Path Planning for Groups using Column Generation

Marjan van den Akker

Roland Geraerts

Han Hoogeveen

Corien Prins

Technical Report UU-CS-2010-019
June 2010

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Path Planning for Groups using Column Generation

Marjan van den Akker, Roland Geraerts, Han Hoogeveen, and Corien Prins

Institute of Information and Computing Sciences, Utrecht University
3508 TA Utrecht, the Netherlands
{marjan,roland,slam}@cs.uu.nl; C.R.Prins@students.uu.nl

Abstract. In computer games, one or more groups of units need to move from one location to another as quickly as possible. If there is only one group, then it can be solved efficiently as a dynamic flow problem. If there are several groups with different origins and destinations, then the problem becomes $\mathcal{NP}$-hard. In current games, these problems are solved by using greedy *ad hoc* rules, leading to long traversal times or congestions and deadlocks near narrow passages. We present an efficient heuristic solution that is based on Integer Linear Programming techniques.

1 Introduction

Path planning is one of the fundamental artificial intelligence-related problems in games. The path planning problem can be defined as finding a collision-free path, traversed by a unit, between a start and goal position in an environment with obstacles. Traditionally, this problem and its variants were studied in the field of robotics. We refer the reader to the books of Choset *et al.* [4], Latombe [16], and LaValle [17] for an extensive overview.

The variant we study is the problem of finding paths for one or more *groups* of units, such as soldiers or tanks in a real-time strategy game, all traversing in the same (static) environment. Each group has its own start and goal position (or area), and each unit will traverse its own path. The objective is to find the paths that minimize the average arrival times of all units.

Current solutions from the robotics field can be powerful but are in general too slow for handling the massive number of units traversing in the ever growing environments in real-time, leading to stalls of the game. Solutions from the games field are usually fast but greedy and *ad hoc*, leading to long traversal times or congestions and deadlocks near narrow passages, in particular when two groups meet while moving in opposite directions. Obviously, such solutions have a negative impact on the gameplay.

One of the first solutions for simulating (single) group behavior was introduced by Reynolds in 1987 [22]. His influential boids model, comprising simple local behaviors such as separation, cohesion and alignment, yielded flocking behavior of the units. While this model resulted in natural behavior for a flock of birds or school of fish moving in an open environment, they could get stuck in cluttered areas. Bayazit and his colleagues [2] improved this model by adding global navigation in the form of a roadmap representing the environment's free space. While the units did not get stuck anymore, they could break up, losing their coherence. By following a point that moves along a backbone path centered in a two-dimensional corridor, coherence was guaranteed by the method proposed by Kamphuis and Overmars [14]. In their method, the level of coherence was controlled by two parameters, namely the corridor width and the group area.

When multiple units are involved, possible interference between them complicates the problem, and, hence, some form of coordination may be required to solve the global problem. From the robotics field, two classes of methods have been proposed. Centralized methods such as references [23,24] compute the paths for all units simultaneously. These methods can find optimal solutions at the cost of being computationally demanding, usually making them unsuitable for satisfying the real-time constraints in games. Decoupled methods compute a path for each unit independently and try to coordinate the resulting motions [20, 26]. These methods are often much quicker than centralized methods but the resulting paths can be far from optimal. Also hybrid methods such as references [11,18] have been proposed. A variant to solving the problem is called prioritized motion planning [3, 9, 19, 28]. According to some prioritization scheme, paths are planned sequentially which reduces the problem to planning the motions for a single unit. It is however not clear how good these schemes are.

Our main contribution is that we propose an efficient solution for the path planning problem with groups. This solution translates the problem into a dynamic multi-commodity flow problem on a graph that represents the environment and uses column generation to identify promising paths in this graph. Our solution can be used to handle difficult situations which typically occur near bottlenecks (e.g. narrow passages) in the environment. The solution is efficient because it provides a global distribution of the paths while local behaviors, such as locally avoiding collisions with other units, can be handled by an external method which can be plugged into the path planning system. Such a method can be any local collision avoidance model such as the Predictive model of Karamouzas *et al.* [15] or the Reciprocal Velocity Obstacles of van den Berg *et al.* [27]. In addition, when characters follow the same homotopic path, Kamphuis' method [14] can be used to introduce coherence if desired.

