (This interpretation seems complicated only at the first glance, at a second glance it is much more natural). With this interpretation Parti-game reaches a state, where the

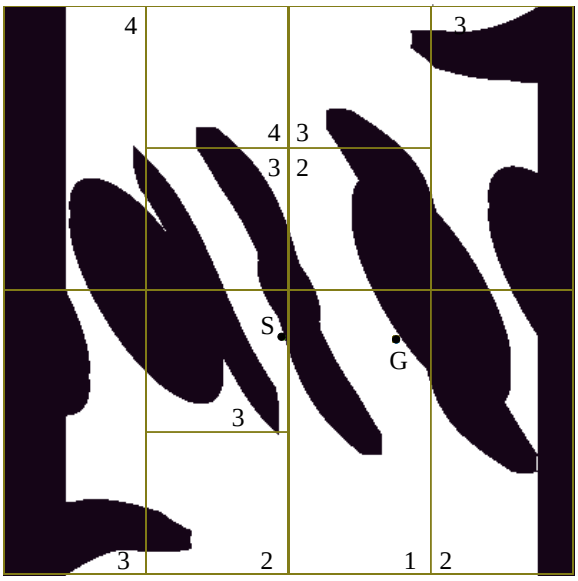

Figure 6: *In this situation Parti-game is stuck, if the set of possible actions $A(i)$ for each partition i is chosen to be 'head for the center of all neighbors $j \in N(i)$ with lowest J_{WC}'.*

partition around start S will be a loser, will be split, and a solution can be found.

A second choice that might lead to completely different behaviour for different implementations of Parti-game is that there is no rule to choose the neighbor that should be headed for, if several neighbors have the minimum J_{WC} value. However, besides different partitioning and different run-times there does not seem to arise different behaviour for different choices. Moreover there does not seem to be a generally advantageous choice at all.

3 Improvements to the Standard Algorithm

3.1 Problems with the Standard Parti-game Algorithm

During the experiments with the Parti-game algorithm we found out some properties of Parti-game that unnecessarily aggravate the planning problem.

First Parti-game stores neither information about the correlation of partitions and obstacles nor about the actual path lengths that vary with the size of the partitions. Running the diffusion algorithm (see paragraph 2.2.4) large partitions pretend to yield short paths. Therefore Parti-game often directly tries to run into large partitions inside obstacles. This behaviour will only change after a lot of small partitions have created at the border of the large obstacle due to continuing refinement of the partitioning, and the database *DB* contains entries that prevent Parti-game entering these large partitions (usually there are no database entries to prevent entering as long as there is no complete border of small partitions around the obstacle, since as long as something changes with the large partitions the information in the database *DB* will be erased after splitting partitions).

A second problem is that Parti-game has difficulties to escape from thin funnels. Unfortunately these are common in the configuration-spaces of manipulators (see Figures 10, p.12, and 14, p.13). If a region R can not be left on basis of the current partitioning, but has an exit in a funnel far inside or far outside of this region, Parti-game has to split many partitions until the form of the funnel is captured and an exit is found. However the progress in direction of the exit of the funnel is very slow due to the split strategy (Figure 7, p.9). Furthermore the ratio between the number of new partitions and progress in

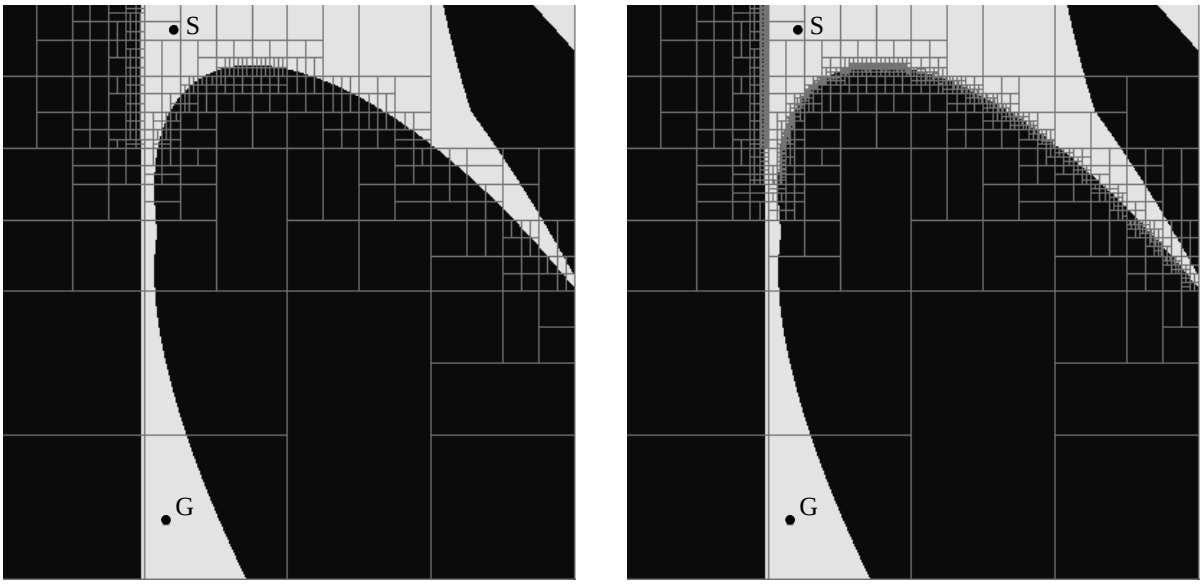

Figure 7: *Example for problems with funnels (taken from the upper left part of Figure 10, p.12). Task is to go from start S to goal G. The right image shows the partitioning after some refinements. Inside the funnel the partitions are already rather small. The right image shows show the partitioning after several further refinements. Though lots of new partitions are generated at the border of the region of losers, the progress into the funnel is marginal.*

direction of the exit decreases with each split. Since always the complete border of the region R is split, lots of partitions are split again and again without any use to escape from R.

A third problem is, that split partitions does not automatically help to improve performance. An example is given in Figure 16 (p.15). Parti-game models a funnel in the lower right corner of the configuration-space. However, there is no exit of that funnel. So lots of partitions are only created but never can help to find a solution. These partitions slow down each diffusion and force Parti-game to run through many partitions until a new split takes place.

A last disadvantage is the assumption of existing trajectories between arbitrary start and goal configurations. However, trajectory generation is especially interesting in unknown environments (in fact, if we knew whether a trajectory exists, usually the hardest task is done). Now, if Parti-game does not succeed, it continues to refine the partitioning forever, we have a non-terminating algorithm. Even if there is a small exit so that Parti-game will finally succeed, due to the exponential growth of the number of partitions with each refinement, we get such a huge number of partitions rather fast that either the machine's memory is exhausted or the administration of the partitions becomes very slow.

3.2 Improvement 1: Maximum Partition Depth

Our first improvement is to introduce a maximum partitioning depth *mpd*. With this limit we can guarantee the termination of the algorithm. Furthermore we define a minimum size of the partitions. If we would use a slightly different greedy controller (which only uses Manhattan movements, i.e. only moves on straight lines parallel to the input dimensions, and therefore also parallel to the borders of the partitions, and always takes trajectories through the partitions' centers), we also could guarantee a certain amount of free-space around the trajectory.

Furthermore *mpd* reduces the attractiveness of large obstacles (see paragraph 3.1), because instead of splitting the small partitions at the border of a losing region R, all partitions of minimum size are kept together with their information in database *DB* that hinders the entry in the large inner-obstacle-partitions.

