Tackling the Premature Convergence Problem in Monte-Carlo Localization

Gert Kootstra and Bart de Boer

Authors:

Gert Kootstra (corresponding author)
Artificial Intelligence
Univesity of Groningen, The Netherlands
Postbus 407
9700 AK Groningen
The Netherlands
Fax: +31 (0)50 363 6687
G.Kootstra@ai.rug.nl

Bart de Boer
Univesity of Amsterdam, The Netherlands
Spuistraat 210
1012 VT Amsterdam
The Netherlands
B.G.deBoer@uva.nl

Keywords: Monte-Carlo localization, particle filter, premature convergence, niching, genetic algorithms

Abstract. Monte-Carlo localization uses particle filtering to estimate the position of the robot. The method is known to suffer from the loss of potential positions when there is ambiguity present in the environment. Since many indoor environments are highly symmetric, this problem of premature convergence is problematic for indoor robot navigation. It is, however, rarely studied in particle filters. We introduce a number of so-called niching methods used in genetic algorithms, and implement them on a particle filter for Monte-Carlo localization. The experiments show a significant improvement in the diversity maintaining performance of the particle filter.

1. Introduction

Many parts of our everyday real-world environment are more or less identical. Many office buildings and apartment complexes consist of identically looking rooms and hallways, as can for instance be seen in the Robotics Data Set Repository (Radish) [16]. If we want our future robots to navigate through these environments, the robot localization techniques need to cope with these symmetries and ambiguous situations. Monte-Carlo localization, based on particle filtering, is currently the most used technique for robot localization (e.g., [28]). A particle filter however cannot deal with ambiguous situations, since it suffers from the problem of premature convergence [2, 28]. The inherent properties of the particle filter make the particle population quickly loose diversity. Because of the loss of diversity, the filter is not able to maintain several possible solutions, needed when there is ambiguity, but it converges to one solution that might not represent the robot's real location. The problem is frequently obscured by using an enormous amount of particles. The KLD method of Fox [9] adaptively changes the population size depending on the spread of the particles measured by the Kullback-Leibler distance. However, it

is unclear if this method is able to maintain diversity. We propose an improvement upon standard Monte-Carlo localization that deals with symmetric environments in a more fundamental and moreover a more computationally efficient way.

Premature convergence is not only a problem in Monte-Carlo localization, but also in genetic algorithms [4, 11]. This is not very surprising, since genetic algorithms and particle filters are very similar techniques. Both algorithms use a set of individuals or particles to represent an underlying probability function. In Monte-Carlo localization this probability function is the probability of the robot's position, in genetic algorithms it is the fitness function. In both methods, a new set of particles or individuals is formed based upon random changes and selection. As a consequence of the similarity between genetic algorithms and particle filters, both techniques face the same problems, but can also benefit from the same solutions as we demonstrate in this paper.

Despite the similarity between both algorithms, the transfer of knowledge from one field to the other has been explored in only a few papers. Higuchi [14], for instance, applied the genetic algorithm's operators *mutation* and *crossover* to the prediction step of a particle filter. Bienvenue *et al.* [2] used fitness sharing, a well known niching method in genetic algorithms, in their particle filter to overcome premature convergence of the population. The stability properties of particle filters and genetic algorithms are discussed in [24]. Finally, Kwok *et al.* [17] aim to preserve diversity by applying crossover and a diversification technique in a robotic Simultaneous Localization and Mapping (SLAM) algorithm. These studies indicate that a systematic exploration of these similarities can be beneficial for both genetic algorithms and particle filters.

We will specifically focus upon the problem of premature convergence. This problem is well-studied in the genetic algorithm field. A number of so-called *niching methods* are developed to tackle the problem [4, 11, 20]. Although these methods proved to be very effective, they are not used in particle filters, with the notable exception of Bienvenue *et al* [2], who use *sharing*, one of

the niching method, in Monte-Carlo filtering. In this paper, we will systematically explore the possibilities to use the niching methods from genetic algorithms in Monte-Carlo localization.

We believe that the proposed methods can be an important improvement for using Monte-Carlo localization in symmetrical environments. More generally, our results will be valid for particle filters at large. Furthermore, we hope to convince the reader of the possibility and relevance of transferring knowledge from the field of genetic algorithms to the field of particle filters.

In the next section, we describe the genetic algorithm and the particle filter and give their similarities. In section 3, we introduce the problem of premature convergence and a number of existing niching methods in the field of genetic algorithms. We demonstrate the applicability of these methods to particle filters for Monte-Carlo localization in section 4, and describe the results in section 5. Finally, we discuss the implications of these results in section 6.

2. Particle Filters and Genetic Algorithms

Particle filters and genetic algorithms are two stochastic search algorithms that use sample points in the search space, that apply variation to these points to *explore* the space, and that use selection and reproduction to *exploit* good solutions. Although the applications of the algorithms might be different, both algorithms, in their essence, are identical.

Particle filters can be traced back to Metropolis & Ulam [22], who invented the Monte-Carlo method. The particle filter tries to represent a distribution by a set of random samples (particles) drawn from that distribution. Through a process of variation, selection and resampling, the particles are redistributed in order to better represent the distribution. Particle filters are widely used for state estimation, for problems such as tracking, and robot localization (e.g. [6]).

The goal of a particle filter is to approximate a distribution by a set of weighted samples (or particles), so that the density of the samples is proportional to the probability density of the distribution. Particle filtering works in three steps: a prediction step, a correction step and a

resampling step. In Monte-Carlo localization, the distribution we try to approximate is that of the position of the robot in its environment. It can be approximated by using knowledge of the previous position of the robot, of the action that was performed, and of the sensor observations that were made. The prediction step is implemented by the *transition model* (what happens to the robot after a given action), and the correction step by the *sensor model* (what would we observe, given the robot's state and the map of the environment). The resampling step selects particles for reproduction based on their weights set by the sensor model. A more detailed description is outlined in section 4.1. For more information on Monte-Carlo localization and particle filters we refer to [6, 28].

Genetic algorithms have been introduced by Holland [15] and a similar technique, evolutionary strategies, by Rechenberg [26]. Inspired by natural evolution, the genetic algorithm implements a population of individuals that is prone to variation and (natural) selection and reproduces generation after generation. Nowadays, different forms of the algorithm are used as effective search mechanisms for a variety of search and optimization problems (e.g., [10, 23]). Within population genetics, a subfield of biology, techniques similar to genetic algorithms are used to study the dynamics of evolutionary processes [12].

The close similarity between particle filters and genetic algorithms has been pointed out by Higuchi [13] (as cited in [14]) and Moral, Kallel & Rowe. [25]. As mentioned before, both particle filters and genetic algorithms are based upon on variation, selection and reproduction. Variation is used in both algorithms for *exploring* the search space. In particle filters, this is taken care of in the prediction step by randomly sampling from the motion distribution. Genetic algorithms use mutation (and crossover) to introduce variation in the population. Secondly, selection and reproduction account for the *exploitation* of good solutions. In particle filters, the probability of selecting a sample is defined by the importance weights, which are assigned to the samples in the correction step (which in Monte-Carlo localization is based on the sensor model). Similarly, in genetic algorithms individuals are selected with a probability that is proportional to

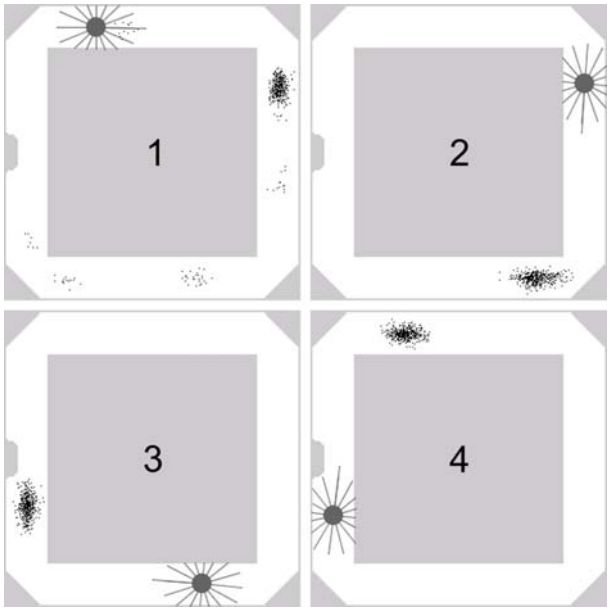

Figure 1. An example of premature convergence in Monte-Carlo localization. In each snapshot, the robot is shown as a circle with its distance measurements. The particles are depicted as black dots. The gray areas are obstacles and the white areas are free space. The environment is highly symmetrical with the exception of an object in the left hallway. In the first situation some diversity is still present in the particle population. However, the diversity quickly disappears as a consequence of random drift in sampling. This results in the system's inability to correctly localize the robot when the disambiguating object is encountered.

their fitness. Finally, individuals in both algorithms reproduce proportionally to their weight (or fitness). Although reproduction is called resampling in particle filters, the implementations are analogous. This demonstrates that particle filters and genetic algorithms are essentially the same. The consequence of this similarity is that both algorithms face similar problems, and that similar solutions can be applied to these problems.

