Machine Learning, 21, 199–233 (1995)

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

ANDREW W. MOORE awm@cs.cmu.edu
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

CHRISTOPHER G. ATKESON cga@cc.gatech.edu
Georgia Institute of Technology, College of Computing, Atlanta, GA 30332

Editor: Andrew G. Barto

Abstract. Parti-game is a new algorithm for learning feasible trajectories to goal regions in high dimensional continuous state-spaces. In high dimensions it is essential that neither planning nor exploration occurs uniformly over a state-space. 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 paths or trajectories to goal regions in high dimensional spaces. 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 ten trials and a few minutes.

Keywords: Reinforcement Learning, Curse of Dimensionality, Learning Control, Robotics, *k*d-trees

1. Reinforcement Learning

Reinforcement learning (Michie & Chambers, 1968), (Sutton, 1984), (Watkins, 1989), (Barto, et al., 1994) 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. The curse of dimensionality is also a problem in other areas of artificial intelligence, such as planning and supervised learning.

Much work has been performed with discrete state-spaces: in particular a class of Markov decision tasks known as grid worlds (Watkins, 1989), (Sutton, 1990). 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 & Chambers, 1968), (Barto, et al., 1983)), the naive grid approach has a number of dangers which will be detailed in this paper.

This paper studies the pitfalls of discretization during reinforcement learning and then introduces the parti-game algorithm. Some earlier work (Simons, et al., 1982), (Moore, 1991), (Chapman & Kaelbling, 1991), (Dayan & 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 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 possible paths 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 region, 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.

Constraints 5 and 6 mark a particularly large departure from the assumptions usually made in the reinforcement learning literature. A more common formulation, especially in discrete spaces, is the optimization of the expected long term sum of state dependent rewards. The state transitions are typically stochastic. This more conventional reinforcement learning formalism is described in many references such as (Barto, et al., 1983), (Watkins, 1989), (Kaelbling, 1990), (Barto, et al., 1994), (Moore & Atkeson, 1993). Parti-game is restricted to a smaller class of tasks, but within that class is designed to attack reinforcement learning problems of much higher dimensionality than previous algorithms. It is hoped that future versions of parti-game will be applicable to stochastic systems and arbitrary reward functions.

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 algorithm learns a controller from a start region to a goal region in a continuous state-space. We now give four increasingly effective algorithms that 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 cells, and generalizes this to the assumption of starting randomly within a cell. 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 cells*

A *partitioning* of a continuous state-space is a finite set of disjoint regions, the union of which covers the whole of state-space. We will call the regions cells, and will label them with integers $1, 2, \ldots N$. Throughout this paper we will assume the cells 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 cells. 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 cell is a discrete entity, and Figure 1 is broken into six cells with identifiers $1 \ldots 6$. Each real-valued state is in one cell and each cell contains a continuous set of real-valued states. Define NEIGHS(i) as the set of cells which are adjacent to i. In Figure 1, NEIGHS$(1) = \{2, 4\}$.

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

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

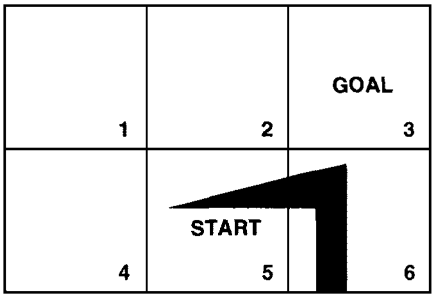


Figure 1. A two-dimensional continuous maze with one barrier: the black polygonal region near the bottom right. State-space has been discretized into six square cells.

The test for sticking can simply be implemented as a test to see if the system has not exited the cell 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 & Tsitsiklis, 1989) in which the discrete MDT states correspond to cells. Actions correspond to neighbors thus: action k in cell i corresponds to starting at the center of cell i and greedily aiming at the center of cell k.

ALGORITHM (1).

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

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

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

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

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

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

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

This simple algorithm has immediate drawbacks. It will minimize the number of cells to the goal, not the real distance. And it can easily find impossible solutions or fail to find

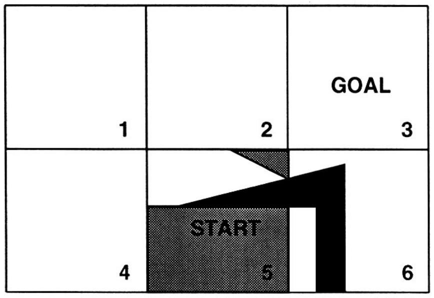


Figure 2. Approximately 65% of the starting states (those in the shaded region) in cell 5 are such that they will enter cell 6 if we aim for the center of 6. Thus $p^6_{56} = 0.65$.

valid solutions. As an example of the former, in Figure 1, Algorithm (1) would find the solution path 5 → 6 → 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 cells begin at the center of the source cell. Suppose we produce a stochastic Markov decision task. Let p^k_{ij} be an approximation of the probability of transition to cell 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 cell j from a uniformly randomly chosen legal start point in cell i. The dynamic programming step of the previous algorithm is altered so that it now solves the stochastic MDT:

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

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 → 6 → 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 → 4 → 1 → 2 → 3.

Other variants of this approximation by a stochastic system 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 cell, i, the neighbors that can be successfully reached depend on more than "i"; they also depend on the current location within i.

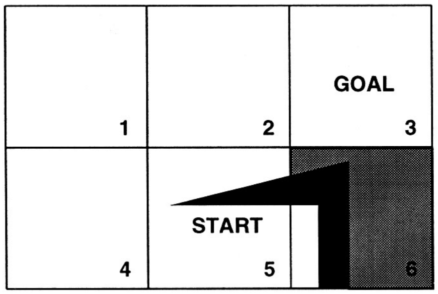


Figure 3. In a similar fashion to Figure 2, $p_{63}^{3} = 0.91$.

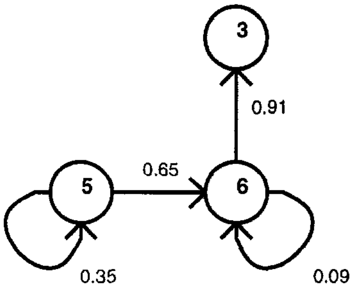


Figure 4. The cell transition probabilities if we follow the 5 → 6 → 3 policy according to the assumptions in the text.

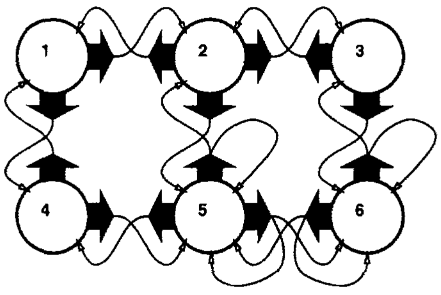


Figure 5. The symbolic representation of the original problem as cells (circles), actions (solid arrows) and outcomes (thin arrows).

