Optimal Reorganization of Agent Formations

DALILA B. M. M. FONTES
LIAAD-INESC Porto L.A. and
Faculdade de Economia
Universidade do Porto
Rua Dr. Roberto Frias, 4200-464 Porto
PORTUGAL
fontes@fep.up.pt

FERNANDO A. C. C. FONTES
ISR-Porto and
Universidade do Minho
Dep. de Matemática para a Ciência e Tecnologia
Campus de Azurém, 4800-058 Guimarães
PORTUGAL
ffontes@mct.uminho.pt

Abstract: In this article we address the problem of determining how a structured formation of autonomous undistinguishable agents can be reorganized into another, eventually non-rigid, formation based on changes in the environment, perhaps unforeseeable. The methodology can also be used to correctly position the agents into a particular formation from an initial set of random locations. Given the information on the agents current location and the final locations, there are $n!$ possible permutations for the n agents. Among these, we seek one that minimizes a total relative measure, e.g. distance traveled by the agents during the switching. We propose a dynamic programming methodology to solve this problem to optimality. Possible applications can be found in surveillance, damage assessment, chemical or biological monitoring, among others, where the switching to another formation, not necessarily predefined, may be required due to changes in the environment.

Key–Words: Combinatorial optimization, dynamic programming, formation reorganization.

1 Introduction

Research in coordination and control of teams of several agents (that may be robots, ground, air or underwater vehicles) has been growing fast in the past few years. The main reason behind such growth is the wide range of military and civil applications where such teams can be used efficiently. Applications of cooperative teams include surveillance, patrol, intruder detection, containment of chemical spills, forest fires, etc. For a recent survey in cooperative control of multiple vehicles systems, see for example the work by Murray [8].

Team formation is a common and critical task in many cooperative agent applications, since shape formation may be considered as the starting point for a team of agents to perform cooperative tasks. Also, formation switching or reconfiguration arises at a variety of applications due to the need to adapt to environmental changes or to new tasks, possibly determined by the accomplished ones. The problem addressed is, therefore, to determine the actions that have to be taken by each individual agent so that the overall group moves into a specific formation. Among the possible actions to reorganize the formation into a new desired geometry, we are interested in finding the ones that optimize a pre-determined performance measure.

The goal of this work is to develop an approach for modeling formation switching of multiple agents. While we realize the importance of dynamic analysis of the control trajectory of each agent, our focus here is on the static optimization problem of deciding *where* each agent should go rather than *how* it should go. The problem addressed here should be seen as a component of a framework for multiagent coordination, incorporating also the trajectory control component [5], that allows to maintain or change formation while following a specified path in order to perform cooperative tasks.

Possible applications arise from reactive formation switching or reconfiguration of a team of autonomous agents. For example, when a team of agents that moves in formation through a trajectory has to change to another formation to avoid obstacles and change then back to the original formation. A possible example, depicted in Figure 1, is the need to perform a reorganization when the formation has to go through a narrow passage.

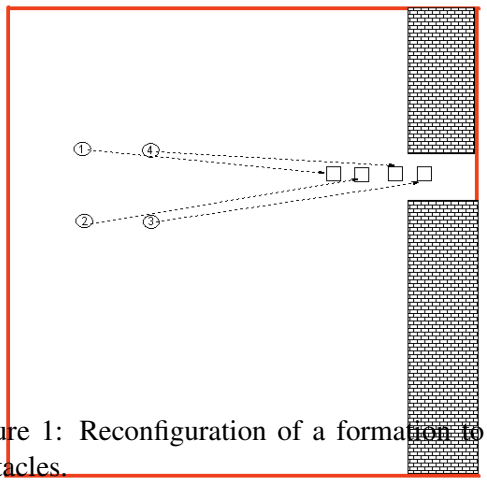

Figure 1: Reconfiguration of a formation to avoid obstacles.

Desai *et al.*, in [3], model mobile robots formation as a graph. The authors use the so-called "control graphs" to represent the possible solutions for formation switching. In this method, for a graph having n vertices there are $n!(n-1)!/2^{n-1}$ control graphs, and switching can only happen between predefined formations.

