# Chapter 5

# Simulation Results

In this chapter we describe the experiments we have carried out through simulation. We have used simulation for three different tasks: firstly, to check that the multiagent Navigation system we have designed works properly; secondly, we have applied Reinforcement Learning techniques in order to learn a policy on the use of the camera; and finally, we have used a Genetic Algorithm approach to tune the parameters of the agents in the Navigation system.

For these different tasks, we have used two simulators. We started using the Webots[1] simulator. On this simulator we implemented the Navigation system and we also used it for the Reinforcement Learning task. However, we found some problems with the Webots simulator, mainly related to batch execution, which made the experimentation very slow. Although we were able to get results when used for Reinforcement Learning, we decided to develop our own simulator, to do extensive simulation with no problems. We used this new simulator to run again the multiagent Navigation system, and for the Genetic Algorithm approach to tune the parameters.

## 5.1 The Simulated System

It has to be pointed out that the overall system (that is, the Navigation, Pilot and Vision systems) used in the simulations is not exactly the same as the one described in the previous chapter (also described in [13]). Since the beginning of this research, four years ago, the Navigation, Pilot and Vision systems have been evolving (agents of the Navigation system have been added, modified and removed, and the capabilities of the Pilot and Vision systems have also changed) until we have reached what, by now, is the definitive version, which has just been described. This evolution has been guided by the experimentation, both on simulation and with the real robot. The simulation experiments described in this chapter show the performance of a previous version of our system [59, 12].

One of the main differences between the simulated system and the definitive one is that in the simulated one the Vision system did not provide information about the

[1] From Cyberbotics, http://www.cyberbotics.com

distance to the visible landmarks; it provided the Navigation system only with angular information. Moreover, the simulated Vision system had no range limitation, that is, it could identify any landmark, no matter how far it was, as long as it was in the view field of the camera. Obviously, this does not hold on the real Vision system.

Due to this lack of distance information, the *Map Manager* agent had to compute the distance to the landmarks using the change in angle of each landmark on successive viewframes. Since the change in angle can vary very little for the landmark the robot is going towards (i.e. the target), it was very difficult to accurately compute the distance to the target. In the simulated system, there was an additional agent, the *Distance Estimator*, that helped on computing the distance to the target. The role of this agent was to move the robot orthogonally with respect to the line connecting the robot and the target landmark while pointing the camera in the direction of the target, so that the change in angle was maximal, permitting the *Map Manager* to compute the distance accurately. The *Distance Estimator* agent computed the imprecision associated to the distance to the target. This imprecision is computed as $I_d = 1 - 1/e^{\kappa\epsilon_t}$, where $\kappa$ is a parameter to control the shape of the function, and $\epsilon_t$ is the error in distance, and, similarly to what the *Target Tracker* does, it is computed as the size of the interval corresponding to the 70% $\alpha$-cut of the fuzzy number representing the distance to the target. The *Distance Estimator* agent bids were a function on this imprecision. If the imprecision was high, it bid high to move the robot orthogonally, so the distance to the target could be computed with a lower error.. On the other hand, if the imprecision was low, so were the bids. This agent played a very important role at the beginning of the navigation, since the distance to the target was unknown, and therefore, the imprecision maximal. Thus, the *Distance Estimator* would bid very high in order to let the *Map Manager* get a first estimate of the distance. This agent was also responsible for deciding if the robot had reached the target, since it had the distance information. On the definitive system, this is responsibility of the *Target Tracker*.

Another important difference is that the simulated system did not use Visual Memory. That is, the Navigation system was only informed about the landmarks currently visible within the view field of the camera. This restriction made it difficult to create "good" beta-units, since all the visible landmarks were within a narrow view field, and thus, very collinear.

The *Rescuer* agent also had some differences: apart from getting active when the robot was blocked and when the imprecision in the target's location was too high, it also got active when the risk (computed and broadcasted by the *Risk Manager*) was over a threshold. Furthermore, its behavior was to always visually scan the surroundings of the robot and, after that, ask for a diverting target, not taking into account the reason of its activation.

There were also differences on the Pilot system. Another partner on the project we are involved in was responsible of building the Pilot system. Therefore, initially, we did not focus on this system, and did not worry about how it was designed. As long as it was able to avoid the obstacles encountered in its way, its design did not affect at all our coordination mechanism nor the design of the agents. For this reason, we started using a built-in pilot system of the Webots simulator that used simulated sonar sensors in order to avoid obstacles. In the real robot, however, such sonar sensors are not available, and,

as explained in the previous chapter, the Pilot system we finally implemented is only able to detect obstacles by bumping into them.

A final difference is that the mapping and navigation method used was not as explained in Chapter 3. Firstly, the criterion used to select topological regions was based only on the collinearity of the region and its size, thus, permitting overlapping regions, and not assuring a complete representation of the environment. And secondly, the computed diverting targets were always single landmarks; the computation of edges as diverting targets was introduced after experimenting with the real robot.

Despite all these differences, the basic elements of our approach have not been drastically modified during the evolution of the system: the bidding coordination mechanism has not been changed at all, and the mapping method has experienced only slight modifications.

## 5.2 Multiagent Navigation System Simulation

The goal of simulation was to check whether our approach, that is, the architecture, the bidding coordination mechanism and the mapping method, could lead to a robust navigation system.

We implemented the agents of the Navigation system and tested the algorithm on the Webots simulator and in our own developed one. Each agent was executed as an independent thread, and they used shared memory for message passing. We also simulated the Pilot and Vision systems on both simulators. We set the parameters of each of the agents by hand. We first set their values intuitively, and slightly modified them after some simulation trials.

As a first step, we checked whether the bidding mechanism was able to adequately coordinate the agents of the Navigation system and the Pilot, so that the task of reaching the target was accomplished. The Pilot system used was not able to inform about the presence of long obstacles between landmarks, although it would avoid them. For this reason, we were not still checking the mapping and navigation capabilities of the system.

Figure 5.1 shows a navigation run in the Webots simulator. It shows the path followed by the robot from a starting point to a target landmark. The environment was composed by a set of landmarks (shown as circles), a river (the thick blue traversing line) with a couple of bridges, and some fences and other obstacles. These obstacles did not occlude the target landmark, so it was visible from any location of the environment. The task to be performed was to reach the target (at the left-hand side of the world) avoiding any obstacle encountered on the way.