3.2. *Algorithm (2): Assuming the worst case*

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

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

In Figure 5, the actions are denoted by black solid arrows and the outcomes by the thin lines. For example, cell 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 that an imaginary adversary will be allowed to make, seen in Algorithm (2).

ALGORITHM (2).

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

Solve the following set of minimax equations:

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

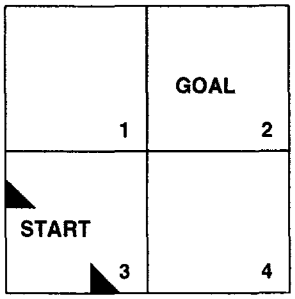


Figure 6. Cell 3 is scored as losing because if it aims for 1 the adversary can place it below the upper triangular block and if it aims for 4 the adversary can place it to the left of the lower triangular block.

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

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 $J_{WC}(i) = n$ then we can be sure that if we follow its policy we will indeed take n or fewer cell transitions to get to the goal starting from cell i. The trivial inductive proof is omitted.

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

It should be observed just how pessimistic the algorithm is. In the almost entirely empty maze of Figure 6 the start cell will be considered a losing cell. So although the minimax assumption guarantees 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 cell 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 cells.

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 **OUTCOMES**(i, j) for each cell i and neighbor j can be obtained empirically. Whenever an outcome given by **OUTCOMES**(i, j) is altered, the game is solved with the new outcomes set. We still assume that the location of the cell containing the goal is known.

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 cell transitions in the lifetime of the system. This is a set of triplets:

(starting in i_0, I aimed for j_0 and actually arrived in k_0)
(starting in i_1, I aimed for j_1 and actually arrived in k_1)
⋮

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 cell $i$ and each neighbor $j \in$ NEIGHS$(i)$ thus:
    • If there exists some $k'$ for which $(i, j, k') \in D$ then:
        OUTCOMES$(i, j) = \{k \mid (i, j, k) \in D\}$        (7)
    • Else, use the optimistic assumption in the absence of experience:
        OUTCOMES$(i, j) = \{j\}$        (8)
2 Compute $J_{WC}()$ for each cell using minimax.
3 Let $i$ := the cell containing the current real-valued state $s$.
4 If $i$ = GOAL then exit, signaling SUCCESS.
5 If $J_{WC}(i) = \infty$ then exit, signaling FAILURE.
```

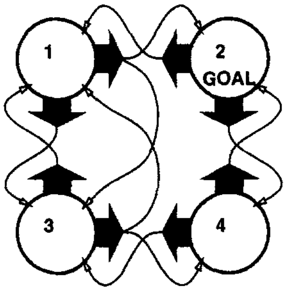


Figure 7. Cells 1 and 3 are losers because the adversary can force a permanent loop between them.

```
6 Else
    6.1 Let $j := \underset{j' \in \mathbf{NEIGHS}(i)}{\mathbf{argmin}} \; \underset{k \in \mathbf{OUTCOMES}(i,j')}{\mathbf{max}} J_{WC}(k)$.
    6.2 WHILE ( not stuck and $s$ is still in cell $i$ )
        6.2.1 Actuate local controller aiming at $j$.
        6.2.2 $s :=$ new real-valued state.
    6.3 Let $i_{\text{new}} :=$ the identifier of the cell containing s.
    6.4 $D := D \cup \{(i, j, i_{\text{new}})\}$
LOOP
```

Step 6.1 computes which neighboring cell is best to aim at, subject to the assumption that the adversary will place us in the worst previously-observed outcome.

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, 1990), (Peng & Williams, 1993), (Moore & Atkeson, 1993), (Barto, et al., 1994).

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

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.

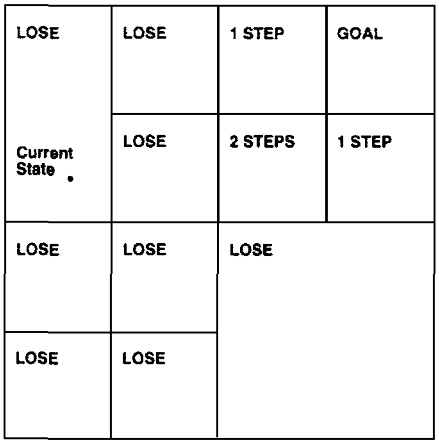


Figure 8. A 12-cell partitioning in which it will not help to split the cell containing the current state.

3.4. Algorithm (4): Varying the Resolution

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

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

It is the cells on the border between losing regions and non-losing regions 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 cells which has been missed by the over-coarseness. This motivates the final algorithm that 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): (Parti-game).

