Task Assignment Problem in Camera Networks

Cerruti Federico 607374, Fabbro Mirko 586668, Masiero Chiara 600608

Abstract—This paper takes the move from a real case study and deals with the problem of task assignment in camera networks. A mathematical model is proposed and different approaches are compared. The attention is mainly focused on algorithms that derive from Constraint Linear Programming and Stable Marriage Problem (SMP). An optimality criterion over static scenarios is defined. Subsequently it is given prominence to problem dynamics, that is a crucial issue. As a matter of fact, we would like to attain a good trade-off between continuity and optimality in matching tasks and agents. The former means that it is advisable to restrain discontinuities in task execution. The latter aims to give preference to high priority tasks. The proposed generalization of the SMP algorithm highlights good performances. It assures a sufficient degree of continuity, it prevents deadlocks and it is tunable. Moreover, it can be recast quite easily as a distributed algorithm.

Index Terms—Camera networks, task assignment.

1 INTRODUCTION

1.1 The video surveillance case study

THIS work is a part of a project about video surveillance: given a set of agents distributed over a certain area and subject to network constraints and agent limitations, this project aims to solve the following problems:

1) dividing the interesting monitored area in subzones according to the positions where the cameras are placed and establishing a connectivity graph (which cameras can communicate), a topological graph (which subzones are adjacent) and a visibility graph (which subzones can be "seen" by every camera);
2) using information from the calibrated cameras and the objects that are shot both by the calibrated and the non-calibrated cameras in order to determine the parameters about the non-calibrated cameras;
3) coordinating agents for attaining an optimal match between tasks that must be executed (i.e. keeping the area monitored or detecting and tracking events that occur in the environment) and agents that can execute them (according to their positions and their graphs);
4) coordinating agents so that, when an event occurs, this event is tracked and all the interested area keeps being monitored with efficiency.

Here, we will focus on the third point.

1.2 Objective and motivations

As we stated above, this work aims to develop an efficient algorithm in order to coordinate a group of cameras distributed over an area to monitor.
We are given a set of agents and a group of requests of execution from tasks that periodically arrives at the network. We have to find a good assignment of the tasks to the agents, according to the positions of the agents and the types of the tasks, so that the number of the executed tasks is maximized and their lifetimes are minimized.
At the present, a nearly brute-force method is used for matching: the first agent that is free and can execute a task starts to execute it; if a critical situation takes place, a special agent (the so called *leader*) solves the problem of assignment.

1.3 A simple description of tasks and agents

Cameras can be considered as agents with limited resources. Hardware structure of the cameras is shown in figure (1).

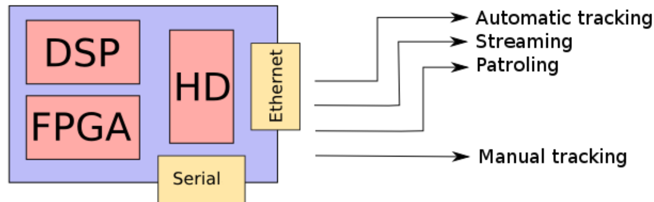


Figure 1: Hardware structure of cameras

Each agent can communicate at least with its (topological) neighbours and can execute a subset of all possible tasks. In fact, the monitored area can be divided into subzones, that are obtained intersecting the visual ranges of the cameras. With this convention, each camera covers one ore more subzones and it is possible to build a covering matrix $\mathbf{V}$ that represents the visibility graph. When a task is generated, it is also given an attribute that specifies its location (or target agent, depending on the type) of interest. A camera can execute only tasks that concern one of the subzones it covers.

Two groups of tasks can be distinguished:

- **asynchronous tasks:** each of them periodically repeats the request of its execution, until it is assigned to an agent (*pulsated broadcast*);
- **synchronous tasks:** they represent the only redundant piece of information of the network: the (synchronous) *heartbeat* contains information about the state of the network; the *snapshots* are used as backup information if necessary.

The assignment problem exists for asynchronous tasks, so we will focus mainly on them. In our simplified formalisation, they include:

1) *patroling:* it employs the hard-disk of the cameras; if an anomalous event takes place (i.e. an *event detection* occurs), it is automatically tracked and a state packet (asynchronous heartbeat) is generated and put in the network;
2) *automatic tracking:* its request is pulsated by a camera that has noticed an anomalous event during its patroling operation;
3) *manual tracking:* it employs the serial port of a camera; it has the highest priority and is executable by only one agent;
4) *streaming:* agents are used as repository of video information.

1.4 Task assignment problem: literature survey

Applications of task assignment problem can be found in many different disciplines. We have focused mainly on works related to information technology and informatics, but some of the most interesting hints were offered by other fields, such as business administration and project management.

We will now summarize the most important ideas that previous works on task assignment suggested to us. Anyway, it is important to point out that we were not able to collect specific research material on camera networks. Probably, application of task assignment algorithms to video-surveillance problems is still not a well-established practice.

Depending on available resources, features and skills of the agents, it is possible to turn to a variety of solving techniques. In this section we recall the most meaningful approaches with regard to our specific aim.

- **Brute-force approach**. Although not optimal, it is frequently used in applications because of its simplicity. A completely brute-force approach would imply the enumeration of all possible solutions to the matching problem, that is usually unfeasible. At the present, the camera networks installed by the case study company [1], are equipped with a nearly brute-force algorithm. It is not necessary to enumerate all possible combinations of task and agents: the underlying idea is that, when a new

1. **Videotec S.p.a.** Via Friuli, 6 - 36015 Schio (VI) Italy

task occurs, the first agent that gets free starts executing it (if the task can be carried out by that specific agent: as it has already been pointed out, tasks are "localized"). In addition, there migth be a special leader agent that can solve dangerous situations in which conflicts take place (i.e. when two or more agents ask to be assigned to the same task). In this cases, the leader decides the winner in the competition, usually by means of a random choice. Brute-force approach does not give any guarantees on the quality of assignment. However, it is quick and robust to node failure (i.e. when a camera loses the contact with the rest of the net). Moreover, the presence of a leader agent prevents deadlocks.

- **Identification based approach**. A completely different point of view is the one offered by M. J. Feiler in his article [6]. This approach was firstly proposed in the field of identification of dynamic systems. Its aim is to match the elements of an agent set $\mathcal{A}$ with elements belonging to an information set $\mathcal{S}$. The standing assumptions is that agents can communicate their positions with each other. A simple algorithm that provides the estimates is:

$$\tilde{\Theta}_i(t+1,t) = (1-\eta_i\left[\Theta(\tilde{t},t))\right])\tilde{\Theta}(t,t) \quad (1)$$

where

$$\eta_i\left[\tilde{\Theta}(t,t)\right] = \frac{||\tilde{\Theta}_i(t,t)||^{-2}}{\sum_{k=1}^{N}||\tilde{\Theta}_k(t,t)||^{-2}} \quad (2)$$

