Solving the mobile robot localization problem using string matching algorithms

Cándida González-Buesa and Javier Campos
Departamento de Informática e Ingeniería de Sistemas
Universidad de Zaragoza
María de Luna 1, 50018 Zaragoza, Spain
Email: jcampos@unizar.es

***Abstract*— In this paper we address the mobile robot localization using some techniques borrowed from the Computational Biology community. The specific problem studied here is also known as the kidnapped robot problem. Our proposal is to solve this problem by string matching algorithms, which have experienced a large advance in the last years due to (for example) the Genoma Project. The paper uses three different algorithms to solve the mentioned problem and shows their advantages, such as the robustness of the results and the memory and time efficiency. These results are validated by real experimentation using panoramic images of indoor buildings, and compared and discussed with existing techniques that have been used over the same test-bed.**

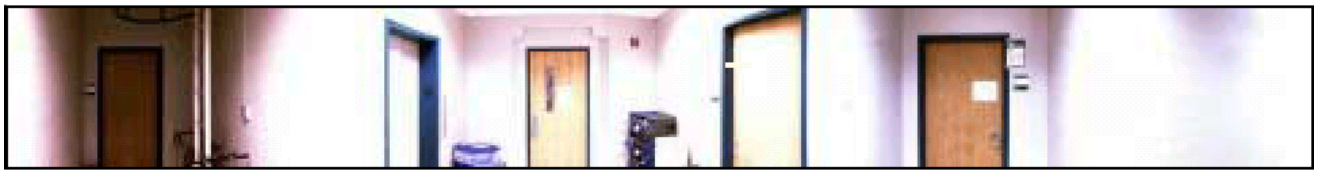
I. Introduction

Many of the current robot applications such as emergency response, transportation, demining, exploration, help care or ludic robots involve the generation of autonomous motion. By using these modules, the mobile devices improve their navigation performance and thus the versatility of their usage. Thus in mobile robotics, we are carrying out a vast effort to improve the robustness of the basic skills of the mobility task, since it is clear that they have a large impact over the whole performance of the system. One of these skills is the localization, that is essential, because knowing where the vehicle is located allows to address any issues concerning mobility [1]. We address here a specific instance of this problem called the kidnapped robot. This problem arises when the robot is located in an environment from which it owns a map. Then, at a given moment, and based on sensory information (e.g. cameras, infrared rays, ultrasounds), the vehicle has to decide where is located within the map. In this work, we assume that the map of the scenario is composed of a set of panoramic images. Thus, given a new image gathered, the vehicle must conclude its current location. Many efforts have been carried out during these last years in this context by using different tools such as Montecarlo [2], graph theory [3], search in interpretation trees [1], RANSAC [4], or topological representations [5].

Our idea is to derive profit from the advances in the field of Computational Biology, based on the large number of projects that have been undertaken (e.g. the Human Genoma Project). Some of these advances resulted in new algorithmic techniques for string processing. This is because, in this context, a major problem is to find genes (that match a given large string) among huge data bases of DNA. These data do not really need to correspond exactly to the gen, so that there are many issues to consider such as the codification, matching, etc. Our proposal is based on reducing the robot localization problem to a string matching problem, applying then those algorithms that better adapt to this situation.

In a recent work [6], the authors take panoramic pictures from the environment by means of a mobile robot, and then they extract some features that are converted into strings. Their map is composed by a set of panoramic pictures taken at given space points, every certain distances, covering all the environment. Next, they process these panoramic pictures to obtain the character strings (called fingerprint sequences). To do it, they propose that some visual features of each space point generate a unique sequence, in such a way that it can be distinguished from the rest of sequences of the environment. Then, they use two types of features: vertical edges (identified as 'v') and color patches (identified as 'A', 'B', ..., 'P'). Extracting those features from panoramic pictures, a character string for every map point can be created (see Fig. 1). With this reduction, it is possible to obtain a database that incorporates both character strings from the pictures taken on the map points, and test strings from the pictures taken on the test points, so that it is possible to compare test and database strings. The database string most similar to each test string will correspond to the map picture most similar to each test picture, and then to the map point nearest to each test point. In this way, we can predict the approximate position of the robot into the map. In other words, at a given moment, the robot obtains a panoramic picture from its most immediate environment, by using a camera linked to the robot; this image is converted into a string, and the similarity of the different character strings is evaluated.

