Proceedings of the 2001 IEEE
International Conference on Robotics & Automation
Seoul, Korea - May 21-26, 2001

Towards a Meta Motion Planner B: Algorithm and Applications

Amit Adam
Dept. of Mathematics
amita@tx.technion.ac.il

Ehud Rivlin
Dept. of Computer Science
ehudr@cs.technion.ac.il

Ilan Shimshoni
Dept. of Industrial Engineering
ilans@ie.technion.ac.il

Technion – Israel Institute of Technology
Haifa 32000 – Israel

Abstract

In a companion paper [1] we have developed a framework for rating or comparing navigation packages. For a given environment a navigation package consists of a motion planner and a sensor to be used during navigation. The ability to rate or measure a navigation package is important in order to address issues like sensor customization for an environment and choice of a motion planner in an environment.

In this paper we present the algorithm which we use in order to rate a given navigation package. Under the framework which was presented in [1], a partially observable Markov decision process (POMDP) is defined. The algorithm searches for an optimal policy to be employed in this decision process.

We briefly review the problem and the framework, develop the algorithm and present experimental results.

1 Introduction

The navigation problem consists of planning and executing a path between two different points in an environment. Many different factors are involved in this problem, of which two important ones are the motion planner and the sensors which will be used by the robot. In a companion paper [1] we have raised and discussed the problem of choosing between different possible combinations of motion planners and sensors. Solving this problem would enable for example to choose among different sensor customizations - for example different placement of visual landmarks in the environment. We have presented a framework in which the basic idea was that each combination of motion planner and sensor (which we have termed a "navigation package") defines a partially observable Markov decision process or POMDP. The navigation package is then rated by the expected payoff which can be obtained in the decision process, while acting under the best possible policy (i.e. the policy which maximizes the expected payoff).

This paper complements [1] by introducing a simple reinforcement learning algorithm which finds a suitable policy for a given POMDP. We present the algorithm, some results which were obtained by using it and a concluding discussion of the work.

2 Approximating an Optimal Policy

Following [1] each navigation package defines a POMDP. The state space consists of a quantization of the environment. The possible actions are movement in different directions, update of the position (by invoking a sensor) or stopping. After every action a reward is collected, the amount of which depends both on the state the robot was in and on the action taken. At each state there is a preferred action (leading to a maximal reward) which is movement in the direction specified by the nominal motion plan. However, since the robot is unsure of its position, it cannot always choose this preferred action. At all times a belief function $b(\cdot)$ is maintained by the robot. This function is the probability distribution of the current state. It is updated after each step based on the dynamics of the robot (where it moved) and on the observations which are generated. In our model an observation is generated only when the robot invokes its sensor.

Our goal now is to develop an algorithm for finding a policy which will maximize the expected payoff. A policy is a mapping which associates an action with each belief function. Let us begin with the way we represent the policy. A standard way of representing a policy is through its value function or Q-values (details may be found in [3] for example). The function $Q(b(\cdot), a)$ represents the total reward the agent may expect if it performs the action a and then continues optimally, when the current state is distributed by $b(\cdot)$. This is a very convenient representation for a policy: using the value function Q, an agent with a current belief function $b(\cdot)$, chooses the optimal action simply by

$$a^* = \arg\max_{a \in A} Q(b(\cdot), a)$$

(A is the set of possible actions).

When using the value function approach to represent a policy, we have to address two issues. The first is the issue of storing and representing the value function Q. The second issue is how to actually compute the value function.

Representing the Q-function is not straight forward since one of the arguments Q accepts is a belief function which comes from an infinite space. Therefore we cannot store the values Q obtains on every possible pair $(b(\cdot), a)$ of belief function and action. Instead, we discretize the belief space into a finite set of beliefs

$$B_d = \{b_1(\cdot), \ldots, b_N(\cdot)\}$$

and store the values Q obtains on the finite set $B_d \times A$. In other words, Q is represented by a finite lookup table.