Some application examples are the following. Consider that a team of agents is performing surveillance using a specific formation and detects some intrusion. In such event it should change to another formation more appropriate to the new task in hands. This new formation may, or may not, be a predefined or structured formation. An example of a non predefined case is described by Yamagishi in [14]. In this work, the formation mission involves containment of chemical spillage. The agents task, which initially is monitoring, after detection occurs becomes to determine the perimeter of the spill. Another type of application requiring switching within specific formations happens, for example, when a robot soccer team looses the ball. In such event the team has to switch from an attack formation to a defense formation with a different geometry more appropriate to the new task.

A similar problem, where a set of agents must perform a fixed number of different tasks on a set of targets, has been addressed by several authors. The methods developed include exhaustive enumeration (see Ramussen *et al.* [11]), branch-and-bound (see Rasmussen and Shima [10]), network models (see Schumacher *et al.* [12, 13]), and dynamic programming (see Jin *et al.* [6]). See also the recent works in this journal using Cooperative/decentralized control [4, 7] and the dynamic programming technique [9].

We propose a dynamic programming approach to solve the problem of formation switching, that is the problem of deciding which agent moves to which place in the next formation. Conditions spotted during the execution of the current task are used to determine the following tasks and therefore formation, at least partially. The formation switching performance is given by the cumulative agents performance. The performance can be measured through any separable and additive function.

This paper is organized as follows. In the next section, the problem of optimal reorganization of agent formations is described and a dynamic programming formulation of such problem is given and discussed. In Section 3, we discuss computational implementation issues of the dynamic programming algorithm, namely an efficient implementation of the main recursion as well as efficient data representations. A detailed description of the algorithms is provided. Computational experiments are reported in Section 4, where the computational implementation options are analyzed and justified. Also, some examples are explored. Some conclusions are drawn in the final section.

2 Problem Description and Formulation

In our problem a team of n identical agents has to switch from their current formation to some other formation, possibly unstructured. Our approach can be used either centralized or decentralized, depending on the agents capabilities. In the latter case, all the agents would have to run the algorithm, which outputs an optimal solution, always the same if many exist, since the proposed method is deterministic.

Regarding the new formation, it can be either a pre-specified formation or a formation to be defined according to the information collected by the agents. In both cases, we do a pre-processing analysis that allows us to come up with the desired locations for the next formation.

Collision avoidance can be addressed at this level either by not allowing trajectories that lead to path crossing or simply by considering that, in such cases, the distance of the trajectory must be increased.

This problem can be restated as the problem of

allocating to each new position exactly one of the agents, located in the old positions.

Consider n agents and a set I of cardinality n indexing the agents in the present location. To each agent $i \in I$ associate a vector containing the initial location coordinates, e.g. a triplet (x_i, y_i, z_i) for each $i \in I$ if we consider the agent to move in a 3D space. Consider also a set J indexing the n possible target locations. Again a triplet (x_j, y_j, z_j) is associated to each $j \in J$. Let $d(i, j)$ denote the value of the function of the desired performance measure between positions i in the current formation and j in the next formation, for example the trajectory distance.

We want to allocate to each position in the new formation, that is to each $j \in J$, exactly one of the agents in the current formation, that is $i \in I$. In total there are $n!$ possibilites, that is $n!$ feasible solutions, from which we want to choose the one that has the best performance according to some given performance criterion.

Dynamic programming provides a framework for decomposing certain optimization problems into a nested family of subproblems, thus providing an effective method to solve combinatorial problems of a sequential nature.

Consider the sets $S \subseteq I$ and $S' \subseteq J$. Define $f(S, S')$ as a best solution of allocating the agents in S to the targets in S'. In order to compute its value efficiently we decompose the problem into a nested family of subproblems. This nested structure suggests a recursive approach for solving the original problem using the solution of the subproblems. The recursion expresses an intuitive principle of optimality for sequential decision processes; that is, once we have reached a particular state, a necessary condition for optimality is that the remaining decisions must be chosen optimally with respect to that state. This is the fundamental principal of dynamic programming, first appearing in the seminal work of Bellman [2].

This approach is computationally advantageous since it combines the solution of subproblems in order to find an optimal solution to the original problem. Subproblem solutions are stored in order to avoid recomputation.