At the very beginning, the distance to the target is unknown, so the *Distance Estimator* agent (DE) bids very high to move the robot orthogonally to the line connecting it to the target and looking to the target, so that the *Map Manager* can estimate the distance to the target. The *Target Tracker* agent (TT) bids for moving and looking towards the target, but the bids of DE are higher and the robot moves orthogonally. As the robot moves, the *Map Manager* computes the distance to the target, and the imprecision computed by the DE decreases, causing its bids also to decay. At a given point, the bids of TT are higher than those of DE, and the robot starts going towards the tar-

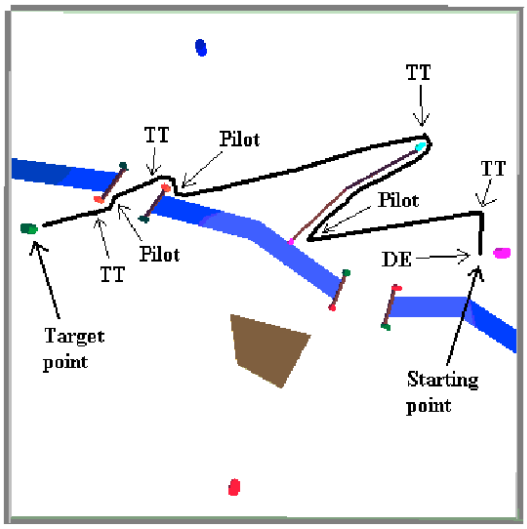


Figure 5.1: Robot's path from starting point to the target

get. Since there are no obstacles around, the Pilot does not bid at all. However, after some advance, the robot encounters an obstacle, and the Pilot bids very high to avoid it, surpassing the bids of TT and DE. When the obstacle has been totally avoided, the Pilot stops bidding, the bids of TT win again, and the robot moves towards the target. This situation is repeated a couple of times until the robot finally reaches the target.

Although the environment used in this first step was simple, mainly because of the constant visibility of the target, simulations showed that the bidding coordination mechanism worked properly, since it was able to coordinate the different agents and the Pilot.

The next step was to test the mapping and navigation capabilities of the Navigation system. In this step we used our own developed simulator, with a better Pilot system, capable of informing about the linear obstacles between landmarks, and with more realistic environments including occluding obstacles, so that the target was not visible all the time.

In Figure 5.2 we see how the Navigation system computes diverting targets for reaching the initial target when this is lost. In this environment, filled polygons are occluding obstacles, and empty ones are non-occluding ones, thus, permitting the visibility of the target from the starting point. At point A, it sees the target and starts going towards it. However, at point B, it detects an obstacle, so the Pilot forces the robot to turn. When it reaches point C, it cannot see the target anymore, as it is behind an occluding obstacle. At this point, a diverting target is computed (in this case, landmark 30 is selected). The robot starts going to this diverting target. Once reached (point D), a new diverting target is computed (landmark 38 is selected), and the robot goes toward it. At point E, after reaching the current diverting target, a new one is computed (landmark 12), which is reached at point F. From this point, it sees the initial target again,

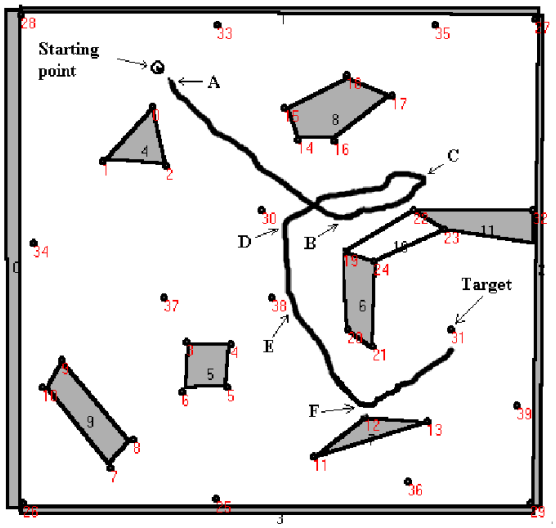


Figure 5.2: Computing diverting targets

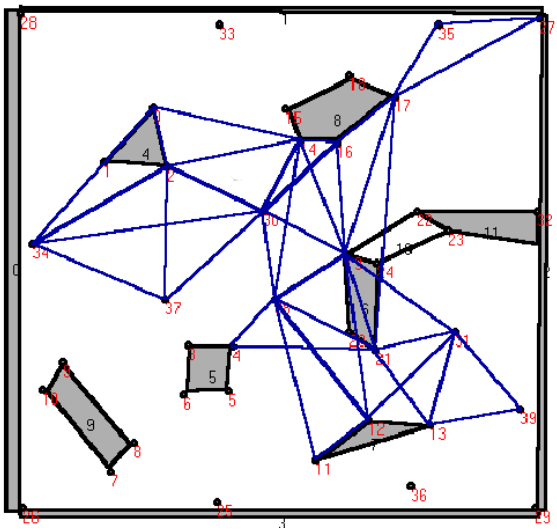

Figure 5.3: Associated map

goes straight towards it, and finally reaches the target.

Someone may ask why the Navigation system computed so many diverting targets, instead of trying to go towards the initial target more frequently. The reason was that the risk was too high very often. This was because of the narrow view field of the camera and the fact that the system was not using Visual Memory, thus, having too few landmarks in sight very often. Although the performance was good enough – the robot reached the target – this behavior of constantly computing diverting targets was not what we really wanted. Moreover, in the situation of the robot being in an area with very few landmarks, possibly seeing only the target, the risk would be very high, but it would not be a wise decision to stop going towards the target and, instead, compute a diverting target. That is why the *Rescuer* agent was modified so that it did not take into account the risk, as presented in the previous chapter.

In Figure 5.3 the map generated while reaching the target is shown. Although internally the *Map Manager* agent stores the map as a graph, here, for clarity, we show the triangular regions corresponding to the nodes of this graph. As can be seen, the map has many overlapping regions, unconnected regions and regions with obstacles inside. Obviously, it is not a very good representation of the environment. In order to obtain a better map of the environment, we modified the mapping algorithm so that it included the constraints presented in Chapter 3. As will be seen in the experimentation with the real robot (Chapter 6), the modified mapping algorithm obtains much better maps.

Although in the simulation we simplified the task in comparison to navigating through a real environment (the Vision system worked perfectly, without any limitation on its view range, the Pilot used sonars for obstacle avoidance), the results obtained, showing that the coordination and mapping worked well, were very promising and encouraged us to keep working on the refinement of the system in order to test it on the real robot. However, even though the main experimentation was to be done with the real robot, we still employed simulation to apply Machine Learning techniques in order to automatically tune the parameters and obtain better performance. In the following sections we describe how we have applied these techniques.

## 5.3 Reinforcement Learning

As mentioned, each of the agents within the Navigation system has a bidding function that is controlled by a set of internal parameters. These parameters need to be tuned in order to achieve the best performance of the Navigation system and of the overall system. Although, as shown in the previous section, we achieved good results with hand-tuned parameters, we wanted to explore if there were other parameter configurations that led to better performance of the system. Adjusting these parameters manually can be very difficult, particularly because of the tradeoffs confronting the top-level agents. An alternative to manual tuning is to employ Machine Learning techniques, specifically Reinforcement Learning methods [64]. In this section, we describe some experiments to test the feasibility of applying Reinforcement Learning within this multiagent system.

Reinforcement Learning is one of the most commonly used learning techniques in Robotics. In Behavior-based architectures learning can be applied at two levels: at the coordination level, where the goal is to apply learning to the coordination system

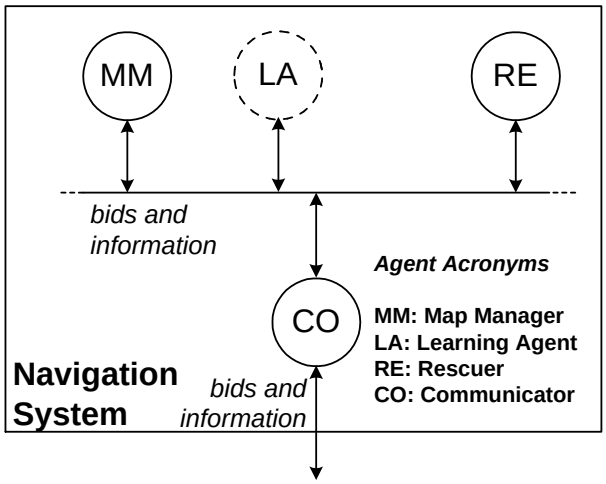


Figure 5.4: Modified navigation system, with the new agent

[44, 28], or at the behavior level, where the goal is to apply learning to the individual behaviors of the system [45, 14]. In our case, we have taken the latter approach [10, 11].