3. Premature Convergence and Solutions

One of the major problems that both genetic algorithms and particle filters suffer from is the problem of premature convergence (see [5, 20] for genetic algorithms and [2] for particle filters).

Premature convergence relates to the loss of diversity in the population. Given a problem with multiple solutions, the entire population will soon converge to just one of these solutions. The reason for this is the presence of randomness in the selection process along with a fixed population size. This causes *random genetic drift* [4, 12, 18 chapter 2]. It can be shown that random genetic drift always results in a homogeneous population, either A or B, with a time to convergence inversely proportional to the population size [12]. The resulting loss in diversity is undesirable, since the maintenance of all potential solutions is crucial to find the global optimum.

As an illustration of the problem of premature convergence, consider the situation in Monte-Carlo localization as shown in Fig. 1, where a loss of diversity results in a sub-optimal solution. As can be seen, the environment is highly symmetric except for a disambiguating object in the left hallway. In this situation, many of the robot's observations could have been made in more than one place. In other words, there are many ambiguous situations, and therefore multiple hypotheses of the robot's position should be maintained. The standard particle filter is used to determine the location of the robot. Initially, the different potential solutions are covered. However, the diversity quickly disappears as a consequence of random genetic drift. In the first situation shown in the figure, there is still some diversity and a number of particles are close to the actual location of the robot. In a short time, however, these particles are lost in favor of only one potential location. As a consequence, the PF settles at a sub-optimal solution and will never be able to find the correct solution. If the PF had maintained more diversity, all ambiguous solutions would be maintained and the optimal solution would be found once the robot observed the disambiguating object.

The main cause of premature convergence is that solutions in different niches (i.e. peaks in the fitness landscape) compete with each other for limited resources (i.e., a limited number of particles). In the remainder of this section we will discuss niching methods from the genetic-algorithm literature that tackle the problem of competition between different niches.

3.1. Niching Methods

The role of niching methods [20] is to find and maintain multiple solutions during the whole search process, even if some of these solutions have lower fitness than others. Niching methods aim to have selection pressure within a region (niche), but not between different regions.

We present three niching methods, i.e., *crowding*, *fitness sharing*, and *local selection*. The first two methods are often used in genetic algorithms and dominate the literature. We included the lesser-known technique of local selection, because we believe that it is especially suited for Monte-Carlo localization.

3.1.1. Crowding

The basic principle of all crowding models is that new individuals enter the population by replacing the most similar individual. Crowding models are so called *steady-state GAs*, meaning that only a subset of the population reproduces at a time, instead of the whole population at once. De Jong's *crowding factor model* [4] was the first model that initiated this branch of niching algorithms. In this model a proportion, called the *generation gap*, of the population is selected for reproduction via fitness proportionate selection. Mutation and possibly crossover are used to generate offspring. For each offspring, a proportion, *cf*, of the population, the *crowding factor*, is randomly sampled. The offspring replaces the most similar individual in that sample, where the similarity can be defined either by the distance between the genotypes or the distance between the phenotypes.

The model is ecologically inspired. Similar individuals in a natural population usually live in the same environmental niche, and thereby compete with each other for limited resources. Dissimilar individuals, *e.g.* different species, on the other hand, live in different niches and do not compete for the same resources.

To get an intuitive idea of why the algorithm promotes diversity, consider the following situation: We have a population with many individuals in niche A and just a few in niche B. In

the reproduction step, a sample of the size of the crowding factor is taken for every individual that reproduces. Since individuals of type A are more frequent in the population, the sample will contain mainly individuals of type A. There is even a fair chance that for a reproducing B, the sample consists of only As. In that case, an individual A is replaced by the individual B, while the reverse rarely happens. This process restores the balance in the population, thus maintaining diversity.

The crowding algorithm turns out to be capable of maintaining multiple solutions, but is of limited use for finding and optimizing multiple solutions. The main reason for this is the fact that old individuals are replaced by new ones irrespective of the mutual fitness values. The only fitness dependent selection is up front when reproducing individuals are selected. This means that fit individuals are selected for reproduction, but they typically will replace similar individuals, which often are similarly fit. This results in fit individuals replacing fit individuals, thereby reducing the selection pressure. To deal with this problem, Sedbrook, Wright & Wright, extended the standard crowding algorithm by selecting the crowding factor sample from the worst part of the population [27]. This method, called *closest-of-the-worst*, only replaces low fitness individuals, thereby increasing the selection pressure.

3.1.2. Fitness sharing

Holland introduced the concept of fitness sharing [15]. The idea behind it is that each niche has an associated total amount of fitness and individuals occupying the same niche share this amount. If the number of individuals exceeds the carrying capacity of a niche, the individuals are better off seeking less crowded niches. A population is stable if each niche contains a number of individuals proportional to its total amount of fitness.

Fitness sharing uses a form of *frequency-dependent selection*. This means that the fitness of a phenotype depends on its frequency in the population relative to other phenotypes. Phenotypes that are rare in the population have a fitness advantage, while common phenotypes have a

disadvantage. This mechanism is often found in nature. Consider, for instance, a predator that can specialize in hunting one of two prey types. It is then most advantageous for the predator to specialize in hunting the most common prey, thereby reducing the fitness of that prey and giving a fitness advantage to the rare type. This type of *negative* frequency-dependent selection results in the balancing of the population and the maintenance of diversity.

Sharing is an effective niching method, which is capable of finding and maintaining multiple solutions. The main drawback is the computational cost of calculating the niche count; the method as it is presented above has $\Theta(n^2)$ complexity. The computational complexity can be reduced by estimating the niche count from a fixed sized sample of the population instead of using the entire population [11]. This results in $\Theta(k \cdot n)$ complexity, where k is the size of the sample. Although this optimization yields good results for the required amount of computation, it comes with the cost of loosing some of the diversity maintaining powers. This is caused by the introduction of additional sample variance, which is a source for extra random genetic drift.

3.1.3. Local selection

The standard genetic algorithm and particle filter have a fixed population size. As discussed, this is one of the causes of premature convergence. The above-mentioned niching methods compensate for this phenomenon in different ways, but keep the population size fixed. The *local-selection algorithm* [21], on the other hand, has a flexible population size. Over time, an individual accumulates "fitness". The rate of accumulation depends on the individual's quality and on the number of nearby other individuals. Based on the accumulated fitness an individual will reproduce or die. The absolute threshold, the flexible population size, and the sharing of fitness among neighboring individuals, eliminate the inherent competition between individuals of different niches, and therefore minimize the influence of premature convergence.

Menczer, Degeratu & Street implemented this as the *evolutionary local-selection algorithm* (ELSA) [21]. We based our algorithm that is described in section 4.7 on ELSA. It is interesting to

note that similar algorithms are used in the field of individual-based modeling, for instance in the Sugarscape simulation [8].

Local selection changes the population size dynamically. The size depends on the carrying capacity of the fitness landscape, as the number of individuals per solution is proportional to the fitness of that solution. Besides the diversity maintenance capabilities, the method has other useful properties for Monte-Carlo localization. MCL has to deal with dynamic fitness landscapes, since the position of the robot constantly changes. Depending on the complexity of the situation, more or fewer members in the population are needed, which is addressed by local selection. Dynamic population size in Monte-Carlo localization is also studied by Fox [9] using KLD-sampling.

In contrast with the other mentioned niching methods, local selection does not select individuals for reproduction by comparing their fitness with the rest of the population in a stochastic selection process. Instead, reproduction and death of an individual only depend on the individual's local environment. Individuals survive as long as the environment is capable of supporting them. Therefore, this niching method is less prone to random genetic drift. Furthermore, the algorithm has a computational complexity of $\Theta(n)$. However, since n is adaptive, experiments need to show whether this really is a computational improvement over the other niching methods. A disadvantage of local selection is the fact that additional parameters need to be set in order to control the magnitude of the population size and the birth and death rate.

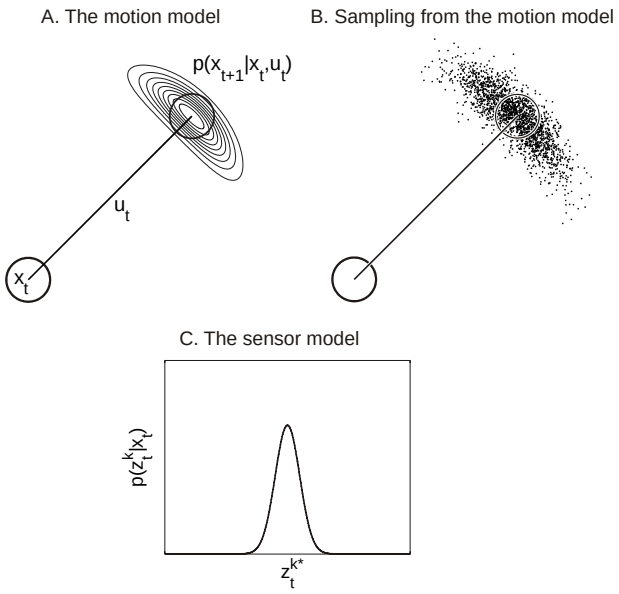

Figure 2. Diagram A shows the distribution of the motion model $p(x_{t+1} \mid x_t, u_t)$. The distribution has a banana shape, caused by the Gaussian distribution on the translation and rotation of the robot. Diagram B shows 1000 samples of this distribution. The sensor model is shown in diagram C. The probability $p(z_t^k \mid x_t)$ is defined by a Gaussian function with the robot's sensor reading, z_t^k, as variable, the particle's reading, z_t^{k*}, as mean and σ_s^2 as variance.