Our paper is organized as follows. In Section 2, we show that path planning for one group can be solved to optimality as a dynamic flow problem. Computing the distribution of paths for multiple groups is more difficult (i.e. $\mathcal{NP}$-hard). We propose a heuristic solution that solves the corresponding dynamic multicommodity flow problem in Section 3. We conduct experiments on some hard problems in Section 4 and conclude in Section 5 that they can be solved efficiently.

2 Path planning for one group

In this section, we discuss the problem of finding an optimal set of paths for one group of units, who all want to move from their origin p to their destination s. We assume that all units in the group are equal. Hence, we do not have to specify a path for each separate unit; our solution simply consists of a set of paths, and each path in our solution can be assigned to each unit. The goal is to maximize the number of units that have reached q for each time t; using this approach, we automatically minimize both the average arrival time and the time by which all units have reached the destination. We assume here that the environment in which the units move is static.

To solve the problem, we first construct a directed graph that resembles the free space in the environment. There are several ways to create such a graph. One possibility is to use tiles, as is done in earlier games, but this is considered to lead to unnatural paths. A better alternative is to use a waypoint graph [21] in combination with a navigation mesh [10]. No matter how the graph has been constructed, we determine for each arc in the graph the traversal time as the time it takes to traverse the arc. We further determine its capacity as the number of units that can traverse the arc while walking next to each other. For instance in [10], the capacity can be computed by the minimum clearance along the arc divided by the width of a character. We choose the time unit as the time a unit has to wait until it can leave after the previous one. For each arc (i, j), we know its traversal time $l(i, j)$ and its capacity c_{ij}. Since the environment is assumed to be static, these data do not change over time. The path planning problem can then be modeled as a *dynamic flow problem* for which we have to determine a flow from the origin, which is called the source, to the destination, which is called the sink. Since we want as many units as possible that have arrived at the destination at each time, we are looking for a so-called *earliest arrival flow*.

The above problem can be solved by a classic algorithm due to Ford and Fulkerson [8], with a small adaptation due to Wilkinson [29]. The algorithm by Ford and Fulkerson computes a dynamic flow in an iterative version: given an optimal dynamic flow for the problem with $T - 1$ periods, an optimal dynamic flow for the T-period problem is constructed. Even though we do not have a deadline T but a number of units that have to go to the destination, we can use this algorithm by increasing the deadline each time by one time unit until all units have arrived.

The algorithm of Ford and Fulkerson [8] resembles the residual-graph algorithm due to Ford and Fulkerson [6] for the *static* maximum flow problem in which time plays no role, or equivalently, all traversal times are zero. The notion of time is included by assigning a time-parameter to each vertex that indicates the ‘current time’ of the vertex; flow can be sent through an arc if the difference between the time-parameters of the vertices are equal to the traversal time of the arc. When an optimal solution to the T-period problem has been constructed, the algorithm by Ford and Fulkerson [8] splits it up in a set of *chain-flows*, which can be interpreted as a set of compatible paths in the graph. The flow (units in our case) are then sent through the graph following the chain-flows, where the last unit leaves the origin such that it arrives at the destination exactly at time T. Although the decomposition in chain flows yields an optimal solution for the T-period problem, this solution does not need to be optimal when it is cut off at time t, even though the algorithm by Ford and Fulkerson [8] did find it as an intermediate product. Wilkinson [29] described a way to store the intermediate information of the algorithm to find an earliest arrival flow.