The left-hand term of equation (1) is the distance, existing at time $t+1$, between agent i and the piece of information $\Theta \in \mathcal{S}$ appeared at time t. The second equation (1) implies that the distance decreases as much as the esimate $\hat{\Theta}$ gets closer to the real parameter Θ. The main result of the article is the following. If the number of agents is greater or equal to $|\mathcal{S}|$, for all $\Theta \in \mathcal{S}$ there is a $\hat{\Theta} \in \mathcal{A}$ such that

$$\lim_{t\to\infty}\hat{\Theta}(t) = \Theta \quad a.s.$$

It should be underlined that the contents of this work are not immediately linked to our specific setting. Nevertheless, reading this article was meaningful, in the first part of our work, in order to define some characteristic features of our project. In particular, we realized that the main problem about our agents was their lack of homogeneousness. A camera can perform only tasks regarding its specific visual zone, or the whole net. This happens because both tasks and agents are "localized", i.e. each of them can affect only a subset of the comprehensive area. Actually, there are some "global" tasks, such as the *heartbeat* broadcasting. However, this kind of tasks can be modelled as default net activities, that do not interfere with the out-and-out tasks, that are involved in our matching problem. Another discrepancy between our context and the work proposed by Feiler, is that, in the latter, it is assumed that each agent knows the position of everyone else in the net exactly and without delays. This is a strong hypothesis; moreover, it is difficult to define a distance function in the case of such non-homogeneous agents. Finally, the number of agents in our case study is usually less than the number of tasks. In conclusion, despite of some proposed and implemented ad-hoc improvements, the approach of [6] does not seem to be suitable to our model of camera networks, but gave us help to undestand the particular features of our project.
- **Market-based approach** The basic idea of this approach is that it is possible to obtain a match between agents and tasks by means of auctions. In the simplest implementation of it, agents play the role of bidders. Depending on the specific context, many different implementations are available. For instance, these methods are frequently used in coordinating mobile vehicles in Search and Rescue missions. Some interesting works in this field are [3], [2] and [7]. Another article that provides an exhaustive description of marked-based methods and their applications to task assignment (paying particular attention to mobile

vehicles) is [5]. In contrast with other non-cooperative methods, market-based methods require that agents can communicate with each other, or at least with their neighbours (i.e. the agents that are interested in the same task and take part in the auction). In [5] different auction heuristics are analysed. The authors deal with the problem of defining adequate cost and utility functions for bidders (agents) and targets (tasks), in order to speed up the auctions. Moreover, it is suggested the idea that, once a matching is found, it can be subsequently improved by swapping tasks between agents, during execution, without reiterating the expensive (in terms of synchronization and communication costs) auction procedure.

- **Game theory based approach**. A point of view that is frequently adopted in mathematical project management is the one that looks to game-theoretical algorithms. A typical situation can be the assignment of tasks to employees. An interesting scheme is proposed in [1]. The key idea is to take into consideration the preferences of both managers and employees. The issue is formulated in order to resume a weighted multiple knapsack problem, in which tasks play the role of items, whose cost is required time for execution; each employee has an allotment of available time, that can be viewed as a knapsack to fill. The proposed strategy is the following:
  1) Since this assignment problem is a *stable marriage problem* (in order to collect information about SMP, we have referred, for instance, to [9] and [4]), it can be handled by means of a Gale-Shapley like algorithm [8].
  2) Each task is given a weight (that corresponds to the time it requires in order to be executed), a preference list and a pointer to the next non-rejected preference. Each agent is equipped with a similar structure, that includes available time, a preference list, a pointer to the next non-rejected preference and a callback list.
  3) The algorithm proceeds taking a task from the set of unattached tasks and recurring to the agent indicated by its pointer to the next non-rejected preference. If the agent is not busy or prefers this task to the one that it is currently performing, it accepts the task. If the replaced task can no longer be executed in the time left to the worker, it becomes unattached and establishes a callback. If the current task is not accepted, it moves to the next non-rejected preference. When an agent has completed its task, it can recall the highest of preferred tasks it has a callback for. The algorithm is iterated, until it ends when all tasks have been executed or rejected by all agents.

  We have developed this approach in order to fit our case study problem. This is fairly more complex, since, even though our assumptions are very simple, it can be shown that we have to face a NP-hard assignment problem. This is due to the fact that charging non-homogeneous agents with localized task implies the presence of incomplete lists with ties (this issue will be developed in section (3.3.1).

2 CONTENTS OF THE FOLLOWING SECTIONS

Here we report what the following sections will deal with.
In Section Two we describe task assignment problem more in detail and specify our assumptions. We also draw a mathematical formalisation of the problem: we show that a possible implementation is represented by a Constraint Linear Programming approach, even though our case study is quite difficult because our problem is dynamic. We introduce some indexes to compare the different implemented approaches and evaluate their performances. Then we make a digression about the Stable Marriage Problem and its variants, the topics that have mainly inspired our work. Finally we go over the approaches we propose and we briefly describe

them, highlighting their *pros* and *cons*, besides their computational complexities.
Section Three contains simulations. Firstly, we desribe the task simulator we have built in order to test our algorithms. Then, simulation results are analysed.
In Sections Four and Five, we conclude our project with some comments on attained results, comparisons among the proposed approaches and some opportunities for future works.

3 MATCHING TASKS AND AGENTS

As we hinted in the previous sections, we are given a group of cameras distributed over an area to monitor. This area is divided into subzones owing to the intersection of the visual ranges of the cameras.

Each of them, that can be considered as an agent with limited resources, is able to cover a subset of all the possible subzones of the monitored area. As a consequence, it is possible to define a covering matrix $\mathbf{V}$.
In our treatment, we suppose that every subzone is covered by at least two cameras. The reason of this assumption is that we would like to assure continuity in monitoring each zone. In fact, if a zone were covered by a single agent and this agent were forced to perform streaming, since this task is not compatible with video-recording, surveillance would fail.
Our problem is to assign the tasks that occur to the agents, obeying to the previously introduced costraints, in order to attain a good match. The quality of the resulting match has to be defined in some ways: we will point out this issue in section (3.3).
Each task is different from the others by some characteristics, i.e. its type (as explained in section (1.3)) and its location (or target agent). We assume that when a task occurs, it is added to a structure called *pool* and is removed from it when it is completed or gets obsolete. Actually, it is meaningful to define a dropping procedure that contrasts obsolescence: when a task exceeds a fixed lifetime, it is removed although not completed, unless it is already associated to an agent. This solution prevents the pool from an excessive growth. Moreover, a task that has not been executed after a certain fixed time, can be reasonably considered unprofitable, so it should be removed.
The *pool* is thought to be a global structure, so all the agents are aware of the existing tasks at every instant and their characteristics. The agents can access to information about all the tasks. However, as we stated above, they cannot perform all of them, but only those ones that concern one of the subzones they cover.
In our treatment, we disregard the aspects related to communication problems: no delays or losses of packets of information are possible and so agents have an instantaneous knowledge of the state of the net and the *pool*. For simplicity, we have simulated a task generation method that may be far from reality in some situations. This happens because tasks are created without correlation (i.e. a task of *manual tracking* on a certain subzone can take place when an identical task is already existing). However, this is not an oversimplification, since it does not affect the pith of the resolving procedure. In conclusion, thanks to this abstraction, we have been able to turn a complex problem about camera network management into a simpler matter: the issue of matching tasks to agents.

3.1 Mathematical formalisation

The next step to take is to draw a mathematical formalisation of the assignment problem we are interested in. A natural implementation is offered by a constraint linear programming (CLP) approach, that aims to build a model that resembles:

$$\max \mathbf{c}^T(\mathbf{x})$$
$$\mathbf{A}\mathbf{x} \leq \mathbf{b}$$
$$\mathbf{x} \geq 0$$

However, we have to face a dynamic problem. The number of task is not fixed: new ones can occur, increasing the length of the pool, whereas others can be completely performed or may be dropped, shortening it. This implies consequences that are not usually taken into

consideration in linear programming. More in details, task dynamicity induces variations of the polyhedron containing decision variables. We are going to analyse this matter after describing the CLP model.

To begin with, the variables of the optimization problem have to be defined.

$$x_{ij} = \begin{cases} 1 & \text{if agent } i \text{ executes task } j \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where $i \in \{1, \ldots, N\}$ and $j \in \{1, \ldots, H\}$. The network has N cameras and we are considering a pool of length H.