```
WHILE ( $s$ not in the goal cell )
```

```
  1 Run Algorithm (3) on $s$ and $P$. Algorithm (3) returns the resulting additions to the
    database $D$, plus the new real-valued state $s$, and a success/failure signal.
  2 If FAILURE was signaled
      2.1 Let $Q$ := All cells in $P$ for which $J_{WC} = \infty$.
      2.2 Let $Q'$ := Members of $Q$ who have any non-losing neighbors.
      2.3 Let $Q''$ := $Q'$ and all non-losing neighbors of members of $Q'$.
      2.4 Construct a new set of cells from $Q''$, of twice the size, produced by splitting
          each cell in half along its longest axis. Call this new set $R$.
      2.5 $P := P + R - Q''$
      2.6 Recompute all new neighbor relations and delete those members of database
          $D$ that contain a member of $Q''$ as a start point, an aim-for, or an actual-outcome.
LOOP
```

3.5. *Parti-game Details*

Initialization
Before the very first trial, parti-game is initialized as just two cells: a goal cell covering the goal region, and one other large cell 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 cells from bordering tiny cells. There could be other algorithms in which the cells to split are chosen differently. The question of which strategy is best remains open for further investigation.

Planning and learning in parti-game
Parti-game performs planning and learning simultaneously.

- **Planning** consists of taking the partitioning that has been learned, and all the cell transitions that have been experienced, and then computing the shortest path to the goal cell.

- **Learning** consists of gathering cell transition data as the system physically moves around its state-space. Learning also involves adapting the representation of the state-space by splitting cells.

Interestingly, these two components are of great help to each other.

- **Learning is helped by planning** because the usefulness of the data gathered is greater than it would be without planning (if, for example, data was gathered by choosing random actions). The planner always computes the shortest path to the goal, subject to the assumption that any cell transitions not yet attempted will work. This means that the learning is always highly relevant to the task: data is gathered about the very cell transitions that are necessary to achieve the shortest possible path according to the current output of the planner. Planning also identifies when cells are losers, which is the engine behind the cell splitting mechanism.
- **Planning is helped by learning** because only the actual empirical experiences need to be considered. Compare this to a conventional robot motion planning formalism in which a geometric model of the configuration space must be first represented and then reasoned with. Instead our planning only needs to look at an empirical sample of real world transitions, meaning less computation if the models are complex, no requirement to accurately model the world, and no need to plan around low probability contingencies (such as the small blocks in Figure 6) unless they actually occur. Planning is also helped by learning because the result of learning (the variable resolution partitioning) changes the representation of the problem.

The goal cell

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

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

4. Experiments

This section evaluates parti-game empirically by means of a number of simulated learning tasks. All the experiments are broken into trials. On each trial the system is placed in an initial state and the trial proceeds until the system enters the goal region. The details of each experiment can be found in the Appendix.

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].

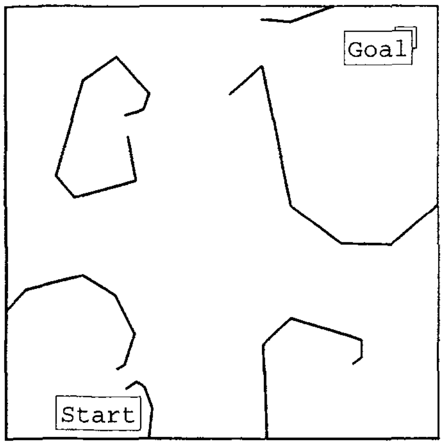


Figure 9. A two-dimensional maze problem. The point robot must find a path from start to goal without crossing any of the barrier lines. Remember that initially it does not know where any obstacles are, and must discover them by finding impassable states.

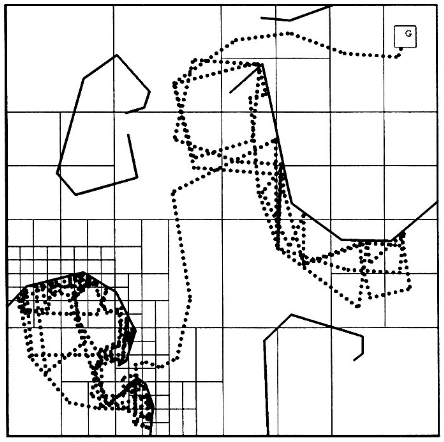


Figure 10. The path taken during the entire first trial. It begins with intense exploration to find a route out of the almost entirely enclosed start region. Having eventually reached a sufficiently high resolution, it discovers the gap and proceeds greedily towards the goal, only to be stopped by the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

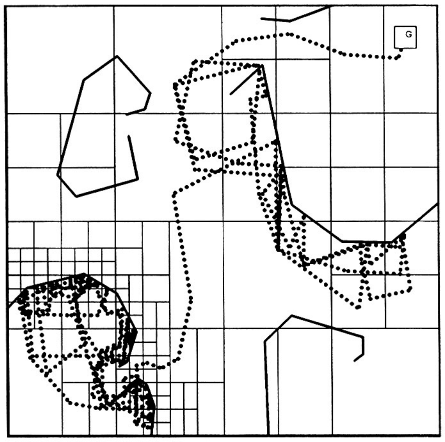


Figure 11. The second trial.

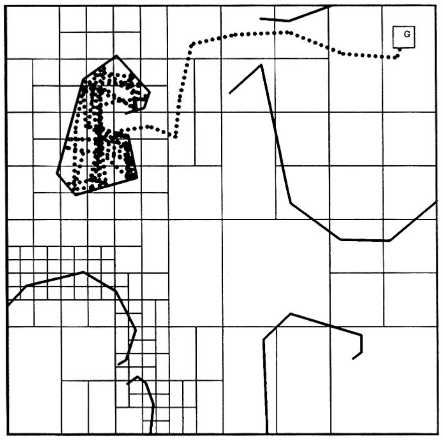


Figure 12. Starting inside the top left barrier.

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.

4.1.1. Comparison with other Algorithms

In an attempt to compare the parti-game algorithm with other reinforcement learning methods, we discretized the maze of Figure 9 and applied two discrete learning algorithms: Q-learning (Watkins, 1989) and prioritized sweeping (Moore & Atkeson, 1993),

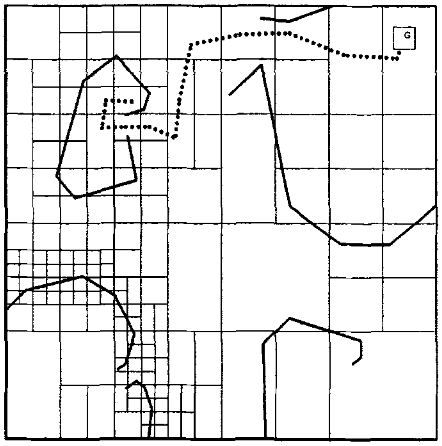


Figure 13. The trial after that.

(Peng & Williams, 1993). The discretization, shown in Figure 14, was at the coarsest resolution (50 × 50) which still permitted a path from start to goal. To prevent the problems caused by the non-Markovian nature of coarse discretizations (described in Section 3) the state transitions were also quantized: moves begin and end at the center of cells, and there are four actions: North, East, South and West. State transitions are deterministic.