The second issue which we now address is how to actually compute the function Q. When the state space is not very small, exact computation of the optimal value function is computationally infeasible (see [3, 2] for example). Therefore we strive to approximate the value function. By finding an approximate value function and using it, we obtain not the optimal policy but an approximation for the optimal policy. We have chosen to use reinforcement learning in order to compute an approximation for the value function.

The reinforcement learning algorithm starts with an initial approximation $\hat{Q}$ for the value function Q. Each iteration of learning involves simulation of actions which were chosen on the basis of the current approximation $\hat{Q}$. The "empirical" rewards obtained in the simulation are used to update the expected value of taking the action - in other words to update the current approximation $\hat{Q}$. More specifically, each learning iteration has the following structure:

A Learning Iteration Starting from Belief $b_0(\cdot)$

- Let $\hat{Q}$ be the current approximation of the value function
- Draw a random state s distributed according to $b_0(\cdot)$
- Let $\tilde{b}(\cdot) = b_0(\cdot)$
- Repeat l times:
  1. Let $b_d(\cdot)$ be the discretized value of $\tilde{b}(\cdot)$.
  2. Based on the current belief $\tilde{b}(\cdot)$ and the current value function $\hat{Q}$, choose the best action a^*.
  3. Based on simulation, update the current value of $\hat{Q}(b_d(\cdot), a^*)$.
  4. From the current state s jump to a new state with probabilities governed by the current state s and the action a^*. Let s now denote the new state.
  5. Obtain an observation o.
  6. Based on the action taken and the observation made, update the belief function. Let $\tilde{b}(\cdot)$ now denote the updated belief function.
  7. Return to step 1 (loop l times).

Let us elaborate on some of the steps.

Step 2: By default we choose the best action based on our current belief and current value function:

$$a^* = \arg\max_{a \in A} \hat{Q}(b_d(\cdot), a)$$

However, recall that $\hat{Q}$ is not the true value function. Therefore we sometimes (say with a 0.1 probability) choose a random action instead. By doing this we maintain a constant "exploring" behaviour.

Step 3: In this step we *learn* the value of $\hat{Q}(b_d(\cdot), a^*)$. This is done by simulating the action a^* at states s which are drawn from the distribution $b(\cdot)$. By simulating the action from a state, we get an immediate reward $R(s, a^*)$. In addition we obtain an observation o. Based on o we update the belief function. Using the updated belief we can look up the expected optimal future reward, based on the current $\hat{Q}$. We add this value to the immediate reward we got, and this is the empirical value obtained from acting a^* while the belief was $b(\cdot)$. We repeat this simulation of acting a^* a number of times, and take the average of empirical values we have obtained. This average value is the updated value of $\hat{Q}(b_d(\cdot), a^*)$.

The above learning iteration is repeated, each time beginning with the same $b_0(\cdot)$. By doing this we obtain updates of the values of $\hat{Q}$, mostly in those areas of the belief space that will actually be traversed when we use the resulting value function. This is a result of the fact that we choose at each step the best action based on the current approximation $\hat{Q}$.

The reinforcement learning algorithm which we have used is rather standard [5]. Reinforcement learning has been applied in various other works on POMDPs including [4, 6, 7]. Although we have used a very basic version of this algorithm, we have obtained very reasonable policies as will be shown in section 3.

Quantizing the Belief Space and Initializing the Value Function Recall that we have to quantize the belief space to a finite set B_d of beliefs. We have found the following quantization useful. For a given belief function we first compute the expected state and check what is the probability of being in that state (recall that our world is represented by a grid of possible

positions). We then compute the probability of being in a neighboring state to the expected state. The two probabilities, namely the probability of being in the expected state and the probability of being in a neighboring state, are quantized and serve as a representation of the belief function. In the experiments which will be described below, we used 10 quantization levels for each probability.