Under our assumptions on the skills of agents and the type of tasks, we can impose that assignment has to obey to the inequalities:

$$\sum_{j=1}^{H} x_{ij} \leq 1 \quad \forall i \tag{4}$$

$$\sum_{i=1}^{N} x_{ij} \leq 1 \quad \forall j \tag{5}$$

The meaning of (4) is that each agent can execute at most one task, whereas (5) implies that each task is assigned at most to one agent.

Our first model was thought in order to describe the constraints imposed by the covering matrix $\mathbf{V}$. This is a $N \times M$ matrix, where M is the number of subzones (whereas N is the number of agents, as we have already stated). Its elements are defined as:

$$V(i,j) = \begin{cases} 1 & \text{if agent } i \text{ covers area } j \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

The covering matrix $\mathbf{V}$ is supposed to be known exactly by the agents. Moreover, w.l.o.g. it is possible to assume that each task is equipped with a structure containing information about it. In particular, it is possible to derive the area to which the task j aims, by inspecting the structure field $task(j).loc$. In order to model covering costraints, other variables had to be introduced:

$$\bar{v}_{ij} = \begin{cases} 0 & \text{if } i \text{ can monitor } task(j).loc \\ 1 & \text{otherwise} \end{cases}$$

The additional costraint on covering was imposed by the equations:

$$\sum_{j=1}^{H} \bar{v}_{ij} x_{ij} = 0 \quad \forall i \tag{7}$$

Finally, the problem involves integer values, such that:

$$x_{ij} \in \{0,1\} \, \forall i,j \tag{8}$$

It should be noticed that equations (7) make the problem more complicated. For instance, they prevent the constraint matrix $\mathbf{A}$ from being TUM (totally unimodular), in general. By a clever choice of the utility function, however, it is possible to skip out equations (7). For now, we postpone the question of defining an adequate (linear) utility function $f(\mathbf{x})$. There is only one assumption we are interested in now. It is that the function strongly penalizes "wrong" assignments. By "wrong", we mean that agents are given tasks that are incident to zones they can not monitor. The resulting model is given by constraints (4), (5), (8), aiming to find $\max f(\mathbf{x})$. A key observation is that now the constraints matrix $\mathbf{A}$ is totally unimodular. Let N be the number af agents and H the length of the pool (number of existing tasks, both waiting or being performed). The expansion of the constraints (4) results in:

$$\begin{gathered} x_{11} + x_{12} + \cdots + x_{1H} \leq 1 \\ x_{21} + x_{22} + \cdots + x_{2H} \leq 1 \\ \vdots \\ x_{N1} + x_{N2} + \cdots + x_{NH} \leq 1 \end{gathered}$$

whereas constraints (5) bring:

$$\begin{gathered} x_{11} + x_{21} + \cdots + x_{N1} \leq 1 \\ x_{12} + x_{22} + \cdots + x_{N2} \leq 1 \\ \vdots \\ x_{1H} + x_{2H} + \cdots + x_{NH} \leq 1 \end{gathered}$$

As a consequence (and ordering variables so that they appear as $x_{11}, \ldots, x_{1H}, x_{21}, \ldots, x_{2H}, \ldots, x_{N1}, \ldots, x_{NH}$),

it follows that the constraint matrix $\mathbf{A}$ has the form:

$$\mathbf{A} = \begin{bmatrix} 1 & \dots & 1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \dots & \dots & \dots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \dots & 0 & 1 & 1 & 1 \\ 1 & \dots & 0 & 1 & \dots & 0 & 1 & \dots & 0 \\ 0 & 1 & \dots & 0 & 1 & \dots & 0 & 1 & \vdots \\ \vdots & \vdots & \dots & \dots & \dots & \dots & \vdots & \vdots & \vdots \\ 0 & \dots & 1 & 0 & \dots & 1 & 0 & \dots & 1 \end{bmatrix}$$

Since all elements are equal to 1 or 0 and each column contains exactly two non-null elements, $\mathbf{A}$ is TUM. As a consequence, conditions (8) are now redundant and the solving procedure is strongly simplified.

It is now interesting to focus on what we meant before when talking about dinamicity of the case study assignment problem. This idea is linked to the fact that, after a match has been found, a new task can occur (increasing the pool) whereas others may be dropped or completed (so that they disappear from the pool). This influences the dimension of the polyhedron containing variables. In order to make the following tractation more intuitive, we can observe that it is possible to represent each combination of the values of x_{ij} by a string of binary digits:

$$x_{11}x_{12}\dots x_{1H}x_{21}\dots x_{2H}\dots x_{N1}\dots x_{NH}$$

Disregarding the constraints (4) and (5), it can be easily seen that there are 2^{NH} different strings, where N is the number of agents and H is the current dimension of the task pool. For simplicity, Z is defined to be the set of all possible strings. Feasible strings with respect to the former constraints can be interpreted as a subset X of Z. The cardinality of X can be obtained by combinatorial calculus. It is given by

$$|X| = \sum_{k=0}^{N} \binom{N}{k} \frac{H!}{(H-k)!} \tag{9}$$

The kth addend corresponds to the number of solutions in which k agents are busy.

A natural representation for the set Z is possible in a NH - dimensional space, where each string can be seen as a vertex of the unit NH-cube (i.e. the unit hypercube in $\mathbb{R}^{NH}$).

New task occurence

Firstly, we analyse what the occurence of a new task implies. The main result is that Z becomes $(HN+N)$-dimensional. Its cardinality is multiplied by 2^N: an idea of how the polyhedron is affected by the new occurrence is given in figure (2).

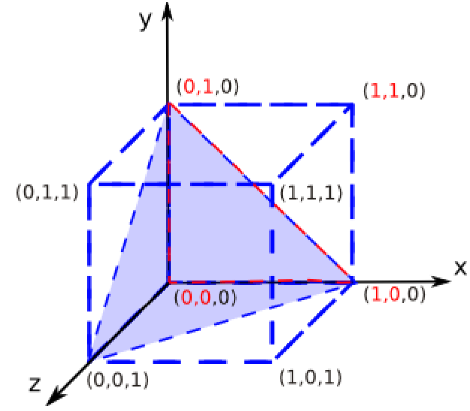


Figure 2: Initial condition: two tasks and one agent. The corresponding polyhedron containing feasible solutions is the red triangle. The new occurence implies that Z becomes a cube (before, it was the square given by the red vertices). The new polyhedron is coloured in blue.

The number of feasible values can be computed by means of equation (9), by replacing H with $H+1$. All feasible combinations corresponding to H tasks in the pool, keep feasibility when a new task occurs (they correspond to new assignments in which the new task is not chosen). However, it is quite difficult to say what happens to the optimal solution. For simplicity, we consider now that the utility function $f(\cdot)$ rewards only assignment that are consistent with the constraints imposed by the covering matrix $\mathbf{V}$. For instance, it could assign a positive value that depends on the task type. Of course, it has to penalise wrong assignment, as we have already assumed. It should also be underlined that the optimal solution could be not unique (there could be permutations of agents and tasks in the same zone, for instance).

It is possible that the occurence of a new task does not imply modification in the optimal solution (for instance, if all agents of the interested area are already performing more convenient tasks). However, the new occurence may cause that all (or nearly all) agents decide to change their tasks with a better one, now available. In fact, when a new task occurs, very different scenarios can tak place.

Let us consider a case in which a new occurence completely mix up the optimal solution.