In [6] the authors suggested the use of string matching algorithms, but they discarded them and implemented their own minimum energy algorithm, based on dynamic programming. This decision was motivated by the fact that they wanted to consider the possibility of different types of mismatch errors, that is, they wanted to calibrate the rank of the discordances. A goal that, they thought, could not be achieved by string matching algorithms. However, in this work we take the other way around exploring the efficacy of these string matching algorithms (described in [7]). We also provide a comparison with the alternative techniques to show that the localization results are improved and



vBvvvOvvvLvBEvvvvvvBvL

Fig. 1. Panoramic image, its visual features and its string (this image, taken from [6], is used here with permission of the first author of that paper)

we discuss the benefits of our approach on the basis of the robustness of the results and the memory and time efficiency of the algorithms. We introduce in Section II the string matching algorithms. In Section III, several tests are presented and the results obtained with the algorithms proposed in this paper are compared with previous results in the literature. Final remarks and conclusions are stressed in Section IV.

II. String Matching Algorithms

There are two types of string matching algorithms: exact string matching algorithms and inexact string matching algorithms. Exact string matching algorithms can be useful for certain restricted types of problems where length of strings is short and the information on characters in the strings is free of errors. However, due to the kind of comparisons that we are going to make, with large strings and possible data errors, it is very unlikely that an exact matching between different strings will occur, because minimum variations in a picture will provoke substantial differences on the extracted strings. Thus, inexact string matching algorithms will better suit the problem that we are facing.

There is a great variety of these algorithms, according to their use, and an in depth description of some of the most representative has recently been published [7]. For this work we have implemented three algorithms: algorithm A, for the estimation of the global alignment with k differences, based on dynamic programming; algorithm B, for inexact matching with k differences, based on hybrid dynamic programming; and algorithm C, for the estimation of the longest common subsequence of two strings. We present the main ideas concerning all of them in the subsequent paragraphs.

A. Algorithm A (global alignment with k differences algorithm)

Given two character strings T and P, of length m and n, respectively, and given a fixed number k, the objective of the algorithm is to find, if it exists, the best global alignment of T and P that contains at most k differences and spaces. The algorithm is based on the estimation of the edit distance between both strings. The edit distance is the minimum number of operations (insertions, deletions or substitutions of characters) needed to convert one string into the other. Dynamic programming is applied to solve it.

Let $P(i)$ and $T(j)$ be the prefix substrings of P and T, that is to say, $P(i) = p_1 p_2 ... p_i$ and $T(j) = t_1 t_2 ... t_j$. Let $D(i,j)$ be the cost of convert $P(i)$ into $T(j)$. The recurrent equation is the next:

$$D(i,j) = \min \begin{cases} D(i-1,j) + \text{del\_cost} & (\text{delete } p_i) \\ D(i,j-1) + \text{ins\_cost} & (\text{insert } t_j) \\ D(i-1,j-1) + d(i,j) & (\text{other case}) \end{cases}$$

for all $1 \leq i \leq n$, $1 \leq j \leq m$, being the trivial cases the next:

$$D(i,0) = i, \text{ for all } i, 0 \leq i \leq n$$

$$D(0,j) = j, \text{ for all } j, 0 \leq j \leq m$$

and being:

$$d(i,j) = \begin{cases} 0, & \text{if } p_i = t_j \\ 1, & \text{if } p_i \neq t_j \end{cases}$$

where *del_cost* and *ins_cost* are two parameters whose value can be adjusted to obtain better results. In the same way, we can alter the value of $d(i,j)$ in function of the similarity between characters p_i and t_j, that is to say, the more similar they are, the lower will be the penalty for mismatching. Making these simple modifications to the equations, we can achieve what Lamon *et al.* [6] were looking for with their algorithm.