3 Path planning for multiple groups

In this section, we consider the path planning problem for multiple groups of units. For each group, we are given the origin, the destination, and the size of the group. Initially, we assume that each unit is available at time zero, and that there are no deadlines for arrival; we later will show how to deal with units that must leave the origin after time zero and/or must have arrived at their destination at a given deadline. The goal is to minimize the average arrival time of all units. Just like in the previous section, we assume that the graph that we use to model the problem is static. We further assume that the graph is directed.

Since there are different groups with different origins and/or destinations, we do not have a dynamic flow problem anymore, but a *dynamic multi-commodity flow problem*, which is known to be $\mathcal{NP}$-hard in the strong sense, since the special case in which all traversal times are zero, the so-called static multi-commodity flow problem, is $\mathcal{NP}$-hard in the strong sense for the case of integral flows [5]. Obviously, one attempt to solve the overall problem is to solve the path planning problem for each group separately using the algorithm of Section 2. If these solutions are compatible, then we have found an optimal solution for the overall problem, but we may expect the combined solution to be infeasible.

We want to present a heuristic for the problem that is based on techniques from (integer) linear programming. We refer the reader to reference [30] for a description of this theory. The basic idea is that we formulate the problem as an integer linear program (ILP), but we restrict the set of variables by eliminating variables that are unlikely to get a positive value anyway. In this way, we make the problem tractable, without loosing too much on quality.

Instead of using variables that indicate for each arc at each time the number of units of group k that traverse this arc (an arc formulation), we use a formulation that is based on *paths* for each origin-destination pair. A path is described by the arcs that it uses and the times at which it enters these arcs. Here we require that the difference in the entering times of two consecutive arcs (i, j) and (j, k) on the path is no less than the traversal time $l(i, j)$ of the arc (i, j); if this difference is larger than $l(i, j)$, then this implies that there is a waiting time at j. Initially, we assume that there is infinite waiting capacity at all vertices. The advantage of using a formulation based on path-usage instead of arc-usage is twofold. First of all, we do not have to model the 'inflow = outflow' constraints anymore for each arc, time, and group. Second, we can easily reduce the number of variables by ignoring paths that are unlikely to be used in a good solution.

Suppose that we know all 'possibly useful' paths for each origin-destination pair. We can now model our path planning problem as an integer linear programming problem as follows. First, we introduce two sets of binary parameters to characterize each path $s \in S$, where S is the set containing all paths. The first one, which we denote by d_{ks}, indicates whether path s does connect origin/destination pair k (then d_{ks} gets value 1), or does not (in which case d_{ks} has value 0). The second set, which we denote by b_{ats}, keeps track of the time t at which arc a is entered by s: it gets value 1 if path s enters arc a at time t, and it gets value 0, otherwise. Note that these are parameters, which are fixed in advance, when the path s gets constructed. Mathematically, these parameters are defined by

$$d_{ks} = \begin{cases} 1 \text{ if path } s \text{ connects origin/destination pair } k \\ 0 \text{ otherwise} \end{cases}$$
$$b_{ats} = \begin{cases} 1 \text{ if path } s \text{ enters arc } a \in A \text{ at time } t \\ 0 \text{ otherwise.} \end{cases}$$

As decision variables we use x_s for each path $s \in S$, which will denote the number of units that follow path s. We use c_s to denote the cost of path s, which is equal to the arrival time of path s at its destination. We formulate constraints to enforce that the desired number y_k of units arrive at their destination for each origin/destination pair k and to enforce that the capacity constraints are obeyed. We define K as the number of origin/destination pairs, and we denote the capacity of arc $a \in A$ by u_a. We use T to denote the time-horizon; if this has not been defined, then we simply choose that is large enough to be sure that all units will have arrived by time T. Since the environment is constant over time, the capacity u_a is independent of t. This leads to the following integer linear program (ILP):

$$\begin{array}{ll} \min \sum_{s \in S} c_s x_s & \\ \text{ subject to} & \\ \sum_{s \in S} d_{ks} x_s = y_k & \forall k = 1, \ldots, K \\ \sum_{s \in S} b_{ats} x_s \leq u_a & \forall a \in A; t = 0, \ldots, T \\ x_s \geq 0 \text{ and integral} & \forall s \in S. \end{array}$$