The recursion of our problem is then written as in equation (1), for which the initial conditions are provided by equation (2). (All other states being initialized as ∞.)

$$f(S, S') = \min_{\substack{i \in S \\ j \in S'}} \left\{ f\left(S \setminus \{i\}, S' \setminus \{j\}\right) + d(i, j) \right\}. \quad (1)$$

$$f(\{i\}, \{j\}) = d(i, j). \quad (2)$$

Hence the optimal value for the performance measure is given by $f^* = f(I, J)$.

In any dynamic programming model, the state depicts the status of a multistage decision system at an intermediate point of the sequential process. The states in such a sequential decision process often group themselves into stages, with a transition occurring from a stage k to a later stage l, with $l > k$. Therefore, it is convenient to label each stage and to think of the process as evolving along the stages.

In the dynamic programming model, equations (1) and (2), a state is represented by $s \equiv (S, S')$ and consists of two state variables: the set of agents S ($S \subseteq I$) and the set of target locations S' ($S' \subseteq J$) . The sets S and S' refer to, respectively, the agents and target locations that have not yet been allocated. The model of the formation switching problem consists of n stages, $1, 2, \ldots, n$, where n is the number of agents in the problem. A stage k contains all states $s_i \equiv (S, S')$ such that $|S| = |S'| = k$. Thus, the set of states at stage k is defined as $F_k = \{s_i \mid s_i \equiv (S, S'), |S| = k, |S'| = k\}$, for all $k = 1, 2, \ldots, n$. The set of all states in the model is called the state space, $V = \{s_1, s_2, \ldots, s_\lambda\}$, where λ represents the cardinality of V (i.e. the total number of states).

3 Computational Implementation

A pure forward Dynamic Programming (DP) algorithm is easily derived from the DP recursion, equations (1) and (2). Such implementation may result in considerable waste of computational effort since, generally, complete computation of the state space is not required. Furthermore, since the computation of a state requires information contained in other states, rapid access to state information should be seek.

The DP procedure we have implemented exploits the recursive nature of the DP formulation by using a backward-forward procedure. Its main advantage is that the exploration of the state space graph, i.e. the solution space, is based upon the part of the graph which has already been explored. Thus, states which are not feasible for the problem are not computed, since only states which are needed for the computation of a solution are considered. The algorithm is dynamic as it detects the needs of the particular problem and behaves accordingly.

States at stage 1 are either nonexistent or initialized as given in equation (2). The DP recursion, equation (1), is then implemented in a backward-

forward recursive way. It starts from the final state (I, J) and while moving backward visits, without computing, possible states until a state already computed is reached. (In the first iteration, the states already computed are just the ones initialized by Eq. (2).) Then, the procedure is performed in reverse order, i.e. starting from the state last identified in the backward process, it goes forward through computed states until a state (S, S') is found which has not yet been computed. At this point, again it goes backward until a computed state is reached. This procedure is repeated until the final state (I, J) is reached with a value that cannot be bettered by any other alternative solution. The main advantage of this backward-forward recursive algorithm is that only intermediate states needed are visited and from these only the feasible ones that may yield a better solution are computed. As it will be shown later, the number of states computed using this method is very small. For our test problems it varies between 12% (for the smallest problems) and 0.05% (for the largest problems) of the state space, i.e. states represented. Furthermore, the subproblems computed, whose solutions are then combined in order to find an optimal solution, correspond to 112.5% and 0.005% of the total number of problem feasible solutions (i.e. $n!$), for the smallest and largest problems, respectively.

As said before, due to the recursive nature of equation (1), state computation implies frequent access to other states. Recall that a state is represented by two sets. Therefore, set operations like searching, deletion, and insertion of a set element must be performed efficiently. A computationally efficient way of storing and operating sets is the bit-vector representation, also called the boolean array, whereby a *computer word* is used to keep the information related to the elements of the set. In this representation a universal set $U = \{1, 2, \ldots, n\}$ is considered. Any subset of U can be represented by a binary string (a computer word) of length n in which the ith bit is set to 1, if i is an element of the set and set to 0, otherwise. So, there is a one-to-one correspondence between all possible subsets of U (in total 2^n) and all binary strings of length n. Since there is also a one-to-one correspondence between binary strings and integers, the sets can be efficiently stored and worked out simply as integer numbers. A major advantage of such implementation is that the set operations, *location*, *insertion* or *deletion* of a set element can be performed by directly addressing the appropriate bit. For a detailed discussion of this representation of sets see, for example, the book by Aho *et al.* [1].