In order to calculate the value $D(i,j)$ we need the values for $D(i-1,j)$, $D(i,j-1)$ and $D(i-1,j-1)$ (see Fig. 2).

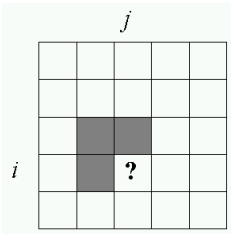


Fig. 2. Dynamic programming table

Therefore, in order to fill the cell (i,j) we need the values of the shaded areas of the table. In this way, we can fill the table by rows from left to right, or by columns from the top.

The cost of filling the table and, therefore, of calculating $D(n, m)$ is O($n \cdot m$), and obtaining the trace or optimal transcription has a cost O($n + m$).

However, we can define limits around the main diagonal of the table, that vary depending on k. This can be done because it is nonsense filling the table beyond k diagonals starting from the main diagonal when we have limited to k the total number of differences. In this way, for $k = 4$, the shaded area in Fig. 3 would be the only to be filled.

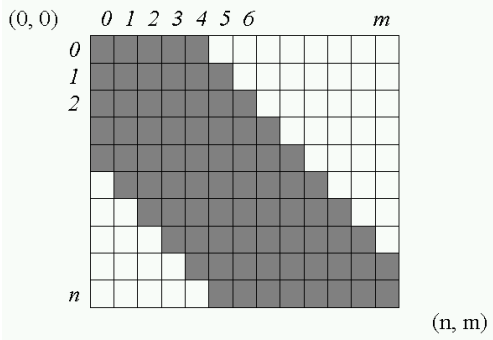


Fig. 3. Limited dynamic programming table

With this modification, that slightly alters the recurrent equation, the cost of filling the table and calculating $D(n, m)$ would be O($k \cdot m$) in time and space.

B. Algorithm B (inexact matching with k differences algorithm)

Given two strings T and P, of length m and n, respectively, and given a fixed number k, the objective of the algorithm is to find, if it exists, all the ways of matching P and T using at most k character substitutions, insertions or deletions, that is to say, to find all the occurrences of P in T with at most k differences or spaces. This algorithm is based on *hybrid dynamic programming*, a combination of classic dynamic programming and suffix trees [8].

A few concepts must be first defined. The main diagonal of the dynamic programming table (DPT) includes all those cells (i, i) for all i such that $0 \leq i \leq n \leq m$. Diagonals above the main diagonal are numbered from 1 to m, whereas diagonals below are numbered from -1 to $-n$. A d–path of the DPT is a path that starts at file 0 and specifies a total of exactly d differences. A d–path is the longest in diagonal i if that path is a d–path that ends in diagonal i, and the index of the column where it ends is greater than or equal to the index of the column of any other d–path that ends in diagonal i.

The main idea of this method is as follows: we have k iterations, and in each iteration $d \leq k$, we must find the longest d–path in diagonal i, for each i from $-n$ to m. The longest d–path in diagonal i can be obtained from the longest $(d-1)$–paths in diagonals $i-1$, i and $i+1$. Every longest d–path (in any diagonal) that reaches the row number n determines the final point (in T) of one occurrence of P with exactly d differences.

When $d > 0$, the longest d–path in diagonal i can be found, as we stated before, from the longest $(d-1)$–paths in diagonals $i-1$, i and $i+1$, by choosing the longest of the next paths:

- Longest $(d-1)$–path in diagonal $i+1$, followed by a vertical edge to diagonal i, followed by the maximum extension along diagonal i that corresponds to an identical substring in P and T. This path can be viewed in Fig. 4.
- Longest $(d-1)$–path in diagonal $i-1$, followed by a horizontal edge to diagonal i, followed by the maximum extension along diagonal i that corresponds to an identical substring in P and T. This path can be viewed in Fig. 5.
- Longest $(d-1)$–path in diagonal i, followed by a diagonal edge, followed by the maximum extension along diagonal i that corresponds to an identical substring in P and T. This path is shown in Fig. 6.