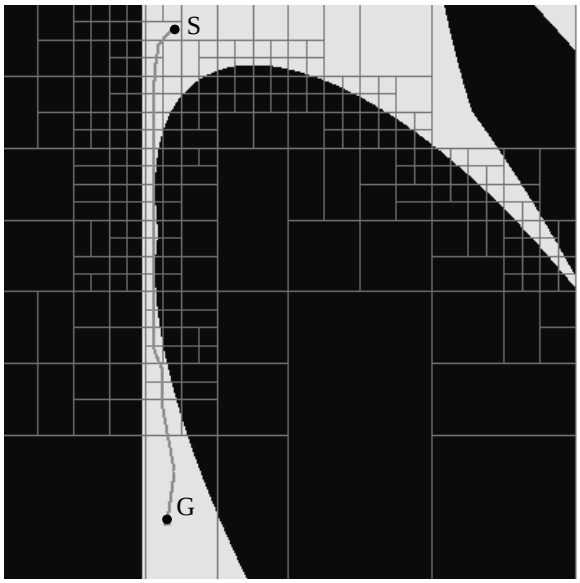

Figure 8: *Solution of the task in Figure 7 (p.9) using a maximum partition depth of* $mpd = 10$.

Besides the exit of funnels is made easier. Due to the frequent refinements mpd is quickly reached. Afterwards only partitions at the border of the losing region R can be split, which have not yet reached maximum partition depth. But splitting exactly these partitions gives the best progress into the funnel (compare Figure 7, p.9).

Should the maximum partition depth be reached in a losing region R, then all partitions in R will receive a value of $J_{WC} = \infty$ (compare paragraph 2.2.4). If start S is included in R, Parti-game will terminate with failure, e.g. no trajectory can be found at this partitioning depth (however, we can now change the partitioning depth from outside and run Parti-game again). If start S lies outside R, just the whole region R will be forbidden for further planning, i.e. we get trajectories with a certain amount of free-space around them, and possibly save to search whole regions for a trajectory.

Figure 8 (p.10) shows the solution for the task presented in Figure 7 (p.9) using $mdp = 10$. We see that the task that could not be solved with a huge number of partitions with the original Parti-game now is solved with only some partitions.

3.3 Improvement 2: Reduced Partition Refinement

Our second improvement is to define a maximum percentage plp of losing partitions that may be split dependent of their size, or partition-depth pd, respectively. Only the largest plp percent partitions of a losing region R will be split.

Given the partition depths pd for all partitions in region R, i.e. all partitions that should be split in original Parti-game, we build all groups of partitions with the same partition depth. Given the minimum partition depth pd_{min} in R and the maximum partition depth pd_{max} in R, we split all partitions i with $100 * \frac{pd_i - pd_{min}}{pd_{max} - pd_{min}} < plp$ for $pd_{max} \neq pd_{min}$. For $pd_{max} = pd_{min}$ all partitions are split as long as $plp > 0$. For $plp = 100$ we do not compute this formula, but split all losing partition as in the original algorithm.

On one hand this improvement is suggestive, since Parti-game is based on the idea that the number of partitions should be as small as possible. On the other hand this improvement runs contrary to Parti-game's other idea that the partitioning should be inhomogeneous, i.e. not the whole space should be covered with nearly equally sized partitions. However, opposite to a maximum partition depth (see paragraph 3.2) the reduced partition refinement does not prevent a solution. If always the same region R is losing, only more runs are necessary until all partitions get really small.

3.4 Modification 3: Fix Partitioning

Our last modification is no improvement in the sense of the word. We implemented the possibility to generally forbid all changes in the partitioning. Setting parameter *nc* (no changes) works like if we set $mpd = 0$ or $plp = 0$ for an already given (not necessarily initial) partitioning. Parameter *nc* can be used to examine the quality of a partitioning for generating different trajectories with the given partitioning (e.g. how many trajectories can be generated for a given number of arbitrary start-goal-combinations). However, even if a refinement of the partitioning is forbidden, we allow to insert new information into the database *DB*. That allows Parti-game to improve a once found trajectory from a certain start to a certain goal, since all experiences from prior planning for this combination are used to compute different values for J_{WC}[1].

4 Example Environments for Experiments

Simulating manipulator environments is not as easy as it looks at a first glance, if one wants to do as realistic simulations as possible. Therefore we use spatially extended obstacles and manipulators (different from Moore (1994)). We simulate the movements with a very small discretization, that guarantees that also the continuous movement is collision free, if we assume a certain thin security zone around the obstacles (this is anyway necessary, since the step motors of real manipulators also have small imprecisenesses). Furthermore we only use sensor information that can easily be measured by industrially available, reasonably prized sensors: First, e.g. potential-sensors in each joint to measure the configuration of our manipulator. Second, force-torque-sensors in each joint that provide collision information.

4.1 Two-Dimensional Example Environment

First experiments are performed within an environment with a two joint manipulator, four obstacles, and surrounding walls (Figure 9 (p.12)). The configuration-space $\mathcal{C}$ (Figure 10 (p.12)) is limited by joint angle ranges of $\varphi_{1,2} = [-\pi, \pi]$.

4.2 Three-Dimensional Example Environment

The three-dimensional example environment contains two walls and a box as obstacles. The manipulator's geometry can be compared to industrial manipulators (without hand), e.g. a Manutec r3. The joint angle ranges are

$$\varphi_1 = [0;\ 2\pi] \quad , \quad \varphi_2 = [-\tfrac{\pi}{2};\ \tfrac{3\pi}{2}] \quad , \quad \varphi_3 = [-1,40;\ 4,54]$$

Figures 11 (p.12) and 12 (p.12) show the manipulator and a section of the configuration-space for $\varphi_3 = 1,57$ and variable $\varphi_{1,2}$. Clearly there are several regions that are critical for Parti-game: there are funnels without exits. So Parti-game will produce fine partitions (see paragraph 3.1) without being rewarded with a trajectory.

4.3 Six-Dimensional Real World Environment

A real world example is the six dimensional ROTEX manipulator from DLR, Oberpfaffenhofen, with allowed joint angle ranges of:

$$\begin{array}{lllllllll} \varphi_1 & = & [-0.17;\ 0.69] & , & \varphi_2 & = & [-0.52;\ 1.66] & , & \varphi_3 = [-3.67;\ 0.52] \\ \varphi_4 & = & [-2.62;\ 3.67] & , & \varphi_5 & = & [-2.09;\ 2.09] & , & \varphi_6 = [-1.57;\ 4.71] \end{array}$$

[1] Unfortunately it also might happen that for a specific start-goal-combination suddenly there is no trajectory any more, if the database entries were changed meanwhile, e.g. due to entries originating from a different trajectory partly using the same partitions. In that case we either have to re-allow split of partitions, or accept that Parti-game can not provide us with a trajectory, although we already know that there is one that Parti-game found earlier.