The flow of the algorithm is managed by Algorithm 1, which starts by labeling all states (subproblems) as not yet computed, that is assigned to them a ∞ value. Then, it initializes states in stage 1, that is subproblems involving sets with cardinality 1, as given by equation (2). After that, it calls algorithms 2 and 3 in turn with parameters (I, J).

Algorithm 2 is a recursive algorithm that computes the optimal solution cost, i.e. it implements equation (1) . This function receives two arguments: the set of agent current locations and the set of target locations, both represented by integer numbers (using bit-vector representation, as discussed previously). It starts by checking whether the specific state (S, S') has already been computed. If so the program returns to the point where the function was called, otherwise the state is computed. To compute state (S, S'), all possible target locations $j \in S'$ that might lead to a better subproblem solution are identified. The function is then called with arguments (S'_j, j), where S'_j represents the set $S' \setminus \{j\}$. The computation of $f(S, x)$ involves determining two terms that are then added: a subproblem represented by $f(S'_j, j)$ and the value of the performance measure if the agent currently positioned in location i is to be moved to target location j, $d(i, j)$. If the latter term $d(i, j)$ by itself exceeds the current best then we move on to another target location without computing subproblem (S'_j, j).

For each subproblem solved, this algorithm also stores the state variables, i.e. the subset of agents and the subset of targets no yet allocated, associated with its best solution. This information is needed to retrieve the solution structure, that is which agent moves to which target location. This is done by algorithm 3.

Algorithm 3 is also a recursive algorithm and it backtracks through the information stored while solving subproblems, in order to retrieve the solutions structure, i.e. the actual agent target allocation. This algorithm works backward from the final state (I, J), corresponding to the optimal solution obtained, and finds the partition by looking at the information stored for this state, namely *agent* and *target*, with which it can build the structure of the solution found. Algorithm 3 also receives two arguments: the set of agent current locations and the set of target locations. It starts by checking whether the agent current locations set is empty. If so, the program returns to the point where the function was called; Otherwise the backtrack information of the state is retrieved and the other needed states evalu-

Algorithm 1: DP for finding agent-target allocations.

Input: The agent set and locations, the target set and locations, and the performance functions;

Compute performance measure for every pair agent-target (d_{ij});

Label all states as not yet computed

$$f(S, S') = \infty \text{ for all } S \in I \text{ and } S' \in J;$$

Initialize states at stage one as in equation (2);

Call $f(I, J)$;
Output: Solution performance;

Call $Allocation(I, J)$;
Output: Solution structure;

ated.

At the end of the algorithm, $f(I, J)$ gives the performance associated with the best agent-target allocation.

4 Computational Experiments

We consider several standard structured formations for a team of agents: line, column, square, diamond, and wedge. We also considered non-rigid formations, which are randomly generated. The agents initial positions were randomly generated and are fixed for all experiments, see Table 1.

	Location	
	x_i	y_i
Agent 1	48	374
Agent 2	15	106
Agent 3	183	64
Agent 4	348	349

Table 1: Agents random initial location.

The target positions of the new formation are given in Tables 2 and 3.

In our experiments we have decided to use the Euclidian distance although any other measure may be used. It should be noticed that the performance function can be any function as long it is separable and additive.

	Target 1		Target 2	
Formation	x	y	x	y
Line	95	258	328	258
Column	95	258	95	348
Square	95	258	207	258
Diamond	95	258	294	258
Wedge	95	258	221	258
Random	95	258	147	78

Table 2: Target locations 1 and 2, for each formation.

	Target 3		Target 4	
Formation	x	y	x	y
Line	172	258	356	258
Column	95	4	95	55
Square	95	370	207	370
Diamond	195	169	195	347
Wedge	158	293	158	368
Random	248	362	293	228

Table 3: Target locations 3 and 4, for each formation.

Algorithm 2: Recursive function: compute optimal performance.