Ideally, we would like to apply Reinforcement Learning to tune all of the parameters of all of the agents in the system. However, this is a very difficult problem, and it is not clear that Reinforcement Learning is the best solution at all levels of the system. Instead, we have chosen to focus on a particular learning problem within the Navigation system. Reinforcement Learning is most needed and most appropriate in cases where there is a complex, quantitative tradeoff between behaviors. In such cases, manual tuning is difficult, and the quantitative criterion of maximizing expected reward, which is the goal of Reinforcement Learning, permits us to represent the tradeoff nicely.

Within the Navigation system, such a tradeoff exists between the *Target Tracker* agent, the *Risk Manager*, and the *Distance Estimator* — recall that we use the initial version of the system, as described in Section 5.1. The *Target Tracker* wants to know the exact heading and distance to the target at all times. This can be achieved by pointing the camera at the target and moving towards it. The *Risk Manager* wants to ensure that the robot is surrounded by a rich network of landmarks so that the robot does not get lost. This can be achieved by pointing the camera in various directions around the robot to identify and track landmarks. Finally, the *Distance Estimator* seeks to know accurate distances to the target landmark. This can be achieved by pointing the camera in the direction of the target while moving the robot orthogonally to the direction of the target. In addition to this conflict, the Navigation system must not monopolize the camera, because the Pilot needs to use it for obstacle avoidance.

Instead of trying to learn the appropriate values for each of the parameters of these agents, we propose to replace the *Target Tracker*, the *Risk Manager*, and the *Distance Estimator* by a new *Learning Agent* that learns its behavior through Reinforcement Learning. We formulate the reward function for this agent so that it is rewarded for

reaching the current target location while minimizing the use of the camera. The two remaining agents have very different roles. The *Map Manager* maintains the beta-coefficient map, but does not bid on actions. The only remaining bidding agent is the *Rescuer*, which is responsible for the higher-level choice of diverting targets whenever the robot becomes blocked. This activity is better-implemented by path planning algorithms than by Reinforcement Learning, so we have not included the *Rescuer*'s responsibilities within the *Learning Agent*. The modified architecture for the Navigation system is shown in Figure 5.4.

### 5.3.1 The Task to be Learned

The task confronting the *Learning Agent* is to choose actions (for both motion and vision) in order to reach the current target location while minimizing the use of the camera. The *Map Manager* informs the *Learning Agent* about the target location. If the robot becomes blocked, the *Rescuer* will ask the *Map Manager* for a new target (a diverting target), and then the *Learning Agent* will take control and choose actions to reach that new target. Once the diverting target is reached, the *Rescuer* may be able to set the current target to be the original goal, and then the *Learning Agent* will attempt to move to that target (and hence, solve the original task).

### 5.3.2 The Reinforcement Learning Algorithm

There are two general types of Reinforcement Learning algorithms: Model-based and Model-free. Model-based algorithms learn a transition model $P(s'|s,a)$ for the environment, where $s$ is the state of the environment at time $t$, $a$ is an action to be executed, and $s'$ is the resulting state of the environment at time $t+1$. Model-based algorithms also learn a reward model $R(s,a,s')$, which gives the expected one-step reward of performing action $a$ in state $s$ and making a transition to state $s'$. Once these models have been learned, dynamic programming algorithms [6] can be applied to compute the optimal value function $V^*$ and the optimal policy $\pi^*$ for choosing actions.

In contrast, model-free methods (such as Q learning and SARSA($\lambda$)) directly learn a value function $V^*$ by repeatedly interacting with the environment without first learning transition or reward models. They rely on the environment to "model itself". For robot learning, however, model-free methods are impractical, because they require many more interactions with the environment to obtain good results. They make sense in simulated worlds where the cost of performing an action can be much less than the cost of storing the transition and reward models, particularly if the environment is evolving over time. But the cost of performing an experimental action with a real robot is very high.

Hence, for our experiments, we have chosen the model-based algorithm known as Prioritized Sweeping [49]. Prioritized Sweeping works as follows. At each time step, the learner observes the state $s$ of the environment, chooses an action $a$, performs the action, receives a one-step reward $r$, and observes the resulting state $s'$. The learner then updates its estimate of $P(s'|s,a)$ and of $R(s,a,s')$ using the observed result state $s'$ and the observed reward $r$. Finally, the learner performs the $k$ most important Bellman

backups to update its estimate of the value function $V$. A Bellman backup in state $s$ is computed as follows:

$$V(s) := \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + V(s')]$$

This is essentially a one-step lookahead that considers all possible actions $a$ and all possible resulting states $s'$, computes the expected backed-up value of each $a$, and assigns the maximum such value to be the new estimate of $V$ at state $s$.

Prioritized Sweeping maintains a maximizing priority queue of states in which it believes a Bellman backup should be performed. First, it performs a Bellman backup for the most recent state $s$. In each Bellman backup, it computes the change in the value $V(s)$ resulting from the backup:

$$\Delta(s) = \left| V(s) - \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + V(s')] \right|$$

After performing the Bellman backup, Prioritized Sweeping considers all states $s^-$ that are known predecessors of $s$, and computes the potential impact $C$ of the change in $V(s)$ on the change in the value of $s^-$ according to

$$C(s^-) = \sum_a P(s|s^-,a)\Delta(s)$$

It then places the state $s^-$ on the priority queue with priority $C(s^-)$. Finally, Prioritized Sweeping performs $k-1$ iterations in which it pops off the state with the maximum potential impact, performs a Bellman backup in that state, and then computes the potential impact of that backup on all predecessor states. In our experiments, $k = 5$. (In our implementation, we actually use the state-action, or $Q$, representation of the value function rather than the state value function $V$. We have described the method using $V$ in order to simplify the presentation.)

Prioritized Sweeping is essentially an incremental form of value iteration, in which the most important updates are performed first. Because every interaction with the environment is applied to update the model, Prioritized Sweeping makes maximum use of all of its experience with the environment. Prioritized Sweeping is an "off-policy" learning algorithm. During the learning process, any exploration policy can be employed to choose actions to execute. If the exploration policy guarantees to choose every action in every state several times, then Prioritized Sweeping will converge to the optimal action-selection policy. We employ $\epsilon$-greedy exploration. In this form of exploration, when the robot reaches state $s$, it executes a random action with probability $\epsilon$. With probability $1-\epsilon$, it executes the action that is believed to be optimal (according to the current value function $V$). Ties are broken randomly.