In the first experiments, the discrete algorithms were not told the location of the goal. The exploration mechanism was "optimism in the face of uncertainty" (Kaelbling, 1990), (Sutton, 1990) in which any unvisited state-action pair was assumed to be zero steps from the goal. In a deterministic problem, this strategy is guaranteed to find the optimal path (Koenig & Simmons, 1993). The deterministic assumption also permitted Q-learning to use a learning rate $\alpha = 1$. Prioritized sweeping was allowed 200 backups per transition.

The results are shown in Table 1. Parti-game has considerably fewer steps for exploration and fewer backups than either of the other algorithms. Those algorithms had a disadvantage: parti-game was told the location of the goal. They were not. To provide a more equal comparison, both discrete algorithms were run with prior knowledge of the goal location. This was implemented by setting the default cost-to-goal value of any unexplored state transition as the Manhattan distance $| x - x_{\text{goal}} | + | y - y_{\text{goal}} |$ from the current state to the goal. Thus, initial exploration is biased towards the goal in a similar manner to the expansion of nodes in an A^* search.

Figure 15 shows the set of states visited before convergence when prioritized sweeping was used. Only about 35% of the states needed to be visited.

As Table 1 shows, performance (measured both as steps and backups) improved considerably when the discrete algorithms were told the location of the goal. In both cases however, the number of physical steps needed for exploration still exceeded that of parti-game. The total computational effort before convergence, measured by number of

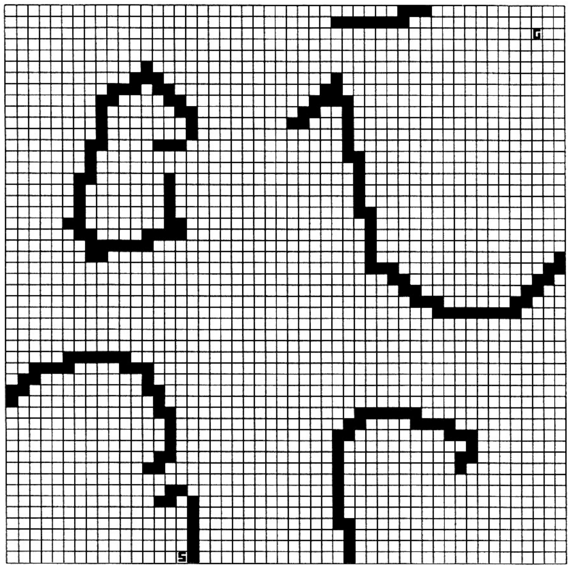


Figure 14. A 50 × 50 quantization of the maze in Figure 9.

backups, was less for Q-learning than for parti-game. If computation cost were the only criterion, and physical movement of the robot was ignored, this Q-learning implementation would therefore be more desirable than parti-game. However, it should be noted that if physical movement is ignored then classic search algorithms (such as A^*, which requires only 857 backups) greatly outperform all reinforcement learning algorithms, including Q-learning, on the same discretized maze.

This comparison provides an informal measure of the strengths and weaknesses of parti-game relative to some discrete reinforcement learning algorithms. It should be noted, though, that many domain-dependent factors prevent general comparative conclusions to be drawn. These include:

- Parti-game was not attempting to find an optimal solution but the discrete methods were[2].

- To help the discrete algorithms, the level of quantization was chosen carefully to be as coarse as possible without blocking possible paths to goal. In general, the knowledge of the correct level of quantization is not known. An advantage of Parti-game is that it creates its own discretization.

- The discrete algorithms are applicable to non-deterministic problems and arbitrary reward functions (though they may take longer to converge in these cases). The current version of parti-game is only applicable to deterministic problems specified by a goal region.

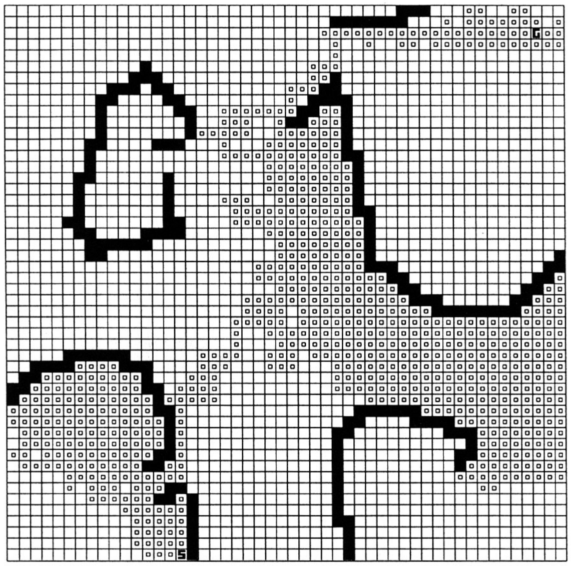


Figure 15. The set of all states visited by Prioritized Sweeping while it was exploring for the shortest route to the goal, the location of which it was told in advance.

Table 1. Comparing parti-game with four discrete reinforcement learning algorithms. The "Steps" column shows the number of steps of exploration before the algorithm had converged on a path from start to goal. One step is one fiftieth the width of the maze (In the continuous simulation, parti-game's step sizes are one two-hundredth the width of the maze, and the total number of its physical steps, 3816, has been divided by 4 to compensate).

Algorithm	Physical Steps before convergence	Backups before convergence
Parti-game	954	20,682
Q, goal unknown	186,140	186,140
PriSweep, goal unknown	11,416	340,680
Q, goal known	13,526	13,526
PriSweep, goal known	4,362	24,879

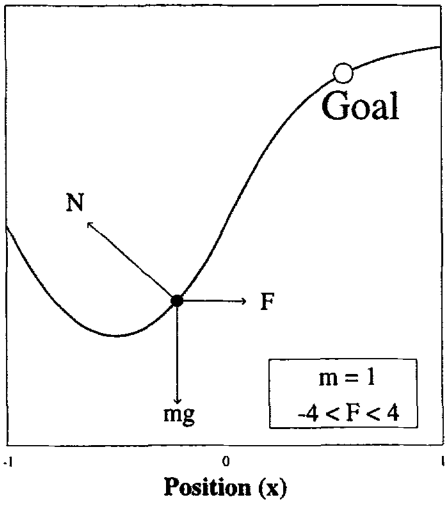


Figure 16. A frictionless puck acted on by gravity and a horizontal thruster. The puck must get to the goal as quickly as possible. There are bounds on the maximum thrust.

- Most importantly, above two dimensions, the advantage of parti-game over straightforward discretization is much more substantial.

4.2. Non-linear dynamics

Figure 16 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 insufficient to accelerate up the slope. Thus if the goal is at the top of the slope, a strategy that proceeded by greedily choosing actions to thrust towards the goal would get stuck.

This is made clearer in Figure 17, 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 solution for the puck task, depicted in Figure 18, 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 cell "north" in state-space—a cell with greater velocity—it thrusts with the maximum permissible force of $+4N$. To aim for a lower velocity cell it thrusts with $-4N$. To aim for an "east" or "west" cell, the local controller merely controls its velocity (using a

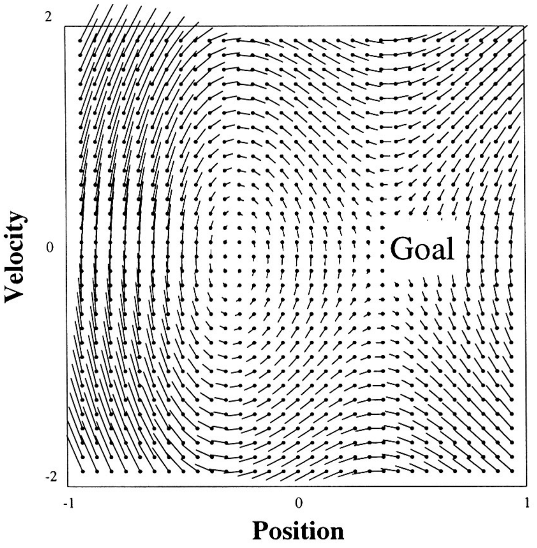


Figure 17. The state transition function for a puck that constantly thrusts right with maximum thrust.

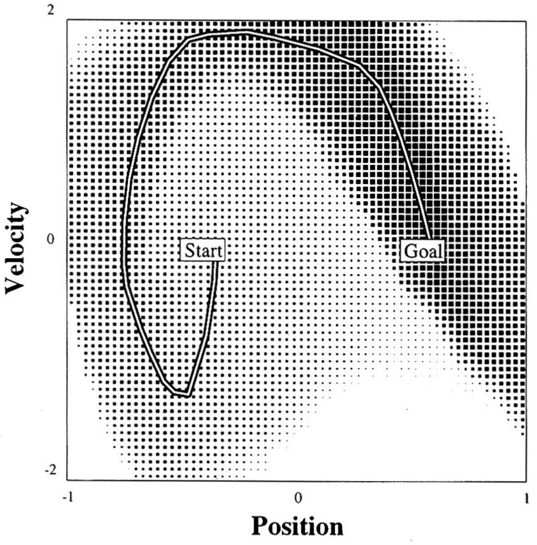


Figure 18. The minimum-time path from start to goal for the puck on the hill. The optimal value function is shown by the background dots. The shorter the time to goal, the larger the black dot. Notice the discontinuity at the escape velocity.

