DRAFT DOCUMENT

The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State-spaces

Andrew W. Moore
Carnegie Mellon University
School of Computer Science &
Robotics Institute
5000 Forbes Ave
Pittsburgh, PA 15213
awm@cs.cmu.edu

Christopher G. Atkeson
Massachusetts Institute of Technology
Artificial Intelligence Lab &
Brain & Cognitive Sciences
545 Technology Square
Cambridge, MA 02139
cga@ai.mit.edu

January 22, 1994

ABSTRACT

Parti-game is a new algorithm for learning from delayed rewards in high dimensional continuous state-spaces. In high dimensions it is essential that learning does not explore or plan over state-space uniformly. Parti-game maintains a decision-tree partitioning of state-space and applies techniques from game-theory and computational geometry to efficiently and adaptively concentrate high resolution only on critical areas. The current version of the algorithm is designed to find feasible solutions to high dimensional problems. Future versions will be designed to find a solution that optimizes a real-valued criterion. Many simulated problems have been tested, ranging from two-dimensional to nine-dimensional state-spaces, including mazes, path planning, non-linear dynamics, and planar snake robots in restricted spaces. In all cases, a good solution is found in less than twenty trials and a few minutes.

1 Reinforcement Learning

Reinforcement learning [Michie and Chambers, 1968, Sutton, 1984, Watkins, 1989, Barto *et al.*, 1991] is a promising method for robots to program and improve themselves. This paper addresses one of reinforcement learning's biggest stumbling blocks: the curse of dimensionality [Bellman, 1957], in which costs increase exponentially with the number of state variables. These costs include both the computational effort required for planning and the physical amount of data that the control system must gather.

Much work has been performed with discrete state-spaces: in particular a class of Markov decision tasks known as grid worlds [Watkins, 1989, Sutton, 1990a]. Most potentially useful applications of reinforcement learning, however, take place in multidimensional continuous state-spaces. The obvious way to transform such state-spaces into discrete problems involves quantizing them: partitioning the state-space into a multidimensional grid, and treating each box within the grid as an atomic object. Although this can be effective (see, for instance, the pole balancing experiments of [Michie and Chambers, 1968, Barto *et al.*, 1983] which break state-space up in this way), the naive grid approach has a number of dangers which will be detailed in this paper.

This paper studies in detail the pitfalls of discretization during reinforcement learning and then introduces the parti-game algorithm. Some earlier work [Simons *et al.*, 1982, Moore, 1991, Chapman and Kaelbling, 1991, Dayan and Hinton, 1993] considered recursively partitioning state-space while learning from delayed rewards. The new ideas in the parti-game algorithm include (i) a game-theoretic splitting criterion to robustly choose spatial resolution, (ii) real-time incremental maintenance and planning with a database of all previous experiences, and (iii) using local greedy controllers for high-level "funneling" actions.

2 Assumptions

The parti-game algorithm applies to learning control problems in which:

1. State and action spaces are continuous and multidimensional.
2. "Greedy" and hill-climbing techniques can become stuck, never attaining the goal.
3. Random exploration can be intractably time-consuming.
4. The system dynamics and control laws can have discontinuities and are unknown: they must be learned. However, we do assume that all paths that the system can travel through state space are continuous.

The experiments reported later all have properties 1–4. However, the initial algorithm, described and tested here, has the following restrictions:

5. Dynamics are deterministic.
6. The task is specified by a goal state, not an arbitrary reward function.
7. The goal state is known.
8. A feasible solution is found, not necessarily a path which optimizes a particular criterion.
9. A local greedy controller is available, which we can ask to move greedily towards any desired state. There is no guarantee that a request to the greedy controller will succeed. For example, in a maze a greedy path to the goal would quickly hit a wall.

This paper begins by giving a series of algorithms of increasing sophistication, culminating in parti-game. We then give results for a number of experimental domains and conclude with discussion of how constraints 5–9 may be relaxed.

3 The Parti-game Algorithm

The parti-game learns a controller from a start region to a goal region in a continuous state-space. We now give four increasingly effective algorithms which attempt to perform this by discrete partitionings of state-space. Algorithms (1) and (2) are non-learning: they plan a route to the goal given a-priori knowledge of the world. Algorithms (3) and (4) must learn, and hence explore, the world while planning a route to the goal. Algorithm (1) is a planner which assumes that state transitions begin at the center of partitions, and generalizes this to the assumption of starting randomly within a partition. Algorithm (2) avoids some of (1)'s mistakes by means of worst-case planning. Algorithm (3) is a learning version of (2). Algorithm (4) is the parti-game algorithm. It develops a variable resolution partitioning in conjunction with the planning and learning of Algorithm (3).

3.1 Algorithm (1): Non-learning and fixed partitions

A *partitioning* of a continuous state-space is a finite set of N disjoint regions labeled $1, 2, ...N$ such that the whole of state space is covered by the union of all partitions. Throughout this paper we will assume the partitions are all axis-aligned hyperrectangles, though this assumption is not strictly necessary. It is important to clarify a potential confusion between real-valued states and partitions. A *real-valued state*, s, is a real-valued vector in a multidimensional space. For example, states from the maze depicted in Figure 1 are two-dimensional (x, y) coordinates. A partition is a discrete entity, and Figure 1 is broken into six partitions with identifiers $1 \ldots 6$. Each real-valued state is in one partition and each partition contains a continuous set of real-valued states. Define **NEIGHS**(i) as the set of partitions which are adjacent to i. In Figure 1, **NEIGHS**$(1) = \{2, 4\}$.

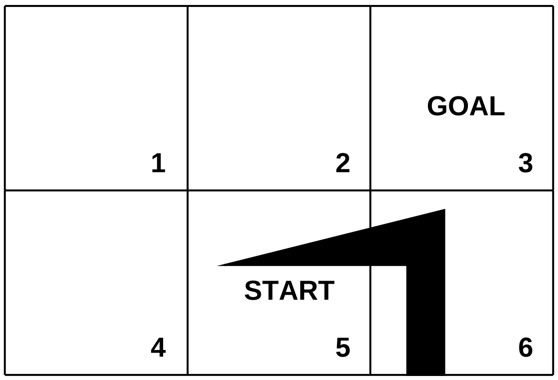


Figure 1: A two-dimensional continuous maze with one barrier: the black polygonal region near the bottom right. state-space has been discretized into six square partitions.

Algorithm (1) takes as input an environmental model and a partitioning P. The environmental model can be any model (for example, dynamic or geometric) which we can use to tell us for any real-valued state, control command and time interval, what the subsequent real-valued state will be. The algorithm outputs a *policy*: a mapping from partition identifiers to the neighboring partitions which should be aimed for. The algorithm depends upon the **NEXT-PARTITION** function, which we define first. **NEXT-PARTITION** tells us which partition we end up in if we start at a given real-valued state and keep moving toward the center of a given partition (using a local greedy controller) until we either exit our initial partition or get stuck. Let i be the partition containing real valued state s. Continue applying the local greedy controller "aim at partition j" either until partition i is exited or until we become permanently stuck in i. Then

$$\textbf{NEXT-PARTITION}(s, j) = \begin{cases} i \text{ if we became stuck} \\ \text{the partition containing the exit state otherwise} \end{cases} \tag{1}$$

The test for sticking can simply be implemented as a test to see if the system has not exited the partition after a predefined time interval. Depending upon the application other sticking detectors are possible, such as an obstacle sensor on a mobile robot.