We represent both the transition model $P(s'|s,a)$ and the reward model $R(s,a,s')$ by three-dimensional matrices with one cell for each combination of $s$, $s'$, and $a$. This technique will only work if the state and action spaces are small. There are two reasons for this. First, the tables must fit into memory. Second, the time required for learning is proportional to the number of cells in these tables, because the *Learning Agent* must

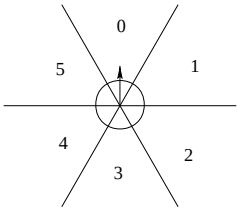


Figure 5.5: Division of environment in sectors. The arrow shows the direction in which the robot is facing (direction of motion, not direction of gaze)

experience multiple visits to each state $s$ so that it can perform each action $a$ several times and gather enough data to estimate $P(s'|s,a)$ and $R(s,a,s')$. Hence, the most challenging aspect of applying Reinforcement Learning is the proper design of the state representation.

**State Representation**

We want the *Learning Agent* to learn a general policy that works for any environment, independently of the locations of the landmarks and targets. Hence, our state representation must not directly employ the locations of the landmarks. Moreover, the robot cannot directly observe the complete state of the environment, which would include the location of the robot, all obstacles, and all landmarks! Instead, the task of the robot is to learn, under conditions of incomplete knowledge, about the locations of obstacles, landmarks, and targets.

State spaces that encode incomplete knowledge are known as "belief state spaces" [15]. The purpose of a belief state representation is to capture the current *state of knowledge* of the agent, rather than the current state of the external world. In our case, the *Learning Agent* is trying to move from a starting belief state in which it knows nothing to a goal belief state in which it is confident that it is located at the target location. Along the way, it seeks to avoid getting lost (which is a belief state in which it does not know its location relative to the target position).

To explain our state representation, we begin by defining a set of belief state variables. Then we explain how these are discretized to provide a small set of features each taking on a small set of values, so that $P(s'|s,a)$ and $R(s,a,s')$ can be represented with small tables.

At any given point in time, the headings to all objects (landmarks and the target position) are divided into six sectors. The field of view of the robot is 60 degrees, so at any point in time, the robot can observe one sector, see Figure 5.5. For each sector, we represent information about the number of landmarks believed to be in that sector and the precision of our beliefs about their headings and distances. This information is gathered from an initial version of the Visual Memory that constantly updates the location of the seen landmarks, and to which the *Learning Agent* has access.

Given these sectors, the following state variables can be defined:

- Distance to target, and its imprecision, $D(t), I_d(t)$

- Heading to target, and its imprecision, $H(t), I_h(t)$
- The landmarks in each sector, $L(s) = \{l_1, ..., l_{n_s}\}$
- Number of landmarks in each sector, $N(s) = \min(4, |L(s)|)$
- Average imprecision of landmarks in each sector, $\overline{I}(s) = \frac{1}{N(s)} \sum_{l \in Best(4,L(s))} I(l)$

We now explain each of these. The distance $D(l)$ to a landmark (or $D(t)$ to the target) is a fuzzy number in the range $[0, \infty]$. The heading to a landmark $H(l)$ (or $H(t)$ to the target) is a fuzzy number with range $[0, 2\pi]$. For each of these, its imprecision ($I_d(l)$ for distance, $I_h(l)$ for heading) is defined by taking the size of the interval corresponding to the 70% $\alpha$-cut of the fuzzy number.

The imprecision of a landmark is computed using the equation 3.3 already given in Section 3.2.2:

$$I(l) = \lambda \cdot \tanh(\beta \cdot I_d(l)) + (1 - \lambda) \cdot \frac{I_h(l)}{2\pi}$$

For an explanation of the equation see the mentioned section.

We summarize the agent's knowledge of the landmarks in each sector by averaging the imprecision of the four most-precisely-known landmarks. The function $Best : N \times 2^L \rightarrow 2^L$ selects a subset, $B = Best(n, L)$, of a group of landmarks, $L = \{l_1, ..., l_m\}$, such that $|B| \leq n \wedge \forall_{l \in B} \forall_{l' \in L-B} I(l) \leq I(l')$. Having 4 landmarks in one sector is already very good, since only 3 landmarks are needed to use the beta-coefficient system network. Furthermore, we do not want these measures to be affected by bad landmarks when we have some that are good enough. That is why we use $Best(4, L(s))$ when computing $\overline{I}(s)$.

**Features**

After computing these state variables, we combine and discretize them to define a small number of features each of which takes on a small number of values. These features define the state space, and they are used to access the tables $P(s'|s, a)$, $R(s, a, s')$ and $V(s)$ in the learning phase, and also to access $\pi(s)$ for policy exploitation.

We employ the following features:

- Target Distance, $D(t)$, discretized to 5 intervals.
- Target Location Imprecision: measure of imprecision on the location of the target, $I(t)$, discretized to 7 intervals.
- Landmark Count: average number of landmarks over the six sectors, $\overline{C} = \frac{1}{6} \sum_{s=0}^{5} N(s)$, discretized to 4 intervals.
- Landmark Imprecision: average imprecision of landmarks' locations in each sector, $\overline{I} = \frac{1}{6} \sum_{s=0}^{5} \overline{I}(s)$, discretized to 7 intervals.

This gives a total of 980 belief states.

**Actions**

Just as Reinforcement Learning requires careful design of the state space to ensure that it is compact, it also requires careful design of the action set to ensure that it is small but also sufficient for the robot to achieve its goals.

Physically, the robot is able to simultaneously perform two types of actions: *moving* actions and *looking* actions. Moving actions make the robot move in a given direction. Looking actions employ the camera to identify or track landmarks in the environment in specified sectors. The Vision system can either search for new landmarks or re-acquire already-detected landmarks, but it is not able to do both things at the same time, because different image processing routines are required for each. In either case, however, the Vision system returns the heading and distance to the landmarks it detects.

An additional constraint on the design of actions is that the Vision system is most effective when the robot is moving in certain directions relative to the landmarks being observed.

Given these constraints, we have designed the following set of actions for the *Learning Agent*:

- Move Blind (MB): move toward the target (i.e., in the direction in which the target is *believed* to be). Do not use the Vision system.
- Move and Look for Landmarks (MLL): move toward the target. Point the camera in the sector that contains the fewest number of known landmarks, and look for new landmarks in this sector.
- Move Orthogonally to Target (MOT): move orthogonally to the direction of the target. Point the camera at the target and attempt to improve the precision of the heading and distance to the target.
- Move and Verify Landmarks (MVL): move toward the target. Point the camera to the sector with the maximum imprecision, $\overline{I}$, and attempt to re-acquire known landmarks and measure their heading and distance more accurately.
- Move and Verify Target (MVT): move toward the target. Point the camera at the target and attempt to re-acquire it and measure its heading and distance more accurately.

These actions should affect the state variables as follows. All actions except MOT make the distance to the target decrease. MB makes all imprecisions grow. MLL should increase the number of detected landmarks. MOT should reduce the imprecision about the target's location, while MVL should reduce the overall imprecision. MVT also reduces the imprecision of the target's location, but not as much as MOT. All actions require that the heading to the target is known (at least approximately). The heading is chosen as the center of the fuzzy interval for $H(t)$. If the heading is completely unknown, the center of this interval is $\pi$. This causes the robot to "pace" back and forth, turning 180 degrees ($\pi$ radians) each time an action is executed.