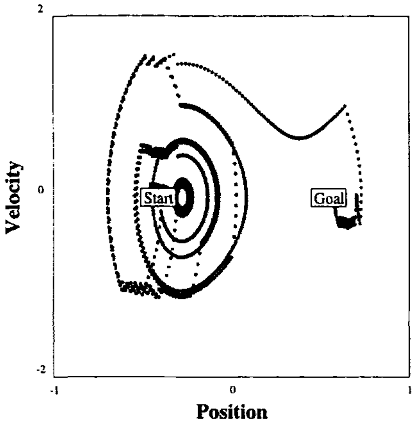


Figure 19. The trajectory of the very first trial, while the system performed its initial exploration of state-space.

trivial linear controller) to be equal to the velocity of the center of the destination cell. Notice that if the velocity is greater than zero in all parts of a cell it is hopeless to greedily aim for the cell on the left. It is also hopeless to aim at the cell on the right if the current cell 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 19 shows the trajectory through state-space during the very first learning trial, while it is exploring and developing its initial partitioning. Figure 20 shows the resulting partitioning and the subsequent trajectory[3]: on its second trial it has already learned the basic strategy of "begin by getting a negative velocity, moving backwards, and only then heading forward with full thrust." Figure 21 shows the interesting result of running many more trajectories, each starting at random parts of state-space. Many cells 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 18). 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 cells which fail to intersect the escape velocity region.

4.3. Higher dimensional state-spaces

Figure 22 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 23 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.

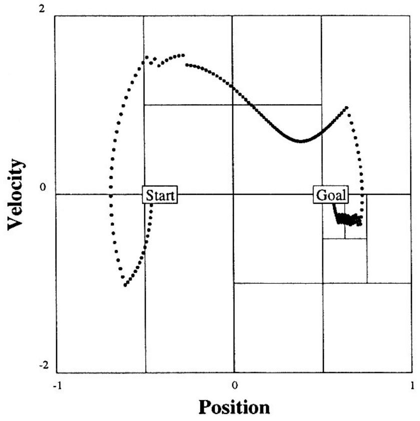


Figure 20. The trajectory and partitioning of the second trial.

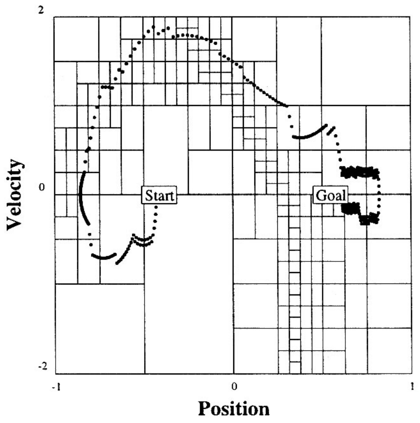
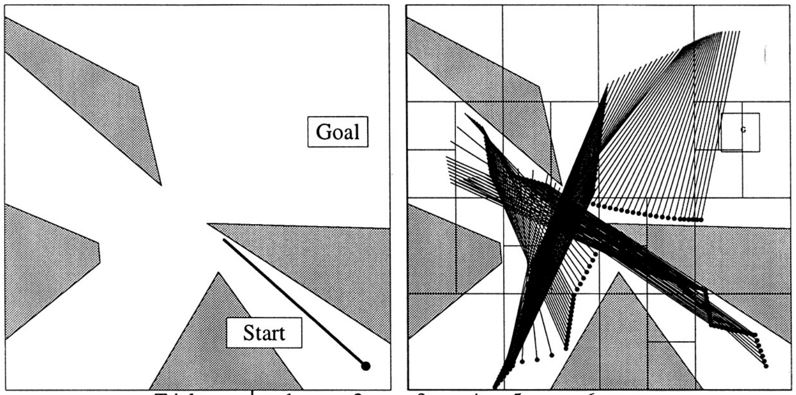


Figure 21. The partitioning after it has learned the task from 200 random start positions.



Trials	1	2	3	4	5	6
Steps	2975	189	187	No further change		
Partitions	149	149	149	No further change		

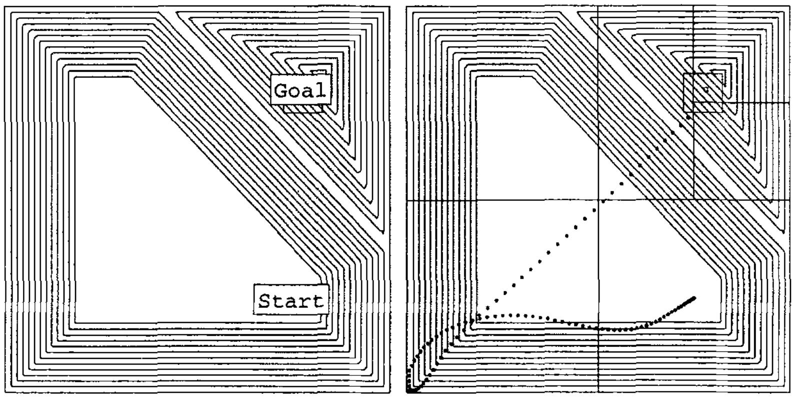
Figure 22. A problem with a planar rod being guided past obstacles. The state-space is three-dimensional: two values specify the position of the rod's center, and the third specifies the rod's angle from the horizontal. The angle is constrained so that the pole's dotted end must always be below the other 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 $\pi/20$ radians, provided it does not hit a barrier in the process. Parti-game converged to the path shown in the right-hand panel after two trials, with 18 times as many exploration steps as there are steps in the final path. The partitioning lines on the right-hand panel only show a two-dimensional slice of the full partitioning.

Greedy strategies, or globally linear control rules, get stuck in limit cycles within a valley. The local greedy controller to navigate between adjacent cells is a 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 24 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 exploration-length versus final-path-length ratio is 60. Interestingly, the final number of cells is only 85. This compares very favorably with the 512 cells that 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 insoluble.

4.4. Comparison to uniform partitioning

In the preceding experiments it is interesting to ask to what extent the performance is due to the adaptive partitioning method. We ran further experiments to find how quickly



Trials	1	2	3	4	5	6
Steps	2040	605	987	127	No further change	
Partitions	15	15	20	20	No further change	

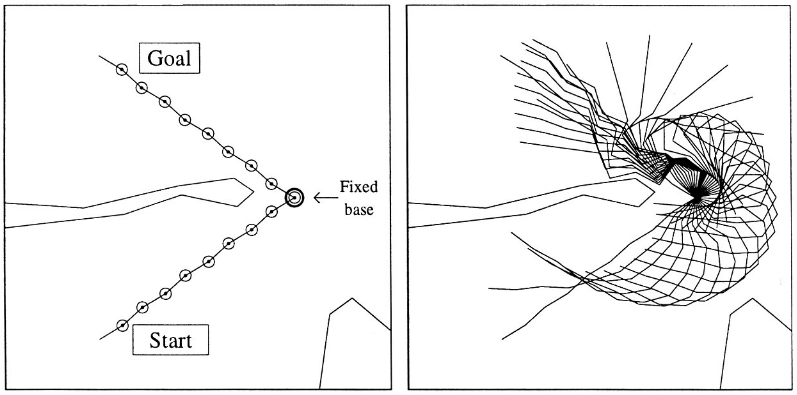
Figure 23. A puck sliding over a hilly surface (hills shown by contours: the surface is bowl shaped, with the start and goal states at the bottoms of distinct valleys). The state-space is four-dimensional: two position and two velocity variables. The controls consist of a force which may be applied in 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 (shown in the right-hand panel).

Table 2. Summary of parti-game's performance on the experiments within the preceding sections. Also shown is the performance of the coarsest uniform quantization of state-space for which parti-game needs to split no cells.

	Regular Parti-game			Uniform Parti-game		
	Steps before converged	Backups before converged	Cells when converged	Steps before converged	Backups before converged	Cells in quantization
Point in maze	3,816	20,682	119	4,205	198,032	1,024
Puck on hill	1,493	353	14	9,895	123,520	256
Rod in maze	3,164	129,981	149	8,594	13.1 Million	8,192
Slider on 2-d surface	3,632	2,466	20	9,337	137,996	256
9-joint snake	3,142	22,236	85	Memory exhausted (> 2^{18} cells)		

parti-game converged on non variable resolution grids. A convenient way to test this is to run a uniform version of parti-game in which, whenever the system is in a losing state, the resolution is doubled in every cell in the entire partitioning. This enables us to see the extent to which the variable resolution component of parti-game is responsible for reducing the exploration and computation needed for convergence.

Table 2 shows that the amount of exploration (steps before convergence) was increased by a factor between 1.1 and 6.5 for the uniform version of parti-game. The factor differ-



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

Figure 24. 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 panel which depicts two configurations of the arm: the start position and the goal position. The solution (see right-hand panel) entails curling, rotating and then uncurling. The robot 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.