Algorithm (1) works by constructing a discrete, deterministic Markov decision task (MDT) [Bellman, 1957, Bertsekas and Tsitsiklis, 1989] in which the discrete MDT states correspond to partitions. Actions correspond to neighbors thus: action k in partition i corresponds to starting at the center of partition i and greedily aiming at the center of partition k.

ALGORITHM (1).

1 Given N partitions, construct a deterministic MDT with N discrete states $1 \cdots N$. The set of actions of partition i is precisely $\mathbf{NEIGHS}(i)$. Define $\mathbf{NEXT}(i,k)$ as

$$\mathbf{NEXT}(i,k) = \mathbf{NEXT\text{-}PARTITION}(\mathbf{CENTER}(i),k) \tag{2}$$

where $\mathbf{CENTER}(i)$ is the real-valued state at the center of partition i.

2 The shortest path to the goal from each partition i, denoted by $J_{SP}(i)$, is determined by solving the set of equations:

$$J_{SP}(i) = \begin{cases} 0 & \text{if } i = \mathbf{GOAL} \\ 1 + \min\limits_{k \,\in\, \mathbf{NEIGHS}(i)} J_{SP}(\mathbf{NEXT}(i,k)) & \text{Otherwise} \end{cases} \tag{3}$$

The equations are solved by a shortest-path method such as dynamic programming [Bellman, 1957, Bertsekas and Tsitsiklis, 1989] or Dijkstra's algorithm [Knuth, 1973].

3 The following policy is returned: Always aim for the neighbor with the lowest $J_{SP}()$ value.

This simple algorithm has immediate drawbacks. It will minimize the number of partitions to the goal, not the real distance. And the discretization can easily find impossible solutions or fail to find valid solutions. As an example of the former, in Figure 1, Algorithm (1) would find solution path $5 \rightarrow 6 \rightarrow 3$. This is because it is possible to travel from the center of 5 and enter 6 (in the part of 6 to the left of the obstacle), and it is possible to travel from the center of 6 and enter 3.

An extension to Algorithm (1) might initially appear to solve the problem. We could remove the assumption that all paths between partitions begin at the center of the source partition. Suppose we produce a stochastic Markov decision task. Let p^k_{ij} be an approximation of the probability of transition to partition j given we have started in i and aimed at the center of k. p^k_{ij} is defined by the probability we end up in partition k from a uniformly randomly chosen legal start point in partition i. The dynamic programming step of the previous algorithm is altered so that it now solves the stochastic MDT:

$$J_{SP}(i) = \begin{cases} 0 & \text{if } i = \mathbf{GOAL} \\ 1 + \min\limits_{k \,\in\, \mathbf{NEIGHS}(i)} \sum\limits_{j=1}^{N} p^k_{ij} J_{SP}(j) & \text{Otherwise} \end{cases} \tag{4}$$

Although intuitively appealing, this refinement does not help. In the example of Figure 1 the resultant policy from state 5 will still be to aim for 6. As we see from Figure 2, $p^6_{56} = 0.65$, and from

Figure 3, $p^3_{63} = 0.91$. The policy $5 \to 6 \to 3$ is interpreted as the transition graph in Figure 4 which has expected length $1/0.65 + 1/0.91 = 2.64$, and so is preferred over the longer but guaranteed policy of $5 \to 4 \to 1 \to 2 \to 3$.

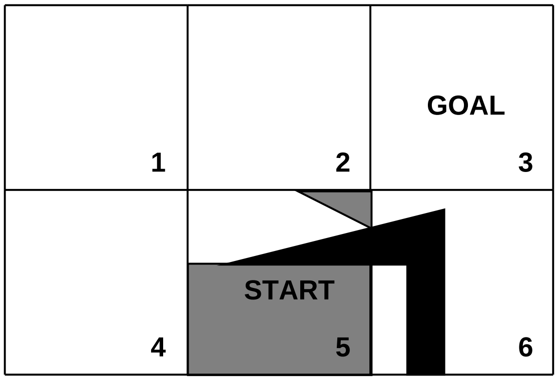


Figure 2: Approximately 65% of the starting states (those in the shaded region) in partition 5 are such that they will enter partition 6 if we aim for the center of 6. Thus $p^6_{56} = 0.65$.

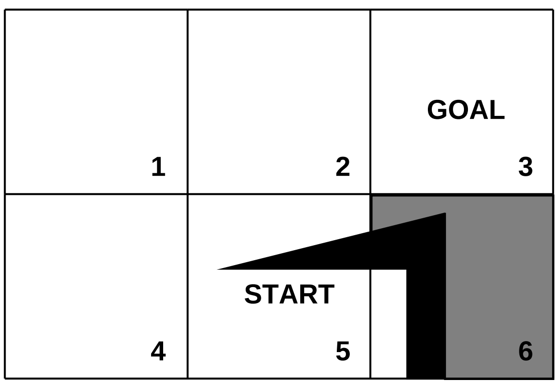


Figure 3: In a similar fashion to Figure 2, $p^3_{63} = 0.91$.

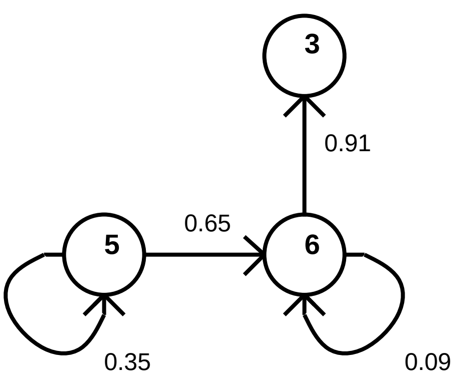


Figure 4: The partition transition probabilities if we follow the $5 \to 6 \to 3$ policy according to the assumptions in the text.

Other variants of this stochastic approximation approach are possible, but they all suffer from the same problem. They are using a Markov decision formalism for something which does not have the Markov property. This is because from a given partition, i, the neighbors which can successfully be reached depends on more than "i", they also depend on the current location within i.

3.2 Algorithm (2): Assuming the worst case

Instead of approximating the steps-to-goal value of a partition by the average steps-to-goal of all real-valued states in the partition, we approximate it by the worst value. As before, each partition has an associated set of actions, each labeled by a neighboring partition. Also, each action now has a set of possible *outcomes*. The outcomes of an action j in a partition i are defined as the set of possible next partitions.

$$\mathbf{OUTCOMES}(i,j) = \left\{ k \;\middle|\; \begin{array}{l} \text{there exists a real valued state } s \text{ in partition } i \text{ for which} \\ \mathbf{NEXT\text{-}PARTITION}(s,j) = k \end{array} \right\} \quad (5)$$

In Figure 5, the actions are denoted by black solid arrows and the outcomes by the thin lines. For example, partition 5 has three actions: "aim at 4", "aim at 2" and "aim at 6". The "aim at 6" action has, in turn, two possible outcomes. We might make it if we are lucky, or else we will remain in 5. The **OUTCOMES**() sets are decisions which an imaginary adversary will be allowed to make, seen in (2).

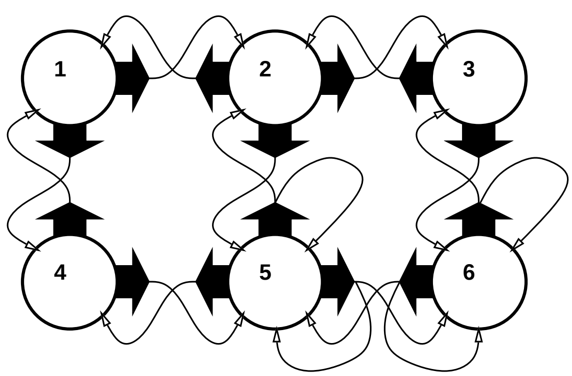


Figure 5: The symbolic representation of the original problem as partitions (circles), actions (solid arrows) and outcomes (thin arrows).