Example 1: We refer to an early, very simple definition of the utility function $f(\mathbf{x})$ that obeys to our assumptions:

$$f(\mathbf{x}) = \sum_{i,j} c_{ij} x_{ij} \tag{10}$$

The gain c_{ij}, associated to the choice of assigning task j to agent i (i.e. putting $x_{ij} = 1$), is given by:

$$c_{ij} = \begin{cases} pr(j) & \text{if } i \text{ can monitor } j.loc \\ -\infty & \text{otherwise} \end{cases} \tag{11}$$

Where $j.loc$ is the area to which the task aims, whereas the value of pr depends only on the type of task j. It can be chosen as:

$$pr(j) = \begin{cases} 0.25 & \text{if } j.type \text{ is patroling} \\ 0.5 & \text{if } j.type \text{ is streaming} \\ 0.75 & \text{if } j.type \text{ is automatic tracking} \\ 1 & \text{if } j.type \text{ is manual tracking} \end{cases} \tag{12}$$

We analyse a simple network configuration, characterized by the covering matrix

$$V = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

We assume that $v_{ij} = 1$ if agent i can monitor area j. As it can be easily seen in figure (3), a new occurence can bring to a new matching that is completely different from the previous one. □

We have modeled patroling activity as a task in order to achieve a better uniformity among tasks[2]. With regard to the utility function (11), it is possible to say that, if patroling tasks are enough to keep all the agents busy, each optimal solution is characterized by the fact that all agents are busy.

Proposition 1: If patroling tasks are enough to keep all agents busy, in each optimal solution all agents are busy.

Proof: By absurd. Suppose that there is an optimal solution s in which at least one agent a is unloaded. Since patroling tasks are enough to keep all agents busy, if a is unloaded it means that other agents have been matched to all the patroling tasks that could be executed by a. As a consequence, there is another solution s' such that:

- all other agents are performing the same kind of task they are assigned to in s.
- there is a permutation in the assignment of patroling task, so that a can perform patroling, too.

This permutation do exist since patroling tasks never change (once they have been create in the first execution) and they are sufficient to keep all agents busy. But s' is characterized by a profit that is $f_{s'}(\mathbf{x}) = f_s(\mathbf{x}) + 1$. So s is not an optimal solution. □

Actually, an agent that can not perform tracking or streaming should automatically change its state to patroling, that should be considered a default state.

The occurence of a new task cause a drastical growth in the number of the possible matching in which all agents are busy.As a matter of fact, it becomes:

$$\frac{(H+1)!}{(H+1-N)!} = \frac{H+1}{H+1-N} \frac{H!}{(H-N)!}$$

An enumerative procedure would imply to compute the utility for the new potentially optimal solutions (in which all agents are busy and the new task is assigned). The number of new cases

2. Our approach creates an initial pool whose elements are patroling tasks, one for each area that has to be monitored. A sufficient condition to assure that patroling tasks are enough to keep all agents busy is that the covering matrix $\mathbf{V}$ contains a permutation of the columns of the identity matrix I_N

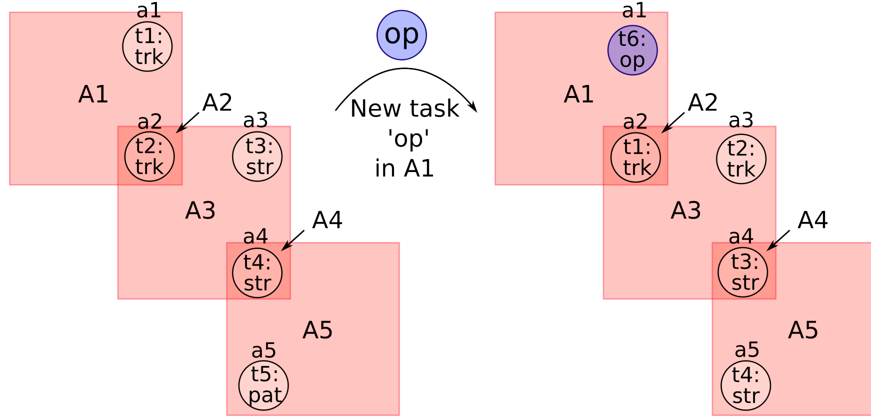


Figure 3: A new occurence can mix up the previous assignment

to be considered is:

$$(\frac{H+1}{H+1-N}-1)\frac{H!}{(H-N)!} =$$

$$\frac{N}{H+1-N}\frac{H!}{(H-N)!}$$

Dropped or completed task

When the length of the pool is decrease by one, the dimension of Z is reduced to $(H-1)N$. So, the number of possible strings in Z becomes $2^{(H-1)N}$ (it is divided by 2^N). This can happen in two cases.

1) **A task is dropped**. If a task is dropped, it means that, during its whole life, other tasks were more advantageous. As a consequence, the vanishing of the dropped task does not affect the optimality of the current solution.
2) **A task is completed**. If a task is completed, it means that an agent becomes unloaded. If the agents are capable to start patroling when no other tasks are available, of course optimality is lost. This happens because, under our assumptions, only streaming and tracking - both manual and automatic - tasks are subjected to dropping [3]. So, the agent that was previously busy was executing a task that was more convenient than patroling.

3. Patroling tasks are supposed to have infinite service and dropping time, since they represent the default state of the net

In conclusion, it is difficult to understand how the change in the length of the pool affects the optimum solution. As a consequence, this formulation of the problem suggests two straightforward choices. The first one is to completely repeat the assignment procedure. This solution assures optimallity. The second one is to implement a greedy heuristic. For instance, if a new task occurs, the agents that can monitor its area should evaluate it, allowing swaps if it is convenient but not affecting the assignment that has already been reached in other subzones. If a task is completed, instead, the agent that has just carried it out should evaluate all the unmatched tasks it is able to perform. Of course, this kind of heuristic can not guarantee the optimality of the final solution. An example can show it clearly.

Example 2: The situation is the one illustrated in figure (4). The covering matrix is:

$$V = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

The pool comprises the following tasks:

1) 'OP' on agent 1;
2) 'STR' in area A1;
3) 'TRK' in area A2;
4) 'TRK' in area A3;

The optimum solution is not unique; for istance, it can be the assignment in which the following pairs have been formed:

1) (a1,'OP' on agent 1)
2) (a2,'TRK' in area 2)

The other optimum solution is (a1,'OP' on agent 1) and (a2,'TRK' in area 3). Referring to the

utility function described by (11), the optimum solutions are characterized by $f(\mathbf{X}) = 1.75$. We consider the first one. What happens if the first task ('OP' on agent 1) is completed? Applying the previously described greedy heuristic, the assignment that involves agent 2 is never questioned. The best that agent 1 can do is to start executing task 'STR'. The final value of $f(\mathbf{x})$ is 1.25. However, if the optimum assignment procedure had been repeated for both the agents, the solution would have been (a1, 'TRK' in A2) and (a2,'TRK' in A3), with $f(\mathbf{X}) = 1.5$.

3.2 Performance metrics and constraints

Before introducing the analysis of the different proposed algorithms, it is fundamental to define some measurement criterion for the reached performances. The intrinsic complexity of the studied application, and in particular the necessity of working with dynamic assignments, imply heterogeneous and often contrasting goals. On one side, in fact, we look for an optimal assignment in the sense of priorities (i.e. it is preferable to execute critical tasks, instead to run the less important ones), on the other one we pursue a certain continuity in task's execution, avoiding situations in which tasks are temporary paused even if almost completed, or continuously swapped between different agents before the end of execution. Nevertheless we may want to avoid unaccomplished tasks queue growing too large, or infinite waiting time for tasks (also if those are low-priority tasks). Those and other aspects are formally presented in the following.

Qualitative constraints

- **Robustness.** Common requirement for every algorithm is the convergence to a stable assignment. In this paper randomized methods are also considered that may lead to continuously swapping configurations. In those cases iteration-breaking mechanisms are implemented to guarantee a finite time convergence.
- **Queue stability.** Another desirable goal is keeping a length-bounded pool (according with reasonable low generation rate of new tasks). This is in general strongly dependent on the particular instance of problem. To ensure the overall stability of the task pool, task dropping after a time-out period is necessary. As a side effect of such a policy, the oldest tasks, that are likely obsolete and no more critical, are cleaned from the queue.