Finally, what function serves as the initial $\hat{Q}$ with which we start the reinforcement learning iterations? Let us first define $Q_{opt}(s)$ to be the value which may be obtained by a robot with perfect control which starts at state s and moves to the goal according to the motion plan. In other words, we assume that the robot always ends up in the position it intended to reach after a movement command. In this case sensing actions are not needed. Now, for a belief function b which after quantization has probability 1 at a given state s, the initial value of $\hat{Q}(b,a)$ is $Q_{opt}(s)$ assuming that a is indeed movement according to the motion plan. If a is another action, then we take $\hat{Q}(b,a)$ to be 0. If the belief b has probability p at the expected state after quantization, then we take the initial value to be $\hat{Q}(b,a) = pQ_{opt}$ for the motion-plan action a and 0 for other actions. This value function is even more optimistic than MDP-based approximations for the POMDP value function, which are sometimes used [2]. Let us call this initial approximation for the value function the "perfect robot value function". Note that this initialization is clearly dependent on the underlying motion planner: the function describes the values obtainable by a perfect robot which acts according to the nominal motion plan.

3 Results

We now present results that were obtained by using the algorithm which we have now described. The results we present were obtained for different environments, sensors and motion planners. Each environment is represented by a grid. Some of the squares in the grid are obstacles. The robot moves between the free cells in the grid. At each time step it may move one square to the left, to the right, up or down. With each movement, position uncertainty grows, since with a probability of 20% the robot ends up in a square which is nearby the square it intended to reach (see Fig. 1).

The robot may invoke its sensor at any time. The sensor returns a grid position which is the estimated current position. The accuracy of this estimate may vary and depends on the actual position of the robot. We have used 3 accuracy levels, depicted in Fig. 2.

For every scenario tested, a value function was computed using the reinforcement learning algorithm described previously. The policy resulting from this value function was used in 1000 runs of the simulated robot. In all runs the robot started in the initial configuration, with the belief function being the initial configuration with probability 1 (i.e. no uncertainty in position). Each run consisted of 30 steps of action/observation. This number of steps is sufficient to permit reaching the goal. The results we show for each scenario are based on the results collected in those 1000 runs.

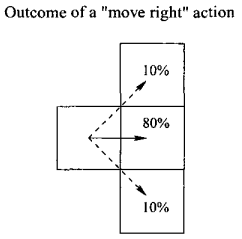


Figure 1: Position uncertainty resulting from a movement to the right. Similar uncertainty develops from movements in other directions

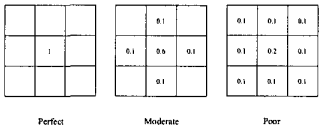
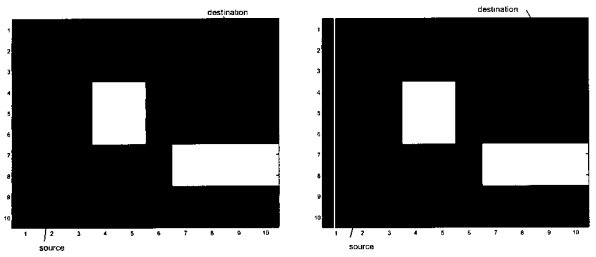


Figure 2: Sensor performance. The robot is in the middle square. The sensor estimates the current position with the probabilities depicted. We use three sensor accuracy levels.

We start with the environment shown in Fig. 3. The asterisk marks the initial configuration and the + marks the goal configuration. A roadmap-based motion planner has been used to plan paths from every free configuration to the goal position. The directions of motion in each configuration as determined by this motion planner are shown in part (a) of the figure.

Let us first present the consequences of moving without any updates. We let the robot use the "perfect robot" value function. The policy associated with this value function is to perform a sequence of motions and then stop, without ever invoking the sensor. Fig. 4 presents the results from 1000 runs using this policy. In part (a) of the figure we see a histogram of the actions performed at each time step. We see that in all 1000 runs we had 14 motion actions and then the robot stopped for the next 16 time steps. Due to inaccurate control, this policy has led to the goal configuration in only about 20% of the runs. This is shown in part (b)



(a) (b)