ALGORITHM (2).

1 Define $J_{WC}(i)$ as the worst case number of partitions to the goal assuming that whenever we have specified our current partition i and our intended next partition j, an adversary is permitted to place us in the worst position within partition i prior to the local controller being activated.

Solve the following set of minimax equations:

$$J_{WC}(i) = \begin{cases} 0 & \text{if } i = \mathbf{GOAL} \\ 1 + \min\limits_{k \,\in\, \mathbf{NEIGHS}(i)} \; \max\limits_{j \,\in\, \mathbf{OUTCOMES}(i,k)} J_{WC}(j) & \text{Otherwise} \end{cases} \tag{6}$$

where $J_{WC}(i)$ is allowed to take the value $+\infty$ to denote a partition from which our adversary can permanently prevent us reaching the goal. Call such a partition a losing partition.

2 The following policy is returned: Always aim for the neighbor with the lowest $J_{WC}()$ value.

The $J_{WC}(.)$ function can be computed by a standard minimax algorithm [Knuth, 1973], which is in turn closely related to deterministic dynamic programming algorithms.

This algorithm is pessimistic, but if it tells us that the current partition is n partition transitions to the goal then we can be sure that if we follow its policy we will indeed take n or fewer partition transitions. The trivial inductive proof is omitted.

In Figure 1, Algorithm (2) will decide that partition 5 is four steps from the goal and will recommend heading towards 4. It avoids partition 6 because the minimax assumption scores partition 6 as being ∞ steps from the goal. This is because if, in partition 6, we decide to use action "aim for 3" the adversary will start us in the bottom left of partition 6. And if we use action "Aim for 5", the adversary will start us in the bottom right of partition 6.

It should be observed just how pessimistic the algorithm is. In the almost entirely empty maze of Figure 6 the start partition will be considered a losing partition. So although the minimax assumption ensures guaranteed success *if* it finds a solution, it may often prevent us from solving easy problems. We will see that Algorithm (3) reduces the severity of this problem because instead of considering the worst of all possible outcomes, the planner only considers the worst of all empirically observed outcomes. Thus a block in a piece of a partition which never was actually visited would not be identified as an outcome available to the adversary. Algorithm (4) fully solves the remaining aspects of the problem by increasing the resolution of losing partitions.

3.3 Algorithm (3): A learning version of Algorithm (2)

An important aim of this work is to have a controller which does not begin with an environmental model, but which manages instead to learn purely from experience. Algorithm (2) can be extended to permit this. The set of $\mathbf{OUTCOMES}(i,j)$ for each partition i and neighbor j can be obtained empirically. Whenever an $\mathbf{OUTCOMES}(i,j)$ is altered, the game is solved with the new outcomes set. We still assume that the location of the partition containing the goal is known.

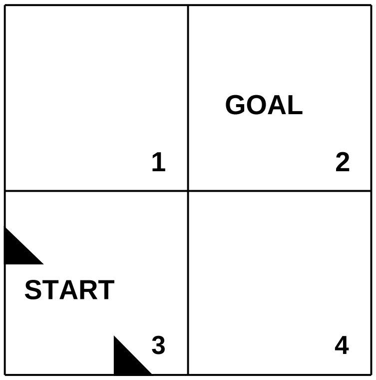


Figure 6: Partition 3 is scored as losing because if it aims for 1 the adversary can place it below the upper triangular block and if it aims for 4 the adversary can place it to the left of the lower triangular block.

A further detail must be resolved. In the early stages, what should be done for those actions which have not yet been experienced? The answer is to assume by default that any neighbor aimed for can be attained. Algorithm (3), based on these ideas, takes three inputs:

- The current real-valued system state s.
- A partitioning of state-space, P.
- A database, D, of all previously observed partition-transitions in the lifetime of the system. This is a set of triplets:

$$(\text{starting in } i_0, \text{I aimed for } j_0 \text{ and actually arrived in } k_0)$$
$$(\text{starting in } i_1, \text{I aimed for } j_1 \text{ and actually arrived in } k_1)$$
$$\vdots$$

The algorithm returns two outputs: The final system state after execution and a binary signal indicating SUCCESS or FAILURE. The database is also updated according to experience.

ALGORITHM (3).
REPEAT FOREVER

1 Compute the **OUTCOMES**(i,j) set for each partition i and each neighbor $j \in$ **NEIGHS**(i) thus:

- If, for any k, $(i,j,k) \in D$ then:

$$\textbf{OUTCOMES}(i,j) = \{k \mid (i,j,k) \in D\} \tag{7}$$

- Else, use the optimistic assumption in the absence of experience:

$$\mathbf{OUTCOMES}(i,j) = \{j\} \tag{8}$$

2 Compute $J_{WC}()$ for each partition using minimax.

3 Let i := the partition containing s.

4 If $i = \mathbf{GOAL}$ then exit, signaling SUCCESS.

5 If $J_{WC}(i) = \infty$ then exit, signaling FAILURE.

6 Else

6.1 Let $j := \underset{j' \in \mathbf{NEIGHS}(i)}{\text{argmin}} J_{WC}(j')$.

6.2 **WHILE** (not stuck and still in partition i)

6.2.1 Actuate local controller aiming at j.

6.2.2 s := new real-valued state.

6.3 Let i_{new} := the identifier of the partition containing s.

6.4 $D := D \cup \{(i, j, i_{\text{new}})\}$

LOOP

An addition to the algorithm can reduce the computational load. If real time constraints do not permit full recomputation of J_{WC} after an outcome set has changed, then the J_{WC} updates can take place incrementally in a series of finite time intervals interleaved with real-time control decisions. Techniques like this are described in [Sutton, 1990b, Peng and Williams, 1993, Moore and Atkeson, 1993].

The following theorem has not been proved but we expect few difficulties: If a solution exists from all real-valued states in all partitions, according to Algorithm (2), then Algorithm (3)will, in fewer than N^3 partition transitions, also find a solution, where N is the number of partitions.

A final note about Algorithm (3) is necessary. More general systems than mazes will produce more interesting games. Later we will see examples of non-uniform partitionings and of dynamics that produce curved trajectories through space. Both cases can produce more detailed game structures than stuck/non-stuck transitions, and Algorithm (3) is applicable in these cases too. Such a structure is shown in Figure 7.

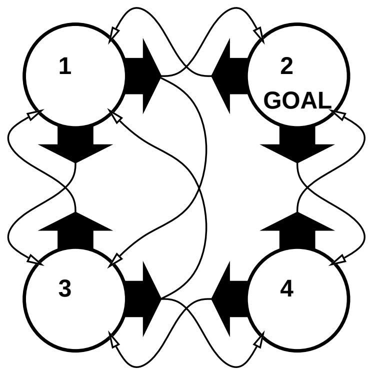


Figure 7: Partitions 1 and 3 are losers because the adversary can force a permanent loop between them.

3.4 Algorithm (4): Varying the Resolution

We do not wish the system to give up when it discovers it is in a partition for which $J_{WC} = \infty$. The correct interpretation of a losing partition is that the planner needs higher resolution, and parti-game gives it that by dividing some coarse partitions in two.