Performance metrics

- **Optimality.** A measure of optimality (in the sense of task's type priority) is given by

$$P(t) = \sum_{\tau=0}^{t} \sum_{n=1}^{N} p_n(\tau) \tag{13}$$

  where p_n is the intrinsic priority of task executed by agent n at the (discrete) time τ and N is the number of agents in the system.
- **IDLE avoidance.** In a real implementation of the surveillance system, there are not agents doing properly *nothing*: if a camera has nothing better to do it is always free to patrol its covered areas. IDLE state for an agent means to us that that agent found no compatible high-priority it can run and all the areas it covers are already patrolled by others. Such a situation reveals an unbalanced assignment configuration (as stated by proposition 1). Let be the IDLE occurrence rate

$$I(t) = \frac{1}{t} \sum_{\tau=0}^{t} \sum_{n=1}^{N} \mathrm{Idle}_n(\tau) \tag{14}$$

  where $\mathrm{Idle}_n(\tau)$ is 1 when the agent n is IDLE at time τ, 0 otherwise.
- **Complexity.** Algorithm's computational complexity (expressed in term of $O(\cdot)$) gives an estimation both of computational load per agent (i.e. necessary iteration to converge to a stable assignment) and scalability of the system (depending on number of agents N, areas M and instantaneous size of the task pool $H(t)$).
- **Assignment (dis-)continuity.** This metric concerns how much agents tend to complete

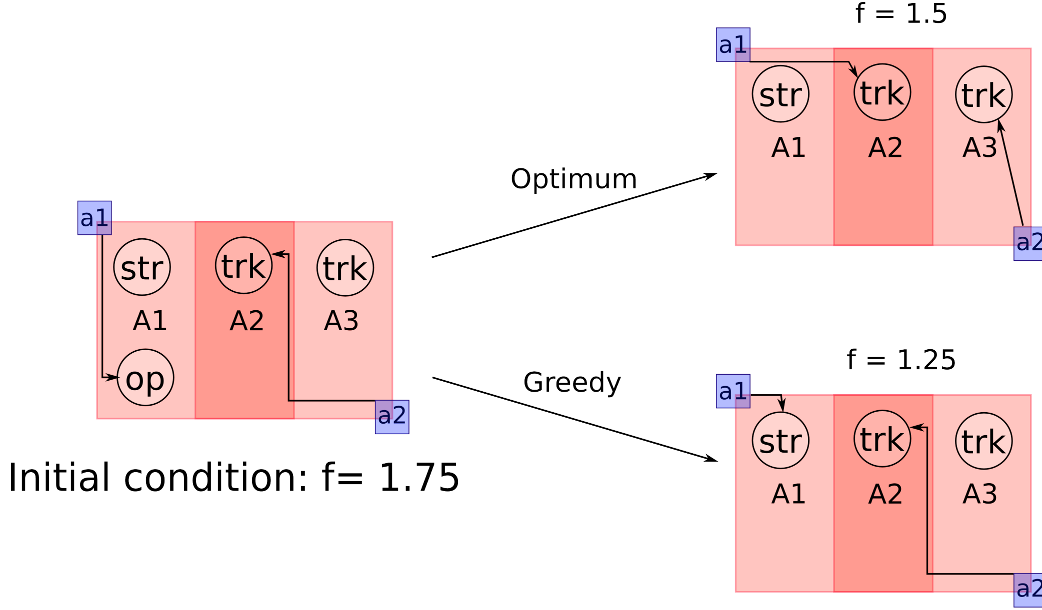


Figure 4: Greedy heuristic vs optimum matching

by themselves tasks they already started to execute. Let define

$$D(t) = \frac{1}{t} \sum_{\tau=0}^{t} \sum_{n=1}^{N} d_n(\tau) \qquad (15)$$

with $d_n(\tau) \in \{0,1\}$. Here $d_n(\tau)$ is 1 if agent n has changed its running task for another one before completion, 0 otherwise [4]. $D(t)$ $(\in [0,1])$ represents in this way a "mean discontinuity rate" indication, that it is preferable to minimize.

- **Average waiting time.** The time every high-priority task spends waiting in queue strongly depends on new tasks arrival rate, but is symptomatic of system inefficiency, too. Let be the average waiting time

$$W = \frac{1}{|\mathcal{H}_C|} \sum_{h \in \mathcal{H}_C} T_{\text{end}}(h) - T_{\text{occ}}(h) - T_{\text{serv}}(h) \qquad (16)$$

where $T_{\text{end}}(h)$, $T_{\text{occ}}(h)$, $T_{\text{serv}}(h)$ are respectively time of completion, time of creation and pure execution time of task h in the set of completed tasks $\mathcal{H}_C$.

4. If an agent has been running a patrolling task, $d_n(\tau) = 1$ only if the new task type is still patrolling (obviously in a different area).

- **Dropping rate.** A well tuned time-out parameter allows to keep the queue size bounded. Nevertheless setting this variable on small values involves the risk of dropping large numbers of tasks, more than necessary. The dropping rate is defined as

$$F = \frac{|\mathcal{H}_{\text{drop}}|}{|\mathcal{H}|} \qquad (17)$$

with $\mathcal{H}_{\text{drop}}$ the set of dropped tasks and $\mathcal{H}$ the set of all high-priority tasks.

Instance parameters

Another interesting goal is value instance size both of the system $(N, M, \mathbf{V})$ and the task occurrence. Since the behaviour of the assignment is strongly dependent on the particular realization, we may ask if, under certain hypothesis, it is possible to conclude on intrinsic feasibility of a good resulting assignment. In order to keep a simple structure of the problem, let assume that task arrival process is a Poisson process, i.e. task generations are independent and memoryless. In addition, the concerning areas of tasks, their *localizations*, are supposed uniformly distributed over all M areas. Let define for each kind of task

- $\overline{T}_{\text{occ}}^{(\cdot)}$ as the average interarrival time. This is the mean interval between the generation of consecutive tasks *of the same type*.

- $\overline{T}_{\text{serv}}^{(\cdot)}$ as the average service time of a task, in other words the mean time such a task need to spend in execution before being considered completed.

Considering the previously introduced typology of task, there will be, in general, different values of $\overline{T}_{\text{occ}}^{\text{OP}}$, $\overline{T}_{\text{occ}}^{\text{TRK}}$, $\overline{T}_{\text{occ}}^{\text{STR}}$ and $\overline{T}_{\text{serv}}^{\text{OP}}$, $\overline{T}_{\text{serv}}^{\text{TRK}}$, $\overline{T}_{\text{serv}}^{\text{STR}}$. Patrolling tasks are particular cases as these are not properly said tasks, however. Nevertheless, to maintain a certain consistence and homogeneousness, consider patrolling job as a set of tasks (one for each area $m \in \{1 \dots M\}$) generated at the initial time ($t = 0$) and never completable, hence $\overline{T}_{\text{serv}}^{\text{PAT}} = \infty$. The system size is represented, quantitatively, by

- Number of total agents N.
- Number of total areas M.
- The covering matrix $\mathbf{V}$ as defined in equation (6) and more specifically, the mean covered area per agent

$$\overline{v}_{\text{agent}} = \frac{1}{N} \sum_{\forall(n,m)} v_{nm} \tag{18}$$

and the mean agent covering an area

$$\overline{v}_{\text{area}} = \frac{1}{M} \sum_{\forall(n,m)} v_{nm}. \tag{19}$$

A suppletory condition is that each agent is able to cover at least two different areas and each area is covered at least by two agents [5].