Obviously, we do not know the entire set S of paths, and enumerating it would last forever. Since the overwhelming majority of the paths will not be used anyway, we will make a selection of the paths that we consider 'possibly useful', and we will solve the ILP for this small subset. We determine these paths by considering the LP-relaxation of the problem, which is obtained by removing the integrality constraints: the last constraint simply becomes $x_s \geq 0$ for all $s \in S$. The intuition behind taking the relaxation is that we use it as a guide towards useful paths, since the problems are so close together that a path which will be 'possibly useful' for the one will also be 'possibly useful' for the other. The LP-relaxation can be solved quickly because there is a clear way to add paths only that improve the solution. We solve the LP-relaxation through the technique of column generation, which was first described by Ford and Fulkerson [7] for the multi-commodity flow problem and by Gilmore and Gomory [12] for the cutting stock problem.

3.1 Column generation

The basic idea of column generation is to solve the linear programming problem for a restricted set of variables and then add variables that may improve the solution value, until these cannot be found anymore. It does not matter which initial set of variables gets selected, as long as it constitutes a feasible solution. For the path planning problem, it is easy to find such a set. We can for example start with the paths that are discovered for solving the single group path planning problem. Then, we let the groups move in turn, that is, group $k+1$ has to wait until all units in group k have arrived at their destination.

Given the solution of the LP for a restricted set of variables, we check if the current solution can be improved, and if this the case, which paths we should add. It is well-known from the theory of column generation, in case of a minimization problem, that the addition of a variable will only improve the solution if its *reduced cost* is negative; if all variables have non-negative reduced cost, then we have found an optimal solution for the entire problem. In our case, the reduced cost of a path s, characterized by the parameters d_{ks} $(k = 1, \ldots, K)$ and b_{ats} $(a \in A; t = 0, \ldots, T)$ has reduced cost equal to

$$c_s - \sum_{k=1}^{K} \lambda_k d_{ks} - \sum_{a \in A} \sum_{t=0}^{T} \pi_{at} b_{ats},$$

where λ_k $(k = 1, \ldots, K)$ and π_{at} $(a \in A; t = 0, \ldots, T)$ are the shadow prices for the corresponding constraints; these values follow from the solution to the current LP. The reduced cost takes the 'combinability' of the path s into account with respect to the current solution.

Since we are testing whether there exists a feasible path with negative reduced cost, we compute the path with minimum reduced cost. If this results in a non-negative reduced cost, then we have solved the LP-relaxation to optimality; if the outcome value is negative, then we can add the corresponding variable to the LP and iterate. The problem of minimizing the reduced cost is called the *pricing problem.*

We break up the pricing problem into K sub-problems: we determine the path with minimum reduced cost for each origin/destination pair separately. Suppose that we consider the problem for the lth origin/destination pair; we denote the origin and destination by p and q, respectively. Since we have $d_{ls} = 1$ and $d_{ks} = 0$ for all $k \neq l$, the term $\sum_{k=1}^{K} \lambda_k d_{ks}$ reduces to λ_l, and we ignore this constant from now on. The resulting objective is then to minimize the adjusted path length $c_s - \sum_{a \in A} \sum_{t=0}^{T} \pi_{at} b_{ats}$. We will solve this as a *shortest path* problem in a directed acyclic graph.