Interestingly, it is not necessarily worth increasing the resolution of the partition the system is in, nor is it necessarily worth splitting all partitions which have $J_{WC} = \infty$. Figure 8 shows a case in which the current state is in a partition which it will not help to split. This is because there is no path to a non-losing state from the current partition anyway (other losing partitions block us) and no matter how high we make its resolution our current partition will remain a loser.

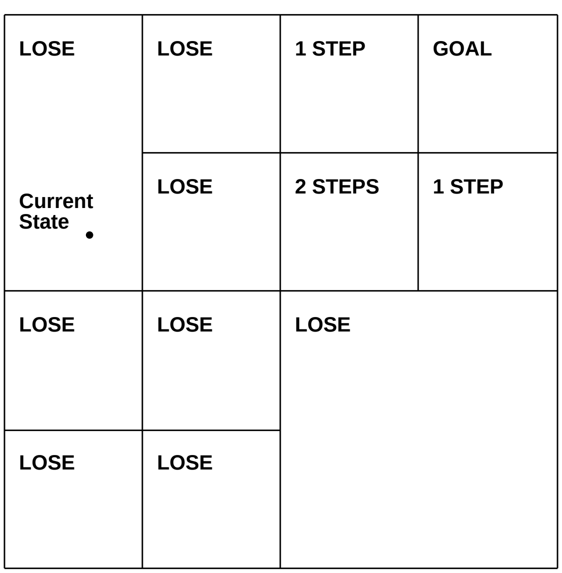


Figure 8: A 12-cell partitioning in which it will not help to split the partition containing the current state.

It is the partitions on the border between losing and not losing which should be split. Under the assumption that all paths through state space are continuous, and also assuming that a path to the goal actually exists, there must currently be a hole in one of the border partitions which has been missed by the over-coarseness. This motivates the final algorithm which we present.

The algorithm takes three inputs:

- The current real-valued system state, s.
- A partitioning of state-space, P.
- A database, D, of all empirically experienced (start, aimed-for, actual-outcome) triplets.

It returns two outputs: a new partitioning of state-space and a new database.

ALGORITHM (4): (PARTIGAME).

WHILE (s not in the goal partition)

1 Compute $J_{WC}()$ for each partition using minimax.

2 Run Algorithm (3) on s and P, retrieving the resulting additions to the database D, plus the new real-valued state s, and a success/failure signal.

3 If FAILURE was signaled

3.1 Let Q := All partitions in P for which $J_{WC} = \infty$.

3.2 Let Q' := Members of Q who have any non-losing neighbors.

3.3 Let Q'' := Q' and all non-losing neighbors of members of Q'.

3.4 Construct a new set of partitions from Q'', of twice the size, produced by splitting each partition in half along its longest axis. Call this new set R.

3.5 P := $P + R - Q''$

3.6 Recompute all new neighbor relations and delete those members of database D which contain a member of Q'' as a start point, an aim-for or an actual-outcome.

LOOP

3.5 Partigame Details

Initialization

Before the very first trial, parti-game is initialized as just two partitions: a goal partition covering the goal region, and one other large partition covering the rest of state-space. At that point, Algorithm (4) is called. Unless the system is very lucky, this trivial partitioning will not be adequate to reach the goal using the greedy controller. At the point when this is detected the initial, trivial partitioning will quickly start splitting.

Increasing the resolution

Notice that this algorithm increases the resolution at both sides of the win/lose border. This prevents enormous partitions from bordering tiny partitions. There could be other algorithms in which the partitions to split are chosen differently. The question of which alternative is best remains open for further investigation.

Planning and learning in parti-game

Partigame performs planning and learning simultaneously. Interestingly, these two components are of great help to each other. The learning consists of gathering data to build up the sets of known possible outcomes of transitions between partitions. This data is gathered by planning paths to "interesting" partitions and executing "interesting" actions. A partition seems interesting if, according to the optimistic assumption that anything we haven't tried will work (Equation 8 in Algorithm (3)), the partition lies on the shortest path to the goal.

The planning is helped by the learning because the computation and representation are concentrated on the parts of the state space which, according to the database of experiences, are most critical.

The goal partition

The goal partition is special. It never changes or gets split. The task is defined to be solved when the system enters any part of the goal partition. In the experimental diagrams in Section 4 it is the box marked "Goal".

When other partitions are split, each new partition has to recompute all the neighbors that it is next to. Any new partition which intersects the goal partition also includes the goal partition as one of its neighbors.

4 Experiments

All these experiments are broken into trials. On each trial the system is placed at the start state and the trial proceeds until the system enters the goal region.

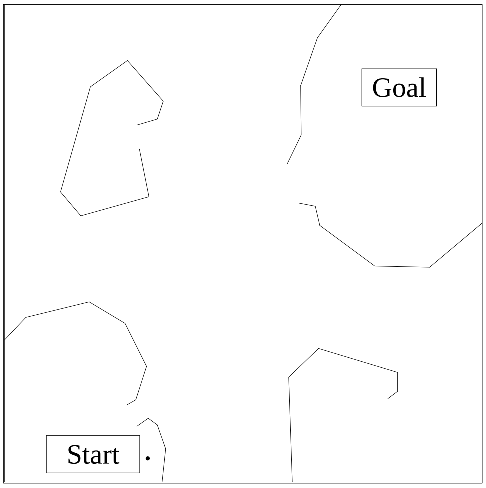


Figure 9: A two-dimensional maze problem. The point robot must find a path from start to goal without crossing any of the barrier lines. Remember that initially it does not know where any obstacles are, and must discover them by finding impassable states.

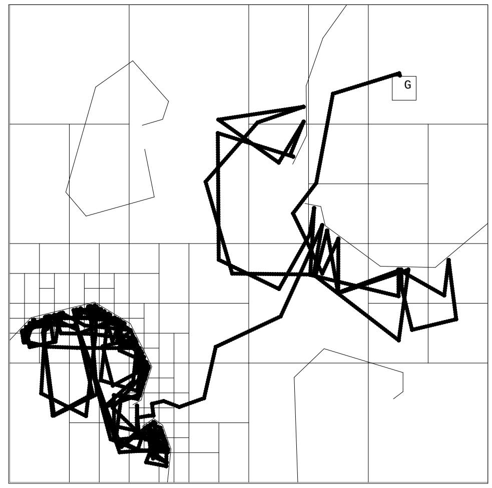


Figure 10: The path taken during the entire first trial. It begins with intense exploration to find a route out of the almost entirely enclosed start region. Having eventually reached a sufficiently high resolution, it discovers the gap and proceeds greedily towards the goal, only to be stopped by the goal's barrier region. The next barrier is traversed at a much lower resolution, mainly because the gap is larger.

4.1 Maze navigation

Figure 9 shows a two-dimensional continuous maze. Figure 10 shows the performance of the robot during the very first trial. Figure 11 shows the second trial, started from a slightly different position. The policy derived from the first trial gets us to the goal without further exploration. The trajectory has unnecessary bends. This is because the controller is discretized according to the current partitioning. If necessary, a local optimizer could be used to refine this trajectory[1].

The system does not explore unnecessary areas. The barrier in the top left remains at low resolution because the system has had no need to visit there. Figures 12 and 13 show what happens when we now start the system inside this barrier.

[1]Another method is to increase the resolution along the trajectory [Moore, 1991].

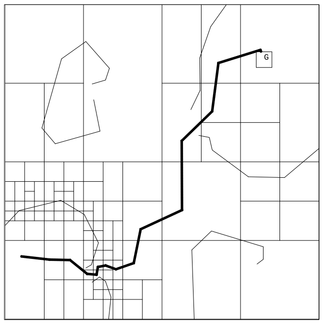


Figure 11: The second trial.