4. Implementation for Monte-Carlo Localization

We have implemented six niching algorithms for Monte-Carlo localization based on the three niching methods given in section 3.1. We first describe the standard particle filters, followed by the six niching algorithms.

Table 1. Functions for the motion model, the sensor model and stochastic uniform sampling.

Algorithm: **SAMPLE_MOTION_MODEL(** u_t, x_t **)**

$u_t = \langle u_t^{trans}, u_t^{rot} \rangle$: robot's motion

$x_t = \langle x, y, \alpha \rangle$: position in polar coordinates

1 $\delta_{trans} \leftarrow$ sample_from($\mathrm{N}(u_t^{trans}, \sigma_\tau^2)$)
2 $\delta_{rot} \leftarrow$ sample_from($\mathrm{N}(u_t^{rot}, \sigma_\gamma^2)$)
3 $\hat{x} \leftarrow x + \delta_{trans} \cdot \cos(\alpha + \delta_{rot})$
4 $\hat{y} \leftarrow y + \delta_{trans} \cdot \sin(\alpha + \delta_{rot})$
5 $\hat{\alpha} \leftarrow \alpha + \delta_{rot}$
6 **return** $x_{t+1} = \langle \hat{x}, \hat{y}, \hat{\alpha} \rangle$

Algorithm: **SENSOR_MODEL(** z_t, x_t **)**

1 $q \leftarrow 0$
2 $z_t^* \leftarrow$ simulate_observation(x_t)
3 **for** all sensors k
4 $p \leftarrow \frac{1}{\sqrt{2\pi\sigma_s^2}} e^{-\frac{1}{2}\frac{\left(z_t^k - z_t^{k*}\right)^2}{\sigma_s^2}}$
5 $q \leftarrow q \cdot p$
6 **end**
7 **return** q

Algorithm: **SAMPLE(** S_t, N **)**

Stochastic Universal Sampling (Baker, 1987)

$S_t = \left\{ p_t^i = \langle x_t^i, w_t^i \rangle \mid i = 1, \ldots, n \right\}$:population at time t .

1 $S_{t+1} \leftarrow \varnothing$
2 $i \leftarrow 1$
3 $c \leftarrow w_t^i$
4 $r \leftarrow$ rand($0, N^{-1}$)
5 **for** $n \leftarrow 1$ to N
6 **while** $r > c$
7 $i \leftarrow i + 1$
8 $c \leftarrow c + w_t^i$
9 **end**
10 $S_{t+1} \leftarrow S_{t+1} \cup p_i$
10 $r \leftarrow r + N^{-1}$
11 **end**
12 **return** S_{t+1}

4.1. Algorithm 1: The Standard Particle Filter

Let $S_t = \{\langle x_t^i, w_t^i \rangle \mid i = 1,...,n\}$ be the population at time t, where x_t^i is the state of particle i and w_t^i is the importance weight of that particle. At $t = 0$, the particles are usually uniformly randomly distributed over the search space with equal importance weights. Every iteration, the algorithm takes the previous population S_{t-1}, the action performed u_{t-1} and the observation z_t. The basic particle filter consists of three steps. First, the motion model repositions the particles. Then, the sensor model determines the weight of the particles. And finally, the set of particles is resampled, and some particles are selected for reproduction. In the following paragraphs, the implementation of these steps is described in detail. The pseudo-code the three steps is given in Table 1, and that of the total particle filter in Table 2.

Motion Model. Every time step, the particles are repositioned based on the action of the robot. As we only want to investigate the dynamics of the particle filter, we use a relatively simple model. In this model, the new position has a mean value equal to the sum of the old position and the action of the robot. Additional Gaussian noise is added to the translation and rotation of the robot. This results in a distribution, $P(x_{t+1} \mid x_t, u_t)$, similar to that depicted in Fig. 2A. The noise in the motion model reflects the expected noise of the robot's action. It is, however, important that the motion model overestimates this noise, so that the particle filter effectively explores the search space. In the experiments we used standard deviations of $\sigma_T = 2$ and $\sigma_\gamma = 0.2$ for translation and rotation respectively. The next position of the particles is determined by sampling from the motion distribution (see also Fig. 2B).

Sensor Model. The sensor model determines the weight of each particle by calculating $w_t^i = P(z_t \mid x_t^i)$, the likelihood that the robot makes observation z_t at position x_t^i. Using ray tracing in the map, an estimate is made of the expected value of each sensor *k* if the robot were at location x_t^i, $z_t^{k*} = f_{RT}\left(x_t^i\right)$, where f_{RT} is the ray-tracing function. The weight of each particle is

then calculated by the product of the individual likelihoods, $P(z_t^k \mid z_t^{k*})$, that the robot's sensor value z_t^k resembles the particle's sensor value z_t^{k*}. This resemblance is calculated using the density function of the Gaussian distribution, with z_t^k as variable, z_t^{k*} as mean and σ_s^2 as the sensor model variance (see Fig. 2C). This results in:

$$P(z_t \mid x_t^i) \quad = \prod_{k \in \text{sensors}} P(z_t^k \mid z_t^{k*}) \quad = \prod_{k \in \text{sensors}} N(z_t^k; z_t^{k*}, \sigma_s^2) \tag{4.1}$$

The sensor model that we describe here consists of a simple Gaussian distribution for a sensor *hit*. Thrun, Burgard & Fox. [28, chapter 5] describe a more elaborate sensor model which better models a real distance sensor. This sophisticated sensor model, however, is unnecessarily complicated for our purposes.

Resampling. Genetic algorithms and particle filters strongly depend on sampling the population. The loss of diversity is partly a consequence of the randomness in sampling. Furthermore, the speed of convergence is proportional to the variance of the sampling method [20]. Therefore, we used a sampling method with low variance throughout this paper. As a substitute for roulette wheel selection, which has a relatively high sample variance, we used *stochastic universal selection* (SUS), proposed by Baker [1]. SUS differs from ordinary roulette wheel selection in that it does not choose a random location on the roulette wheel for every new selection, but initially chooses only one random point on the roulette wheel, and then selects new samples by moving the wheel with fixed-sized steps. Both selection methods have the same expected values, but the variance of SUS is reduced. The stochastic universal selection algorithm is given in Table 1. An additional advantage of SUS is that its complexity for a population of size *n* is $\Theta(n)$, whereas the complexity of roulette wheel selection is $\Theta(n^2)$.

Table 2. Pseudocode of the standard Particle Filter and the additional code for sharing and frequency-dependent selection.

Algorithm Standard particle filter

$S_t = \left\{ p_t^i = \left\langle x_t^i, w_t^i \right\rangle \mid i = 1, \ldots, n \right\}$: population at time t.

x_t^i : position of particle i at time t.

w_t^i : weight of particle i at time t.

u_t : motion of the robot at time t.

z_t : observation of the robot at time t.

```
1  S_0 ← randomized_population
2  repeat
3    for all particles i
4       x^i_{t+1} ← SAMPLE_MOTION_MODEL( u_t, x^i_t )
5       w^i_{t+1} ← SENSOR_MODEL( z_{t+1}, x^i_{t+1} )
6    end
7    S_{t+1} ← SAMPLE( S_t, n )
8  until(finished)
```

Algorithm Sharing – 20%

$$5.1\ w_{t+1}^i \leftarrow w_{t+1}^i \Big/ \sum_{j=0}^{20\% \cdot n} \frac{1}{\mathrm{dist}(i, \mathrm{rand\_particle})}$$

Algorithm Frequency Dependent Selection – 20%

$$5.1\ w_{t+1}^i \leftarrow w_{t+1}^i \cdot \sum_{j=0}^{20\% \cdot n} \mathrm{dist}(i, \mathrm{rand\_particle})$$

Algorithm Frequency Dependent Selection – 1

$$5.1\ w_{t+1}^i \leftarrow w_{t+1}^i \cdot \mathrm{dist}(i, \mathrm{rand\_particle})$$

The presented motion and sensor model are used in all variants of the Monte-Carlo localization algorithms discussed below. The presented sampling method is used for resampling in the standard particle filter, the sharing algorithm and both frequency-dependent selection algorithms. Furthermore, it is used for fitness proportional selection in the crowding and closest-of-the-worst algorithms.

4.2. Algorithm 2: Crowding

Where the standard particle filter uses stochastic universal sampling for selection and reproduction, the crowding algorithm uses a rather different approach. After the motion and sensor model are applied to all particles, only a proportion of the population is selected for reproduction. A proportion of gg particles is selected with fitness proportionate selection using SUS. For each selected particle, a proportion, *cf*, of the population is uniformly sampled. The selected particle replaces the particle in the *cf*-sample with the shortest distance in Euclidean space. In the experiments, we used a generation gap of $gg = 0.2$ and a crowding factor $cf = 0.01$. The pseudo-code is given in Table 3.