Recursive $Compute(S, S')$;

if $f(S, S') \neq \infty$ **then**
  return $f(S, S')$ to caller;
end

Set $min = \infty$;

i is first element in S;

$S_i = S - \{i\}$;

for *each* $j \in S'$ **do**
  $S'_j = S' - \{j\}$;
  if $d_{i,j} \leq min$ **then**
    Call $Compute\left(S'_j, j\right)$;
    if $f\left(S'_j, j\right) + d_{i,j} \leq min$ **then**
      $min = f\left(S'_j, j\right) + d_{i,j}$;
      ag=i;
      tar=j;
    end
  end
end

Store information on:
  agent$(S, S') = ag$,
  target$(S, S') = tar$,
  $f(S, S') = min$.

Return: $f(S, S')$;

In figures 2 to 7 we have plotted the initial and final positions of the agents as well as the assignments of the agents to the target positions.

The algorithm was implemented in Matlab and all examples run in between 0.016 seconds (column) and 0.063 seconds (line). We note that the Matlab functions were interpreted and not compiled. In the case of faster solutions being required the code could be compiled.

A question that might arise is whether for small dimensional systems a complete enumeration search would be feasible or even faster. So, for problems of different dimensions (number of agents), with the agents generated in random location, we have also implemented a total enumeration search. The results are shown in Table 4. It can be seen that for lower dimension problems both algorithms are very fast, with the DP starting to outperform the total enumeration for problems with 6 or more agents. For larger problems the better efficiency of DP becomes obvious being almost 2000 times faster for problems including 10 agents.

Regarding the memory requirements, there is also a considerable difference as problem dimension grows. The full representation of the state space comprises $(2^n - 1)^2$ states. However, in our implementation, the percentage of the state space actually computed is very small and decreases with problem size. Therefore, if memory requirements become a problem we can change the way the state space is

Algorithm 3: Recursive function: retrieve agent-target allocation.

Recursive $Allocation(S, S')$;

if $S \neq \varnothing$ **then**
$i = agent(S, S')$;
$j = target(S, S')$;
$Alloc(i) = j$;
$S_i = S - \{i\}$;
$S'_j = S' - \{j\}$;
CALL $Allocation(S_i, S'_j)$;
end

Return: $Alloc$;

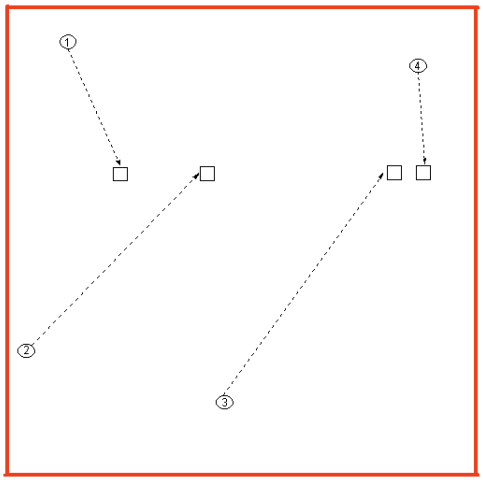

Figure 2: Line formation.

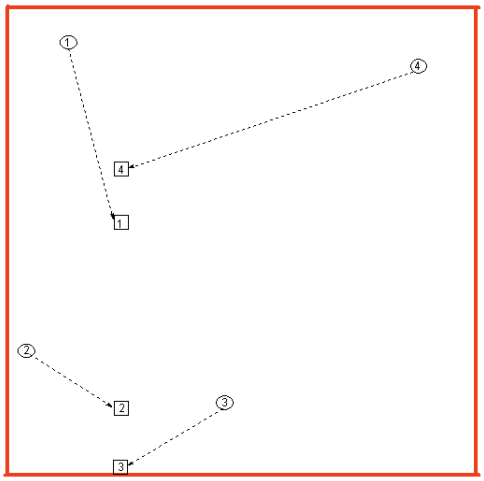

Figure 3: Column formation.

Number of agents	DP running time	Total enum. running time
4	0.031	0.015
5	0.032	0.016
6	0.047	0.125
7	0.062	0.937
8	0.14	8.781
9	0.281	93.359
10	0.562	1095.2
11	1.344	

Table 4: DP algorithm running time vs. total enumeration running time (in seconds).

represented. Currently, all state space is represented, however we can represent the state space dynamically, as needed, which drastically reduces the memory requirements.

In Table 5 we report on the total percentage of the state space used in the implementation proposed in this work, i.e the percentage ratio between the number of states computed and the total number of states represented.

As explained before, the original problem is solved by combining, in an efficient way, the solution to some subproblems. The method efficiency improves with the number of times the subproblems are reused, which obviously increases with the number of feasible solutions.