We have assigned an immediate reward to each action to reflect the load on the Vision system and the motion system. The rewards are negative, because they are costs.

MB is the cheapest action, since it does not use the camera. It has a reward of $-1$. MVL and MVT produce a reward of $-5$, since they make moderate demands on the Vision system. MOT gives a reward of $-6$, because it requires more motion in addition to the same image processing as MVL and MVT. Finally, MLL is the most expensive, with a reward of $-10$, because it must do extensive image processing to search for new landmarks and verify that they are robust to changes in viewpoint.

The system receives a reward of 0 when it reaches the target location. The Reinforcement Learning objective is to maximize the total reward. In this case, this is equivalent to minimizing the total cost of the actions taken to reach the target.

### 5.3.3 Experimentation

We have employed the Webots simulator to perform our experiments. The environment contains a set of landmarks, one of which is designated as the target. There is also a wall that surrounds the region in which the robot is navigating. The landmarks are the only objects in the environment. There are no obstacles, as obstacle avoidance is handled by the Pilot system. However, the robot can be blocked by the landmarks or by the wall. In each trial, the robot starts at a random location in this environment, and it has to reach the target. The trial terminates under three conditions: (a) if the robot reaches the target (and is confident that it has reached the target), (b) if the robot takes 500 steps without reaching the target, or (c) if the robot is blocked. When the trial is finished, the next one begins with another random initial location for the robot.

In order to see if the performance of the system improves after learning, we compared it with a hand-coded policy. The hand-coded policy used the same discretized features as the learning algorithm (Target Distance, Landmark Count, Landmark Imprecision and Target Location Imprecision). The following table shows the policy for choosing an action depending on the values of these features :

| Target Distance | Landmark Count | Landmark Imprecision | Target Loc. Imprecision | Action |
|---|---|---|---|---|
| $high$ | $low$ | $*$ | $*$ | MLL |
| $high$ | $\neg low$ | $high$ | $*$ | MVL |
| $high$ | $\neg low$ | $\neg high$ | $high$ | MOT |
| $high$ | $\neg low$ | $\neg high$ | $\neg high$ | MB |
| $\neg high$ | $*$ | $high$ | $high$ | MVL |
| $\neg high$ | $*$ | $\neg high$ | $high$ | MVT |
| $very\ low$ | $*$ | $*$ | $\neg high$ | MVT |
| $low$ | $*$ | $*$ | $\neg high$ | MB |

where $high$, $low$ and $very\ low$ are defined as follows:

| **Variable** | *very low* | *low* | *high* |
|---|---|---|---|
| Target Distance | $< 1$ | $\leq 2$ | $> 2$ |
| Target Location Imprecision | – | $< 5$ | $\geq 5$ |
| Landmark Count | – | $< 2$ | $\geq 2$ |
| Landmark Imprecision | – | $< 5$ | $\geq 5$ |

The reader should note that this hand-coded policy is not the same as the policy produced by the hand-coded bidding functions described in Chapter 4. We have chosen this policy because it allows us to debug and test the *Learning Agent* separately from the rest of the multi-agent system.

The *Learning Agent* was trained for 2000 simulated trials. At regular intervals, the learned value function was tested by placing the robot in 100 randomly-chosen starting locations, running one trial from each location, and measuring the total reward, the total number of actions, and whether the robot succeeded in reaching the target position. The same set of 100 starting locations was employed in each testing period. The hand-coded policy was also evaluated on these 100 starting locations.

First, let us consider the fraction of successful trials. Figure 5.6 shows that even after only 100 trials, the *Learning Agent* is already out-performing the hand-coded policy. After 2000 trials, the *Learning Agent* succeeds in reaching the target in 84 of the trials, compared to only 24 for the hand-coded policy. From these results we also see that our hand-coded policy was pretty bad. Although we could have tried to rewrite the policy to improve its performance, the results show that Reinforcement Learning can greatly help on solving complex tradeoffs, very difficult to handle manually.

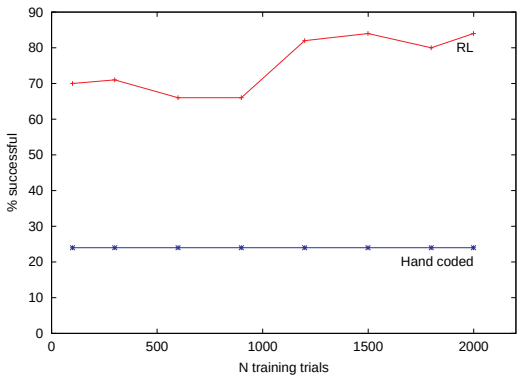


Figure 5.6: Number of successful test trials as a function of the amount of training

A second way of analyzing the performance of the *Learning Agent* is to compute the average reward per trial, the number of actions per trial, and the number of actions of each type. Table 5.1 displays this information after 2000 training trials. Each value is averaged over five test runs. The only difference between test runs is the random number seed for the Webots simulator. We see that while the hand-coded policy receives an average of $-858$ units of reward, the learned policy only receives $-336$ units, which is a huge improvement. In addition, the *Learning Agent* on the average only requires

Table 5.1: Comparison of the *Learning Agent* (LA) and the hand-coded policy (HC) after 2000 training trials.

| | Reward per trial | Actions per trial | MB | MOT | MVT | MVL | MLL |
|---|---|---|---|---|---|---|---|
| HC | -858 | 153.33 | 4.94 | 18.59 | 0.52 | 121.96 | 7.32 |
| LA | -336 | 49.95 | 11.41 | 6.52 | 5.61 | 4.97 | 21.43 |

50 steps to terminate a trial (reach the goal, become blocked, or execute 500 steps) compared to 153 steps for the hand-coded policy. Actually, the *Learning Agent* never terminates because of reaching the 500-step limit.

Table 5.1 contains other interesting information. In particular, we see that the *Learning Agent* has learned to perform fewer MOT and MVL actions and more MB, MVT, and MLL actions. Note particularly that the *Learning Agent* is executing an average of 11.4 MB (Move Blind) actions per trial, compared to only 4.9 for the hand-coded policy. One of the goals of applying Reinforcement Learning was to find a policy that freed the camera for use by the low-level obstacle avoidance routines, and this is exactly what has happened: the hand-coded policy uses the camera 96% of the time, while the *Learning Agent* uses it only 77% of the time. On the other hand, we were surprised to see that the *Learning Agent* chooses to execute the most expensive action, MLL, so often (21.4 times per trial, compared to only 7.3 times per trial for the hand-coded policy). Certainly, it has found that a mix of MLL and MB gives better reward than the combination of MVL and MOT that is produced by the hand-coded policy. The *Learning Agent* spends much more time looking for new landmarks and much less time verifying the direction and distance to known landmarks.

### 5.3.4 Future Work

Although the obtained results show that the *Learning Agent* has learned to select actions to resolve the complex camera tradeoff, we need to integrate it into the overall multi-agent system (as depicted in Figure 5.4), to see if the performance of the whole system is also improved. Even though the *Learning Agent* knows which actions it has to bid for (following the learn policy), it is not clear how its bidding function should be (e.g. constant, depending on the values of $V(s)$).