The complexity of crowding is quadratic, $\Theta(gg \cdot cf \cdot n^2)$. However, since gg and cf are small proportions of the population, it is not very drastic for small population sizes. In practice, the algorithm was even faster then the standard particle filter for the maximum population size that we used ($n = 2500$). This is caused by the fact that only $gg \cdot n$ copy functions need to be applied every time step.

Table 3. Pseudocode for crowding and the additional code for closest-of-the-worst. In the experiments, we used $gg = 0.2$ and $cf = 0.01$. For the sampling in line 3, we used stochastic universal sampling.

Crowding

```
1   S_0 ← randomized_population
2   repeat
3     for all particles i
4       x^i_{t+1} ← SAMPLE_MOTION_MODEL(u_t, x^i_t)
5       w^i_{t+1} ← MOTION_MODEL(z_{t+1}, x^i_{t+1})
6     end
7     G_t ← SAMPLE(S_t, gg · n)
8     for all particles i in G_t
9       C ← uni_sample(S_t, size ← cf · n)
10      j ← argmin_{k∈C} (dist(i, k))
11      p^j_t ← p^i_t    // replace j by i
12    end
13  until(end)
```

Closest-of-the-Worst

```
9   Ŝ_t ← worst_particles(S_t, size ← n/3)
9.1 C ← uniform_sample(Ŝ_t, size ← cf · n)
```

4.3. Algorithm 3: Closest-of-the-Worst

The closest-of-the-worst algorithm applies more selection pressure than standard crowding. Instead of a random uniform selection of the *cf*-sample over the whole population, the *cf*-sample is taken from the worst third of the population. This increases the complexity of the algorithm, because the population needs to be sorted on its weights, adding $n \cdot \log n$ steps. The algorithm is given in Table 3.

4.4. Algorithm 4: Sharing

This algorithm is an extension of the standard particle filter. It introduces the sharing of weights between particles that are close together. We base our algorithm on that of Goldberg and Richardson [11], who implemented fitness sharing by calculating the *shared fitness*, $\hat{w}_t^i$, of individual i at time *t*, as the normal fitness value, w_t^i, divided by the *niche count*:

$$\hat{w}_t^i = \frac{w_t^i}{\sum_{j=1}^{n} s\big(d(i,j)\big)} \tag{4.2}$$

where $d(i,j)$ is the Euclidean distance between particle *i* and *j*. The niche count is the sum of all *sharing function* values, $s(d)$, between individual i and all n members of the population.

In the original sharing algorithm [11], a parameter defining the niche radius, r, is used. Inspection of the algorithm on some pilot experiments showed the best result with high values of r. With a small radius, many irrelevant clusters appear. Moreover, it is desirable to reduce the number of free parameters in the models. We therefore decided to eliminate the niche radius and let the particles share their weights with all other particles, but more with nearby particle and less with distant particles. This results in the following *sharing function*:

$$s(d(i,j)) = \frac{1}{d(i,j)} \tag{4.3}$$

Originally, the weight of particle i is adjusted by dividing it by the sum of $s(d(i,j))$ over all particles j in the population. To reduce the computational costs, we choose to calculate the niche count by summation over a fraction χ of randomly selected particles, instead of over all particles:

$$\hat{w}_t^i = \frac{w_t^i}{\sum_{j=0}^{\chi \cdot n} s\left(d\left(i, \text{rand}(0,n)\right)\right)} \tag{4.4}$$

where w_t^i and $\hat{w}_t^i$ are, respectively, the weight and adjusted weight of particle i at time t. In our experiments, we used $\chi = 0.2$. Although this reduces the computational demands, the complexity is still quadratic, $\Theta(\chi \cdot n^2)$. Table 2 shows the code for sharing in addition to the standard particle filter.

4.5. Algorithm 5: Frequency-Dependent Selection

The rationale behind frequency-dependent selection is that infrequent particles have a fitness advantage, whereas frequent particles do not have this advantage. Sharing, as described above, is a possible implementation of frequency-dependent selection. However, it can also be implemented differently. Instead of dividing a weight *i* by the niche count, the weight can be recalculated by multiplying it with the distance from *i* to the rest of the population. Like in the sharing algorithm, we used the distances towards a fraction χ of randomly selected particles to reduce the amount of computation. We used $\chi = 0.2$ in our experiments. This gives:

$$\hat{w}_t^i = w_t^i \cdot \sum_{j=0}^{\chi \cdot n} d\left(i, \text{rand}(0,n)\right) \tag{4.5}$$

The complexity of the algorithm, like in sharing, remains quadratic, $\Theta(\chi \cdot n^2)$. The code for frequency-dependent selection is given in Table 2.

4.6. Algorithm 6: Frequency-Dependent Selection, Sample Size=1

In the previous algorithm, the frequency of a particle is determined by taking 20 percent of the population into account. Although this reduces the amount of computation, the algorithm still has

a quadratic complexity. To reduce the complexity to $\Theta(n)$, we need to measure frequency by sampling a fixed size from the population. In this algorithm, we chose to have a sample size of one particle. This means that the weight of the particles is multiplied by the distance towards one randomly selected particle. The algorithm is described in Table 2.

We should mention that a larger sample size is expected to perform better. However, as it is not our aim to find the best tradeoff between accuracy and computation, we chose to show the simplest possible solution for frequency-dependent selection. The sharing algorithm could be adjusted in a similar way to also have a sample size of one. Although this has not been tested, one can expect a similar change in performance.

4.7. Algorithm 7: Local Selection

Local selection has a different method to fight premature convergence. Instead of compensating for the inherent competition between different niches as is done by algorithms 2–6, local selection adapts the population size. This removes the inherent competition for resources between different niches, and lets reproduction and death depend only on the carrying capacity of the environment. The pseudo code of local selection for Monte-Carlo localization is given in Table 4.

As is necessary for local selection, the world is divided into bins to count the number of particles in a local area. These bins are of size 2 by 2 by 36º (the total environment is 150 × 150 × 360º). The algorithm works in two sweeps through the population. In the first sweep, the motion and sensor model are applied to all particles. At the same time, the bins are filled. In the second sweep, the energy of particles is calculated. The accumulated energy, E_t^i, is increased by E_{in}, which is the weight of the particle divided by the bin count,. The accumulated energy is furthermore decreased by a fixed amount, E_{out}. Depending on the energy of a particle, it either reproduces (if $E_t^i > \theta$), dies (if $E_t^i < 0$) or continues living (otherwise). If the particle reproduces, an exact copy is added to the population. Both particles share the fitness of the parent. θ and

E_{out} together influence the population size. The rate of generating new individuals, is set by θ. E_{out} controls the rate with which individuals die. The complexity of the algorithm is linear in the number of particles $\Theta(n)$, but two sweeps through the particle population are needed, one for filling the world bins and one for reproduction.

We changed the original ELSA algorithm [21] with respect to two points. Firstly, Menczer *et al.* propose to use a fixed amount of energy that enters the environment every time step. This fills an energy reservoir. If the reservoir is empty, the members of the population cannot accumulate more energy. This energy replenishment sets a maximum size to the population. However, we would like the maximum population size to emerge from the fitness landscape. We therefore omit the limited energy replenishment. Note that this does not mean that the population size can grow indefinitely. Secondly, we supply the members of the initial population with θ (the threshold for reproduction) energy instead of half of this amount, as used in [21]. The reason for this is that it gives the initial population a bit more time to explore the environment. If a particle does not gain any energy itself, it will die in $\frac{1}{2}\theta / E_{out}$ time steps in the original algorithm. With our settings, this result in a life expectancy of 2.5 time steps. This gives the particles too little time to explore the environment, thereby increasing the probability that none of the particles near a potential robot position finds this position before it "starves". An initial life expectancy twice as big yields better results.

After some initial experimental runs, it turned out that the best results were achieved when E_{out} depended on θ. We had good results with $E_{out} = 0.2 \cdot \theta$. A smaller fraction results in too many particles, which makes it infeasible to achieve real-time performance. Higher fractions cause the particles to disappear quickly, resulting in a population that is too small. Altogether, this results in an algorithm with θ as the only variable to control the population size.

Table 4. Pseudo code for the local-selection algorithm. In the experiments, θ was the variable and $E_{out} = 0.2 \cdot \theta$. The used value of E_{out} was derived from a number of pilot experiments.

Local Selection

$S_t = \left\{ p_t^i = \left\langle x_t^i, E_t^i \right\rangle \mid i = 1, \ldots, n \right\}$:population at time t.