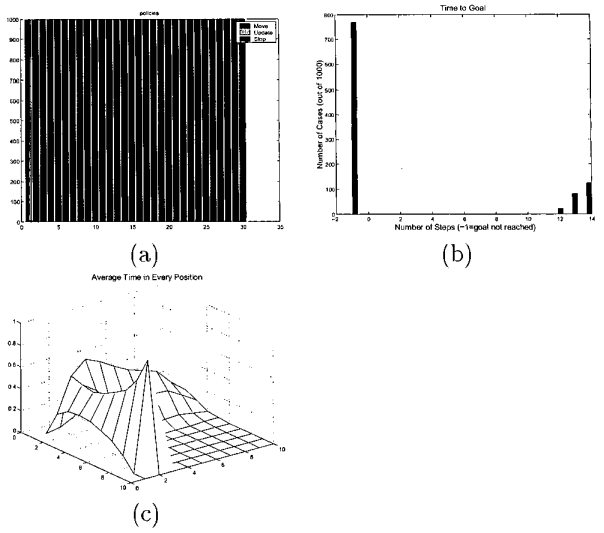
Figure 3: First environment. (a) Roadmap-based motion planner. (b) Visibility graph based motion planner

of the figure. Part (c) of the figure shows the average number of time steps (per run) the robot has spent in every configuration. Note that in the initial configuration this number is 1. Then due to inaccurate control the presence of the robot is "spread out" on a wide strip around the nominal path. Notice that around the goal configuration the spread is rather wide.

Next we used the value function which was computed using the reinforcement learning algorithm for 100000 learning iterations. We assumed perfect sensing capability across the environment. Fig. 5 shows the results. Part (a) of the figure shows that the policy executed consists of movement actions for 8 steps and then the robot invokes its sensor. Part (b) of the figure shows that now the goal has been reached in almost all of the runs. In part (c) we see the "presence" of the robot along the path. Compare this figure with with Fig. 4(c). Part (d) shows the positions in which the sensor was invoked. Note the effect of position update on the "presence" of the robot as seen in part (c) of the figure. Near the final goal the policy calls for an update since stopping in a non-goal position is much less rewarding than stopping in the goal. Part (e) of the figure shows a histogram of the number of times an obstacle was hit during a run. This, together with the histogram in part (b), are "operational" criteria which might be of interest.

We now change the sensor in this scenario. A sensor which does not operate properly in part of the environment is introduced in Fig. 6. We now have a new navigation package, consisting of the first environment with the roadmap-based motion planner (see Fig. 3), and the new non-perfect sensor which we have now defined.

The results for this environment are shown in Fig. 7. Notice in parts (a) and (d) of the figure how the policy has changed to invoke the sensor earlier in the path. This is due to the fact that it makes no sense to invoke the sensor where it performs poorly.



(a) (b)

(c)

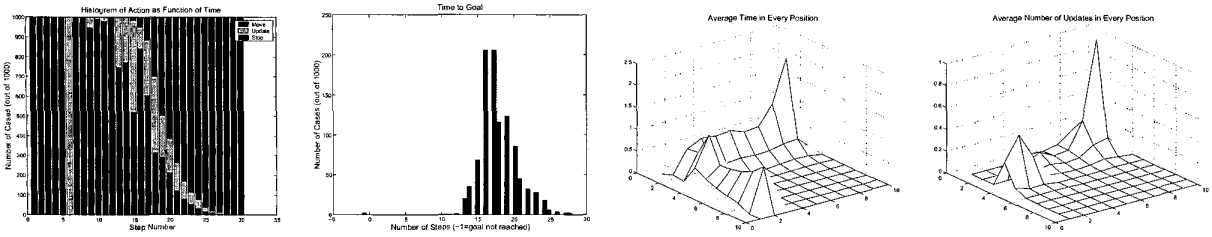
Figure 4: Results of using the "perfect robot" value function on the first environment. (a) Histogram of the actions performed in each time step. (b) The time it took to reach the goal. Note that in nearly 80% of the runs the goal was not reached. (c) The average "presence" of the robot along the path.