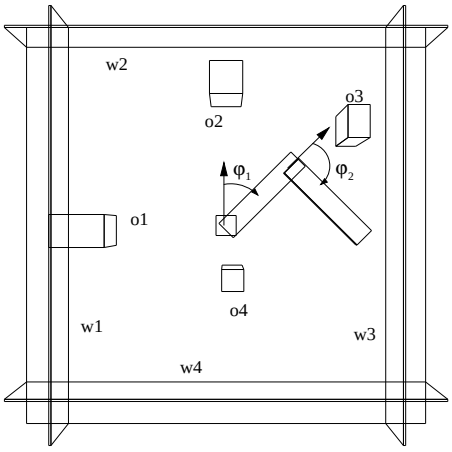

Figure 9: *Two dimensional manipulator in example environment. The current configuration is denoted with "C" in Figure 10 (p.12).*

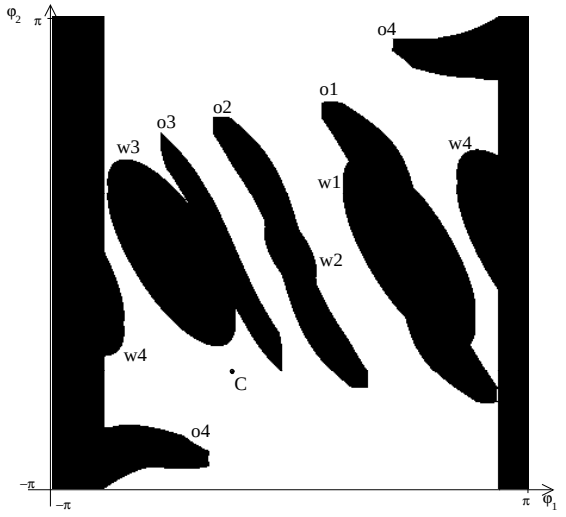

Figure 10: *Configuration-space of Figure 9 (p.12). Black areas represent the configuration-space obstacles.*

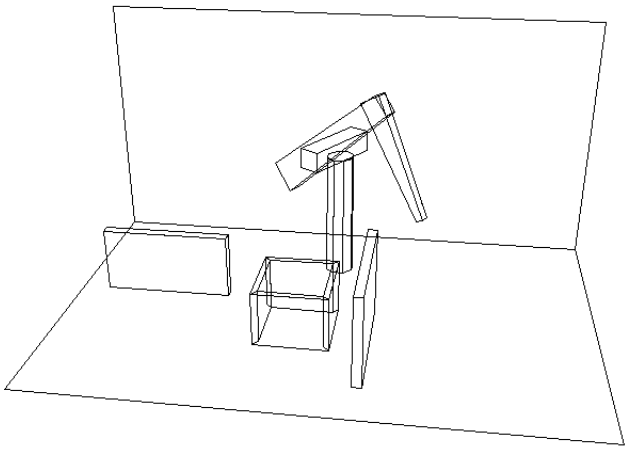

Figure 11: *Work-space of three-dimensional example environment. Joint angles are* $(\varphi_1; \varphi_2; \varphi_3) = (2,62;\ 0,78;\ 3,14)$.

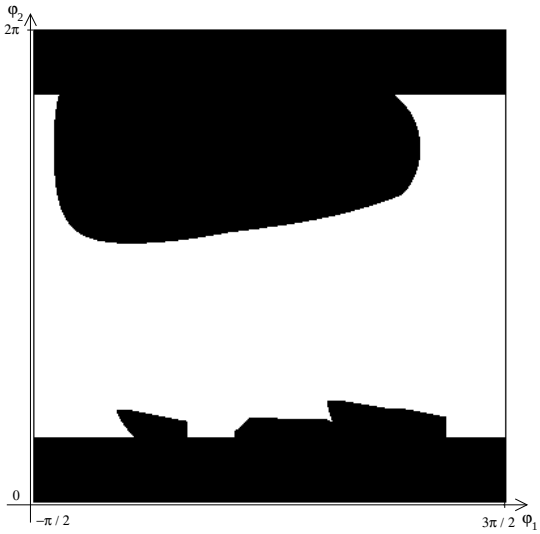

Figure 12: *Cut of the configuration-space of the three-dimensional example environment from Figure 11 (p.12) for fixed* $\varphi_3 = \frac{\pi}{2}$.

Since the ROTEX environment was used in the german space mission D2, where extension and weight were heavy problems, the working-cell (Figure 13 (p.13)) is very narrow, compared to the manipulator. Due to the many small obstacles (that were used to do some experiments, e.g. grasping, pressing buttons, etc.) the configuration-space furthermore is very cluttered (see Figure 14, p.13). Therefore the ROTEX environment is very dis-advantageous for Parti-game, nevertheless, it is real world example.

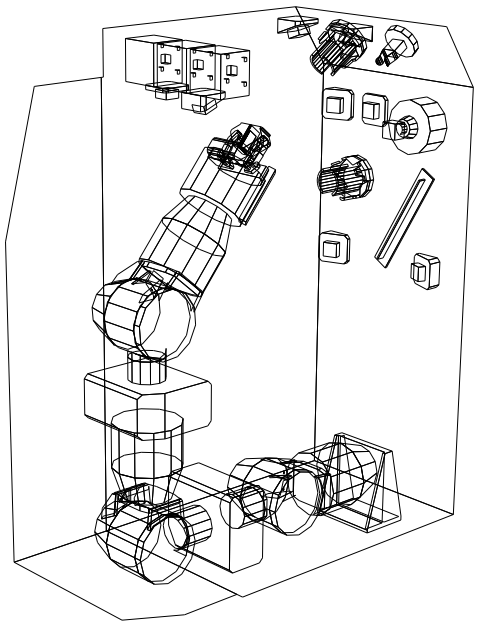

Figure 13: *Work-space of the six-dimensional real word example ROTEX. Joint angles are* $\vec{\varphi} = (0;\ \frac{\pi}{2};\ -\pi;\ 0;\ -0,69;\ 0)$.

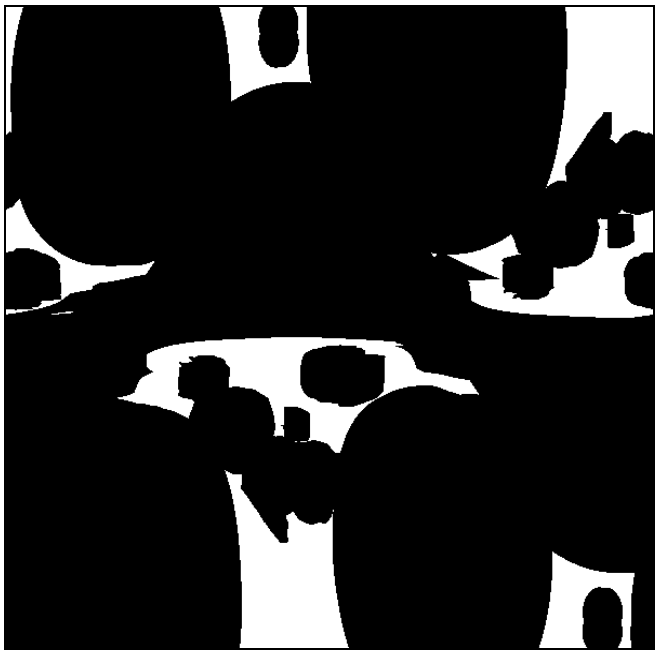

Figure 14: *2D-slice of the configuration-space of ROTEX from Figure 13 (p.13) for variable joints number four and five. Abscissa:* $\varphi_4 \in [-2,62;\ 3,67]$. *Ordinate:* $\varphi_5 \in [-2,09;\ 2,09]$.

5 Results

The task is to generate kinematic trajectories for arbitrary start-goal-combinations in the configuration-space of manipulators. We will test the ability of Parti-game to construct a partitioning of the configuration-space that can serve as basis for planning trajectories. I.e. for testing a partitioning we select a number of arbitrary start and goal configurations and try to generate a (of course collision-free) trajectory using only the diffusion algorithm, database entries, and the greedy-controller (but without further refining the partitioning, which we enforce by setting parameter *nc*, see paragraph 3.4).

5.1 Learning Strategies

The first learning strategy selects an arbitrary start-goal-combination. Starting with the current partitioning Parti-game is run once until a refinement of the partitioning took place. Then a new start-goal-combination is selected. This strategy seems sound, since during training the trajectory is not the primary interest but the construction of the partitioning. However, it turned out that the Parti-game's exploration efforts do not decrease from run to run for the same start-goal-combination. Instead again and again there are situations where sometimes several refinements are needed in order to approach the goal. Of course these refinements aggregate in the partitioning and therefore are necessary for the final partitioning.

So we used a second strategy: we plan a trajectory for the current start-goal-combination, i.e. several runs of Parti-game are performed until a solution is found. Then a new random start-goal-combination is selected and the procedure is iterated.