E_t^i : energy of particle i at time t.

```
S_0 ← randomized_population
for all particles i
  E_t^i ← θ
repeat
  for all particles i
    x_{t+1}^i ← SAMPLE_MOTION_MODEL( u_t, x_t^i )
    w_{t+1}^i ← MOTION_MODEL( z_{t+1}, x_{t+1}^i )
    b_i ← get_world_bin(i)
    wbin[b_i] ← wbin[b_i] + 1
  end
  for all particles i
    E_in ← w_{t+1}^i / wbin[b_i]
    E_{t+1}^i ← E_{t+1}^i + (E_in − E_out)
    if( E_{t+1}^i > θ )
      î ← copy(i)
      E_{t+1}^i ← E_{t+1}^î ← E_{t+1}^i / 2
      S_{t+1} ← S_{t+1} ∪ { p_{t+1}^i, p_{t+1}^î }
    else if( E_{t+1}^i > 0 )
      S_{t+1} ← S_{t+1} ∪ p_{t+1}^i
    else
      // particle i dies
    end
  end
until(finished)
```

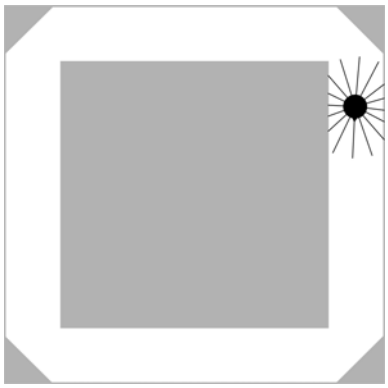

Figure 3. The map with square symmetry used in the experiments.

5. Experiments

We tested the niching methods using a robotic simulation, in order to create well-controlled experiments and be able to focus on the niching methods. A path to an implementation on real robots is given in section 7.1. Different experiments were conducted to test the diversity maintenance, compactness, and estimation accuracy of the methods.

5.1. The Robotic Simulation

The map we used for the experiments has square symmetry (see Fig. 3). In this symmetric environment there are always four ambiguous situations for the robot. The size of the map is 150 by 150. The robot in the simulation has 16 distance sensors, whose values are determined by ray tracing in the map, with a maximum distance of 20. Gaussian noise is added to the sensor readings with standard deviation of 1.0. Furthermore, the robot has differential drive, with which it can navigate through the environment. The robot provides odometry information, consisting of the translational and rotational speed of the robot measured in units per time step and radians per time step respectively. There is Gaussian noise on the odometry, with standard deviation of 1.0 and 0.04 respectively. The robot is controlled by a simple obstacle avoidance behavior, similar to Braitenberg Vehicle 3b, the “Explorer” [3]. The default translational speed is 8 units per time step.

Table 5. The carrying capacity using the local-selection algorithm with different values of θ.

θ	0.7	0.65	0.6	0.55	0.5	0.45	0.4	0.35
n	232	333	499	725	959	1277	1675	2236

5.2. Experimental Setup

5.2.1. Diversity Maintenance

The goal of the experiments is to show that solutions for premature convergence in genetic algorithms can successfully be applied to Monte-Carlo localization. Moreover, we would like to measure the performance of the niching algorithms in terms of their ability to maintain diversity. This is the reason why the environment has square symmetry. It provides four ambiguous situations to the robot. The ability of the niching algorithms to maintain diversity is measured by the number of timesteps (with a maximum of 500) over which they maintain all four possible solutions. The performance is measured as a function of the number of hypotheses. For algorithm 1–6, the performance is measured for 100, 200, 500, 1000, 1500, 2000 and 2500 particles. For algorithm 7, the situation is a bit more complicated, since the population size is not constant. It is, however, possible to control the population size by the threshold θ, as discussed in section 4.2. To test the carrying capacity for different thresholds, we ran experiments with large numbers of initial particles. The reason to use a large number of initial particles is that the carrying capacity is strongly influenced if one of the possible solutions is not covered in the initial phase. The carrying capacities for different thresholds are given in Table 5. In the main experiments, we used these thresholds, and an initial number of particles equal to the associated carrying capacity.

We consider that a solution is maintained by the particle filter if at least one particle is in its vicinity, where vicinity is defined as a sphere with radius 10 centered on the location and orientation of that solution, as calculated using the *real* position of the simulated robot. To be able to compare location and orientation, a difference of 180° in orientation is considered commensurate to a distance in location of 50. Two performance measures are used: the percentage of successful runs, and the time to premature convergence. A successful run is a run where the particle population maintains all four ambiguous solutions throughout the whole period of a run (500 cycles). The time to premature convergence is the time, measured in cycles, when

the particle filter loses one or more of the four potential solutions. Each run is terminated after 500 cycles. If no premature convergence has occurred, the time to convergence is set to 500. A total of 100 runs is used to test all algorithms for each population size. To estimate variation in the percentage of successful runs, the 100 runs are split up in 10 chunks of 10 runs, over which the variance is calculated.

5.2.2. Compactness of the Particle Subpopulations

Besides the performance measures for diversity preservation, we measured the compactness of the particle subpopulations. The compactness is an interesting measure, since it tells us something about the uncertainty of the niching methods. A higher compactness of a subpopulation relates to a lower uncertainty. The preferred compactness is that which best reflects the positioning uncertainty considering the sensory and odometric noise of the robot. Since this is well estimated by the standard particle filter, we will take its compactness as the baseline. The niching methods should not result in particle populations that are too incompact. Consider for instance a niching method that would simply uniformly place particles on the map. That method would maintain all possible solutions, but its total lack of compactness would make it impossible to correctly estimate the robot's position. Neither should it result in too compact populations, since that would reduce the noise robustness of the filter. We used two measures for compactness. First, the proportion of all particles that are within the vicinity, $\rho = 10$, of the four optima. The distance from each particle to the nearest optimum is calculated by the Euclidean distance in three dimensions, the x- and y- values of the location as well as the orientation. The proportion of particles within the vicinity gives a measure of the compactness of the subpopulations. The second measure is the mean sum of squared errors (MSSE), where the error is taken as the distance towards the nearest optimum. This gives the variance of the particles in the subpopulations, which is inversely related to the compactness of the subpopulations.

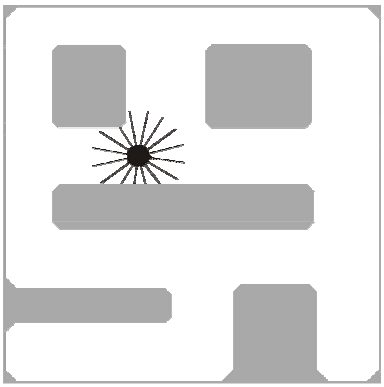

Figure 4. The non-ambiguous map used in experiments C.

5.2.3. Estimation Accuracy

Finally, we tested the different niching methods on their accuracy in estimating the robot's position. Although this research focuses on the ability of the different algorithms to maintain diversity, this should not conflict with the power to correctly estimate the position of the robot. We tested the estimation performance on the symmetrical map that is used in the previous experiments, as well as on a non-symmetrical map (see Fig. 4). The latter does not provide any long-lasting ambiguous situations for the robot. To estimate the position of the robot, we used a kernel-density estimation [7]. The error is then measured by the distance from the estimation to the nearest of the four possible robot positions in the symmetrical map. In the non-symmetrical map, the error is the distance from the estimation to the robot's position. This method estimates the position of the robot by finding the highest density of particles.

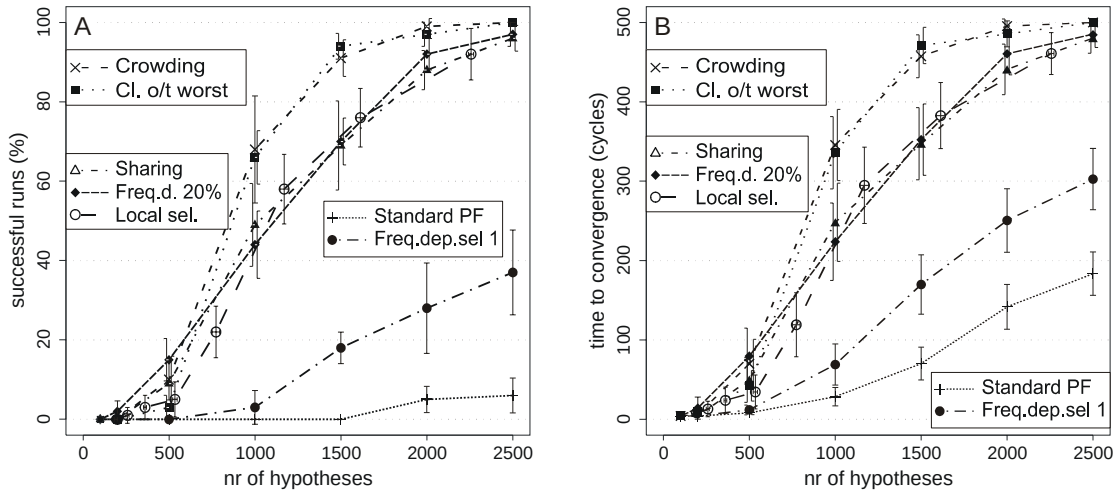

Figure 5. (A) The percentage of runs without convergence plotted against the number of hypotheses. The error bars give the 95% confidence intervals. Means and confidences are calculated from 10 sets of 10 runs. The number of hypotheses for the local-selection algorithm is variable and depends on θ. The corresponding horizontal error bars show the 95% confidence interval for the average number of hypotheses per run. (B) The time to convergence as a function of the population size. A run is considered converged if one or more of the four optima are lost. The run has a maximum length of 500 cycles. Non-converged runs receive a time to convergence of 500. The error bars give the 95% confidence intervals. The data comes from a total of 100 runs.