ence was even greater for computational and memory requirements. In all cases, more than ten times the number of backups and more than ten times the number of cells were needed by the uniform methods. In the three-dimensional and nine-dimensional state-spaces these factors were particularly high—both measures were more than a hundred times worse for the uniform parti-game.

5. Related work

A few other researchers in Reinforcement Learning have attempted to overcome dimensionality problems by decompositions of state-space. An early example was (Simons, et al., 1982) who applied it to 3-degree-of-freedom force control. Their method gradually learned by recording cumulative statistics of performance in cells. 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 has the drawback that, unlike parti-game, it requires a guess at an initially valid trajectory through state-space. (Chapman & 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 & 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 & Lozano-Perez, 1983), (Kambhampati & 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 & Lozano-Perez, 1983) involve a 3-degree-of-freedom navigation problem and in (Kambhampati & Davis, 1986), a fairly difficult 2-dimensional maze.

Finally, some relation can be seen between parti-game and adaptive multigrid methods used to accelerate the convergence of solutions to partial differential equations. Adaptive multigrid methods that allow variations in resolution across the space typically use quad-tree or oct-tree data structures (McCormick, 1989). These approaches subdivide a cell by splitting in all dimensions simultaneously. (Arcilla, et al., 1991) describe adaptive triangulation approaches. Multigrid approaches have been used for dynamic programming in solving for the value function specified by Bellman's equation (Hoppe, 1986), (Akian, et al., 1988), (Chow, 1990). As with the robot motion planning approaches described above, it is not yet clear how multigrid approaches can be adapted to handle the learning problem in which the system dynamics are not completely known in advance.

6. Discussion

6.1. Splitting

Given a cell we have decided to split, which axis should be split? Algorithm (4) states that we 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 25 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.

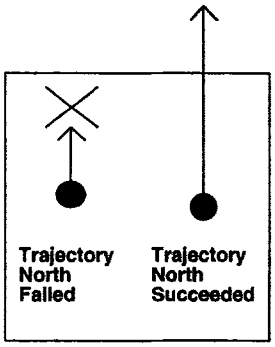


Figure 25. We have had two experiences of attempting to move North from two different points in the cell. Only one succeeded.

6.2. *Not forgetting*

When a cell is split, the outcomes of its children are initialized to be empty and so old data is forgotten. This is neither desirable nor 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 a large sacrifice of autonomy? We argue not: provided the continuity assumption holds, 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. It is particularly easy given our working assumption of deterministic system dynamics, but even in stochastic cases, developing a local linear model from data may not be hard (Schaal & Atkeson, 1994).

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

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

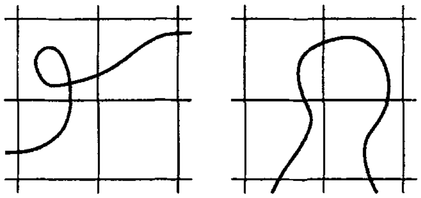


Figure 26. Parti-game cannot produce either of these kinds of suboptimality. No loops, and no unblocked adjacent cells that contain separate parts of a solution trajectory.

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 cell with

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

breaks on each axis and encourages exploration by estimating the J_{WC} value of all unvisited cells as zero. At this resolution at least one initial cell must be a proper subset of the goal region, and so once the system has entered any part of each initial cell the goal must have been discovered.

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 that parti-game will not produce. In the case of navigation, for example, parti-game cannot produce loops or meanders, as shown in Figure 26.

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 navigation problems, a proof would involve geometric assumptions. In dynamics problems a proof might need to assume local linearizability of the dynamics within cells, and could use then use Linear Quadratic Gaussian (LQG) local control design (see, for example, (Sage & White, 1977)).

For systems which can neither be characterized as geometric motion planning problems nor dynamics problems, it is also possible that optimality might be provable. Future research into this might incorporate *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 cells, 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 cell produces multiple results, how do we decide if this is due to inherent randomness or due to overly coarse cells? 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 principle, though, any amount of noise could break the parti-game 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 that try to explain outcomes in terms of location within the cell. This might help, but further research is needed.

If the randomness is something that occasionally teleports the system to a random place (breaking the assumption of paths being continuous through state-space), then parti-game would probably need an entirely different splitting criterion. One possibility is a version of the "G" splitting rule of (Chapman & 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

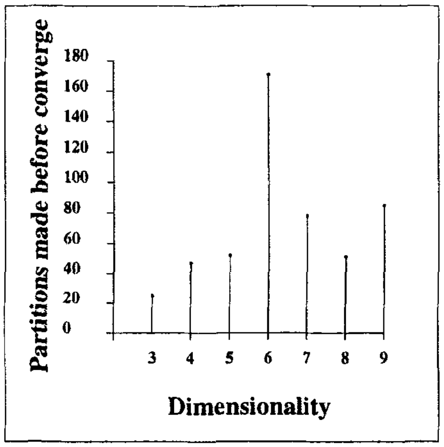


Figure 27. The number of cells finally created against degrees of freedom for a set of snake-like robots. The partitionings built were all highly non-uniform, typically having maximum depth nodes of twice the dimensionality. The relation between exploration time and dimensionality (not shown) had a similar shape.