For the same environment and sensor, we now consider changing the motion planner. Fig. 3(b) shows a second motion planner for this environment. A new navigation package is now defined and a value function was computed for it. Results of using this value function are presented in Fig. 8.

We see that the time in goal and the chances of reaching the goal are quite similar (compare Figs. 8(b) and 5(b)). However, the chances for colliding with an obstacle are higher when using the second motion planner (Figs. 8(e) and 5(e)). Therefore, we might conclude the first navigation package is better than the second.

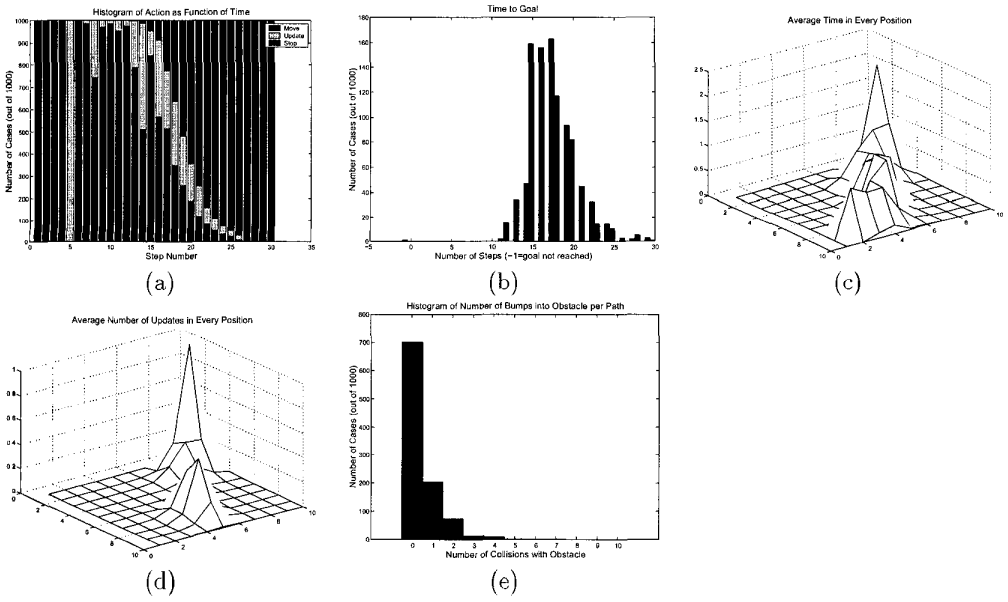
Fig. 9 shows a different environment with two motion plans. The first is based on a potential field planner and the second on the visibility graph. The sensor we have used on this environment (with both motion plans) is abstractly represented by the performance map shown in Fig. 10.

The first navigation package on this environment used the potential-field based motion plan (depicted in Fig. 9(a)). The results obtained by using this navigation package are shown in the top row of Fig. 11. The bottom row shows the results for the second motion plan. We can see that when using the potential-field based planner, the robot had a chance of approximately 8% of not reaching the goal. When it did reach the goal,



(a) (b) (c) (d)

Figure 7: First environment, sensor as in Fig. 6. (a) Histogram of the actions performed in each time step. Notice earlier sensing as compared with Fig. 5(a). (b) The time it took to reach the goal. (c) The average "presence" of the robot along the path. (d) Places where the sensor was invoked. Compare with Fig. 5(d).



(a) (b) (c) (d) (e)

Figure 8: First environment with a second motion planner (as in Fig. 3(b)). (a) Histogram of the actions performed in each time step. (b) The time it took to reach the goal. (c) The average "presence" of the robot along the path. (d) Places where the sensor was invoked. (e) Histogram of the number of times an obstacle was hit along the path.