6. Results

6.1. Diversity Maintenance

The results of the experiments are shown in Fig. 5. (A) shows the percentage of successful runs plotted against the number of particles. (B) gives the time to convergence plotted against the number of particles. The error bars show the 95% confidence intervals. All graphs in both plots show an increase in performance with the number of hypotheses. This is expected, since the time to premature convergence is extended with larger population size. For small numbers of

hypotheses, all algorithms perform badly. This reflects the fact that there needs to be a minimal number of particles to find and maintain the four optima. This is especially acute in the initial phase, when all particles are randomly placed on the map and enough particles must be present such that at least one particle is near each optimum.

As can be seen in both plots, the diversity preserving ability of the standard particle filter is very limited. Even for 2500 particles, the algorithm is successful in maintaining the diversity in only 6% of the runs, with an average time to convergence of 183 cycles. EXTRAPOLATION. Even the simplest and computationally cheapest niching method, frequency-dependent selection with a sample size of 1, significantly outperforms the standard algorithm. However, this algorithm does not achieve more than 37% success on average, with a time to convergence of 303 cycles for the maximum population size. The two sharing algorithms with equal complexity, standard sharing and frequency-dependent selection, both with a sample size of 20%, perform equally well. Their performance in maintaining diversity is similar to that of local selection. Among these three algorithms, no significant differences are found. With the maximum population size, the algorithms have an average success of 96%, 97% and 92% and a time to convergence of 480, 485 and 461 cycles respectively. It must be kept in mind that the maximum population size of local selection is on average 2258 in contrast with 2500 for all other niching methods. The best performing algorithms are the two crowding algorithms, standard crowding and closest-of-the-worst. For population sizes of 1000 particles and more, both algorithms outperform the other algorithms. Both algorithms show a significant difference, using the t-test with $p < 0.05$, with all other algorithms for 1500 particles. With more particles, the performance is not significantly better than the other algorithms, but this is caused by the fact that the performance is close to what is maximally possible, which inherently flattens the curves. The difference would remain if a larger maximal number of runs was chosen. In the case of 2500 particles, both algorithms have a perfect score. In 100% of the runs, the diversity is maintained over all 500 cycles.

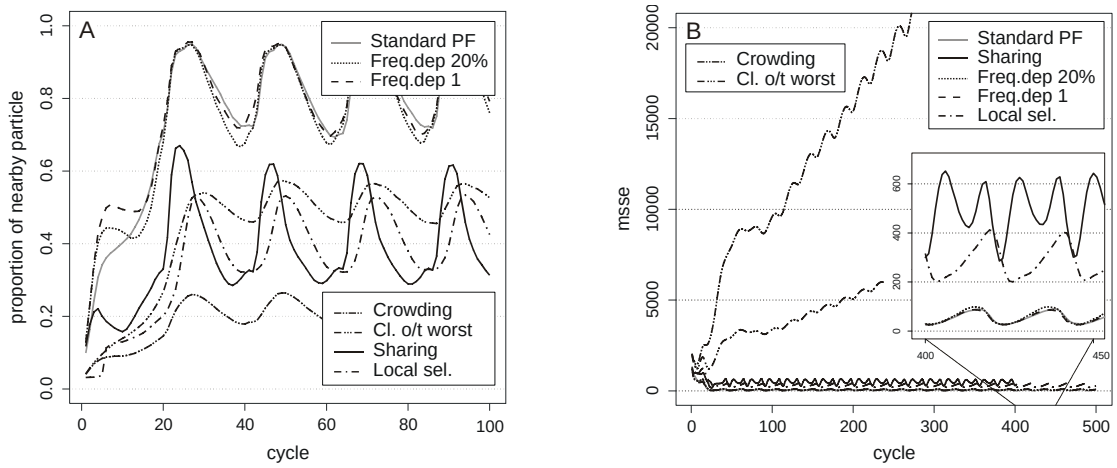

Figure 6. Compactness of the particle subpopulations. Experiments are performed with 2500 particles, and $\theta = 0.35$ for local selection. (A) Proportion of particles that is within the radius $\rho = 10$ of the four possible solutions. This is a measure of the compactness of a subpopulation. After an initial phase, the graphs show periodic behavior. This is a result of the shape of the robot's environment. When the robot approaches a corner, the subpopulations get more compact, while they expand in the corridors. (B) The mean sum of squared errors (MSSE). This shows the variance within a subpopulation and is inversely proportional to the compactness of the population. The two crowding algorithms show a constant increase in the MSSE. The other algorithms remain stable after the initial phase.

Local selection has the best performance in maintaining diversity of all algorithms with linear complexity. Its performance is comparable to that of standard sharing and frequency-dependent selection with 20% sample size, which both have quadratic complexity.

6.2. Compactness of the Particle Subpopulations

Fig. 6 reveals that the two sharing algorithms, standard sharing and frequency-dependent selection, with a sample size of 20% have a different compactness of the subpopulations. Both measures of compactness show that the subpopulations are more compact with frequency-dependent selection than with standard sharing. The compactness with frequency-dependent

selection is similar to the standard particle filter, whereas sharing results in significantly less compact subpopulations. The difference in compactness can be explained by the difference between equation (4.5), used by frequency-dependent selection, and (4.4), used by sharing. The latter demotes particles that are close to other particles more than the former. This results in less compact particle populations.

The diversity maintenance of local selection was similar to that of sharing and frequency-dependent selection. The compactness of its subpopulations is also similar to that of sharing, but worse than that of frequency-dependent selection.

Fig. 6B reveals a problem that both crowding algorithms have. The MSSE of both standard crowding and closest-of-the-worst constantly increases. This phenomenon is caused by the fact that a proportion of the population is not selected for replacement. Particles that reproduce replace similar particles, which are nearby. Particles that are far from successful particles therefore have a high probability to remain in the population. This results in a fixed proportion of 'free' particles that follow a random walk through the environment. Since we have not bounded the positions of the particles, we see a constantly increasing MSSE. Adding more selection pressure as is done in closest-of-the-worst reduces the number of such particles, but the problem remains. Additional techniques providing more selection pressure need to be applied to prevent or solve this phenomenon. Alternatively, the free particles could be detected and subsequently repositioned close to more successful particles.

6.3. Estimation Accuracy

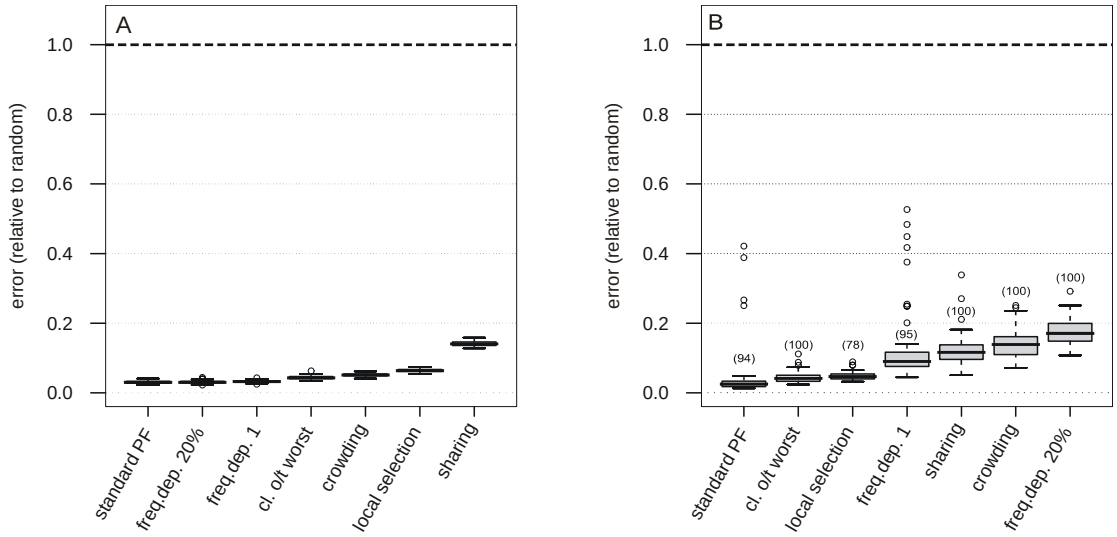

Figure 7. The estimation error, using kernel density estimation, of the different algorithms. The error is relative to the expected error with a random estimation. Experiments are performed with 2500 particles. For local selection, $\theta = 0.35$. The figure shows standard box-and-whisker plots. On the horizontal axis, the algorithms are ordered by their median error. (A) shows the estimation error in the ambiguous environment. (B) shows the estimation error in a non-ambiguous environment. The numbers above the boxes show the number of successful runs. An unsuccessful run is a run where there are less than 10 particles within the vicinity of the robot ($\rho = 10$) over an extended period of time. All unsuccessful runs are caused by the fact that, by chance, the position of the robot is not covered by the random initialization of particles. Since this is an initialization problem, and not an estimation problem, only successful runs are considered in the error estimation.