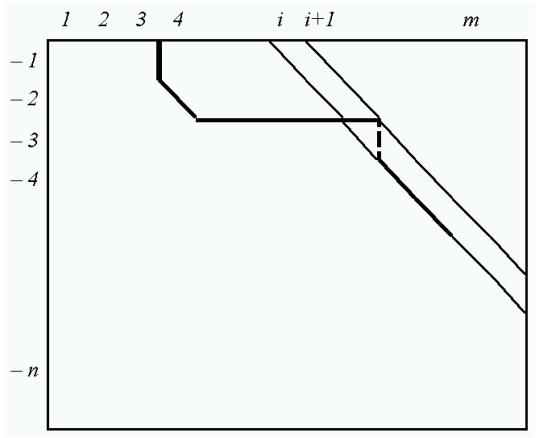


Fig. 4. First d–path in diagonal i

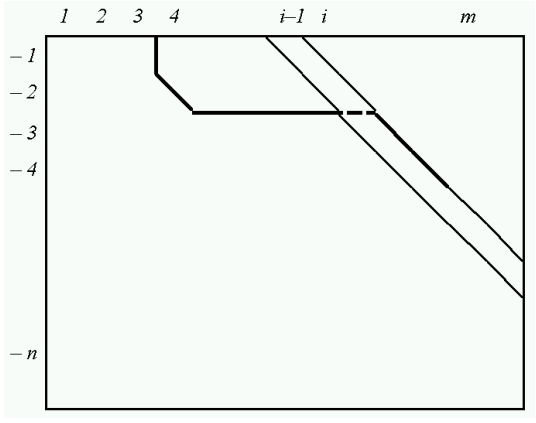


Fig. 5. Second d–path in diagonal i

The trivial case of this recurrence corresponds to $d = 0$, that is to say, when we do not allow any difference between P and T. In this case, the longest 0–path in diagonal i corresponds to the longest common extension of $T[i \,..\, m]$ and $P[1 \,..\, n]$.

The cost of this algorithm is O($k \cdot m$) in time and space.

To estimate the common substrings of P and T, as well as to estimate the longest common extension of the trivial

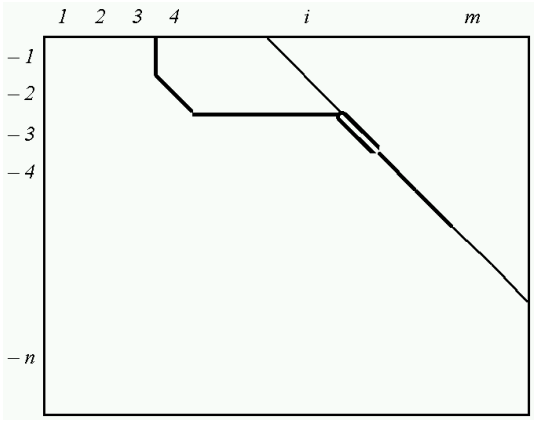


Fig. 6. Third d–path in diagonal i

case, generalized suffix trees can be used. Basically, these trees are suffix trees including more than one string, that is, trees containing in a very compact representation the suffixes from two or more character strings.

C. Algorithm C (Longest common subsequence algorithm)

This algorithm calculates the longest common subsequence (LCS) of two given strings and, contrary to those presented above, it is not based on dynamic programming. The basic idea, already described in [7], consists on reducing first the LCS problem to the problem of the longest increasing subsequence (LIS) of a list of numbers. Then, by solving the LIS problem, we may solve the LCS problem.

Given two strings T and P, of length m and n, respectively, for each character of T we create a list, in decreasing order, with the positions in P of that character. If this character does not appear in P, then it will be associated to an empty list and no number will be added to the list. We can then obtain the LIS of this list. Numbers on LIS correspond to the positions in the string P of the characters that form the LCS of T and P.

Now we must solve the LIS problem. Let Π be a list of numbers. We define a cover of Π as a set of increasing subsequences of Π such that they contain all numbers of list Π. We define the size of a cover as the number of subsequences included on it, and a minimum cover as the cover with the smallest size.