Next step is computing a load factor that takes into account all of the variable listed above. Let be

$$\begin{aligned} c_{\text{op}} &= \overline{T}_{\text{serv}}^{\text{OP}} / \overline{T}_{\text{occ}}^{\text{OP}} \\ c_{\text{trk}} &= \overline{T}_{\text{serv}}^{\text{TRK}} / \overline{T}_{\text{occ}}^{\text{TRK}} \\ c_{\text{str}} &= \overline{T}_{\text{serv}}^{\text{STR}} / \overline{T}_{\text{occ}}^{\text{STR}} \end{aligned} \tag{20}$$

the single load of different tasks. Since each operator task is compatible with only one agent and tracking and streaming tasks are feasible by all agents covering relative area, we may conclude that:

5. This to avoid overdetermined solutions.

- OP tasks are equally distributed on the N agents;
- TRK and STR tasks are equally distributed on the areas and executed by the agents according to the topology.

So

$$C = \frac{1}{N} c_{\text{OP}} + \frac{\overline{v}_{\text{agent}}}{M \cdot \overline{v}_{\text{area}}} \left(c_{\text{trk}} + c_{\text{str}} \right) \tag{21}$$

is taken as a global load factor. When $C > 1$ the incoming rate of new tasks is faster than the completion rate, hence, even ignoring limitation represented by the covering matrix $\mathbf{V}$, the system won't be able to satisfy all tasks.

3.3 Proposed approach

3.3.1 The stable matching problem

Our work was mainly inspired by the *stable marriage problem* [8].

It considers a set of n men and n women. Each person has a list (ordered by his preference), where all the persons of the opposite sex appear. We have to find a set of stable marriages between the men and the women.

A marriage is said to be stable if there are no dissatisfied pairs, that is there is no pair of a man and a woman who both prefer another partner to their current one (according to their preference lists).

The algorithm proposed in [9] is the following: an unpaired man X considers the first woman on his list and removes her from it. If the woman is not engaged, she accepts his proposal, otherwise she considers her preference list: if she prefers X to her current partner, she breaks it with her mate (who gets unpaired) and marries X, otherwise X is still unpaired because the woman is happier with her husband. These operations are repeated while there are men unpaired (see also alorithm 1).

In this way, once a woman becomes attached, she remains married, altough she can change her partner (if she receives a better marriage proposal). As a man eliminates one woman from his list during every iteration, if the rounds continue long enough, all men and women will be married and it is assured the termination of the algorithm. This marriage is shown to be

stable: let us assume we have a dissatisfied pair, i.e. a man *X* that prefers a woman *b*, to his current wife *a*. This implies that *X* must have proposed to *b* before *a*. Woman *b* either rejected him or, at first, she accepted him but then refused him for another better man. So, *b* must prefer her current husband to *X*, contradicting the initial assumption that *b* id dissatisfied: so the marriage is stable.

Algorithm 1 Gale-Shapley Algorithm

```
1:  while there is an unpaired man do
2:    pick an unpaired man X
3:    remove the first woman w from his list
4:    if w is engaged then
5:      if w prefers X
        to her current partner Y then
6:        set (X, w) as paired
7:        set (Y, w) as unpaired
8:      else
9:        X is still unpaired
10:     end if
11:   else
12:     set (X, w) as paired
13:   end if
14: end while
```

This algorithm was modified by Brent Lagesse that in [1] faces the problem of the assignment of tasks to employees, taking into consideration manager and employees preference, employee time and employee skills. A new more specific approach was proposed, which utilises two-sided matching: each task is given a weight, that is the estimated time required to complete the task; each employee has an allotment of time available to fill with tasks. The algorithm is the one described in section 1.4. This algorithm terminates when all tasks have been assigned or they have been rejected by all workers. Termination is assured since the preference lists mean that are no deadlock.

In video-surveillance case study, cameras (agents) and tasks play the role of duties and employees.

Nevertheless, a direct application of this algorithm to our case study problem was not possible, because it is more complex owing to dinamicity and the characteristics of tasks and agents. So we have studied some variants of the *stable marriage problem* (referring to [4], in particular):

- **Stable marriage problem with incomplete lists (SMI)**. This is the case when the involved preference lists can be incomplete: the number of men and women does not have to be the same and each person's preference list consists of a subset of the members of the opposite sex in strict order. Now, a matching M is a one-one correspondance between a subset of men and a subset of women, where each member of a pair is acceptable to the other one, that is each one appears in the preference list of the other one.
  M is said to be a stable marriage if there is no acceptable pair (m, w), such that m and w are either both unmatched in M or prefer the other to his/her partner in M.
  As in the classical case, there is always at least one stable matching for an instance of the *stable marriage problem with incomplete lists*, that can be found in a polinomial time.
- **Stable marriage problem with ties (SMT)**. This extension of the *stable matching problem* arises when lists contain *ties*. In this case, a person, we say a man, prefers each woman of a tie to all the women belonging to every subsequent tie and is indifferent among the women of a single tie.
  A matching M is (weakly) stable if there is no pair of m and w, each of whom prefers the other to his/her partner. For an instance of this problem, a weakly stable matching is bound to exist, and can be found in polinomial time by breaking all ties in an arbitrary way.
- **Stable marriage problem with ties and incomplete lists (SMTI)**. This variant is obtained combining both extensions described above.
  In this context, a matching M is stable if there is no pair of m and w, each of

whom is either unmatched in M and finds the other acceptable or strictly prefers the other to his/her partner in M. A stable matching can be found by breaking all the ties, but the ways in which ties are broken affect the solution. Finding a solution of maximum cardinality is a NP-hard problem.

In the end we have chosen an algorithm that draws on from these concepts and tries to avoid some problems related to the NP-hard complexity of the problem. It will be described in the following section.

3.4 Proposed Algorithms

We have developed two algorithms, that are based on the Stable Marriage Problem. In order to evaluate their performances, we are going to compare them with other task assignment procedures. Some of them have been already introduced, the others will be explained subsequently. The algorithms that are going to be studied are the following.

- Centralized assignment.
- Nearly Brute-force (Purely random assignment).
- Greedy assignment.
- SMTI Revised.
- Randomized SMTI Revised.

They have been implemented and simulated using Mathworks MATLAB™. Before we show the results of these simulation, it is meaningful to premise a brief description and some considerations about them.

Centralized assignment

This algorithm follows directly from the formulation of the problem given in subsection (3.1). It aims at founding the best solution with regard to the maximization of a utility function similar to (11). Actually, it has been improved in order to weigh the life span of tasks. The resulting function is:

$$f(\mathbf{x}) = \sum_{i,j} c_{ij} x_{ij}$$

where

$$c_{ij} = \begin{cases} \alpha pr(j) + (\gamma)\frac{T_{lftm}(j)}{T_{drop}(j)} & \textit{if } j.loc \in i.v \\ -\infty & \textit{otherwise} \end{cases} \quad (22)$$

We say that $j.loc \in i.v$ if agent i can monitor the area affected by task j. The term $pr(j)$ is the priority associated to the type of the task, $T_{lftm}(j)$ is the duration of the life of task j and T_{drop} is the fixed time after that task j is dropped. The choice of this kind of function has been motivated by some considerations about the features we would like to attain. The former addend weighs the role of the intrinsic priority of the task, whereas the latter one assigns higher priority to tasks whose life span is close to drop time (they might be dropped soon, if no agent carry them out). It should be pointed out that α and γ are positive or null, such that $\alpha + \gamma = 1$ [6].

Since the problem can be reduced in each time interval in a CLP issue without integer conditions, we can solve it simply by means of the simplex algorithm, for instance [7].

Let us analyse the features of this algorithm.