However for the ability to generate trajectories these different learning strategies do not show large differences after several thousand runs of Parti-game, since in both cases there is a very fine partitioning.

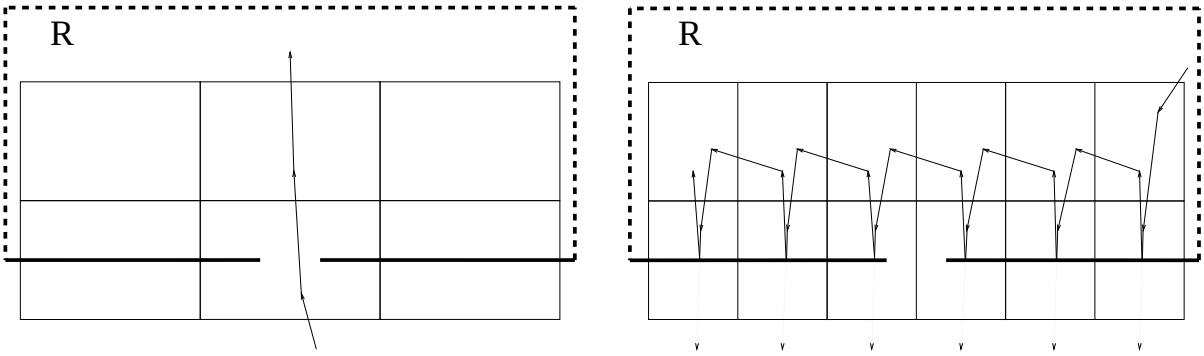

Figure 15: *Example for problems with leaving a region R with a small entry/exit. On the left side Parti-game entered the region. On the right side Parti-game can not leave R (planned with dotted arrows) though a first refinement of the partitioning took place. Even the next (horizontal) refinement will not help.*

5.2 Impacts of the Maximum Partition Depth

Choosing a maximum partition depth *mpd* is sound in order to guarantee termination of Parti-game. The difficulty is to choose the right value for *mpd*. Principally *mdp* would have to be adapted to the difficulty of the environment, i.e. the size of the passages from one region in configuration-space to another. However, the user usually has no idea of that size, since having some small passages does not automatically mean that there do not exist other, larger passages. Furthermore, it is nearly impossible to imagine the configuration-space of a multi-dimensional manipulator environment (compare Figure 14, p.13). If *mpd* is chosen too small then we risk not to find an existing trajectory. If *mpd* is chosen too large then we may have to accept exploding memory consumption and extreme computation time as in original Parti-game.

In our experiments it turned out that for *mpd* a value of five or six times the number of joints is a good choice. Since the number of potential trajectories increases with an increasing number of joints, it is likely that also slightly smaller values show a good performance for higher dimensional environments.

If *mpd* is fixed there arises a typical phenomenon. Parti-game captures itself in a region with only small exits. This might happen due to the manner we split cells. An example is shown in Figure 15 (p.14). Parti-game entered somehow accidently region R (on the right side) with small exits. Due to the interplay of partitioning and greedy-controller it might now happen that Parti-game does not find one of these exits. Even some refinements of the partitioning may not help (partitioning after refinement in Figure 15 (p.14) right side, even the next horizontal split will not help to find the example exit). If the maximum partition depth is reached for an insufficient fine partitioning, Parti-game is trapped in region R.

However, ending a run being trapped in a region is not necessarily a disadvantage, if we set back to start S for the next run. For successive runs all partitions in region R are marked with $J_{WC} = \infty$, i.e. R is not entered any more. Therefore R must not be searched any more. If there is some other trajectory around R, this can even be an advantage, save partitions, and speed up trajectory generation. The only reason to stop planning attempts is given, if Parti-game yet stops in the partition around start S due to the maximum partition depth.

5.3 Impacts of the Reduced Partition Refinement

Similarly to the maximum partition depth *mdp* also the percentage *plp* of losing partitions that will be split must be chosen in advance without any knowledge. Opposite to *mdp* it is not critical to choose *plp*, since necessary splits of partitions can be made up in consecutive refinements (see paragraph 3.3). The advantage of using *plp* is that much information is kept in the database *DB* , namely all information for all partitions that are not split. All these partitions must not be re-examined in the next run. A typical

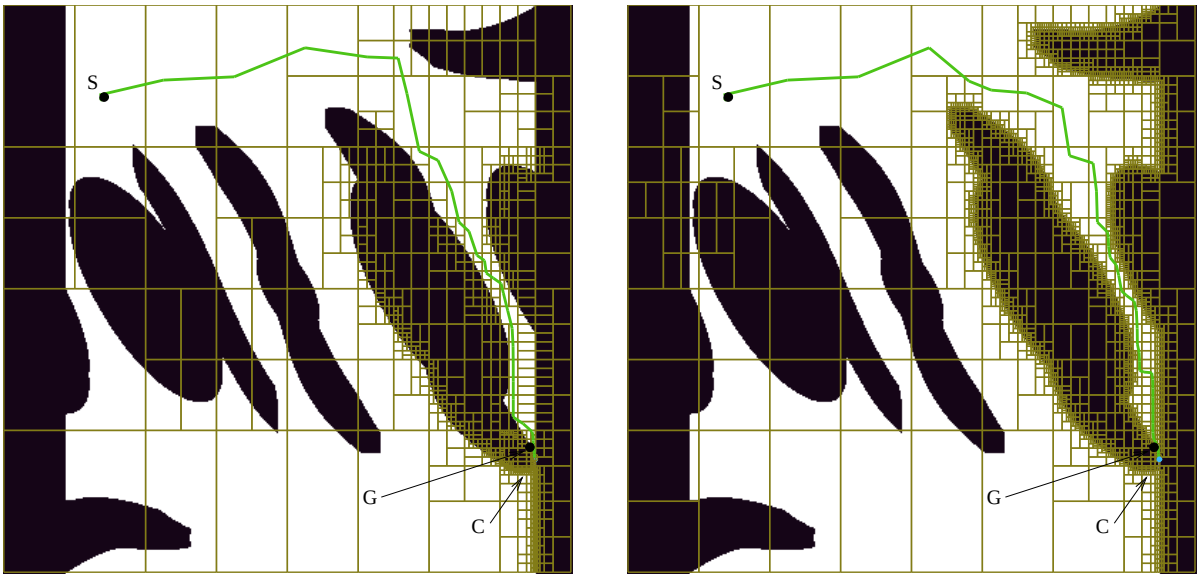

Figure 16: *Extreme example for problems of Parti-game with* $plp \neq 100$. *On the left side an original Parti-game succeeds with 1065 partitions and smallest partitions with partition depth* $pd = 19$, *trying to use the putative narrow channel* C *below goal* G, *which in reality is no channel. On the right side Parti-game with* $plp = 30$ *succeeds with 2348 partitions but smallest partitions of partition depth* $pd = 15$.

good value for plp is $10 < plp < 40$. Of course there is additional effort for sorting the partition according their size. But only for very small values of $1 < plp < 3$ this extra effort is not negligible. In nearly all experiments using plp helped to find solutions with a smaller final partition depth. Furthermore mostly the solution is found in less time. Nevertheless the total number of partitions usually is not smaller but significantly higher.

The reason is quite obvious. If there is only one trajectory from start to goal that leads through a very narrow channel the original Parti-game and Parti-game with $plp \neq 100$ are equal. However, if there is also another trajectory using larger partitions original Parti-game will do that, opposite to Parti-game with $plp \neq 100$ that unnecessarily splits all partitions to nearly that minimally necessary size. Furthermore, even if such narrow channels are present in the environment, it is not sure, that the algorithm will be forced to find a solution through that channel, due to the random selection of start-goal-combinations. An extreme example for this behaviour is shown in Figure 16, p.15 (the putative but extremely narrow channel C is more obvious in Figure 10, p.12).