In order to evaluate the efficiency of our method,

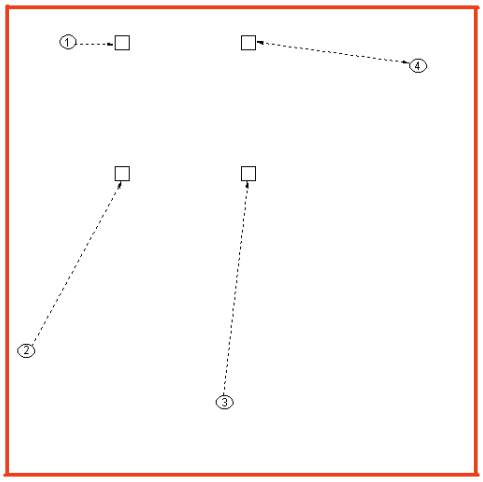

Figure 4: Square formation.

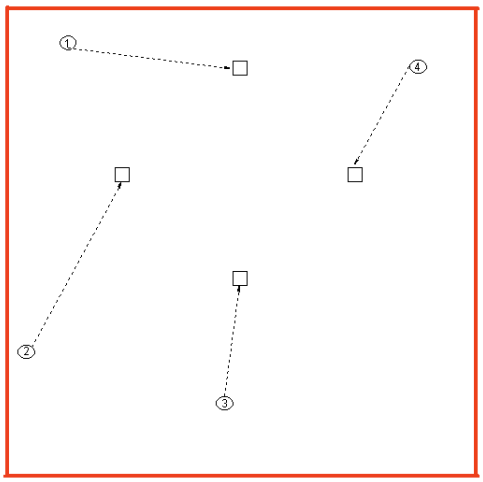

Figure 5: Diamond formation.

Agents number	State space percentage computed	initialized
4	12.00	7.11
5	5.31	2.60
6	2.34	0.91
7	1.05	0.30
8	0.48	0.09
9	0.22	0.03
10	0.11	0.019
11	0.05	0.003

Table 5: Used states as a percentage ratio of the state space.

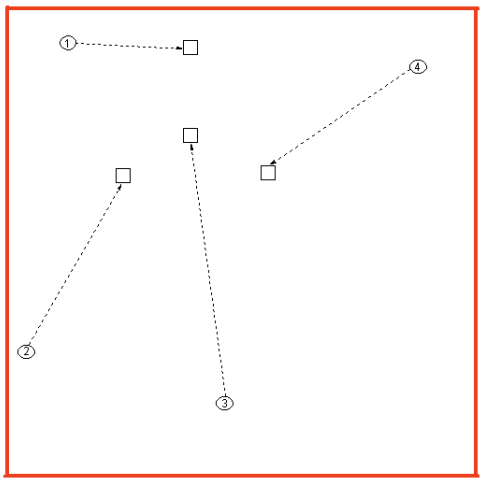

Figure 6: Wedge formation.

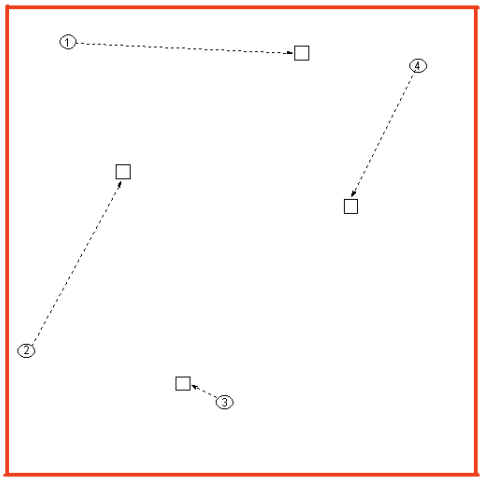

Figure 7: Random formation.

Agents number	Feasible solutions percentage computed	initialized
4	112.50	66.67
5	42.50	20.83
6	12.92	5.00
7	3.35	0.97
8	0.77	0.16
9	0.16	0.02
10	0.03	0.003
11	0.005	0.0003

Table 6: Used states as a percentage ratio of the number of feasible solutions.

we compute the ratio between the number of subproblems used and the number of feasible solutions for the problem($n!$). These ratios are reported in Table 6. As it can be seen, for very small problems the complete enumeration would provide a better method (ratio over 100). However, as problem size grows the advantage of our method becomes clear. The percentage ratios of initialized states is also reported in this table.