We construct the following graph, which is called the *time expanded* graph. The basis is the original graph, but we add a time index to each vertex: hence, vertex i in the original graph corresponds to the vertices $i(t)$, with $t = 0, \ldots, T$. Similarly the arc (i, j) with traversal time $l(i, j)$ results in a series of arcs connecting $i(t)$ to $j(t + l(i, j))$. We further add waiting arcs $(i(t), i(t + 1))$ for each i and t. The length of the arc is chosen such that it corresponds to its contribution to the reduced cost. As c_s is equal to the arrival time of the path s in the destination, this term contributes a cost $l(i, j)$ to each arc $(i(t), j(t+l(i, j)))$ and cost 1 to each waiting arc. With respect to the term $-\sum_{a\in A}\sum_{t=0}^{T}\pi_{at}b_{ats}$, suppose that arc a corresponds to the arc (i, j). Then b_{ats}, with $a = (i, j)$, is equal to 1 if the path uses the arc $(i(t), j(t + l(i, j)))$ and zero otherwise; and therefore, this term contributes $-\pi_{at}$ to the length of the arc $(i(t), j(t+l(i, j)))$, given that $a = (i, j)$.

Summarizing, we put the length of the arc $(i(t), j(t+l(i, j)))$ equal to $l(i, j) - \pi_{at}$, where $a = (i, j)$; the waiting arcs $(i(t), i(t + 1))$ simply get length 1. The path that we are looking for is the shortest one from $p(0)$ to one of the vertices $q(t)$ with $t \in \{0, \ldots, T\}$. We use the A^* algorithm [13] to solve this problem.

Theorem 1. *The shortest path in the time expanded graph corresponds to the feasible path with minimum reduced cost for the kth origin/destination pair and vice versa.* $\square$

We compute the reduced cost by subtracting λ_l. If this path has negative reduced cost, then we add it to the LP. Since we know the shortest paths from $p(0)$ to each vertex $q(t)$, we do not have to restrict ourselves to adding only the path with minimum reduced cost, if there are more paths with negative reduced cost. If in all K sub-problems the shortest paths have non-negative reduced cost, then the LP has been solved to optimality.

When we have solved the LP to optimality, then we have found a large number of paths that at least were interesting enough to be generated during the column generation phase; therefore, all these paths are included in the ILP. Even though these path constitute a solution to the LP, there is no guarantee that they will enable a feasible solution to the ILP. Therefore, we add some additional paths as well. These paths are constructed in the following way. First, we round down all decision variables, which will lead to an integral solution that satisfies the capacity constraints, but in which not all units will arrive at their destination (if all units make it to their destinations, then we have found an optimal solution). For the remaining units, we construct additional paths using Cooperative A* by Silver [25]. These paths are added to the ILP, which is then solved to optimality by the ILP-solver CPLEX [1].

3.2 Extensions

We have shown above how we can solve the basic problem. In this subsection, we will present some extensions.

Time constraints on the departure and arrival. It is possible to specify an earliest departure and/or latest possible arrival time for each group of units. These can easily be incorporated in the paths by restricting the time expanded graph. The only possible drawback is, if we put these limits too tight, that we may make the problem infeasible. Since the problem of deciding whether there is a feasible solution is $\mathcal{NP}$-complete, we apply a computational trick. We replace the y_k in the constraint that y_k units have to move from the origin to the destination by $y_k + Q_k$, where Q_k is an artificial variable measuring the number of units of group k that did not reach their destination. We now add a term $\sum_{k=1}^{K} w_k Q_k$ to the objective function, where w_k is a large penalty weight, which makes it very unattractive for the units to not reach their target.

Changes in the environment. A change in the environment may lead to a change in the capacity of an arc (for example that it drops to zero if the arc gets closed) or to a change in the traversal time in a certain period. If we know the changes beforehand, then these are easily incorporated in our model. A change in the capacity can be modeled by making the capacity of arc a time dependent; the right-hand side of the capacity constraint becomes then u_{at} instead of u_a. A change in the traversal times can be modeled by changing the arcs in the graph that we use to solve the pricing problem.