5.4 Results in the Two-Dimensional Example Environment

For the two-dimensional example environment (see paragraph 4.1) we set $mdp = 36$. This limit is never reached. For several thousand runs (the start-goal-combination is changed after a trajectory for the current task was found) training on a Sun Sparc 10 workstation in multi-user-operation takes some minutes. The size of the database DB is negligible compared to the size of the program.

First we test the original Parti-game. We take 2000 random start-goal-combinations for testing. Figure 17 (p.16) shows the number of partitions, the partition depth of the smallest partition multiplied with factor 100, and number of test runs needed to find a collision free trajectory (i.e. no changes are allowed in the partitioning, but information is stored into the database DB, see paragraph 3.4) multiplied with factor 1000. The number of test runs is computed after building the partitioning. The parameter nc is set in order to prevent further refinement of the partitioning (see paragraph 3.4). If the partitioning is good, the solution will be found at the first test run. For a less good partitioning further runs may be necessary to change the entries in the database DB. For even worse partitions there might be no solution

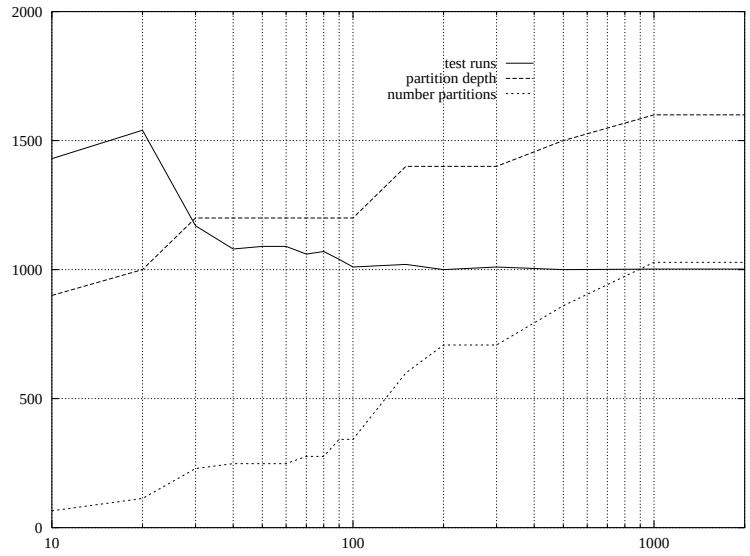

Figure 17: *Results for the original Parti-game in the two-dimensional example environment. X-Axis: Number of tasks solved with Parti-game. Y-Axis: Number of test-runs, partition depth, and number of partitions. In order to fit into one scale, the number of test-runs to find a solution for a task is multiplied with factor 1000, the partition depth pd of the smallest partitions is multiplied with factor 100.*

for some start-goal-partitions (However, since we do partitioning as long as one task is solved or the improved Parti-game succeeds, even the first task is solved). Since in the two-dimensional environment trajectories for all start-goal-combinations are found, the number of test-runs alone is an indicator how good the partitioning already is.

The partitioning for the original Parti-game after construction with 50 and 1000 start-goal-combinations is shown in Figure 18 (p.17).

Second we test Parti-game with a reduced partition refinement with value $plp = 50$. We use 5000 random start-goal-combinations for building up the partitioning. The results are depicted in Figures 19 (p.17) and 20 (p.18).

The original as well as the modified Parti-game algorithm give good results. There are no substantial differences in the partitions, however, as expected the modified Parti-game clearly uses more partitions (see paragraph 5.3). For both algorithms the refinement of the partitions could have been stopped after 1000 runs, since no substantial improvements with respect to trajectory generation were done afterwards (but the number of partitions was further increased). There are two interesting results: First, the partition depth *pd* of the smallest partitions remains constantly at 14 from 10 to 4000 runs with the modified Parti-game, while the number of partitions steadily increases. This is due to a difficult task in the beginning of the training that enforced a high partition depth, while subsequent runs did not need to raise that number. Second, between run 100 and 200 of the modified algorithm on the average 1,18 test runs are necessary to generate each trajectory. This is an interesting phenomenon, since a trajectory must be included in this test set that only can be generated after substantially altering the entries in the database *DB* (these changes are kept for the following tasks). It is not likely that this number 1.18 is produced by several different trajectories each altering the database *DB* a little bit exactly between 100 and 200 runs, since in that case generally the partitioning should need refinement and we should have an increased number of test-runs all the time.

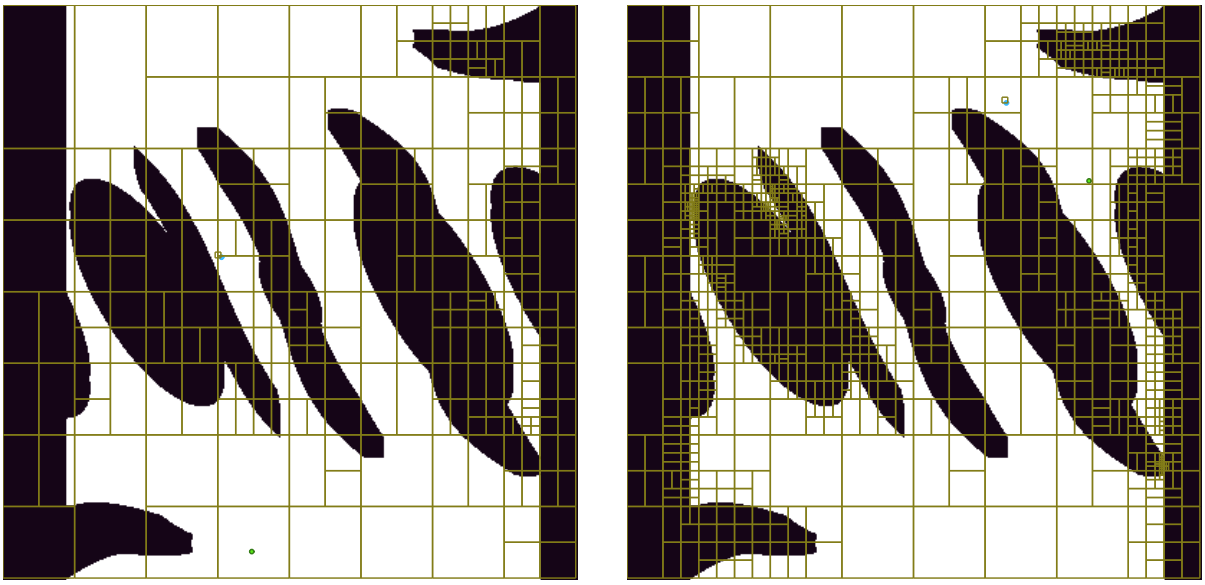

Figure 18: *Partitioning with original Parti-game after building the partitioning with 50 (left) and 1000 (right) start-goal-combinations.*