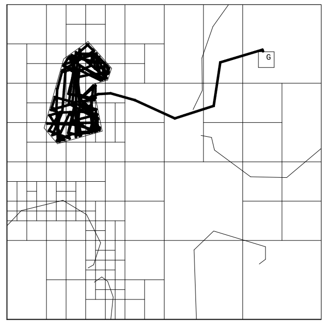


Figure 12: Starting inside the top left barrier.

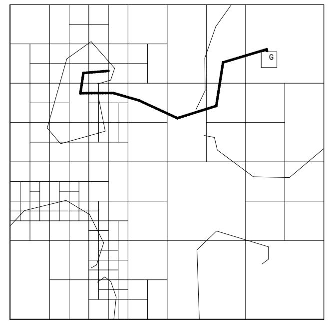


Figure 13: The trial after that.

4.2 Non-linear dynamics

Figure 14 depicts a frictionless puck on a bumpy surface. It can thrust left or right with a maximum thrust of ± 4 Newtons. Because of gravity, there is a region near the center of the hill at which the maximum rightward thrust is not strong enough to accelerate up the slope. Thus if the goal is at the top of the slope, a strategy which proceeded by greedily choosing actions to thrust towards the goal would get stuck.

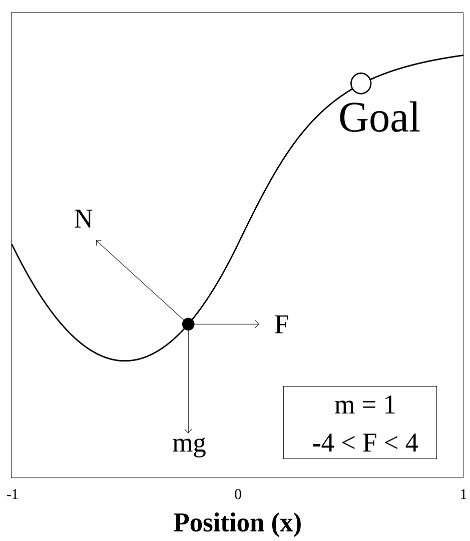


Figure 14: A frictionless puck acted on by gravity and a horizontal thruster. The puck must get to the goal as quickly as possible. There are bounds on the maximum thrust.

This is made clearer in Figure 15, a *phase space diagram*. The puck's state has two components, the position and velocity. The hairs show the next state of the puck if it were to thrust rightwards with the maximum legal force of 4 Newtons. Notice that at the center of state space, even when this thrust is applied, the puck velocity decreases and it eventually slides leftwards. The optimal

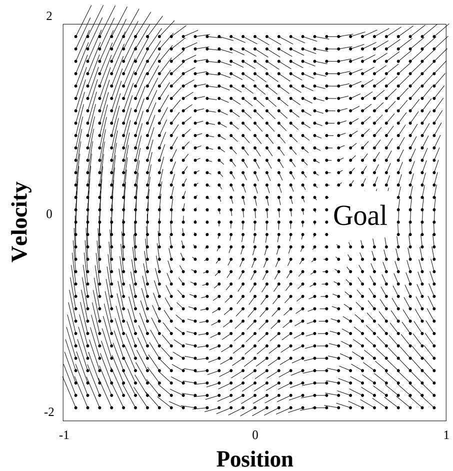


Figure 15: The state transition function for a puck which constantly thrusts right with maximum thrust.

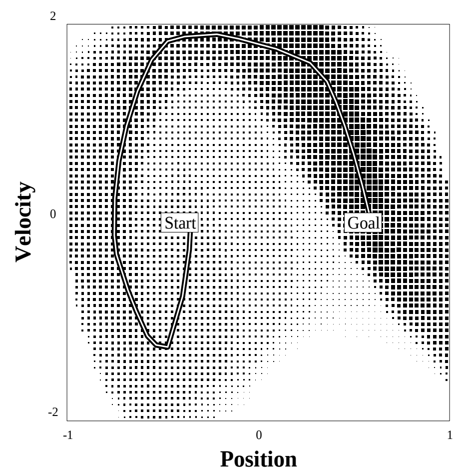


Figure 16: The "minimum-time" path from start to goal for the puck on the hill. The optimal value function is shown by the background dots. The shorter the time to goal, the larger the black dot. Notice the discontinuity at the escape velocity.

solution for the puck task, depicted in Figure 16, is to initially thrust away from the goal, gaining negative velocity, until it is on the far left of the diagram. Then it thrusts hard right, to build up sufficient energy to reach the top of the hill.

The local greedy controller which parti-game uses is bang-bang. To aim for a partition "north" in state space—a partition with greater velocity—it thrusts with the maximum permissable force of $+4N$. To aim for a lower velocity partition it thrusts with $-4N$. To aim for a "east" or "west" partition, the local controller merely controls its velocity to be equal to the velocity of the center of the destination partition. Notice that if the current partition's velocity is greater than zero it is hopeless to greedily aim for the partition on the left. It is also hopeless to aim at the partition on the right if the current partition has negative velocity. In the experiments below Parti-game is given this extra information. Forcing parti-game to learn this from experience approximately doubles the learning time.

Figure 17 shows the trajectory through state space during the very first learning trial, while it is exploring and developing its initial partitioning. Figure 18 shows the resulting partitioning and the subsequent trajectory[2]: on its second trial it has already learned the basic strategy of "begin

[2] Careful inspection of this diagram reveals that the trajectory changes direction not at the borders of cells but instead within cells. This is because the current implementation waits until it is well within a cell before applying the cell's recommended action.

by getting a negative velocity, moving backwards, and only then heading forward with full thrust". Figure 19 shows the interesting result of running many more trajectories, each starting at random parts of state space. Many partitions are created and refined, but only around the critical border in state space which serves as the escape velocity of the problem (also visible as the discontinuity in Figure 16). This high resolution line arises not out of any pre-programmed knowledge of the escape velocity but because the system does not need to increase the resolution of partitions which fail to intersect the escape velocity region.

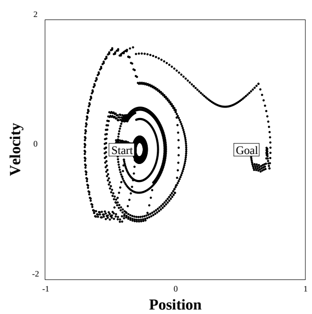


Figure 17: The trajectory of the very first trial, while the system performed its initial exploration of state space.

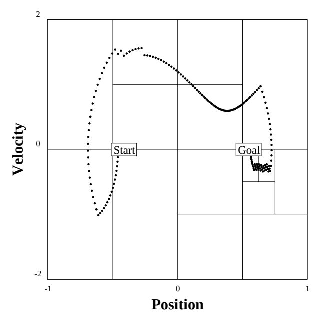


Figure 18: The trajectory and partitioning of the second trial.

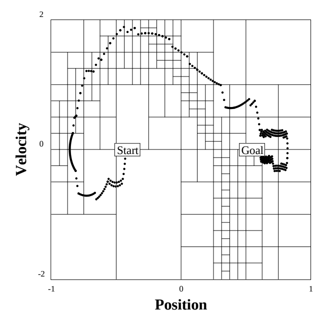


Figure 19: The partitioning after it has learned the task from 200 random start positions.