The benefits of our particular implementation can be seen both from (1) the fact that computing an optimal solution with it is much faster than a total computation implementation (Table 4); and (2) from the fact that only a small part of the states is actually computed, either when this number is compared to the total number of states in our representation (Table 5), or when when it is compared to the number of possible feasible solutions (Table 6).

5 Conclusion

We have developed an optimization algorithm to decide how to reorganize a formation of vehicles into another formation of different shape. The method proposed here should be seen as a component of a framework for multiagent coordination/cooperation, which must necessarily include other components such as a trajectory control component. The algorithm proposed is based on dynamic programming which performs efficiently for small dimensional problems and is much more efficient than total enumeration search as the problem dimension increases. For problems with 10 agents, it is already 2000 times faster than total enumeration. Another advantage is that the dynamic programming formulation structure is independent of the particular performance functions considered (as long as they are separable and additive). Moreover, constraints on each agent — such as maximum displacement or nonadmissible locations for each agent — might be easily incorporated.

Acknowledgements: The research was supported by FCT/POCI 2010/FEDER through project POCTI/EGE/61823/2004.

References:

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, USA, 1957.

[3] J. P. Desai, P. Ostrowski, and V. Kumar. Modeling and control of formations of nonholonomic mobile robots. *IEEE Transactions on Robotics and Automation*, 17(6):905–908, 2001.

[4] A. Errahmani, H. Ouakka, M. Benyakhlef, and I. Boumhidi. Decentralized adaptive fuzzy control for a class of nonlinear systems. *WSEAS Transactions on Systems and Control*, 2(8):411–418, 2007.

[5] F. A. C. C. Fontes, D. B. M. M. Fontes, and A. C. D. Caldeira. Model predictive control of vehicle formations. In P. Pardalos M.J. Hirsch, C.W. Commander and R. Murphey, editors, *Optimization and Cooperative Control Strategies*, Lecture Notes in Control and Information Sciences , Vol. 381 ISBN: 978-3-540-88062-2. Springer Verlag, Berlin, 2009.

[6] Z. Jin, T. Shima, and C. J. Schumacher. Optimal scheduling for refueling multiple autonomous aerial vehicles. *IEEE Transactions on Robotics*, 22(4):682–693, 2006.

[7] Adrian Korodi, Alexandru Codrean, Liviu Banita, and Constantin Volosencu. Aspects regarding the object following control procedure for wheeled mobile robots. *WSEAS Transactions on Systems and Control*, 3(6):537–547, 2008.

[8] R. M. Murray. Recent research in cooperative control multivehicle systems. *Journal of Dynamic Systems, Measurement and Control*, 129:571–583, 2007.

[9] Monica Parvan, Florian Ghionea, and Cristina Flaut. The relevant request determination in the public transport system. *WSEAS Transactions on Systems and Control*, 3(6):517–527, 2008.

[10] S. J. Rasmussen and T. Shima. Branch and bound tree search for assigning cooperating UVAs to multiple tasks. Minneapolis, Minnesota, USA, 2006. Institute of Electrical and Electronic Engineers, American Control Conference 2006.

[11] S. J. Rasmussen, T. Shima, J. W. Mitchell, A. Sparks, and P. R. Chandler. State-space

search for improved autonomous UAVs assignment algorithm. Paradise Island, Bahamas, 2004. IEEE Conference on Decision and Control.

[12] C. J. Schumacher, P. R. Chandler, and S. J. Rasmussen. Task allocation for wide area search munitions via iterative network flow. Reston, Virginia, USA, 2002. American Institute of Aeronautics and Astronautics, Guidance, Navigation, and Control Conference 2002.

[13] C. J. Schumacher, P. R. Chandler, and S. J. Rasmussen. Task allocation for wide area search munitions with variable path length. New York, New York, USA, 2003. Institute of Electrical and Electronic Engineers, American Control Conference 2003.

[14] M. Yamagishi. Social rules for reactive formation switching. Technical Report UWEETR-2004-0025, Department of Electrical Engineering, University of Washington, Seattle, Washington, USA, 2004.