Some more further work will be focused on the design of the state and feature representation and the set of available actions. Asada et al. [5] proposed a solution for coping with the "state-action deviation problem", in which actions operate at a finer grain than the features can represent, having the effect that most actions appear to leave the state unchanged, and learning becomes impossible. We plan to evaluate the suitability of this approach in our experiments. Regarding the action set design, we found that the set of available actions was maybe too small and some more actions may be needed. We are working on an "action refinement" method [20] that exploits prior knowledge information about the similarity of actions to speed up the learning process. In this approach, the set of available actions is larger, but in order to not slow down the learning, the actions are grouped into subsets of similar actions. Early in the learning process, the Reinforcement Learning algorithm treats each subset of similar actions as a single "abstract" action, estimating $P(s'|s, a)$ not only from the execution of action $a$, but also

from the execution of its similar actions. This action abstraction is later on stopped, and then each action is treated on its own, thus, refining the values of $P(s'|s,a)$ learned with abstraction.

## 5.4 Evolving the Multiagent Navigation System

As we have already mentioned previously, our Navigation system is decomposed into a set of different agents that are responsible for different tasks. Each of these agents has certain parameters that affect its bidding behavior. Trying to manually find the best values for the parameters of the bidding functions is an extremely difficult task. In this section we describe the application of an evolutionary approach to do this optimization.

### 5.4.1 Navigation Tasks

For a given environment we consider two different navigation tasks. Each one of them with a different level of complexity. The best parameter set may change depending on the complexity of the task. We conjecture that the parameters found depend mainly on the complexity of the navigation task and not so much on the structure of the overall environment. This complexity is dependent, though not equal, to the cartographic complexity of the world in which the agent moves, and is based on the following factors:

1. Number of visible landmarks at any time
2. Density of obstacles in the region of navigation
3. Visibility of the target at any time

Using this notion of navigational complexity, the total space of all navigation tasks can be split into two representative classes: going towards the target free of obstacles, and reaching targets located behind obstacles. In our experiments we use clusters $C_1$ (encircled targets in Figure 5.7) and $C_2$ (encircled targets in Figure 5.8) as representatives of the two task complexity classes. The best parameter set is determined for both these classes. The aim of the experiments is to endow the Navigation system of the robot with the capability to switch between these two parameter sets according to the actual task complexity it is facing.

### 5.4.2 The Agents

Although a detailed description of the agents was already given in Chapter 4, as well as the description of the differences between the simulated system and the final system, (given at the beginning of this chapter), we review the parameters of each of the agents:

- **Target Tracker ($\alpha, \beta, \kappa_1, \kappa_2$)**
  - $\alpha$: controls how rapidly the bids for moving towards the target decrease, $bid(move(\theta)) = \kappa_1(1 - I_a^{1/\alpha})$; high values of $\alpha$ make bids increase fast, while low values make bids increase slowly

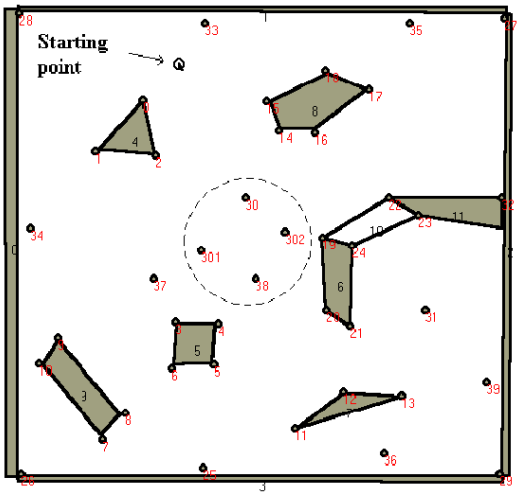


Figure 5.7: Cluster C1

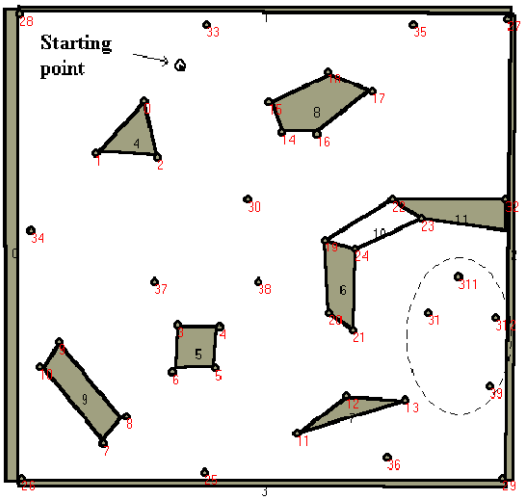


Figure 5.8: Cluster C2

  - $\beta$: controls the shape of the imprecision function, $I_a = \left(\frac{\epsilon_\theta}{2\pi}\right)^\beta$; high values make it increase slowly, while low values make it increase fast
  - $\kappa 1$: maximum value for moving actions bids
  - $\kappa 2$: maximum value for looking actions bids

- **Distance Estimator ($\kappa, \phi, \delta$)**
  - $\kappa$: controls the shape of the distance imprecision function, $I_d = 1 - 1/e^{\kappa\epsilon_t}$; high values of $\kappa$ make the imprecision grow fast, while low values make it increase slowly
  - $\phi, \delta$: controls the *at target* computation; it considers that the robot has reached the target if the upper bound of the $\alpha$-cut of level $\phi$ of the fuzzy number modeling the distance to the target is less than $\delta$ times the body size of the robot

- **Risk Manager ($\gamma_A, \gamma_B, \gamma_r$)**
  - $\gamma_A, \gamma_B$: control the relative importance of the position of landmarks, ahead and around, respectively, used in the risk computation,

$$R = 1 - \min\left(1, q_A\left(\frac{|A|}{6}\right)^{\gamma_A} + q_B\left(\frac{|B|}{6}\right)^{\gamma_B}\right)$$

  - $\gamma_r$: maximum value for looking actions bids

- **Rescuer ($\overline{I}_a, \overline{R}$)**
  - $\overline{I}_a$: imprecision threshold, above which this agent gets active
  - $\overline{R}$: risk threshold, above which this agent gets active

### 5.4.3 The GA algorithm

**Representation**

We seek to optimize the Navigation system with respect to its 10 parameters: *Target Tracker* ($\alpha$, $\beta$, $\kappa_1$, $\kappa_2$), *Distance Estimator* ($\kappa$), *Risk Manager* ($\gamma_A$, $\gamma_B$, $\gamma_r$), and *Rescuer* ($\overline{I}_a$, $\overline{R}$). The *Distance Estimator*'s parameters $\phi$ and $\delta$ are fixed to 0.7 and 2 respectively since they do not affect the efficiency of the system. We use a real valued chromosome, each chromosome being a vector of 10 dimensions (see Figure 5.9). The initial population is generated randomly.

### Evaluation

Each individual in the population specifies a particular parameter set for the system, and is evaluated by running a simulation with the specified parameters in a given environment. Consider that the agent navigates from an initial position $p_0$ to the target cluster C containing the $n$ target positions ($t_1$, $t_2$, ..., $t_n$) and that it takes $d_i$ steps to reach the target $t_i$ from $p_0$ with a success value $s_i$. A threshold is defined for the number of steps that are taken to reach the target, above which the agent is said to have failed in its attempt to navigate to the target (i.e. its success value is 0, otherwise it is 1).