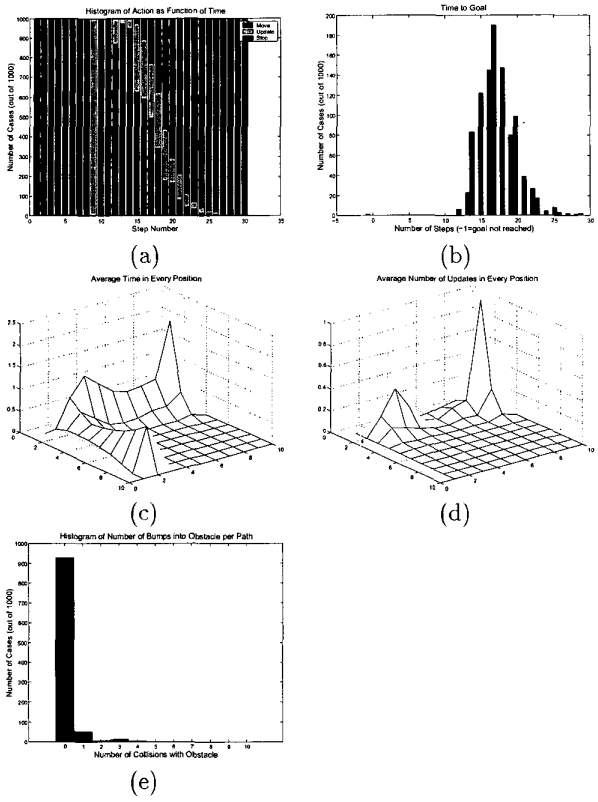


Figure 5: First environment with perfect sensing. (a) Histogram of the actions performed in each time step. (b) The time it took to reach the goal. Note that now in almost all of the runs the goal was reached. (c) The average "presence" of the robot along the path. (d) Places where the sensor was invoked. (e) Histogram of the number of times an obstacle was hit along the path.

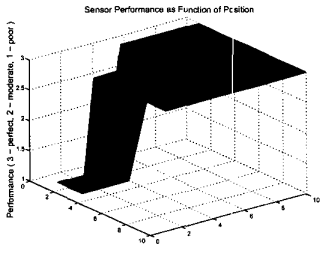


Figure 6: Sensor performance in the first environment.

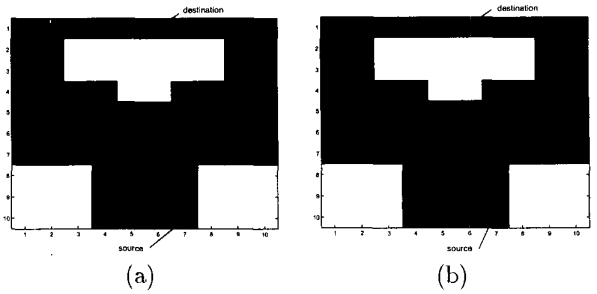


Figure 9: Second environment with two motion plans on it. (a) Potential field based motion plan. (b) Visibility graph based motion plan.

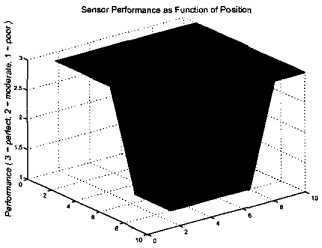


Figure 10: Sensor used on the second environment

it usually took over 20 time steps. However, when using the visibility graph based planner, the robot had only a 2% chance of not reaching the goal, and when it did reach the goal, it usually took less time steps. Looking at the chances of colliding with an obstacle, we see that using the visibility graph planner our chances of colliding are above 60% (in less than 400 runs of the 1000 there were no collisions), while the potential field planner gives us a chance of collision of approximately 45%. Depending on our preferences, we may now decide which of the motion planners should be used in this environment with this specific sensor.