Undirected edges in the graph. An undirected arc can be traversed both ways, which makes it much harder to model the capacity constraint. If for example the traversal time is l and we want to send x units through the edge at time t, then this is possible only if the number of units that start(ed) to traverse the edge from the other side at times $t-l+1, t-l+2, \ldots, t+l-1$ does not exceed the remaining capacity. To avoid having to add this enormous number of constraints, we split such an edge e in two arcs e_1 and e_2 that have a constant capacity over time. We do not fix the capacity distribution beforehand, but we make it time-independent by putting the capacities equal to u_{e_1} and u_{e_2}, which are two non-negative decision variables satisfying that $u_{e_1} + u_{e_2}$ is equal to the capacity of the edge. We can modify this time-independent capacity distribution a little by making u_{e_1} or u_{e_2} equal to zero until the first time it can be reached from any origin that is part of an origin/destination pair which is likely to use this arc.

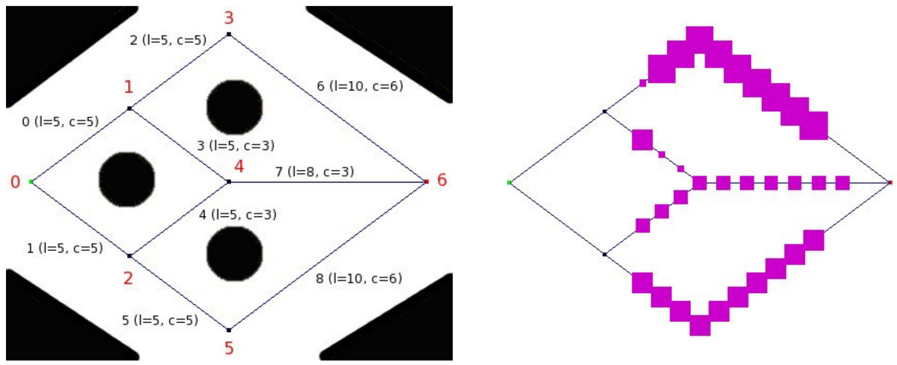
4 Experiments

In this section, we will describe the experiments we have conducted. In particular, we investigated the efficiency of our solution on three difficult problems. The solution from Section 3 was implemented in C++ using the ILOG CPLEX Concert Technology library version 11.100 for solving the LPs and ILPs [1]. All the experiments were run on a PC (CentOS linux 5.5 with linux kernel 2.6.18) with an Intel Core 2 Duo CPU (3 GHz) with 2 GB memory. Only one core was used.

Each experiment was deterministic and was run a small number of times to obtain an accurate measurement of the average integral running times (in ms). These times include the initialization of the algorithm (such as the data structures, CPLEX, heuristics, building the initial LP), the column generation, the solving of the LP and ILP, path finding, and making an integer solution.

One group

Fig. 1(a) shows the problem where one group moved from node 0 to node 6. The experiment was carried out for a single group with 100 through 500 units. Because it may be inefficient to let all units use the shortest path (e.g. when the capacity of the shortest path is low), it may be better to let a few units take an alternative path. Indeed, as is shown in Fig. 1(b), the group (with 100 units) was split to minimize the average arrival times. The algorithm took 10ms for 100 units, 40ms for 200 units, 100ms for 300 units and 250ms for 400 units. Even with 500 units the algorithm took less than half a second. In a game situation, the units should already start moving when the algorithm is executed to avoid stalls.