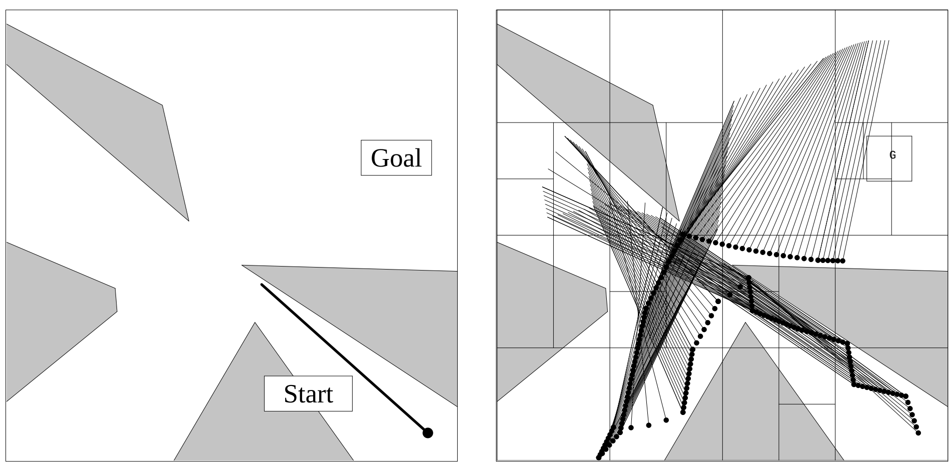
4.3 Higher dimensional state spaces

Figure 20 shows a three-dimensional state-space problem. If a standard grid were used, this would need an enormous number of states because the solution requires detailed maneuvers. Parti-game's total exploration took 18 times as much movement as one run of the final path obtained.

Figure 21 shows a four-dimensional problem in which a ball slides on a tray with steep edges. The goal is on the other side of a ridge. The maximum permissible force is low. Greedy strategies, or globally linear control rules, get stuck in limit cycles within a valley. The local greedy controller to navigate between adjacent partitions is bang-bang controller. Parti-game's solution runs to the far end of the tray, to build up enough velocity to make it over the ridge. The exploration-length versus final-path-length ratio is 24.

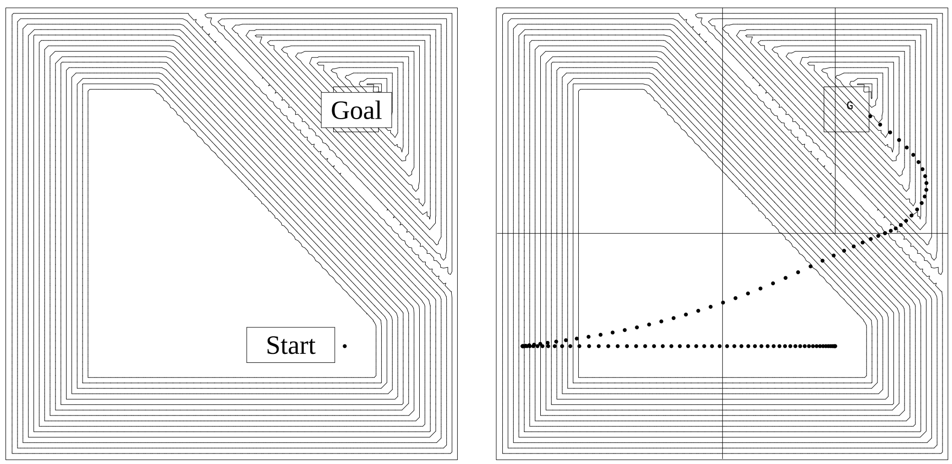
Figure 22 shows a 9-joint snake-like robot manipulator which must move to a specified configuration on the other side of a barrier. Again, no kinematics model or knowledge of obstacle locations are given: the system must learn these as it explores. It takes seven trials before converging on the solution shown, which requires about two minutes run-time on a SPARC-I workstation. The

Figure 20: A problem with a planar rod being guided past obstacles. The state-space is three-dimensional: two values specify the position of the rod's center, and the third specifies the rod's angle from the horizontal. The angle is constrained so that the pole's dotted end must always be below the other end. The pole's center may be moved a short distance (up to 1/40 of the diagram width) and its angle may be altered by up to 5 degrees, provided it does not hit a barrier in the process. Parti-game converged to the path shown below after two trials, with 18 times as many exploration steps as there are steps in the final path. The partitioning lines on the second diagram only show a two-dimensional slice of the full partitioning.



Trials	1	2	3	4	5	6	7	8	9	10
Steps	2975	189	187	no further						
Partitions	149	149	149	change						

Figure 21: A puck sliding over a hilly surface (hills shown by contours below: the surface is bowl shaped, with the start and goal states at the bottoms of distinct valleys). The state-space is four-dimensional: two position and two velocity variables. The controls consist of a force which may be applied in any direction, but with bounded magnitude. Convergence time was two trials, with 24 times as much exploration as there are steps in the final path.



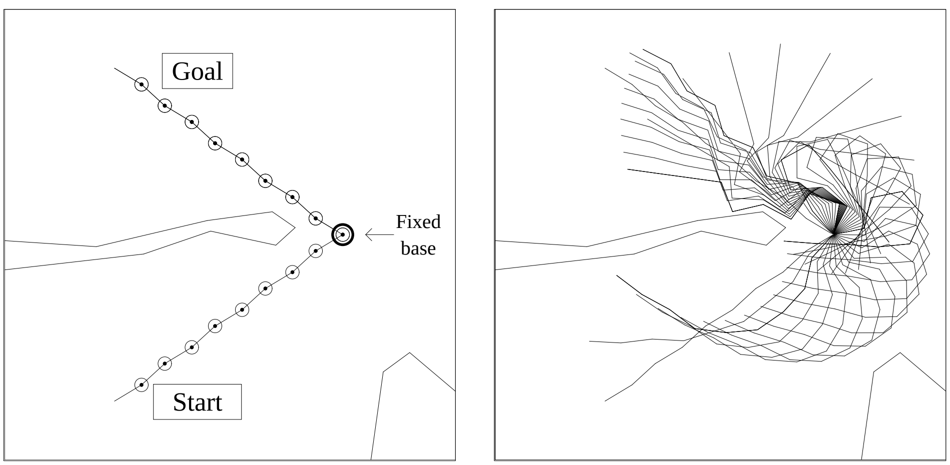
Trials	1	2	3	4	5	6	7	8	9	10
Steps	2609	115	no further							
Partitions	13	13	change							

exploration-length versus final-path-length ratio is 60. Interestingly, the final number of partitions is only 85. This compares very favorably with the 512 partitions which would be needed if the coarsest non-trivial uniform grid were used: $2 \times 2 \times \cdots \times 2$. Unsurprisingly, for the 9-joint snake, this 512 uniform grid is too coarse, and in experiments we performed with such a grid the system became stuck, eventually deciding the problem was impossible.

5 Related work

A few other researchers in Reinforcement Learning have attempted to overcome dimensionality problems by decompositions of state space. An early attempt was [Simons *et al.*, 1982] who attempted it for 3-degree-of-freedom force control. Their method gradually learned by recording cumulative statistics of performance in partitions. More recently, we produced a variable resolution dynamic programming method [Moore, 1991]. This enabled conventional dynamic programming to be performed in real valued multivariate state spaces where straightforward discretization would fall prey to the curse of dimensionality. This is another approach to partitioning state space but

Figure 22: A nine-degree-of-freedom planar robot must move from the shown start configuration to the goal. The joints are shown by small circles on the left-hand diagram which depicts two configurations of the arm: the start position and the goal position. The solution entails curling, rotating and then uncurling. It may not intersect with any of the barriers, the edge of the workspace, or itself. Convergence occurred after seven trials, with 60 times as much exploration as there are steps in the final path.