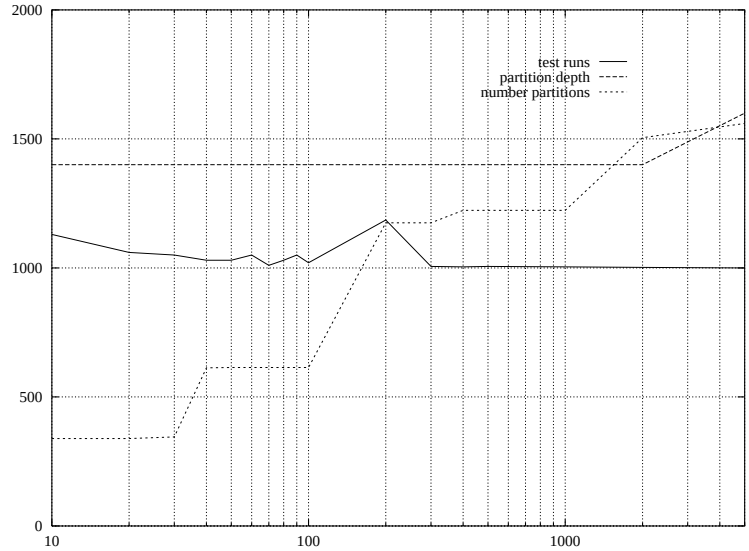

Figure 19: *Results for Parti-game with* $mpd = 36$ *and* $plp = 50$ *in the two-dimensional example environment. The number of test-runs is multiplied with factor 1000, the partition depth pd of the smallest partitions is multiplied with factor 100.*

5.5 Results in the Three-Dimensional Example Environment

For our three dimensional example environment (see paragraph 4.2) memory consumption and processing time is already no more tolerable. Constructing more than 7000 partitions after 2000 runs with original Parti-game takes nearly an hour on the Sun Sparc 10 in multi user operation (see Figure 21, p.18). 5000 runs produce 12380 Partitions and a maximum partition depth of 23. For this partitioning more than 8 MB memory are necessary. Its construction takes about 1.3 hours.

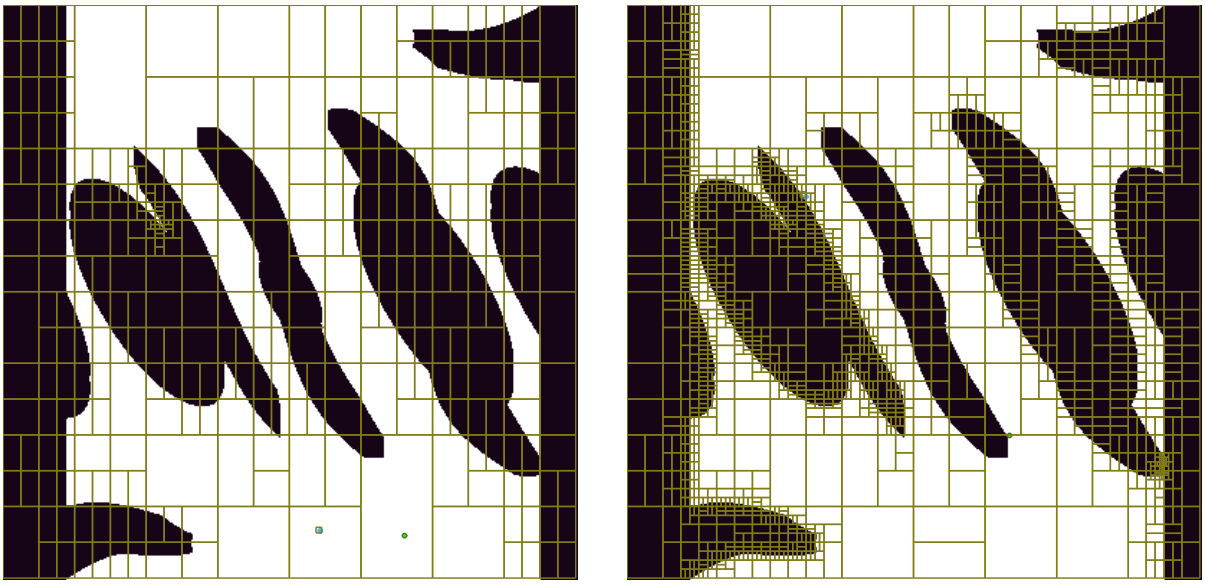

Figure 20: *Partitioning with Parti-game with* $mpd = 36$ *and* $plp = 50$ *after building the partitioning with 20 (left) and 5000 (right) start-goal-combinations. Compared to Figure 18 (p.17) much more partitions are generated.*

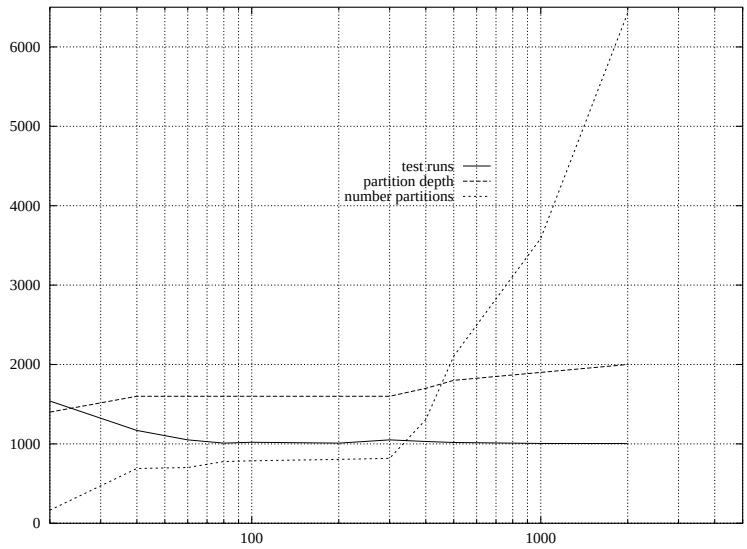

Figure 21: *Results for the original Parti-game in the three-dimensional example environment. The number of test-runs is multiplied with factor 1000, the partition depth pd of the smallest partitions is multiplied with factor 100.*

The modified Parti-game with $plp = 30$ only produces 2880 partitions within less than 30 minutes (Figure 22, p.19).

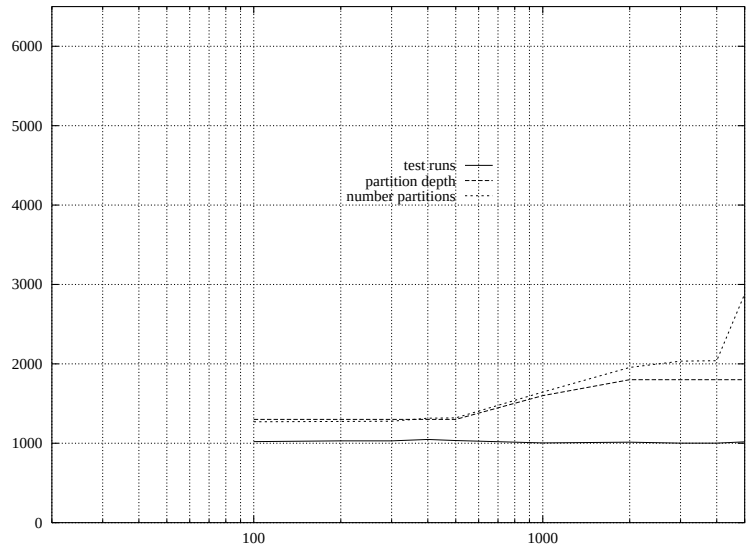

Figure 22: *Results for Parti-game with plp = 30 in the three-dimensional example environment. The number of test-runs is multiplied with factor 1000, the partition depth pd of the smallest partitions is multiplied with factor 100.*

5.6 Results in the Six-Dimensional Real World Environment

As expected from the increase in resources' consumption from the two- to the three-dimensional example environment, we get heavy difficulties with the real world example. For relatively simple tasks as in Figure 23 (p.20), left side, Parti-game succeeds in short time with 21 partitions and a partition depth of $pd = 8$ for the smallest partitions. However, using random start-goal-combinations we quickly get situations as in Figure 23 (p.20) right side, where the manipulator has to be completely turned around in most of its joints. In none of our experiments Parti-game was able to solve such a task in less than an hour (our limit for a single task). This was independent of any modification we were testing.