Let I be an increasing subsequence of Π with same size than a cover of Π, called C. Then I is a LIS of Π and C is a cover of Π. Therefore, we must find a minimum cover of Π, and, after that, starting from that cover, an increasing subsequence I of Π with the same size than C, that will be the subsequence that we are looking for.

In order to obtain the minimum cover of Π we must take each number of Π and insert it at the end of the first possible non increasing subsequence that we can. If it can not be inserted into any subsequence, a new subsequence that contains that number is created.

The constructing cost of this cover is O($n \cdot \log p$), being p the size of the longest increasing subsequence (LIS). Furthermore, the LIS can be obtained in time O(n) given the minimum cover.

III. Tests and Results

The first test reproduces one of the last performed by Lamon in his work. The rest of the tests are replica of the tests in [6]. In this way, we could compare our results, in terms of success percentages, with their results.

A. Rooms test

The environment of the rooms test is composed by ten separated rooms. Previously, four fingerprint sequences were obtained by placing the camera (the mobile robot) near the center of each room. One fingerprint was assigned to the environment database, while the others were considered test points. In this way, a database that includes those strings corresponding to ten map points, plus thirty test strings, is generated.

We consider a test point to be correct when the exact solution is included within the set of optimal matchings between the test string and all the database strings. A test point is exact if there is only one optimal matching between the test string and the database strings, being this matching the exact solution.

Lamon obtained a 73% of success rate, because their algorithm was right in 73% of the strings tested.

The percentages of success for each implemented algorithm are presented in Table I.

TABLE I
Our results for Rooms test

	Algorithms			
	***Alg. A*$_1$**	***Alg. A*$_2$**	***Alg. B***	***Alg.C***
Exact	83.3 %	93.3 %	73.3 %	80 %
Correct	96.7 %	96.7 %	90 %	90 %
Wrong	3.3 %	3.3 %	10 %	10 %

The difference between algorithms A_1 and A_2 rests on the election of the parameters of algorithm A. Algorithm A_1 includes the default parameters (unitary cost for insertion, deletion, color difference and discordance), while algorithm A_2 differs only in the color difference cost, that has a lower value (0.75).

As we can see in Table I, the exact results for all algorithms, except for B, have given us better percentages, in the case of consider the exact ones, than the 73% reported from Lamon. This is also evident if we consider the correct percentages of success. It has to be noted that algorithm A_1 has had the smallest error rate, and algorithm A_2 the greatest success rate.

B. Ground Floor and White Hall tests

The Ground Floor and White Hall tests were performed by Lamon *et al.* [6] in two different environments that are outlined in Fig. 7.

The map on the left represents the Ground Floor, and covers all the way from the main entry to the Conference Room in the first floor of the Smith Hall Building. The map on the right represents the White Hall, and covers the entrance hall of the same building. For the Ground Floor 21 map points and 22 test points were taken, and for the White Hall 15 map points and 18 test points were considered.

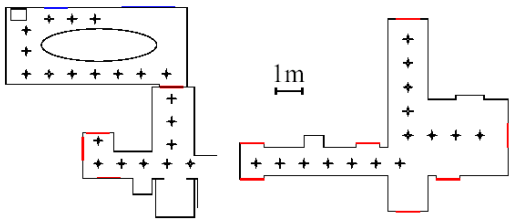


Fig. 7. Maps of Ground Floor and White Hall

For Lamon *el at*. [6], a test point was *topologically correct* if the best matching among the map points was a point adjacent to the test point, and a test point was *geometrically correct* if the best matching among the map points was that one closest to the map point.

However, we have also considered the option of multiple best matching. A test point is defined as *topologically correct* if, among the set of best matchings, there is one that corresponds to a map point adjacent to the test point. A test point is taken as *topologically exact* when there is only one map point adjacent to the test point within the set of best matchings.

In the same way, a test point is considered *geometrically correct* if, among the set of best matchings, there is one that corresponds to the map point that is closest to the test point. A test point is termed *geometrically exact* when there is only one map point in the set of best matchings that is closest to the test point.