This formalization gives the clues to define the fitness function that permits the selection of the best parameter sets. It is clear that the average cost of reaching a target from the initial position $p_0$ is defined as the summation of the steps required to reach each target divided by the number of targets. That is,

$$\overline{c} = \frac{\sum_{i=1}^{n} d_i}{n}$$

Similarly, we can naturally define the average success value as:

$$\overline{s} = \frac{\sum_{i=1}^{n} s_i}{n}$$

The best behavior for a navigation system is the one that has a high success rate with a low average cost and with a low standard deviation for this average cost, $\sigma_c$. Thus, we define the fitness function as follows:

$$f = \frac{\overline{s}}{\overline{c} + \sigma_c}$$

### Evolution

We follow an elitist approach. That is, from a population of individuals, the fittest individual is passed to the next generation. The remaining individuals form the pool from which the new generation offspring are created. We randomly select two individuals from the mating pool whose fitness is over a randomly determined value. Then we apply crossover and mutation on them to generate new individuals:

```
begin
    counter := 0;
    repeat
        r := generate a random number;
        i := find the first individual whose fitness ≥ r;
        r' := generate a random number;
        i' := find the first individual whose fitness ≥ r';
        apply crossover operator on i and i';
        apply mutation operator on i and i';
        counter := counter+1;
    until counter = population_size / 2
end
```

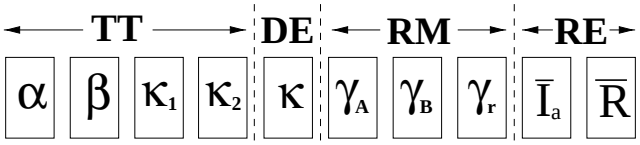


Figure 5.9: Chromosome with the set of parameters

**Crossover**

A simple two point crossover is used with the two parents exchanging their genetic material between two randomly generated breakpoints in the gene string. A point to note is that the chromosomes are broken only at agent boundaries (see Figure 5.9). The idea is that one of the parents may have good genes for a particular agent while the other parent may have good genes for another agent. This way the crossover could result in an offspring having a higher fitness value than both its parents.

**Mutation**

The mutation operator for the genetic algorithm has been adopted from the Breeder Genetic Algorithm [53]. Given any set of parameters as a chromosome, we can view it as a point *x* within a 10 dimensional space. Using our mutation operator, we seek to search for optimality within a "small" hypercube centered at *x*. How small this hypercube is, depends on the ranges in each parametric dimension within which we allow the chromosome to mutate. The parametric dimensions are not homogeneous, hence mutation ranges differ for each dimension, being directly proportional to the variance allowed in that parameter. Another feature of this mutation operator is that while it searches within the hypercube centered at *x*, it tests more often in the very close neighborhood of *x*, the idea being that, while we want to conduct a global search for optimum using our recombination, mutation is used for a more restricted local search. Having understood the broad features which the mutation operator should demonstrate, we formally define the mutation as follows:

Given a chromosome *x*, each parameter $x_i$ is mutated with probability 0.1. The number of parameters being 10 implies that at least one parameter will be probably mutated. Further, given the mutation range for the parameter $x_i$ as $range_i$, the parameter $x_i$ is mutated to the value ${x_i}^*$ given by

$$ {x_i}^* = x_i \pm range_i \cdot \rho $$

As previously discussed, $\rho$ should be such that it lies between 0 and 1 (to generate the hypercube centered at $x$) and also it should probabilistically take on small values so as to test more often in the close neighborhood of $x$. This is realized by computing $\rho$ from the distribution

$$ \rho = \sum_j \alpha_j 2^{-j} $$

where each $\alpha_j$ is probabilistically either 0 or 1.

| | $\alpha$ | $\beta$ | $\kappa_1$ | $\kappa_2$ | $\kappa$ | $\gamma_A$ | $\gamma_B$ | $\gamma_R$ | $\overline{I}_a$ | $\overline{R}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| C1 | 1.731 | 2.03 | 0.314 | 0.493 | 0.355 | 0.240 | 0.521 | 0.054 | 0.386 | 0.215 |
| C2 | 1.231 | 2.12 | 1.0 | 0.564 | 0.178 | 1.377 | 4.39 | 0.707 | 0.871 | 0.906 |

Table 5.2: Optimal parameter values for each of the clusters for one execution of the GA over 100 generations

**Diversity**

The convergence of the genetic algorithm is estimated through its population diversity. Initially, the population has a high diversity since all the individuals are randomly selected. As the algorithm converges, the individuals in the population converge towards the best solution, thus decreasing the diversity. In our case, the individuals are points in a heterogeneous dimension space, with $\alpha$, $\beta$, $\gamma_A$ and $\gamma_B \in \Re^+$ while the other parameters ranging between 0 and 1. Hence we use the Mahalanobis distance measure to determine the diversity of a population [22].

The Mahalanobis distance takes into account the heterogeneity in dimensions and correspondingly scales each dimension while estimating the distance between two points. Given a set of data points $\{z_i\}$ with each data point $z_i$ being an n-tuple $\langle z_{ij} | 1 \leq j \leq n \rangle$, the Mahalanobis distance $d_m$ between two points $z_k$ and $z_l$ is given as

$$d_m(z_k, z_l) = (z_k - z_l)^T \Sigma^{-1} (z_k - z_l)$$

Here $\Sigma$ is the $n \times n$ variance-covariance matrix for the given data points. To compare the diversity of populations across generations, the covariance matrix is computed taking into account all the chromosomes over all generations. The diversity of a population is then calculated as the average Mahalanobis distance of each chromosome from the mean chromosome.

### 5.4.4 Results

The genetic algorithm was run on the two task complexity classes represented by the target clusters $C_1$ and $C_2$ in our simulator. The population size was of 20 individuals, and we ran the genetic algorithm for 100 generations. The initial position was the same for both tasks, with the crossover and the mutation rates being 0.8 and 0.1 respectively. In the algorithm, four of the parameters — $\alpha$, $\beta$, $\gamma_A$ and $\gamma_B$ lie on the positive real axis and hence we have to choose an upper limit on the real line. This upper limit is important since a low upper limit value implies that we implicitly restrict our real valued parameters to that limit, while a high upper limit value may increase the number of generations for which the genetic algorithm may have to be run since the initial random generation will be very disperse. $\alpha$ and $\beta$ are exponents of numbers less than 1 and hence their large values will not be useful. Keeping these factors in consideration, the upper limit value has been fixed to 5 in our simulations.