- **Pros:** The assignment is optimum with regard to the function (22).
  It assures the absence of deadlock: it is sufficient to make it obey to the Bland law (which prevents cyclic degeneration of simplex, by fixing the order of choice of the variables to be taken as basis, when there are doubtful situations).
- **Cons:** The worst case is characterized by the complexity $\binom{n}{k}$ where n is the number of variables and k is the number of constraints (in particular, in the case study problem we have $n = NH$ and $k = N + H$. In the worst case, it means that the number of possible iterations is:
  $$\frac{(NH)!}{(N+H)!(NH-(N+H))!}$$
  Moreover, it cannot be implemented in a distributed version.

6. Since we are interested in comparing the role of priority and life span, α and γ are chosen to provide a convex linear combination of them

7. We have chosen Simplex algorithm as a solving procedure in our code implementation

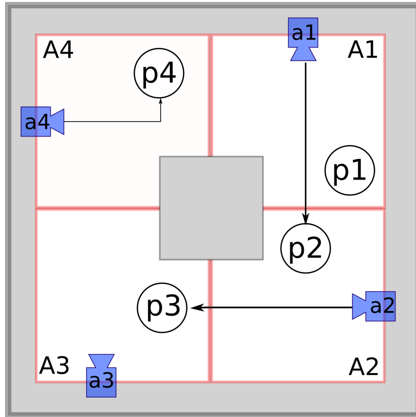


Figure 5: With Nearly Brute-force agents can be IDLE. An example is given by the case in which there are only patroling tasks, one per area, and agent a_i can patrol areas i and $(i+1)\ mod\ 4$. A possible result of random assignment in this situation lets agent a_3 unloaded.

Nearly Brute-force (Purely random assignment)

This is the simplest assigment procedure that can be implemented. Each agent a_i, with $i \in \{1, \dots, N\}$, performs the sequence of operations that is described by the flow chart in figure (6).

This algorithm is very similar to the procedure that is presently put in practise (it resembles the *brute-force* approach we have described in section (1.4), under our assumptions on the pool structure). Let us analyse the features of the Nearly Brute-force algorithm.

Its computational complexity in the worst case is $O(NH)$, where N is the number of agents and H is the pool length. This is due to the fact that an agent can look through the whole length of the pool before finding a task it can perform. This can be easily understood referring to our implementation. Patroling is considered a task like the others and we initialize the pool inserting a task of patroling for each subzone to be monitored. Since these tasks cannot be brought to a close (since they have infinite service time), they never leave the pool. Other tasks queue up. The structure of the resulting pool is shown in figure (7).

The number of zones to be monitored is given by M and the total number of existing tasks

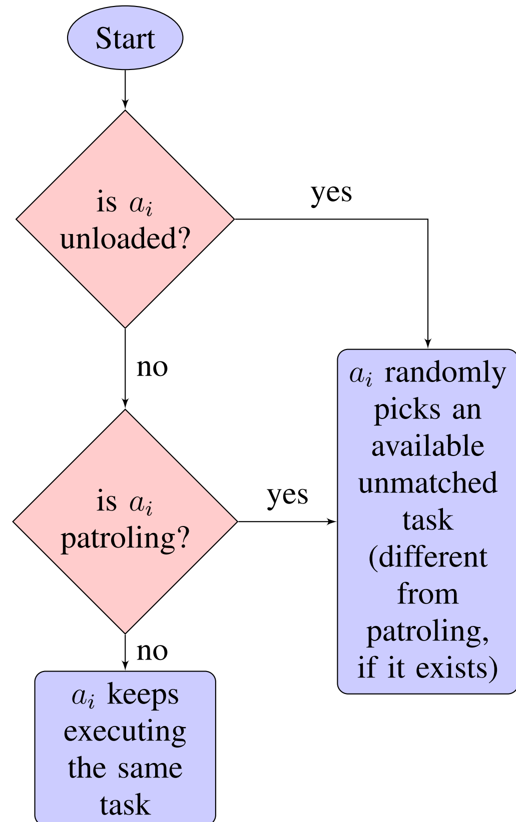


Figure 6: Nearly Brute-force

is given by H. If we suppose that an agents randomly chooses tasks that are different from patroling, if there are any, the worst case occurs when an agent does not find any suitable tasks of this type and has to glance through all the tasks before considering the patroling ones (so it takes into consideration at first $(H-M)$ tasks, then at most all M patroling tasks - actually, if we suppose that an agent covers more than one zone, it has to consider less than M tasks before finding a patroling task that is suitable for it).

- **Pros**: It assures maximum continuity in matching tasks and agents. An agent cannot drop the task that it has chosen, apart from the patroling tasks (that are to be considered as an astraction of the default state of the cameras).
  This approach easily allows scaling, since it does not request communications among the agents (except for the synchronization of global information regarding the pool, eventually). Of course, the scaling cost de-

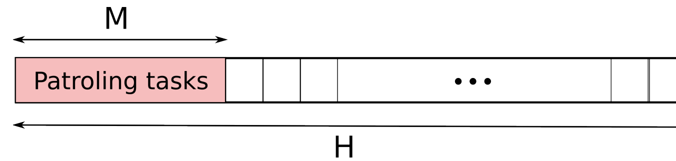


Figure 7: Example of the pool structure

pends on the dimension of the pool.
Deadlocks could occur only if two or more agent asked for the same task simultaneously. However, this situation can be easily avoided. On the one hand, in the centralized implementation agents are considered sequentially; on the other one, it is sufficient to recur to a *leader* agent to solve conflicts. For instance, this hierarchically higher agent can settle differences by operating a random choice.

- **Cons**: No optimality criterion directs the assignment, that is completely random.
  There could be IDLE agents (under the assumption that patroling is a task like the others and not the default state of each agent). An example is exhibited in figure (5).

Greedy assignment

This algorithm is quite similar to the previous one, except for the fact that assignment is not chosen randomly. Each agent evaluates all the tasks, than it chooses the best one (of course, it has to be a task that aims to a zone covered by the agent). The profit associated to a task can be established by means of the function (22). The sequence of operations performed by each agent are shown in flow chart (8)

The computational complexity of this algorithm is always $O(NH)$, since each agents need to glance through the whole pool to evaluate the tasks.

- **Pros**: This algorithm, as the previous one, easily allows scaling, since the cost of communication is low and each agent behaves almost regerdless of other agents' choices. Conflicts can be solved, as we have already stated, by means of a *leader* agent, so deadlocks are prevented.

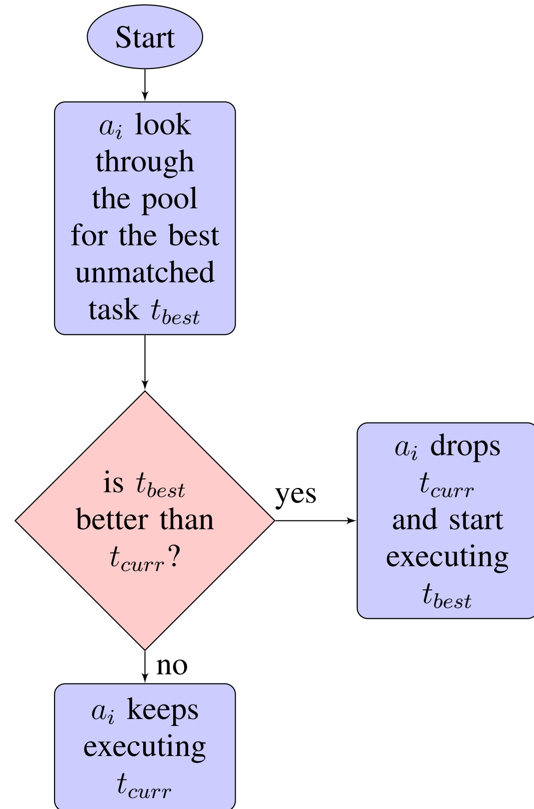


Figure 8: Greedy Assignment. The task that is currently performed by a_i is indicated by t_{curr}.