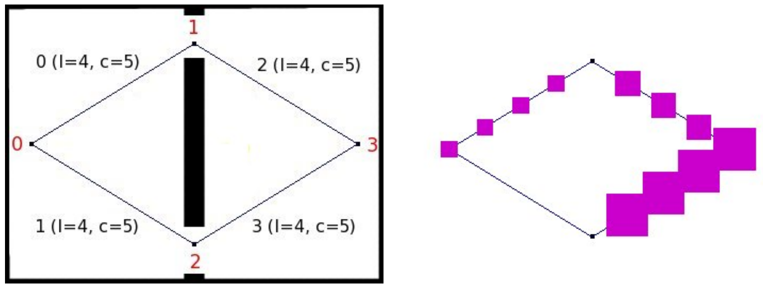


(a) Test environment (b) Output algorithm

Fig. 1. (a) The environment used for testing the division of units among the arcs. The (large) red numbers show the node numbering and the black numbers show the arc numbering. For every arc we give the length l and capacity c. (b) The output of the algorithm for 100 units at timestep 16. The pink squares symbolically represent the units, and the width of a square is proportional to the number of units.

Two groups moving in opposite directions

In the following case, as is displayed in Fig. 2(a), two groups moved in opposite directions while switching their positions. One group started at node 0 and the other one at node 3. Since the arcs had limited capacities, the units had to share some arcs. There were two different homotopic paths between these two nodes, and both paths could be used by only 5 units per timestep. On the left side we placed 10 units and on the right side we had 50 units. Computing the solution took only 10ms. Also other combinations were tested, e.g. 20 versus 50 units (20ms), 20 versus 100 units (40ms), 40 versus 100 units (70ms), and 40 versus 200 units (230ms). The latter case is visualized in Fig. 2(b). Here, most units from the right side used the lower path, and some units from the right side used the upper path. All the units from the left side used the upper path. Again, these running times were sufficiently low for real-time usage.



(a) Test environment (b) Output algorithm

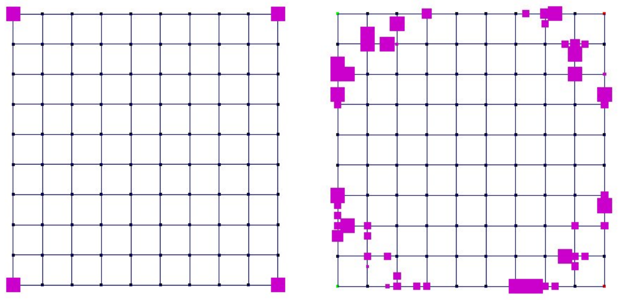
Fig. 2. (a) The environment and graph used for testing two groups moving in opposite directions, i.e. one group starts at node 0 and the other one starts at node 3. (b) The output of the algorithm for 40 versus 200 units at timestep 3.

Four groups with many units moving in a big graph

We created a large graph whose structure was a raster with arcs between the raster points. The length of these arcs was set to 3 with capacity 20. We refer the reader to Fig. 3 for an illustration of this graph. In every corner we placed 100 units that needed to move to their diagonally opposite corners. Computing their paths took 510ms. We also tested the algorithm with 1000 units placed at each corner (where the capacities were scaled with the same proportion), which took 480ms. The results clearly illustrated the scaling power of the algorithm as it did not slow down when both the number of units and the capacities were scaled proportionally.

5 Conclusion

We have presented a centralized method based on techniques from ILP for path planning problems involving multiple groups. The crux is that the LP-relaxation can be solved quickly by using column generation. The solution to the LP-relaxation can be used as a basis to construct a heuristic solution. We have described a method to find a good approximation by solving a restricted ILP. If the instance is so big that solving the ILP would require too much time, then



(a) Timestep 0 (b) Timestep 9

Fig. 3. The test environment is a raster. The output of the algorithm is displayed at timestep 0 and 9.

we can still use the solution to the LP-relaxation to find a solution by clever rounding. The units can then already start moving according to this solution while a good solution for the remaining units is determined in the meantime.

In future work, we will integrate a local collision-avoidance model, such as proposed in references [15, 27], to test whether our solution leads to visually pleasing motions. We think that our solution enhances the gameplay in difficult situations involving one or multiple groups.

Acknowledgments

This work was partially supported by the ITEA2 Metaverse1 (www.metaverse1.org) Project.