Trials	1	2	3	4	5	6	7	8	9	10
Steps	1090	430	353	330	739	200	52	no further		
Partitions	41	66	67	69	78	85	85	change		

has the drawback that, unlike parti-game, it requires a guess at an initially valid trajectory through state space. [Chapman and Kaelbling, 1991] proposed an interesting algorithm which used more sophisticated statistics to decide which attributes to split. Their objectives were very hard because they wished to avoid remembering transitions between cells and they did not assume continuous paths through state space, and so they obtained only limited empirical success.

In [Dayan and Hinton, 1993] a 2-dimensional hierarchical partitioning was used on a grid with 64 discrete squares, and [Kaelbling, 1993] gives another hierarchical algorithm. These references both attempt a different goal than parti-game: they try to accelerate Q-learning [Watkins, 1989] by providing it with a pre-programmed abstraction of the world. The abstraction, it is noted in both cases, may sometimes indeed lead to faster learning and can improve Q-learning if there are multiple goals in the problem. In contrast, parti-game is able to build its own abstraction using geometric reasoning and so learns more quickly (typically in fewer than ten trials and a few minutes of real time) and on significantly higher dimensional problems than have been attempted elsewhere. The price parti-game pays is that it is limited to geometric abstractions, whereas both Kaelbling's and Dayan's methods may eventually be applicable to other abstraction hierarchies.

Geometric Decompositions have also been used fairly extensively in Robot Motion Planning (e.g. [Brooks and Lozano-Perez, 1983, Kambhampati and Davis, 1986]), summarized in [Latombe, 1991]. The principal difference is that the Robot Motion Planning methods all assume that a model of the environment (typically in the form of a pre-programmed list of polygons) is supplied to the system in advance so that there is no learning or exploration capability. The experiments in [Brooks and Lozano-Perez, 1983] involve a 3-degree-of-freedom navigation problem and in [Kambhampati and Davis, 1986], a fairly difficult 2-dimensional maze.

Finally, some relation can be seen between parti-game and multigrid methods (e.g. [Hoppe, 1986]) used in numerical analysis to accelerate the convergence of solutions to partial differential equations. Multigrid methods typically increase the resolution of the grid everywhere, and like robot motion planning, do not learn: the correct system dynamics must be programmed in.

6 Discussion

6.1 Splitting

Given a partition we have decided to split, which axis should be split? Algorithm (4) states that we should split along the longest axis. This begs the question of where to split in the case of ties. The current algorithm resolves ties with a fixed ordering on axes, but we could be cleverer. In Figure 23 it is clear that a vertical split would be more useful than a horizontal split. This kind of intelligent split choice, which pays attention to the locations of outcomes, would not be difficult to incorporate.

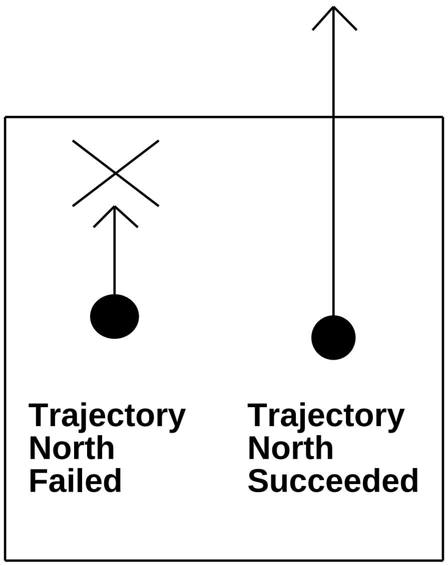


Figure 23: We have had two experiences of attempting to move North from two different points in the partition. Only one succeeded.

6.2 Not forgetting

When a partition is split, the outcomes of its children are initialized to be empty and so old data is forgotten. This is not desirable or necessary. Old trajectories could be retained and used to initialize the **OUTCOMES**() sets of those children within which earlier trajectory segments lay.

6.3 Learning the local greedy controllers

The parti-game algorithm requires that the user defines local greedy controllers. Is this not a large sacrifice of autonomy? We argue not: learning greedy controllers merely requires gathering enough local experience to form a local linear map of the low level system dynamics. This can be done with relative ease, both in a statistical and computational sense.

6.4 Dealing with an unknown goal state

There is no difficulty for parti-game in removing the assumption that the location of the goal state is known. Convergence will be considerably slowed down if it is not given, but this is not the fault of the algorithm. If there are D state variables and the goal is signaled when all state variables are simultaneously within $\pm\delta\%$ of an unknown goal value, then it is clear that an exploration of at least

$$\left(\frac{100}{2\delta}\right)^D \tag{9}$$

points on a grid in state-space are needed to ensure the goal is reached even once, whatever the learning algorithm.

A simple supplement to parti-game can be made to implement this kind of uniform exploration. It begins with a uniform grid partition with

$$\frac{100}{\delta} \tag{10}$$

breaks on each axis and encourages exploration by estimating the J_{WC} value of all unvisited partitions as zero.

6.5 Attaining Optimality

Parti-game is designed to find solutions to delayed reward control problems in reasonable time without needing help in the form of initial human-supplied trajectories. The algorithm works hard to find a solution but makes no attempt to optimize it. Empirically, all solutions found have been good. There are a number of kinds of suboptimality which parti-game will not produce. In the case of navigation, for example, parti-game cannot produce loops or meanders, as shown in Figure 24.

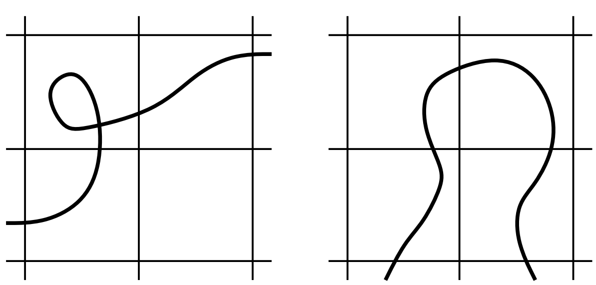


Figure 24: Partigame cannot produce either of these kinds of suboptimality. No loops, and no unblocked adjacent partitions which contain separate parts of a solution trajectory.

The lack of guaranteed optimality in parti-game is a concession to the fact that there is unlikely to be sufficient time in the lifetime of a reinforcement learning system to explore every possible solution. Future research may reveal ways to achieve weaker optimality guarantees:

- That the solution is locally optimal.
- A proof that even if the solution is not globally optimal, the global solution can be no better than factor K (in terms of cost units for the task being learned) over parti-game's solution.

Both these optimality statements will require extra assumptions about the state space. In the case of navigational problems, this can come in the form of geometric reasoning. In dynamics problems it will be by means of local linearizations within partitions, and subsequent Linear Quadratic Gaussian (LQG) local control design (see, for example, [Sage and White, 1977]).

It is also possible that other domains will be able to use similar reasoning by means of *admissible heuristics*: a classical method in AI for formally reasoning about the optimality of proposed solutions [Nilsson, 1971].

6.6 Multiple Goals

Because it builds an explicit model of all the possible state transitions between partitions, it is a trivial matter for parti-game to change to a new goal. We have performed a number of experiments (not reported here) that confirm this.