The last example relates to the environment shown in Fig. 12. We have used this environment and motion planner in conjunction with a perfect sensor. In this example we want to illustrate the effects of choosing different rewards. We have used two sets of rewards, where the second set penalizes more severely collisions with obstacles and uses of the sensor. The results from the first set of rewards are shown in parts (a)-(c) of Fig. 13, while the results that were obtained with the second set of rewards are shown in parts (d)-(f) of the figure. As may be seen, the original motion plan has been used in part (a) of the figure, but it has been abandoned completely in the second case (part (d)) because it called

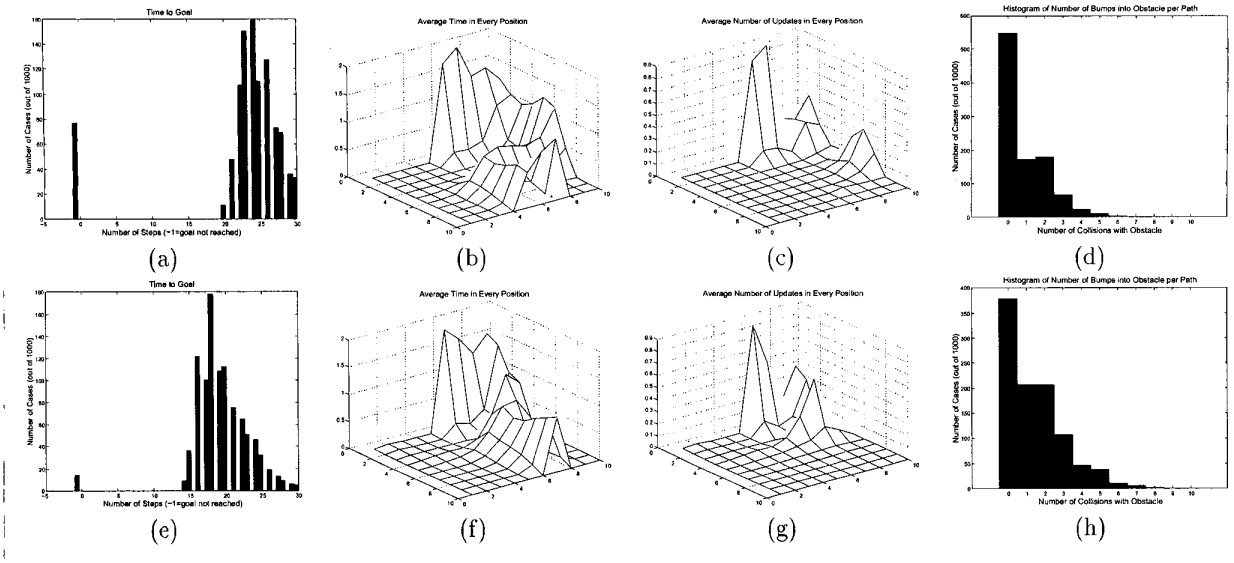


Figure 11: Second environment. Top row refers to a potential field planner and bottom row to a visibility graph motion planner. (a),(e): The time it took to reach the goal. (b),(f): The average "presence" of the robot along the path. (c),(g): Places where the sensor was invoked. (d),(h): Histogram of the number of times an obstacle was hit along the path.

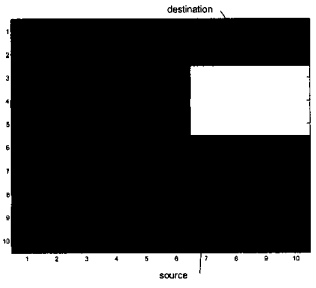


Figure 12: Third environment

for travel near an obstacle, which is very undesirable under this choice of rewards.

4 Discussion and Conclusions

The navigation problem involves a number of different aspects and factors such as the environment, the sensors and the algorithms. We have chosen the term navigation package to denote a specific combination of these factors. In this paper and a companion paper [1] we have discussed the problem of comparing navigation packages.

The algorithm we have presented in this paper uses a simple version of reinforcement learning. We have applied this simple algorithm to a number of test environments and have shown that it produces rather "reasonable" policies.

Besides enabling the comparison of different navigation packages, the POMDP framework allows us to augment a nominal path with sufficient sensing. In contrast to other approaches, we recognize the fact that in some cases the robot may not need to know its exact position. An optimal policy will choose sensing actions only in cases where reduction of the position uncertainty is actually required.