If we present only tasks (start-goal-combinations) that can be solved (unfortunately only after trajectory generation we know if a task can be solved by Parti-game with the given resources), we quickly end up with more than 10000 partitions. Sometimes we were using more than 20 M main memory. Memory handling alone significantly slows down the path-planning process. Also the frequent diffusion (taking place after each collision and after each partitioning refinement, see paragraph also 2.2.4) then takes more than a second each. Therefore Parti-game is not at all a universal, quick, resources saving algorithm for arbitrary, real world manipulator trajectory generation problems.

6 Discussion

Parti-game is based on the assumption that there is always a trajectory between arbitrary start and goal configurations. Without the improvement of a maximum partitioning depth *mpd*, Parti-game would never stop and end up in an infinite loop, if there is no trajectory. However, also if after a refinement of the partitioning there is a chance to find a trajectory, Parti-game throws away some information. Parti-game does not keep track of the experiences where obstacles have been. After the refinement collisions with the same obstacles have to completely re-explore the space took by the old (and now the new) partitions.

Regarding the advantage of guaranteed termination but risking to miss an existing trajectory with a fixed maximum partition depth *mpd*, it might be tempting to try the following idea: First set *mpd* to a

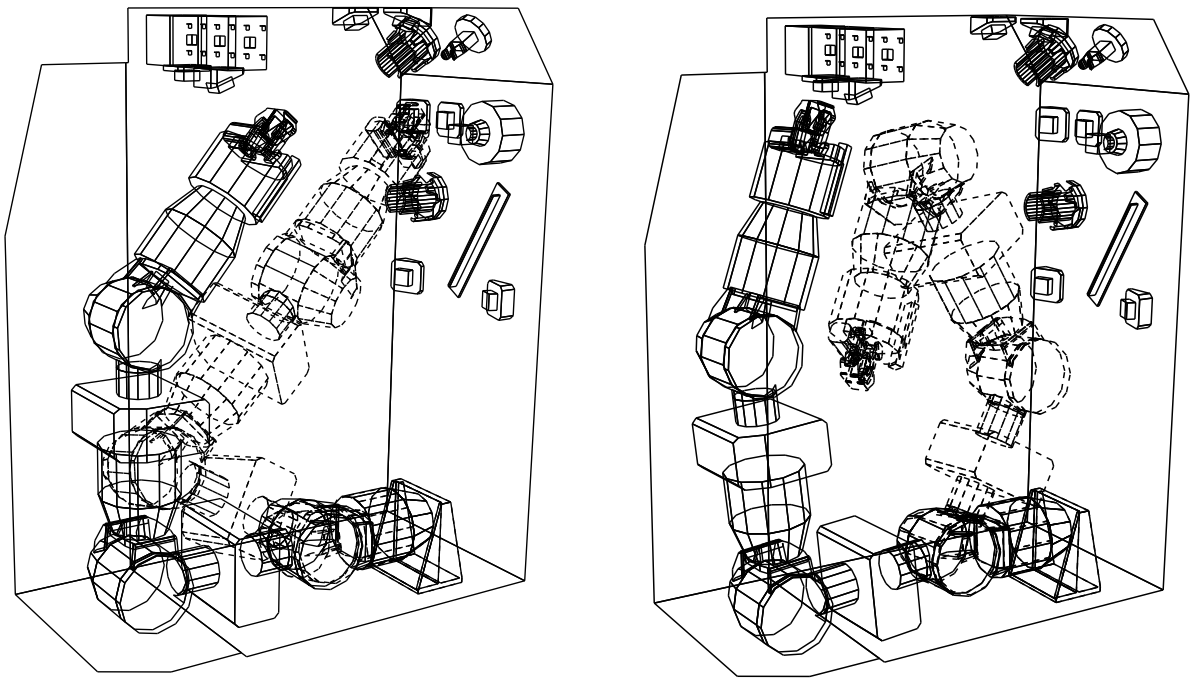

Figure 23: *Examples for problems that Parti-game can solve (left) or not solve (right), respectively. The drawn through manipulator represents the start configuration, the dotted manipulator represents the goal configuration. In the right image the manipulator has to turn around completely in most of its joints.*

small value in order to save partitions. If there is no trajectory found, we could increase *mpd* and run Parti-game again. Unfortunately this procedure yields partitionings with almost equally sized partitions. Therefore it is contrary to the idea of Parti-game and useless.

Especially in cluttered real world examples (e.g. the six-dimensional environment in paragraph 4.3) Parti-game runs into two-fold difficulties. On one hand too many partitions are created in small ravines or around large obstacles that do not at all help to build trajectories (this is obvious from the fact that the number of tests does not increase any more after some training, i.e. the trajectories are found quickly, but the mean number of partitions steadily increases with ongoing training, i.e. these new partitions do not help to make finding a trajectory easier, compare also paragraph 3.1). Since the partitioning refinement bisects the partition size with each refinement, the storage needed often increases exponentially. On the other hand also the diffusion algorithm (paragraph 2.2.4) takes more and more time, since it runs through more and more cells in the inhomogeneous, but fully covered space. Therefore, though obvious improvements in experiments with snake robots of different dimensionality but fixed robot length (Moore and Atkeson,1995), we can not really overcome the curse of dimensionality (Bellman,1957) using Parti-game.

Reading Moore and Atkeson (1995), the question arises why the example of the nine-dimensional manipulator works so well? Well a first answer may be, because the simulation with line-segments is very cheap, i.e. not much time must be spent on simulating the manipulator. A second answer is that there is a nine dimensional manipulator in a two-dimensional environment. Therefore the manipulator has abundant redundant degrees of freedom to solve the trajectory generation problem. A third answer is that the configuration-space is rather easy. There is only one thin wall as an obstacle, but a large (nine-dimensional) free-space to move around. A fourth answer is that Moore and Atkeson (1995) generally only do examples with one goal. Fiddling around a little bit with the software provided by Moore and Atkeson (1995) one can easily find start-goal-combinations that also cause massive problems (e.g. start several runs with angles $\vec{\varphi} = (-3, 0, 0, 0, 0, 0, 0, 0, 0)$. After finding a trajectory with 80 partitions and

a partition depth $pd = 19$ in the first run, after the second run we end up with 660 partitions and a partition depth of $pd = 44$ after about 11 minutes CPU-time on a Sun Sparc 20 Model 701! Afterwards the partitioning seems to stabilize, we end up with constant partition depth $pd = 44$ but 664, 666, 667 partitions in the next runs after all together 16 minutes CPU-time. Though the next runs do not need a further refinement of the partitioning to find the goal, they only stop after another 5, 2, 0.5 minutes CPU-Time. The ninth run stops after another 2 minutes CPU-time with 771 partitions. Run 13 alters the partitioning to 675 partitions. From run 15, after consuming more than 28 minutes CPU-time for one single start-goal-combination, we arrived at a stable partitioning and database *DB* with 675 partitions and partition depth $pd = 44$. It now takes a little bit more than 1 second CPU-time to generate the final path.).

This last point shows a general disadvantage of Parti-game for the trajectory generation task. Parti-game does not produce symmetric solutions, i.e. if a certain partitioning is good to find a trajectory from start to goal, this partitioning generally does not help too much to find a trajectory from goal to start. Therefore we can not exploit the problem inherent symmetry. This is mainly an outcome of the interplay of the used kind of greedy-controller (that only goes in half the way to the partition's center. We could remedy this using a greedy-controller as described in paragraph 3.2). Furthermore relatively coarse partitionings found for one goal seldomly are directly usable for even only slightly changed start configurations but the same goal (compare problems in paragraph 3.1, Figure 7, p.9, where special partitioning for exactly one goal at the other side of a small passage is created). If we substantially change the goal (i.e. the new goal is on another side of an obstacle than the previous goal), we often can not take profit from most of the partitioning at all.