References

1. CPLEX 11.0. User's manual. Technical report, ILOG SA, Gentilly, France, 2008.
2. O. Bayazit, J.-M. Lien, and N. Amato. Better group behaviors in complex environments using global roadmaps. In *Artificial Life*, pages 362–370, 2002.
3. M. Bennewitz, W. Burgard, and S. Thrun. Priority schemes for decoupled path planning techniques for teams of mobile robots. *Robotics and Autonomous System*, 41:89–99, 2002.
4. H. Choset, K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, first edition, 2005.
5. S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal of Computation*, 5:691–703, 1976.
6. L. Ford Jr. and D. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.
7. L. Ford Jr. and D. Fulkerson. Constructing maximal dynamic flows from static flows. *Operations Research*, 6:419–433, 1958.
8. L. Ford Jr. and D. Fulkerson. A suggested computation for maximal multicommodity network flows. *Management Science*, 5:97–101, 1958.

9. T. Fraichard. Trajectory planning in a dynamic workspace: A 'state-time' approach. *Advanced Robotics*, 13:75–94, 1999.
10. R. Geraerts. Planning short paths with clearance using explicit corridors. In *IEEE International Conference on Robotics and Automation*, pages 1997–2004, 2010.
11. R. Ghrist, J. O'Kane, and S. LaValle. Pareto optimal coordination on roadmaps. In *International Workshop on the Algorithmic Foundations of Robotics*, pages 171–186, 2004.
12. P. Gilmore and R. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9:849–859, 1961.
13. P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
14. A. Kamphuis and M. Overmars. Finding paths for coherent groups using clearance. In *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, pages 19–28, 2004.
15. I. Karamouzas, P. Heil, P. van Beek, and M. Overmars. A predictive collision avoidance model for pedestrian simulation. In *Motion in Games*, volume 5884 of *Lecture Notes in Computer Science*, pages 41–52. Springer, 2009.
16. J.-C. Latombe. *Robot Motion Planning.* Kluwer, 1991.
17. S. LaValle. *Planning Algorithms.* Cambridge University Press, 2006.
18. S. LaValle and S. Hutchinson. Optimal motion planning for multiple robots having independent goals. *Transaction on Robotics and Automation*, 14:912–925, 1998.
19. Y. Li. *Real-time motion planning of multiple agents and formations in virtual environments.* PhD thesis, Simon Fraser University, 2008.
20. J. Peng and S. Akella. Coordinating multiple robots with kinodynamic constraints along specified paths. *International Journal of Robotics Research*, 24:295–310, 2005.
21. S. Rabin. *AI Game Programming Wisdom 2.* Charles River Media Inc.
22. C. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21:25–34, 1987.
23. G. Sánchez and J.-C. Latombe. Using a PRM planner to compare centralized and decoupled planning for multi-robot systems. In *IEEE International Conference on Robotics and Automation*, pages 2112–2119, 2002.
24. J. Schwartz and M. Sharir. On the piano movers' problem: III. Coordinating the motion of several independent bodies: The special case of circular bodies moving amidst polygonal obstacles. *International Journal of Robotics Research*, 2:46–75, 1983.
25. D. Silver. Cooperative pathfinding. In *Artificial Intelligence for Interactive Digital Entertainment*, pages 117–122, 2005.
26. T. Siméon, S. Leroy, and J.-P. Laumond. Path coordination for multiple mobile robots: A resolution complete algorithm. *IEEE Transactions on Robotics and Automation*, 18:42–49, 2002.
27. J. van den Berg, M. Lin, and D. Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *IEEE International Conference on Robotics and Automation*, pages 1928–1935, 2008.
28. J. van den Berg and M. Overmars. Prioritized motion planning for multiple robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2217–2222, 2005.
29. W. Wilkinson. An algorithm for universal maximal dynamic flows in a network. *Operations Research*, 19:1602–1612, 1971.
30. L. Wolsey. *Integer Programming.* Wiley, New York.