The genetic algorithm converges to an optimal solution for each cluster as can be seen in Figures 5.10-5.15. By optimal solution we refer to the best solution the algorithm has found, which may not necessarily be the optimal solution to the navigation

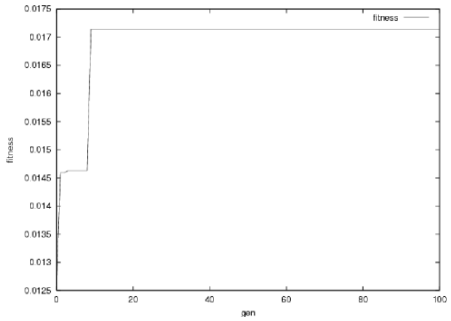


Figure 5.10: Fitness of the fittest individual along generations (cluster $C_1$)

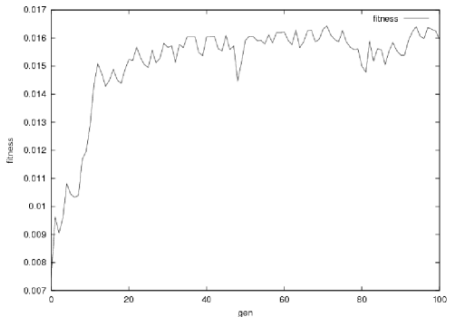


Figure 5.11: Average fitness of the population along generations (cluster $C_1$)

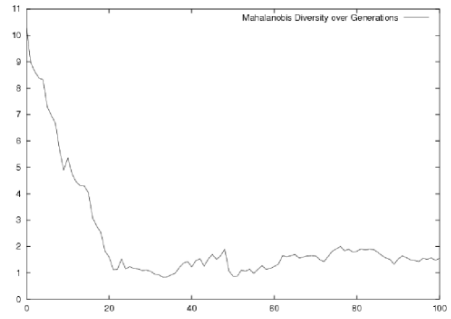


Figure 5.12: Mahalanobis diversity (cluster $C_1$)

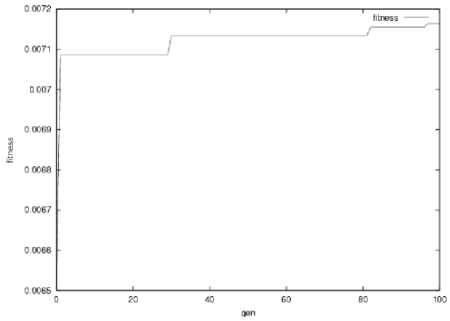


Figure 5.13: Fitness of the fittest individual along generations (cluster $C_2$)

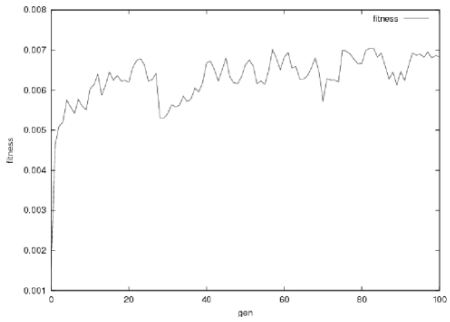


Figure 5.14: Average fitness of the population along generations (cluster $C_2$)

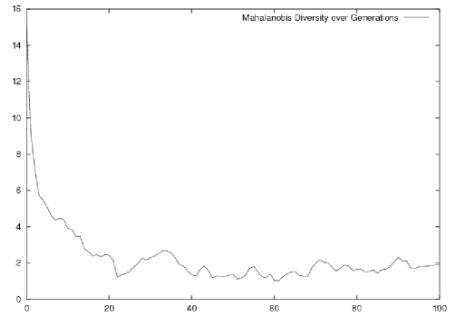


Figure 5.15: Mahalanobis diversity (cluster $C_2$)

| | Going to $C_1$ | | | Going to $C_2$ | | |
|---|---|---|---|---|---|---|
| | $\overline{s}$ | $\overline{c}$ | $f$ | $\overline{s}$ | $\overline{c}$ | $f$ |
| $C_1$ set | 1 | 50.5 | 0.017 | 0.5 | 127.5 | 0.003 |
| $C_2$ set | 0.5 | 42.5 | 0.011 | 1 | 122 | 0.007 |
| HT set | 0.5 | 69 | 0.005 | 0 | – | 0 |

Table 5.3: Results obtained by the different parameter sets

task. The optimal values for some of the parameters differ significantly for the two clusters as shown in Table 5.2. The parameters associated to the bidding function of the *Risk Manager* agent differ the most between the two clusters. This is so because the *Risk Manager* is very sensitive to the complexity of the task. The more obstacles, the higher the risk of losing sight of landmarks.

In order to check the results obtained for each of the clusters, we have tested the two parameter sets found by the genetic algorithm on the two different navigation tasks (going to cluster $C_1$ and going to cluster $C_2$). We have also tested our original parameter set, which we set by hand, on the same two navigation tasks. The results obtained by each set on each of the tasks are shown in Table 5.3. For each task, the mean average success value ($\overline{s}$), average cost ($\overline{c}$) and the fitness value ($f$) is computed. As expected, the parameter set found for cluster $C_1$ performs perfectly when going to cluster $C_1$ and it only reaches the targets of cluster $C_2$ 50% of the time. On the other hand, the parameter set found for cluster $C_2$ reaches the targets of cluster $C_2$ all the times, while it only reaches the targets of cluster $C_1$ 50% of the time. Finally, the hand-tuned parameter set reaches 50% of the time for targets in cluster $C_1$, and never reaches the targets of cluster $C_2$. Therefore, the evolutionary approach has improved the global navigation behavior.

In Figures 5.16 and 5.17 we can see some paths followed by the robot using each of the parameter set on each of the tasks. Successful paths are only shown for those parameter set with a success value of 1. Otherwise, an example of a failing path (marked with a cross at its end) is shown.

### 5.4.5 Future Work

We will analyze the generality, in terms of different environments and starting points, of the parameters obtained by the genetic algorithm. Further work should also focus on designing an agent capable of identifying the complexity of the task being performed, so that the parameters can be switched from one set to another. We will explore the use of Case Base Reasoning techniques on this “situation identifier” agent.

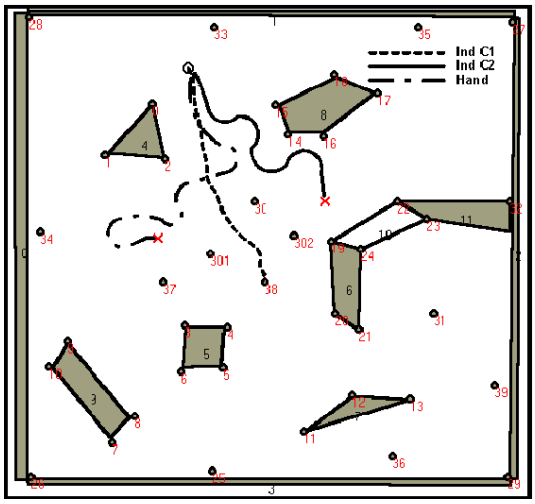


Figure 5.16: Going to cluster $C_1$

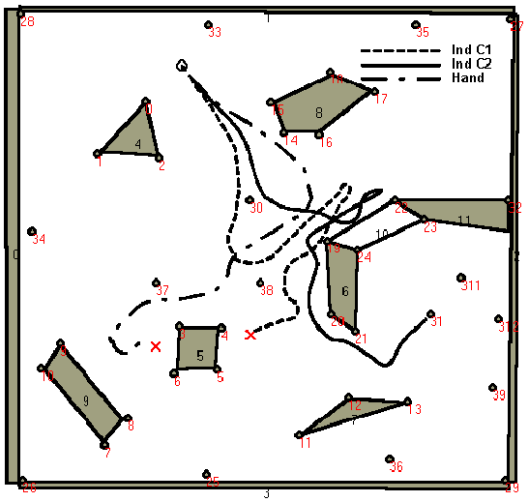


Figure 5.17: Going to cluster $C_2$