Fig. 7 shows the accuracy of all algorithms for both the ambiguous environment, (A), and a non-ambiguous environment, (B). The error is plotted relative to the expected error of a random estimation for both environments. In Fig. 7A, we see that the estimation with the standard particle filter is most accurate. Although this measure does not say anything about the diversity

maintenance, it illustrates the good properties of the standard algorithm for Monte-Carlo localization. When performing a standard t-test, all niching methods, except the frequency-dependent selection with a sample size of 20%, show a significantly worse result (with p-values << 0.01). This loss of accuracy can be explained by the fact that less particles represent one solution, causing larger errors in the kernel density estimation. However, the overall performance in estimating the robot's position of all algorithms can be considered good.

In Fig. 7B, the results are shown for the non-ambiguous environment. Again, the performance of the standard particle filter is best and similar to its performance in the ambiguous environment. For the niching methods, we see a slight decrease in accuracy and an increase in variance. This decrease in performance can be explained by the fact that the niching methods, in contrast to the standard particle filter, often maintain a second particle cluster in a local maximum. These 'ghost clusters' usually have a smaller population size and do not follow the same trajectory as the robot. However, every now and then, these ghost clusters have a higher density of particles than the main cluster, which causes the kernel density estimation method to incorrectly estimate the position of the robot.

Especially the change in performance of frequency-dependent selection with a sample size of 20% is remarkable. Where it performed among the best in the ambiguous environment, it performs worst in the non-ambiguous environment. The reason is that, more often than with the other algorithms, the ghost cluster contains the highest density of particles. Because of the nature of the algorithm, the particle populations, including the ghost clusters, are very compact. This increases the probability on an incorrect estimation.

Another notable result is the number of successful runs of the local-selection algorithm (see Fig. 7B). A run is considered unsuccessful when there are fewer than 10 particles within the vicinity of the robot ($\rho = 10$) over an extended period of time (> 250 cycles). All unsuccessful runs are the result of a failure to place particles near the robot's position in the initial phase. Local selection is especially vulnerable to this, because particles 'die' if their fitness is low. In the other

algorithms, the particle population is fixed and the particles are able to recover the position of the robot, but with local selection, quickly there are no particles left to recover. Although this might seem a huge disadvantage of local selection, it can be avoided quite easily by increasing the initial amount of energy for each particle. Moreover, in the cases where the initial phase was successful, local selection results in good estimation accuracy.

The fact that all the runs of all methods in the ambiguous environment are successful can be explained by the form of the fitness landscape. In the ambiguous environment, the fitness landscape is quite broad. There are more possible solutions and there is a broad and gradual increase in fitness towards the maxima, which can be used by the hill-climbing properties of the particle filter. The fitness landscape of the non-ambiguous environment, on the other hand, is much spikier, and the maximum is therefore harder to find. Apart from the problem with unsuccessful runs, local selection shows a good accuracy in estimating the robot's position. Overall, all niching methods show a good, and by far better than random, performance in estimation.

7. Discussion

In this paper, we discussed the problem of premature convergence in particle filters, and we demonstrated the successful transfer of a number of niching methods originally from the domain of genetic algorithms to tackle the problem. We applied the methods for maintaining diversity in the particle population to Monte-Carlo robot localization in symmetric environments that provide ambiguous situations.

All niching methods that we used show a significantly better performance in maintaining diversity than the standard particle filter. The two crowding algorithms, standard crowding and closest-of-the-worst, have the best performance, despite their problems with optimization, reported in the genetic algorithms literature [5, 19]. Further research needs to be done to see how the crowding algorithms perform in larger and more complex environments. Adding more

selection pressure with the closest-of-the-worst algorithm does not yield an improvement in the diversity preserving performance, but does improve the compactness. Despite the problem of 'free' particles that both algorithms have, their accuracy in estimating the robot's position is good, with closest-of-the-worst outperforming standard crowding.

Local selection is the best performing algorithm with linear complexity. Its method for reproduction is very different from the other algorithms and involves a dynamic population size. The number of particles adapts to the carrying capacity of the environment. Since in robot localization the complexity constantly changes, a dynamic population size is a useful property for Monte-Carlo localization (e.g., [9]). Furthermore, local selection has the useful property that it can be applied to environments differing in size without adjusting the parameters, whereas for all other implementations of the particle filter the number of particles needs to be set. The three important properties of local selection, its linear complexity, its good diversity preserving capabilities, and the adaptation to the complexity of the environment, suggest that the algorithm is very suitable for Monte-Carlo localization. We therefore believe that it is useful to further investigate this algorithm and test its performance in different and more complex environments, as well as to compare it to KLD-sampling [9].

The sharing algorithms, standard sharing and frequency-dependent selection, are very simple variations on the standard particle filter that yield good diversity preserving performance. Frequency-dependent selection results in more compact subpopulations, which makes for more accurate estimation of the robot's position in the ambiguous environment, but causes more problems with ghost clusters in the non-ambiguous environment. Our experiments show that diversity preserving performance depends strongly on the sample size. We tested sample sizes of 1 particle and 20% of the population. The latter results in significantly better performance, but makes the complexity of the algorithm quadratic in the number of hypotheses.

For Monte-Carlo localization, and particle filters in general, a sufficient number of particles in the initial phase is crucial for finding all optima. With small population sizes, the probability that

there are no particles near one of the potential solutions is large. If this happens, the particle filter will never be able to recover that solution. This is especially problematic for local selection, since it may result in the annihilation of the particle population. It is therefore necessary to start with a very large number of particles, and for local selection, with a high initial energy of the particles. After the initial phase, which usually lasts for a small number of time steps, the number of particles can be reduced. Local selection already does this automatically. The other methods would have to be modified slightly to incorporate this.

Because of the diversity preserving forces in the niching methods, they produce different clusters, even when there is no ambiguous situation. The so-called ghost clusters that are formed fill a local maximum in the fitness landscape. Although the ghost clusters sometimes cause incorrect localization in our experiment, this can quite easily be avoided with additional techniques. The position of the ghost cluster does not change in accordance with the odometry information of the robot and can therefore be easily classified as incorrect.

7.1. The Path to Implementation

In this paper, we used a simulated robot to analyze the dynamics of the niching methods. We decided to use a simulation in order to create the best controlled experiments possible, and thus make the fairest comparison between methods. Exactly symmetrical situations that, in addition, are constant over time are hard to recreate in the real world. This does not mean that our results are only applicable to such artificial situations. On the contrary, they are very relevant for robot navigation in the real world. Particle filters are increasingly used in robot navigation (e.g., [29-31]), and many real world environments create ambiguous situations for the robot. Consider for instance an office building with many identical-looking offices or a building with large open spaces (or unreflective walls) where distance sensors do not return a reading. Such situations also cause premature convergence in particle filters on real robots. The methods investigated in this

paper are expected to work as well in the real world as they work in simulation. After all, the dynamics of a particle filter do not depend on the source of the input.

The implementation of the niching methods on a real-world application is straightforward. Only the motion and sensor models need to be adjusted to better model the actuators and sensors of a real robot. We refer to [28] for a description of a motion and sensor model for a mobile robot. The rest of the particle filter and the applied niching methods can be used exactly as in their presented form.

7.2. Conclusion

Our research demonstrated the applicability of a number of niching methods that are well-known in genetic algorithms, to solve the problem of premature convergence in particle filters. Not only does this paper contribute to an improvement of particle filters in symmetric environments, it also demonstrates the possibility to transfer knowledge between the fields of genetic algorithms and particle filters. We discussed that both algorithms are, in their essence, identical. They are both based on variation, selection and reproduction. This similarity causes similar problems for both algorithms, and these can be solved by using similar techniques. Besides the examples we gave in this paper, there might well be other techniques that are worth sharing. We hope this paper will kick off the search for, and the application of, techniques shared between genetic algorithms and particle filters.

References

[1] J. E. Baker, Reducing bias and inefficiency in the selection algorithm, in *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, 1987, pp. 14-21.

[2] A. Bienvenue, M. Joannides, J. Berard, E. Fontenas and O. Francois, Niching in Monte Carlo Filtering Algorithms, in *Proceedings of 5th Int. Conf. on Artificial Evolution*. Creusot, 2001, pp. 19-30.

[3] V. Braitenberg, *Vehicles: Experiments in Synthetic Psychology*: MIT Press, 1984.

[4] K. A. de Jong, An analysis of the behavior of a class of genetic adaptive systems.: University of Michigan, 1975.

[5] K. Deb and D. E. Goldberg, An investigation of niche and species formation in genetic function optimization, in *Proceedings of the third international conference on Genetic algorithms*, 1989, pp. 42-50.

[6] A. Doucet, J. F. G. de Freitas and N. J. Gordon (Eds), Sequential Monte Carlo Methods in Practice. New York: Springer Verlag, 2001.

[7] R. O. Duda, P. E. Hart and D. G. Stork, *Pattern Classification, Second Edition*: Wiley Interscience, 2000.