6.7 Stochastic Dynamics

This is the hardest issue for parti-game to cope with. If a given action in a given partition produces multiple results, how do we decide if this is due to inherent randomness or due to overly coarse partitions? In the latter case it will be helpful to increase the resolution and in the former case it will not.

The easiest case will be noise in the form of

$$\text{next-state} = f(\text{state}, \text{action}) + \text{noise-signal}() \tag{11}$$

An example is an environment which randomly jogs a mobile robot between each movement. We have performed some experiments with parti-game under this scenario (not reported here), and have not yet seen it get stuck even when quite substantial noise was added. In principal, though, any amount of noise could break the partigame algorithm—if trials were run indefinitely, eventually all of state space would become partitioned to unboundedly high resolution. An improvement to parti-game might use statistical tests which try to explain outcomes in terms of location within the partition. This might help, but further research is needed.

If the randomness is something which occasionally teleports the system to a random place (breaking the assumption of paths being continuous through state space), then partigame would probably need an entirely different splitting criterion. One possibility is a version of the "G" splitting rule of [Chapman and Kaelbling, 1991].

6.8 The Curse of Dimensionality

We finish by noting a promising sign involving a series of snake robot experiments with different numbers of links (but fixed total length). Intuitively, the problem should get easier with more links, but the curse of dimensionality would mean that (in the absence of prior knowledge) it becomes exponentially harder. This is borne out by the observation that random exploration with the three-link arm will stumble on the goal eventually, whereas the nine link robot cannot be expected to do so in tractable time. However, Figure 25 indicates that as the dimensionality rises, the amount of exploration (and hence computation) used by parti-game does not rise exponentially. It is conceivable (but not supported by further evidence in this paper) that real-world tasks may often have the same property: the complexity of the ultimate task remains roughly constant as the number of degrees of freedom increases. If so, this might be the Achilles' heel of the curse of dimensionality.

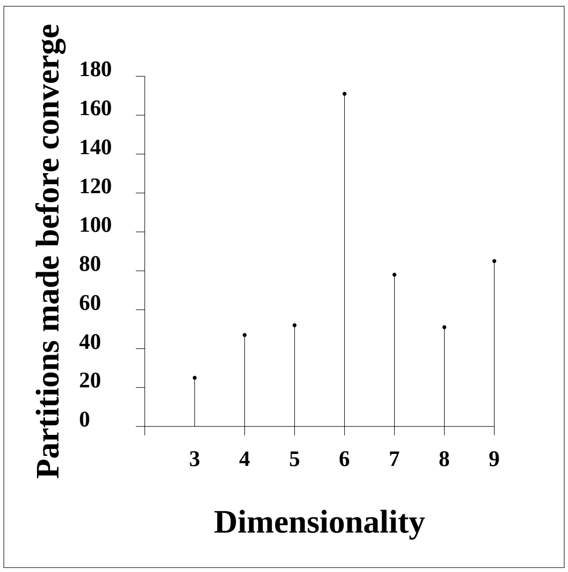


Figure 25: The number of partitions finally created against degrees of freedom for a set of snake-like robots. The partitionings built were all highly non-uniform, typically having maximum depth nodes of twice the dimensionality. The relation between exploration time and dimensionality (not shown) had a similar shape.

7 Conclusion

This paper began with the problems of coarse partitionings of state space. It then showed how worst-case assumptions can solve these problems, and very effectively identify partitions which need to have their resolutions increased. There are many interesting avenues arising from these ideas which remain open for further investigation.

References

[Barto *et al.*, 1983] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike Adaptive elements that that can learn difficult Control Problems. *IEEE Trans. on Systems Man and Cybernetics*, 13(5):835–846, 1983.

[Barto *et al.*, 1991] A. G. Barto, S. J. Bradtke, and S. P. Singh. Real-time Learning and Control using Asynchronous Dynamic Programming. Technical Report 91-57, University of Massachusetts at Amherst, August 1991.

[Bellman, 1957] R. E. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[Bertsekas and Tsitsiklis, 1989] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation.* Prentice Hall, 1989.

[Brooks and Lozano-Perez, 1983] R. A. Brooks and T. Lozano-Perez. A Subdivision Algorithm in Configuration Space for Findpath with rotation. In *Proceedings of the 8th International Conference on Artifical Intelligence*, 1983.

[Chapman and Kaelbling, 1991] D. Chapman and L. P. Kaelbling. Learning from Delayed Reinforcement In a Complex Domain. Technical Report, Teleos Research, 1991.

[Dayan and Hinton, 1993] P. Dayan and G. E. Hinton. Feudal Reinforcement Learning. In S. J. Hanson, J. D Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5.* Morgan Kaufmann, 1993.

[Hoppe, 1986] R. H. W. Hoppe. Multi-Grid Methods for Hamilton-Jacobi-Bellman Equations. *Numerical Mathematics*, 49, 1986.

[Kaelbling, 1993] L. Kaelbling. Hierarchicial Learning in Stochastic Domains: Preliminary Results. In *Machine Learning: Proceedings of the Tenth International Workshop.* Morgan Kaufman, June 1993.

[Kambhampati and Davis, 1986] Subbarao Kambhampati and Larry S. Davis. Multiresolution Path Planning for Mobile Robots. *IEEE Journal of Robotics and Automation, Vol. RA-2, No. 3*, 2(3), 1986.

[Knuth, 1973] D. E. Knuth. *Sorting and Searching.* Addison Wesley, 1973.

[Latombe, 1991] J. Latombe. *Robot Motion Planning.* Kluwer, 1991.

[Michie and Chambers, 1968] D. Michie and R. A. Chambers. BOXES: An Experiment in Adaptive Control. In E. Dale and D. Michie, editors, *Machine Intelligence 2.* Oliver and Boyd, 1968.

[Moore and Atkeson, 1993] A. W. Moore and C. G. Atkeson. Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time. *Machine Learning*, 13, 1993.

[Moore, 1991] A. W. Moore. Variable Resolution Dynamic Programming: Efficiently Learning Action Maps in Multivariate Real-valued State-spaces. In L. Birnbaum and G. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop*. Morgan Kaufman, June 1991.

[Nilsson, 1971] N. J. Nilsson. *Problem-solving Methods in Artificial Intelligence.* McGraw Hill, 1971.

[Peng and Williams, 1993] J. Peng and R. J. Williams. Efficient Learning and Planning Within the Dyna Framework. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. MIT Press, 1993.

[Sage and White, 1977] A. P. Sage and C. C. White. *Optimum Systems Control.* Prentice Hall, 1977.

[Simons *et al.*, 1982] J. Simons, H. Van Brussel, J. De Schutter, and J. Verhaert. A Self-Learning Automaton with Variable Resolution for High Precision Assembly by Industrial Robots. *IEEE Trans. on Automatic Control*, 27(5):1109–1113, October 1982.

[Sutton, 1984] R. S. Sutton. Temporal Credit Assignment in Reinforcement Learning. Phd. thesis, University of Massachusetts, Amherst, 1984.

[Sutton, 1990a] R. S. Sutton. Integrated Architecture for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In *Proceedings of the 7th International Conference on Machine Learning.* Morgan Kaufman, June 1990.

[Sutton, 1990b] R. S. Sutton. Integrated Architecture for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In *Proceedings of the 7th International Conference on Machine Learning.* Morgan Kaufman, June 1990.

[Watkins, 1989] C. J. C. H. Watkins. Learning from Delayed Rewards. PhD. Thesis, King's College, University of Cambridge, May 1989.