We believe that this work is a first step towards a *meta-algorithm* for choosing between different navigation algorithms and/or sensors. In order to completely achieve this goal, more research is required. Firstly, the POMDP framework we have used can compare two different navigation packages for navigating from a given initial position to a final position. If we want our meta-algorithm to customize a sensor for an environment for example, we must consider all possible combinations of initial and goal positions. Solving a POMDP for every possible combination is not feasible currently. Therefore our framework may not be used in a straight-forward manner for such tasks.

Additionally, solving large POMDPs (thousands of states) even approximately is still beyond our capability. For larger environments techniques such as multi-resolution or division of the environment into sub-

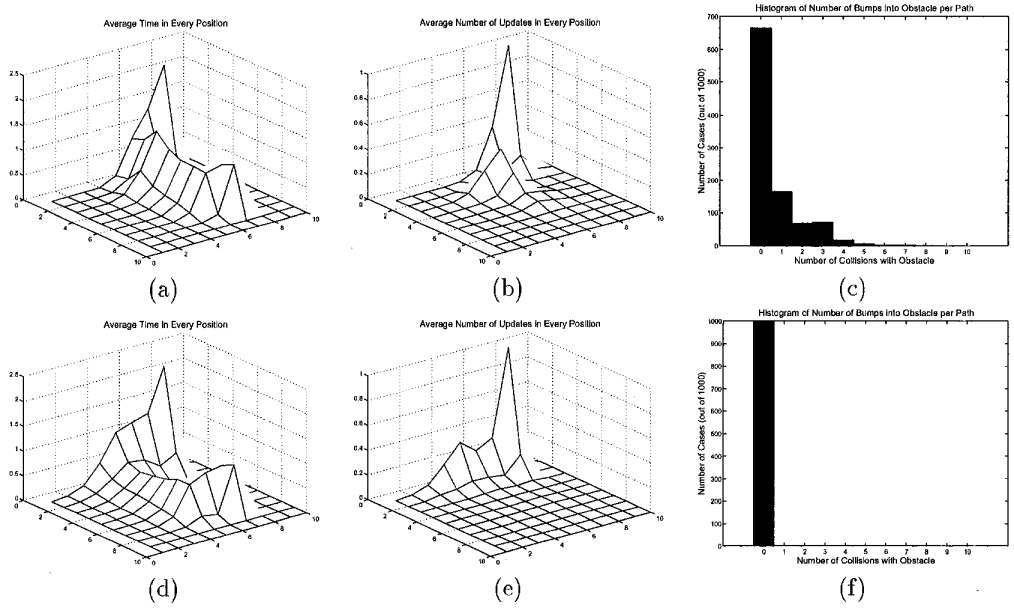


Figure 13: Using different rewards, environment as in Fig. 12. The bottom row shows the results of using a policy which was found for rewards which penalize collisions and position updates more severely. (a),(d): Paths taken. Notice in part (d) how the robot first moves to the left in order to keep a safe distance from the obstacle region. In part (a) the robot "sticks" to the nominal motion plan. (b),(e): Where the sensor was invoked. (c),(f): Collisions with obstacles.

environments may be required. Our future research will focus on these issues in order to enable the use of a meta algorithm for navigation in larger and more "real life" environments.

Acknowledgment A. Adam and I. Shimshoni were supported in part by Israeli Ministry of Science Grants no. 9766 and no. 2104.

References

[1] A. Adam, E. Rivlin, and I. Shimshoni. Towards a Meta Motion Planner A: Model and Framework. Proceedings of ICRA 2001.

[2] M. Hauskrecht. *Planning and control in stochastic domains with imperfect information.* PhD thesis, MIT, 1997.

[3] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

[4] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In *Proc. of 12'th Int. Conf. on Machine Learning*, pages 362–370, 1995.

[5] T. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[6] R. Parr and S. Russell. Approximating optimal policies for partially observable stochastic domains. In *Proc. of 14'th IJCAI*, 1995.

[7] S. Thrun. Monte Carlo POMDPs. In *Proc. of NIPS*, 1999.