[8] J. M. Epstein and R. Axtell, *Growing Artificial Societies: social science from the bottom up*. Washington, D.C.: Brookings Institution Press, 1996.

[9] D. Fox, Adapting the Sample Size in Particle Filters Through KLD-Sampling., *International Journal of Robotics Research (IJRR)*, vol. 22, pp. 985-1003, 2003.

[10] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[11] D. E. Goldberg and J. Richardson, Genetic algorithms with sharing for multimodal function optimization, in *Proceedings of the Second International Conference on Genetic Algorithms and their application*. Cambridge, Massachusetts, United States, 1987, pp. 41-49.

[12] D. L. Hartl and A. G. Clark, *Principles of Population Genetics*, second ed. Sunderland, Mass.: Sinauer Associates, Inc., 1989.

[13] T. Higuchi, Genetic Algorithm and Monte Carlo Filter (in Japanese with English Abstract), in *Proceedings of the Institute of Statistical Mathematics*, vol. 44, 1996, pp. 19-30.

[14] T. Higuchi, Monte Carlo Filter Using the Genetic Algorithm Operators, *Journal of Statistical Computation and Simulation*, vol. 59, pp. 1-23, 1997.

[15] J. H. Holland, *Adaptation in natural and artificial systems*: Ann Arbor: University of Michigan Press, 1975.

[16] A. Howard and N. Roy, The Robotics Data Set Repository (Radish), http://radish.sourceforge.net/, 2003.

[17] N. M. Kwok, W. Zhou, G. Dissanayake and G. Fang, Evolutionary Particle Filter: Re-sampling from the Genetic Algorithm Perspective, in *IEEE/RSK International Conference on Intelligent Robots and Systems (IROS 05)*. Edmonton, Canada, 2005.

[18] R. Lande, Natural Selection and Random Genetic Drift in Phenotypic Evolution, *Evolution*, vol. 30, pp. 314-334, 1976.

[19] S. W. Mahfoud, Crowding and preselection revisited, in *Parallel problem solving from nature, 2*, R. Männer and B. Manderick, Eds. Amsterdam: Elsevier, 1992, pp. 27-36.

[20] S. W. Mahfoud, Niching Methods for Genetic Algorithms, in *Department of General Engineering*, vol. PhD: University of Illinois, 1995.

[21] F. Menczer, M. Degeratu and W. N. Street, Efficient and Scalable Pareto Optimization by Evolutionary Local Selection Algorithms, *Evolutionary Computation*, vol. 8, pp. 223-247, 2000.

[22] N. Metropolis and S. Ulam, The Monte Carlo Method., *Journal of the American Statistical Association*, vol. 44, pp. 335-341, 1949.

[23] M. Mitchell, *An Introduction to Genetic Algorithms*: MIT Press, 1996.

[24] P. D. Moral and A. Guionnet, On the Stability of Interacting Processes with Applications to Filtering and Genetic Algorithms, *Annales de l'Institut Henri Poincaré*, vol. 37, pp. 155-194, 2001.

[25] P. D. Moral, L. Kallel and J. Rowe, Modeling Genetic Algorithms with Interacting Particle Systems, in *Natural Computing Series: Theoretical Aspects of Evolutionary Computing*, L. Kallel, B. Naudts and A. Rogers, Eds. Berlin: Springer-Verlag, 2001, pp. 10-67.

[26] I. Rechenberg, *Evolutionsstrategie. Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart: Frommann-Holzboog, 1973.

[27] T. A. Sedbrook, H. Wright and R. Wright, Application of a genetic classifier for patient triage, in *Proceedings of the Fourth International Conference on Genetic Algorithms*, 1991, pp. 334-338.

[28] S. Thrun, W. Burgard and D. Fox, *Probabilistic Robotics*. Cambridge, MA: MIT Press, 2005.

[29] S. Thrun, D. Fox, W. Burgard and F. Dellaert, Robust Monte Carlo Localization for Mobile Robots, *Artificial Intelligence*, vol. 128, pp. 99-141, 2001.

[30] N. Vlassis, B. Terwijn and B. Krose, Auxiliary particle filter robot localization from high-dimensional sensor observations, in *Proceedings of IEEE International Conference on Robotics and Automation*, W. R. Hamel and A. A. Maciejewski, Eds. Washington D.C., USA, 2002, pp. 7-12.

[31] J. Wolf, W. Burgard and H. Burkhardt, Robust vision-based localization for mobile robots using an image retrieval system based on invariant features, in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, vol. 1, 2002, pp. 359- 365.

Biography

Gert Kootstra is a Ph.D. student at the Artificial Intelligence institute of the University of Groningen. He received his M.Sc. in Artificial Intelligence in 2001 from the University of Groningen. His graduation research was conducted at the AI-lab of Rolf Pfeifer at the University of Zurich on the topic of visual landmark navigation. From 2001 to 2005, Gert was a junior lecturer in Robotics and Autonomous Systems. His current research interests are in robotic navigation, simultaneous localization and mapping and visual perception.

Bart de Boer has a M.Sc. in Computer Science. He received his Ph.D. in Artificial Intelligence from the University of Brussels, with Luc Steels as promoter. From 2003 to 2007, he was assistant professor in Cognitive Robotics at the University of Groningen. Bart received a prestigious VIDI grant from the Netherlands Organization for Scientific Research in 2007. He is currently at the Amsterdam Centre of Language and Communication at the University of Amsterdam, where he is working on modeling the evolution of speech and language.

Figure and table captions

Figure 1. An example of premature convergence in Monte-Carlo localization. In each snapshot, the robot is shown as a circle with its distance measurements. The particles are depicted as black dots. The gray areas are obstacles and the white areas are free space. The environment is highly symmetrical with the exception of an object in the left hallway. In the first situation some diversity is still present in the particle population. However, the diversity quickly disappears as a consequence of random drift in sampling. This results in the system's inability to correctly localize the robot when the disambiguating object is found.

Figure 2. Diagram A shows the distribution of the motion model $p(x_{t+1} \mid x_t, u_t)$. The distribution has a banana shape, caused by the Gaussian distribution on the translation and rotation of the robot. Diagram B shows 1000 samples of this distribution. The sensor model is shown in diagram C. The probability $p(z_t^k \mid x_t)$ is defined by a Gaussian function with the robot's sensor reading, z_t^k, as variable, the particle's reading, z_t^{k*}, as mean and σ_s^2 as variance.

Figure 3. The map with square symmetry used in the experiments.

Figure 4. The non-ambiguous map used in experiments C.

Figure 5. (A) The percentage of runs without convergence plotted against the number of hypotheses. The error bars give the 95% confidence intervals. Means and confidences are calculated from 10 sets of 10 runs. The number of hypotheses for the local-selection algorithm is variable and depends on θ. The corresponding horizontal error bars show the 95% confidence interval for the average number of hypotheses per run. (B) The time to convergence as a function of the population size. A run is considered converged if one or more of the four optima are lost. The run has a maximum length of 500 cycles. Non-converged runs receive a time to convergence of 500. The error bars give the 95% confidence intervals. The data comes from a total of 100 runs.

Figure 6. Compactness of the particle subpopulations. Experiments are performed with 2500 particles, and $\theta = 0.35$ for local selection. (A) Proportion of particles that is within the radius $\rho = 10$ of the four possible solutions. This is a measure of the compactness of a subpopulation. After an initial phase, the graphs show periodic behavior. This is a result of the shape of the robot's environment. When the robot approaches a corner, the subpopulations get more compact, while they expand in the corridors. (B) The mean sum of squared errors (MSSE). This shows the variance within a subpopulation and is inversely proportional to the compactness of the population. The two crowding algorithms show a constant increase in the MSSE. The other algorithms remain stable after the initial phase.

Figure 7. The estimation error, using kernel density estimation, of the different algorithms. The error is relative to the expected error with a random estimation. Experiments are performed with 2500 particles. For local selection, $\theta = 0.35$. (A) shows the estimation error in the ambiguous environment, where the error is measured by the distance from the estimation to the nearest of the four possible robot positions. (B) shows the estimation error in a non-ambiguous environment. Here, the error is the distance from the estimation to the robot's position. The numbers above the boxes show the number of successful runs. An unsuccessful run is a run where there are less than 10 particles within the vicinity of the robot ($\rho = 10$) over an extended period of time. All unsuccessful runs are caused by the fact that, by chance, the position of the robot is not covered by the random initialization of particles. Since this is an initialization problem, and not an estimation problem, only successful runs are considered in the error estimation.

Table 1. Functions for the motion model, the sensor model and stochastic uniform sampling.

Table 2. Pseudocode of the standard Particle Filter and the additional code for sharing and frequency-dependent selection.

Table 3. Pseudocode for crowding and the additional code for closest-of-the-worst. In the experiments, we used $gg = 0.2$ and $cf = 0.01$. For the sampling in line 3, we used stochastic universal sampling.

Table 4. Pseudocode for the local-selection algorithm. In the experiments, θ was the variable and $E_{out} = 0.2 \cdot \theta$. The used value of E_{out} was derived from a number of pilot experiments.

Table 5. The carrying capacity using the local-selection algorithm with different values of θ.