eventually, whereas the nine-link robot cannot be expected to do so in tractable time. However, Figure 27 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.

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 cells that need to have their resolutions increased. There are many interesting avenues arising from these ideas which remain open for further investigation.

Appendix

Experimental Details

All systems, though continuous in time and space, were simulated in small discrete time steps.

A.1. Point in maze

This system is used in section 4.1. The state, $\mathbf{s} = (x, y)$, is two-dimensional and must lie in the unit square $0 \leq x \leq 1$, $0 \leq y \leq 1$. The action $\mathbf{a} = (\delta x, \delta y)$ is also two-dimensional, and represents small displacements in the state. Actions are constrained such that $\mid \mathbf{a} \mid = \sqrt{\delta x^2 + \delta y^2} \leq \frac{1}{200}$. The goal region is the square $0.9 \leq x \leq 0.95$, $0.9 \leq y \leq 0.95$. Writing $\mathbf{s}(t)$ as the state at time t and $\mathbf{a}(t)$ as the action at time t, the dynamics are:

$$\mathbf{s}(t+1) = \begin{cases} \mathbf{s}(t) & \text{If any barrier intersects the} \\ & \text{line between } \mathbf{s}(t) \text{ and } \mathbf{s}(t) + \mathbf{a}(t). \\ \mathbf{s}(t) + \mathbf{a}(t) & \text{Otherwise.} \end{cases} \tag{A.1}$$

The local greedy controller chooses the action which moves closest to the local goal $\mathbf{s}_{\text{GOAL}}$:

$$\mathbf{a}(t) = \lambda(\mathbf{s}_{\text{GOAL}} - \mathbf{s}(t)) \text{ where } \lambda = \min(\frac{1}{200 \mid \mathbf{s}_{\text{GOAL}} - \mathbf{s}(t) \mid}, 1) \tag{A.2}$$

A.2: The puck on the hill

This system is used in section 4.2. The state, $\mathbf{s} = (x, \dot{x})$, is two-dimensional and must lie in the region $-1 \leq x \leq 1$, $-2 \leq \dot{x} \leq 2$. x denotes the horizontal position of the puck in Figure 16. The action a is one-dimensional and represents the horizontal force applied to the puck. Actions are constrained such that $-4 \leq a \leq 4$. The goal region is the rectangle $0.5 \leq x \leq 0.7$, $-0.1 \leq \dot{x} \leq 0.1$. The surface upon which the puck slides has the following height as a function of x:

$$H(x) = \begin{cases} x^2 + x & \text{if } x < 0 \\ x/\sqrt{1+5x^2} & \text{if } x \geq 0 \end{cases} \tag{A.3}$$

The puck's dynamics are given by

$$\ddot{x} = \frac{a}{M\sqrt{1 + (H'(x))^2}} - \frac{gH'(x)}{1 + (H'(x))^2} \tag{A.4}$$

where $M = 1$ and $g = 9.81$. This equation is integrated by

$$\begin{aligned} x(t+1) &= x(t) + h\dot{x}(t) + \tfrac{1}{2}h^2\ddot{x}(t) \\ \dot{x}(t+1) &= \dot{x}(t) + h\ddot{x}(t) \end{aligned} \tag{A.5}$$

where $h = 0.01$ is the simulation time step. If any of the state variables exceed their limits, the system is deemed to have crashed, and the trial is restarted at the start state.

The local controller is essentially bang-bang: it is a linear controller with very high gains, the output of which is clipped to remain in the legal range $-4 \leq a \leq 4$.

$$\begin{aligned} a_{\text{nominal}}(t) &= -K_p\left(x(t) - x_{\text{goal}}(t)\right) - K_v\left(\dot{x}(t) - \dot{x}_{\text{goal}}(t)\right) \\ a(t) &= \begin{cases} -4 & \text{if } a_{\text{nominal}}(t) < -4 \\ a_{\text{nominal}}(t) & \text{if } -4 \le a_{\text{nominal}}(t) \le -4 \\ 4 & \text{if } a_{\text{nominal}}(t) > 4 \end{cases} \end{aligned} \tag{A.6}$$

where $K_p = 10^4$ and $K_v = 10^5$.

A.3: Rod in maze

This system is used in section 4.3. The state, $\mathbf{s} = (x, y, \theta)$, is three-dimensional. (x, y) denotes the position of the center of the rod and must lie in the unit square $0 \le x \le 1$, $0 \le y \le 1$. θ denotes the angle of the rod from the horizontal, and is constrained to lie in the range $0 \le \theta \le \pi$. Thus full rotation of the rod is not permitted: its bottom must never be above its top. The action $\mathbf{a} = (\delta x, \delta y, \delta\theta)$ is also three-dimensional, representing small displacements in the state. Actions are constrained such that $\sqrt{\delta x^2 + \delta y^2} \le \frac{1}{40}$ and $-\frac{\pi}{20} \le \delta\theta \le \frac{\pi}{20}$. The goal region is $0.8 \le x \le 0.9$, $0.6 \le y \le 0.7$, and θ may be any value. The rod has length 0.4. The dynamics are:

$$\mathbf{s}(t+1) = \begin{cases} \mathbf{s}(t) & \text{If any barrier intersects the area swept out} \\ & \text{when moving between } \mathbf{s}(t) \text{ and } \mathbf{s}(t) + \mathbf{a}(t). \\ \mathbf{s}(t) + \mathbf{a}(t) & \text{Otherwise.} \end{cases} \tag{A.7}$$

The local greedy controller chooses the action which moves closest to the local goal $\mathbf{s}_{\text{GOAL}}$:

$$\begin{aligned} (\delta x, \delta y) &= \lambda[(x_{\text{goal}}, y_{\text{goal}}) - (x, y)] \\ &\text{where} \\ \lambda &= \min\left(\frac{1}{100 \mid (x_{\text{goal}}, y_{\text{goal}}) - (x, y) \mid}, 1\right) \end{aligned}$$

$$\delta\theta = \begin{cases} -\frac{\pi}{20} & \text{if } \theta_{\text{goal}} - \theta < -\frac{\pi}{20} \\ \theta_{\text{goal}} - \theta & \text{if } -\frac{\pi}{20} \le \theta_{\text{goal}} - \theta \le \frac{\pi}{20} \\ \frac{\pi}{20} & \text{if } \theta_{\text{goal}} - \theta > \frac{\pi}{20} \end{cases} \tag{A.8}$$

A.4: Puck sliding on two-dimensional surface

This system is used in section 4.3. The state, $\mathbf{s} = (x, y, \dot{x}, \dot{y})$, is four-dimensional. The dynamics and local controller are all essentially two-dimensional analogs of the puck on the hill. One important exception is that the system does not enter a crash state if one of the state variables exceeds its limits: instead the state variable is clipped to the closest legal value.