We must note that only those points topologically correct have been taken into account for the evaluation of their geometric correction. Thus, the percentages of geometrically exact, correct and wrong points have not been calculated from the total of test points, but only from those that were topologically correct.

Lamon *et al*. [6] carried out three tests: one with the points of the Ground Floor, another with the points of the White Hall, and another one blending the points from both environments, and they obtained the results shown in Table II.

TABLE II

LAMON'S RESULTS FOR GROUND FLOOR, WHITE HALL AND GLOBAL ENVIRONMENT TESTS

	Ground Floor	White Hall	Global Environ.
Top. exact	91 %	94 %	90 %
Top. wrong	9 %	6 %	10 %
Geom. exact	70 %	82 %	75 %
Geom. wrong	30 %	18 %	25 %

We have performed two kind of tests, one taking the default parameters of each algorithm, and a second choosing these parameters in order to obtain a better solution. The election of these parameters has depended directly on the string database, and has varied from one database to other. Parameters have differed slightly in each test because we have not been able to unify their values.

Default parameters for algorithm A_1 are shown in Table III and parameters for algorithm A_2 for each test are shown in Table IV. With those parameters, the final percentages of the tests are shown in Table V, Table VI and Table VII.

TABLE III

PARAMETERS FOR ALGORITHM A_1

Deletion Cost	1.0
Insertion cost	1.0
Colour difference	1.0
Coincidence cost	0.0
Discordance cost	1.0

TABLE IV

PARAMETERS FOR ALGORITHM A_2

	Ground Floor	White Hall	Global Env.
Deletion Cost	0.55	0.7	0.55
Insertion cost	0.45	0.6	0.45
Colour difference	0.7	1.0	0.7
Coincidence cost	– 0.15	0.0	0.0
Discordance cost	3.0	1.5	3.0

TABLE V

OUR RESULTS FOR GROUND FLOOR TEST

	Algorithms			
	***Alg. A*$_1$**	***Alg. A*$_2$**	***Alg. B***	***Alg. C***
Top. exact	68.2 %	90.9 %	54.5 %	50 %
Top. correct	95.5 %	90.9 %	72.7 %	81.8 %
Top. wrong	4.5 %	9.1 %	27.3 %	18.2 %
Geom. exact	52.3 %	85.0 %	37.5 %	38.9 %
Geom. correct	71.4 %	85.0 %	75.0 %	83.3 %
Geom. wrong	28.6 %	15.0 %	25.0 %	16.7 %

TABLE VI

OUR RESULTS FOR WHITE HALL TEST

	Algorithms			
	***Alg. A*$_1$**	***Alg. A*$_2$**	***Alg. B***	***Alg. C***
Top. exact	61.1 %	77.8 %	61.1 %	38.9 %
Top. correct	83.3 %	88.9 %	77.8 %	77.8 %
Top. wrong	16.7 %	11.1 %	22.2 %	22.2 %
Geom. exact	60.0 %	75.0 %	50.0 %	35.7 %
Geom. correct	73.3 %	87.5 %	78.6 %	71.4 %
Geom. wrong	26.7 %	12.5 %	21.4 %	28.6 %

In general, the number of failures has been lower than those obtained with the algorithm of Lamon *et al*. [6]. However, precision has also been lower, specially for algorithms B and C. Nevertheless, in terms of correction percentages, our results are quite similar, and in some cases even better than those from Lamon *et al*. [6]. The reason why algorithm C has had a lower exactness is that this algorithm is very sensitive to length differences between the strings, and in these tests, length differences between some strings are considerable. This situation has negatively affected the results of matching with algorithm C.

Algorithm A has offered very good results, specially when we have adjusted the parameters (algorithm A_2). In spite of that, in some cases the wrong rate is slightly greater than that obtained with other algorithms; as its exactness rate is also greater, it compensates for that wrong rate.