Much better results for manipulator trajectory generation are provided by recently published (first results were published about the time Parti-game was published first), graph-based trajectory generation algorithms as e.g. Kavraki and Latombe (1994), Horsch et al. (1994), Eldracher (1994), Eldracher and Pic (1995), but also Baginski (1996a), Kinder and Brychcy (1994), Brychcy and Kinder (1995). Especially Kavraki and Latombe (1994), (Kinder and Brychcy,1994), and Eldracher (1994) easily handle ten-dimensional environments for arbitrary start-goal-combinations. The graph construction process (preprocessing similar to the construction of a general partitioning) with these approaches takes between five and fifteen minutes. Afterwards trajectories are produced in some tenths of a second, without need to run any time-consuming diffusion algorithms in large partitionings (a special form of graphs). In our opinion the main reason for this far better performance is that the graph based algorithms only examine one-dimensional sub-spaces, but never run any algorithms through the whole dimensionality of the search space. Furthermore usually the structures to be stored are relatively small and simple. Interestingly also the only graph based approach that explicitly stores regions of supposed free-space (Kinder and Brychcy,1994) runs into difficulties with memory consumption. However, on one hand these difficulties only arise for ten-dimensional environments (opposite to Parti-game that already has difficulties with three-dimensional environments) and on the other hand there are complex structures for each cell (comparable to the partitions), while the number of partitions needed to cover the space is significantly lower compared to Parti-game.

An advantage of Parti-game is that for the solution of one task it does not search the whole configuration-space. Therefore Parti-game still remains an interesting alternative to model free approaches (e.g. Glavina,1990; Baginski,1996a), if only some trajectories should be planned. However, this advantage vanishes, if we want to construct a general trajectory planner, where Parti-game is forced to search the whole configuration-space by the presentation of many thousands of arbitrary start-goal-combinations in an external loop.

7 Conclusion

What can be concluded from these results? A surely wrong conclusion would be that Parti-game is only an algorithm clearly outperformed by others. Parti-game is a very general algorithm for any reinforcement-problems. Examples of a wide area of applications are shown in Moore and Atkeson

(1995). Parti-game is a well structured approach. Many details can be changed without changing the basic idea, especially the greedy controller and the partition refinement scheme, which seem to be the most important sources of problems. Furthermore Parti-game allows to transfer the idea of approximate cell-decomposition, known since long in the robotics environment, to that general class of reinforcement-problems. Besides, even for robotics tasks the prerequisite of a known geometric model (in order to do the well known cell decomposition) is dropped and substituted by the database *DB* of all experiences during planning.

The algorithms that outperform Parti-game for manipulator trajectory generation (Kavraki and Latombe,1994; Eldracher,1994; Brychcy and Kinder,1995) are clearly specialized approaches. Therefore Parti-game remains an interesting alternative for all that tasks, where no specialized alternatives are available. It should be pointed out, that the simple greedy-controller does not interact favorably with the partitioning. But the idea of Parti-game is not based on this controller and can be altered, as many of the details described in Section 2.

However, for manipulator trajectory generation Parti-game is not the algorithm of choice, since it is too time- and memory-consuming. Especially small ravines and cluttered spaces, causing problems for Parti-game, are very common in the configuration-spaces of manipulators.

8 Acknowledgment

This work was conducted in the laboratory of Prof. Dr. Brauer at Technische Universität München, Germany. Without the support of Prof. Dr. Brauer and the fruitful discussions with our colleagues Dieter Butz, Dr. Daniel Hernández, Dr. Hans Geiger, Margit Kinder, Dr. habil. Jürgen Schmidhuber, and Dr. Gerhard Weiß, as well as the students in our laboratory, who conducted most of the experiments, this work would not have been possible. Furthermore special thanks to Andrew W. Moore, the inventor of Parti-game, who helped us a lot to get the details of his algorithms right and provided us with his software. Furthermore my thank belongs to DLR Oberpfaffenhofen, mainly Prof. Dr. Hirzinger, for providing the geometry of the ROTEX environment. Last but not least I want to thank Leslie Kaelbing for encouraging me to write this report during a common stay at IDSIA, Lugano.

References

Baginski, B. (1996a). Local motion planning for manipulators based on shrinking and growing geometry models. In *Proceedings of IEEE Conference on Robotics and Automation* Minneapolis.

Baginski, B. (1996b). The Z^3–method for fast path planning in dynamic environments. In *IASTED Aplications of Control and Robotics*, pages 47–52 Orlando, Florida.

Bellman, R. (1957). *Dynamic Programming.* Princeton University Press, Princeton, NJ.

Brooks, R. A. (1983). Find-path for a puma-class robot. In *Proceedings of AAAI 83*, pages 40–44.

Brychcy, T. and Kinder, M. (1995). A neural network inspired architecture for robot motion planning. In Bulsari, A. and Kallio, S., editors, *Engineering Applications of Artificial Neural Networks: Proceedings of the International Conference EANN '95, 21-23 August 1995, Otaniemi/Helsinki, Finland*, pages 103–110. Finnish Artificial Intelligence Society.

Canny, J., editor. (1988). *The Complexity of Robot Motion Planning* (2 edition). MIT Press, Cambridge, MA.

Chapman, D. and Kaelbing, L. P. (1991). Learning from delayed reinforcement in a complex domain. Technical report, Teleos Research.

Dayan, P. and Hinton, G. (1993). Feudal reinforcement learning. In Hanson, S. J., Cowan, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*. Morgan Kaufmann Publishers, San Mateo, CA.

Eldracher, M. (1994). Neural subgoal generation with subgoal graph: An approach. In *World Congress on Neural Networks WCNN-94, San Diego, 1994*, Volume II, pages II–142 – II–146. Lawrence Erlbaum Associates, Inc., Publishers, Hillsdale.

Eldracher, M. and Pic, T. (1995). How neural networks speed up a randomized incremental graph-based motion planner. *Zeitschrift für angewandte Mathematik*, *To appear*.

Glavina, B. (1990). Solving findpath by combination of goal-directed and randomized search. In *IEEE International Conference on Robotics and Automation*, pages 1718–1723 Cincinnati, USA.

Horsch, T., Schwarz, F., and Tolle, H. (1994). Motion planning with many degrees of freedom – random reflections at c-space obstacles.. In *1994 IEEE International Conference on Robotics and Automation, San Diego*, pages 3318–3323.

Kavraki, L. and Latombe, J.-C. (1994). Randomized preprocessing of configuration space for fast path planning. In *1994 IEEE International Conference on Robotics and Automation, San Diego*, pages 2138–2145.

Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. *International Journal for Robotics Research*, *5*(1), 90–98.

Kinder, M. and Brychcy, T. (1994). Path planning for six-joint manipulators by generalization from example paths. Forschungsberichte Künstliche Intelligenz FKI-192-94, Institut für Informatik, Technische Universität München.

Latombe, J.-C. (1991). *Robot Motion Planning* (3 edition). Kluwer Academic Press, Boston.

Lozano-Peréz, T. (1981). Automatic planning of manipulator transfer movements. *IEEE Transactions on Systems, Man and Cybernetics*, *SMC-11*(10), 681–698.

Moore, A. W. (1994). The partigame algorithm for variable resolution reinforcement learning in multidimensional state-spaces. In *Neural Information Processing Systems 6*, pages 711–718. Morgan Kaufmann Publishers, San Mateo, CA.

Moore, A. W. and Atkeson, C. G. (1995). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, *21*, 119–223.

Schwartz, J. and Sharir, M. (1983). On the piano movers' problem: II. general techniques for computing topological properties of real algebraic manifolds. In *Advances in Applied Mathematics*, Volume 4, pages 298–351. Academic Press.