A.5: Multi-joint arm kinematics

This system is also used in section 4.3. In the given example, the state-space dimension $N = 9$. The state $\mathbf{s} = (\theta_1, \theta_2 \ldots \theta_N)$ denotes a set of joint angles. θ_1 denotes the angle from horizontal of the joint connected to the base (shown in Figure 24). For $i > 1$, θ_i denotes the angle between joint i and joint $i - 1$. The action $\mathbf{a} = (\delta\theta_1, \delta\theta_2 \ldots \delta\theta_N)$ represents small displacements in the state. Actions are constrained such that $\mid \mathbf{a} \mid \leq \frac{\pi}{15}$. The state transition function is:

$$\mathbf{s}(t+1) = \begin{cases} \mathbf{s}(t) & \text{If any barrier intersects the area swept out} \\ & \text{when moving between } \mathbf{s}(t) \text{ and } \mathbf{s}(t) + \mathbf{a}(t), \\ & \text{or the arm self intersects during the move.} \\ \mathbf{s}(t) + \mathbf{a}(t) & \text{Otherwise.} \end{cases} \tag{A.9}$$

The local greedy controller chooses the action which moves closest to the local goal $\mathbf{s}_{\text{GOAL}}$:

$$\begin{aligned} \mathbf{a}(t) &= \lambda(\mathbf{s}_{\text{GOAL}} - \mathbf{s}(t)) \\ &\text{where} \\ \lambda &= \min(\frac{\pi}{15 \mid \mathbf{s}_{\text{GOAL}} - \mathbf{s}(t) \mid}, 1) \end{aligned}$$

Acknowledgments

We would like to thank Mary Soon Lee, Justin Boyan, Stefan Schaal, Andy Barto, Satinder Singh, Sebastian Thrun and the reviewers for their suggestions and comments during the preparation of this paper. Andrew Moore acknowledges the following sources of support: U. K. Science and Engineering Research Council, a National Science Foundation Research Initiation Award, and 3M Corporation. Christopher Atkeson acknowledges support provided under Air Force Office of Scientific Research grant F49-6209410362, by the ATR Human Information Processing Research Laboratories, and by a National Science Foundation Presidential Young Investigator Award.

Notes

1. Another method is to increase the resolution along the trajectory (Moore, 1991).
2. In practice, parti-game's solution was shorter than the optimal path in the discretized maze. This is because the continuous actions of parti-game permit travel in arbitrary directions, not just North, East, South and West.
3. 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.

References

Akian, M., Chancelier, J.P. & Quadrat, J.P., (1988). Dynamic Programming Complexity and Application. In *Proceedings of the 27th Conference on Decision and Control, Austin, Texas.*

Arcilla, A.S., Hauser, J., Eiseman, P.R. & Thompson, J.F., (1991). *Numerical Grid Generation in Computational Fluid Dynamics and Related Fields.* North-Holland.

Barto, A.G., Bradtke, S.J. & Singh, S.P., (1994). Real-time Learning and Control using Asynchronous Dynamic Programming. *AI Journal, to appear (also published as UMass Amherst Technical Report 91-57 in 1991).*

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

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

Bertsekas, D.P. & Tsitsiklis, J.N., (1989). *Parallel and Distributed Computation.* Prentice Hall.

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

Chapman, D. & Kaelbling, L.P., (1991). Learning from Delayed Reinforcement In a Complex Domain. Technical Report, Teleos Research.

Chow, C.S., (1990). Multigrid algorithms and complexity results for discrete-time stochastic control and related fixed-point problems. Technical report, M.I.T. Laboratory for Information and Decision Sciences.

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

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

Kaelbling, L. (1993). Hierarchicial Learning in Stochastic Domains: Preliminary Results. In *Machine Learning: Proceedings of the Tenth International Workshop.* Morgan Kaufmann.

Kaelbling, L.P., (1990). Learning in Embedded Systems. PhD. Thesis; Technical Report No. TR-90-04, Stanford University, Department of Computer Science, June 1990.

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

Knuth, D.E., (1973). *Sorting and Searching.* Addison Wesley.

Koenig, S. & Simmons, R.G. (1993). Complexity Analysis of Reinforcement Learning. In *Proceedings of the Eleventh International Conference on Artificial Intelligence (AAAI-93).* MIT Press.

Latombe, J. (1991). *Robot Motion Planning.* Kluwer.

McCormick, S.F., (1989). *Multilevel Adaptive Methods for Partial Differential Equations.* SIAM.

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

Moore, A.W., (1991). 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 Kaufmann.

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

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

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

Sage, A.P. & White, C.C., (1977). *Optimum Systems Control.* Prentice Hall.

Schaal, S. & Atkeson, C.G., (1994). Assessing the Quality of Local Linear Models. In *Advances in Neural Information Processing Systems 6.* Morgan Kaufmann.

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

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

Sutton, R.S., (1990). Integrated Architecture for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In *Proceedings of the 7th International Conference on Machine Learning.* Morgan Kaufmann.

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

Received January 31, 1994
Accepted March 8, 1995
Final Manuscript March 24, 1995