TABLE VII
OUR RESULTS FOR GLOBAL ENVIRONMENT TEST

	Algorithms			
	Alg. A_1	***Alg. A_2***	***Alg. B***	***Alg. C***
Top. exact	60.0 %	75.0 %	50.0 %	35.0 %
Top. correct	85.0 %	77.5 %	75.0 %	70.0 %
Top. wrong	15.0 %	22.5 %	25.0 %	30.0 %
Geom. exact	47.1 %	83.9 %	43.3 %	39.3 %
Geom. correct	70.6 %	87.1 %	80.0 %	85.7 %
Geom. wrong	29.4 %	12.9 %	20.0 %	14.3 %

In spite of some cases the wrong rate is lightly greater than the other algorithms, the exactness rate is also greater, which rewards that wrong rate.

IV. CONCLUSIONS

The main contribution of this paper is to apply advanced algorithms developed in the Computational Biology field to mobile robot localization problems. The results have shown that the algorithms are robust and its memory and time efficiency is good.

Some of the implemented algorithms offered good results when compared with previous tests, whereas others were less adequate. Conceptually simple algorithms, such as algorithms A and C, gave in general better results than the more complicated algorithm B. However, these results were conditioned by the structure of the environment in each test.

Algorithm C behaved the best in the Rooms Test. This may be due to the regularity of the strings of that environment, since, as we have mentioned before, this algorithm is very sensitive to size differences between the strings to be compared.

In the Rooms environment we have noticed the effectivity of all algorithms, specially that of algorithms A and C. However, in all other tests (Ground Floor, White Hall and Global Environment tests) results were slightly worse. This may be secondary to environments differences. Whereas the Rooms environment is composed of a few independent areas, other environments are composed of a single area, with extracted strings having too many points in common, specially those located in the same or nearby areas (for instance, along the same corridor).

As a consequence, and as it has been reported by others [6], our results were the better the more separate were the map points chosen. If the points were separated enough, the effectivity of the results increased substantially. Therefore, large environments, where we can distribute the map points properly, are most suitable for the implemented algorithms. Thus, an interesting future work to consider is the to give some insights on the better way of selecting points to build the maps.

ACKNOWLEDGMENTS

Thanks to Pierre Lamon and Roland Siegwart, from the *Swiss Federal Institute of Technology*, for their interest in this work, for their inestimable help in the part of pictures processing, and for providing us with the code of their project and their test data.

We wish to thank also José Neira, Javier Mínguez, and Luis Montesano, from the *Robotics Group* of our Department, for their valuable comments and suggestions.

This work has been developed within the project TIC-2003-05226 of the Spanish Ministry of Science and Technology.

REFERENCES

[1] J.A. Castellanos and J.D. Tardós, Mobile Robot Localization and Map Building. A Multisensor Fusion Approach. Kluwer Academic Publishers, Boston, MA, 2000.

[2] D. Fox, W. Burgard, F. Dallaert and S. Thrun, "Montecarlo localization: efficient position estimation for mobile robots", Proc. Nat. Conf. Artificial, pp. 343-349, Orlando, Florida, 1999.

[3] T. Bailey, E.M. Nebot, J.K. Rosenblatt and H.F. Durrant-Whyte, "Data association for mobile robot navigation: a graph theoretic approach", Proc. ICRA 2000, pp. 2512-2517, San Francisco, California, 2000.

[4] J. Neira, J.D. Tardós and J.A. Castellanos, "Linear time vehicle relocation in SLAM", Proc. ICRA 2003, pp. 427-433, Taipei, Taiwan, September 2003.

[5] H. Choset and K. Nagatani, "Topological simultaneous localization and mapping (SLAM): toward exact localization without explicit localization", IEEE Trans. on Robotics and Automation, 17(2):125-137, April 2001.

[6] P. Lamon, I. Nourbakhsh, B. Jensen and R. Siegwart, "Deriving and matching image fingerprint sequences for mobile robot localization", Proc. ICRA 2001, pp. 1609-1614, Seoul, Korea, May 2001.

[7] D. Gusfield, Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York, 1997.

[8] E. Ukkonen, "On–line construction of suffix trees", Algorithmica, vol. 14, pp. 249–60, 1995.