- **Cons**: Continuity is not assured. A task may be dropped by an agent before it comes to the end, since the agent may find a better task to perform.
  As in the previous case, there can be IDLE agents, even if the optimal assignment would match all the agents with a task.

Our algorithm: SMTI Revised (a.k.a. The Wedding Planner algorithm)

The algorithm we have implemented to solve our task assignment problem is the following (see

also algorithm 3).
Each task has a fixed preference list, that is created once and for all at the moment of the creation of the task; it contains all the agents that can execute that task (according to the covering matrix, as explained in the previous sections).
Each agent draws up a preference list, that contains the tasks it can execute, ordered by a criterion that keeps into account the lifetimes of tasks and their priorities (see the algorithm 2 for details); ties and incomplete lists are possible.
So an agent, chosen at random, builds temporary lists, containing only tasks with the same scores as defined in its entire list. The temporary lists are scrolled, beginning from those containing tasks with the highest scores (in descending order).
Three situations are possible:

1) the current agent is executing a task of the temporary list it is considering: in this case, no change takes place and the algorithm switches to the next agent;
2) some tasks belonging to the current temporary list are unassigned: in this case, the current agent takes the first of these tasks (and if it was executing a task, this gets unmatched) and the algorithm switches to the next agent;
3) all the tasks belonging to the current temporary list are assigned. A swap can occur in two situations:
    a) the current task in the temporary list appears in a lower position in the list of the current agent A than in the list of the agent B that is executing it. In fact, if A did not take this task, its list would risk to be empty and so A to get IDLE.
    b) the current task in the temporary list appears in the same position of both agents' lists and the current agent can execute a lower number of tasks; that is because we can think that an agent that can perform few tasks will be subject to a lower number of requests from them.

    If a swap does not take place, the agent goes on in considering the following tasks in the temporary list, then the lower placed items in its global preference list. If the end is reached without a match, the algorithm switches to the next agent

These operations are repeated until a stable matching is attained.

Algorithm 2 Compute Score

1: **function** $score$ = computeScore($task$)
2: $score \leftarrow \alpha * (task\ priority) +$
$(1-\alpha) * (task\ lifetime)/(task\ drop\ time)$;

Algorithm 3 SMTI Revised

```
1: function SMTI Revised
2: global X  pool   % structures containing agents and existing tasks respectively;
3: for each agent do
4:    compute scores of tasks that are compatible with the current agent (using function computeScore in algorithm 2);
5:    build a list containing tasks that are compatible with the current agent, placed by a score decreasing order;
6: end for
7: while (matching is unstable) do
8:    A ← vector containing all the agents placed with a random order;
9:    for n ∈ A do
10:      build temporary lists for X(n) containing only its compatible tasks with the same scores, beginning from those with the highest scores;
11:      loop
12:         consider the current temporary list of X(n) (tempList);
13:         if X(n) is executing a task that appears in tempList then
14:            break: switches to another agent;
15:         end if
16:         if no assignment has just taken place then
17:            if a task in tempList is unassigned then
18:               t ← first unassigned task in tempList;
19:               set (X(n), t) as paired;
20:               break: switches to another agent;
21:            else
22:               consider the following temporary list;
23:            end if
24:         end if
25:         if no assignment has just taken place then
26:            if all tasks in tempList are assigned then
27:               ft ← first task in tempList;
28:               while no swap takes place do
29:                  t ← current task tempList;
30:                  posTn = position of t in X(n).list;
31:                  posTcurr = position of t in the list of the agent is executing it;
32:                  if (posTn ≺ posTcurr) OR (posTn = posTcurr   &&
                     X(n) can execute a lower number of tasks)  then
33:                     set (X(n), t) as paired;
34:                     break: switches to another agent;
35:                  else
36:                     consider the following temporary list;
37:                  end if
38:               end while
39:            end if
40:         end if
41:      end loop
42:   end for
43: end while
```

The computational complexity of this algorithm is $O(N^2)$, where N is the number of agents that monitor the area of interest.
It is interesting to point out that, if we assume that each agent has complete lists with no ties, the worst case occurs when agents' lists contain tasks with exactly the opposite order than tasks'lists.
Every time the function is invocated, each agent builds temporary lists containing only tasks with the same scores, beginning from those with the highest scores: for our hypotheses, each of them contains only one task. Actually, agents do not build H temporary lists (where H is the number of task in the *pool* at the current time), because, as soon as an assignment takes place, the algorithm switches to the next agent and the temporary lists with lower scores are not created. However, the first agent builds one temporary list, the second one creates two temporary lists and so on. The nth agent builds:

$$N + (N-1) + \cdots + 1 =$$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

Computational complexity in the worst case of the proposed SMTI Revised algorithm can not fall away, since lists are likely to be shorter. However, optimality is not assured any more, since the final solution may not be of maximum cardinality.

- **Pros** This algorithm allows scalability, since it requests communications only between an agent and its neighbours.
  No deadlock is possible: two agents can prefer the same task, but the conflict is solved by breaking the ties in the lists, building temporary lists and defining a swap policy.

- **Cons** Continuity is not assured: tasks can be swapped in order to attain a stable match.
  Agents can be IDLE, i.e. if an agent's list gets empty.
  Finally, the achieved solution may be non-optimum from the point of view of the maximization of function (22), as shown in the following examples.

Example 3: The swapping policy of SMTI Revised may not be sufficient to achieve the best assignment. For instance, the situation could be the one shown in figure 9. The best matching plans that task 1 is assigned to a_1, task 2 to a_2 and task3 to a_3. However if the algorithm starts from agent a_1, the final assignment is not optimal. As a matter of fact, a swap between a_1 and a_2, that would lead to the best configuration, is not possible, after a_1 has been matched to task 2. This is because task 2 is in the same position of a_1 and a_2' preference lists and agents are all equivalent in terms of the number of task they can be matched to.

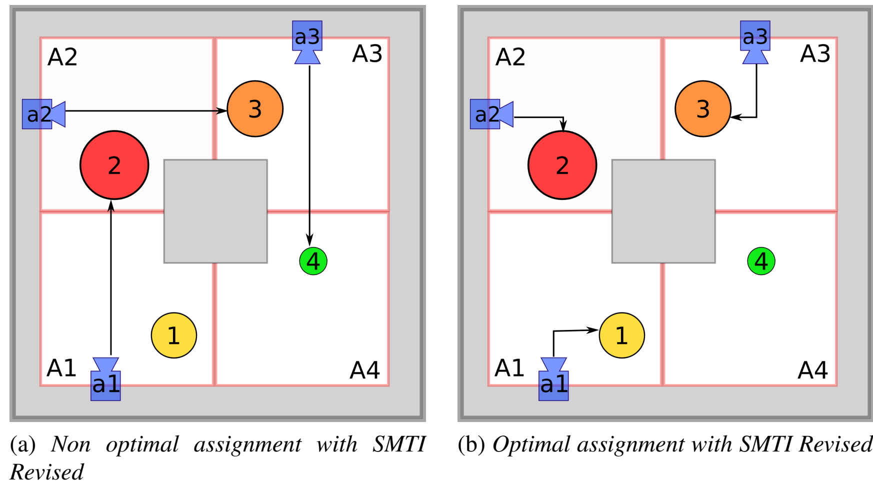
Example 4: Another example is given in figure 10, showing how the sequence in which agents are taken into consideration and the equivalence of the agents themselves can affect the final solution. The left figure shows the worst possible stable matching. This can be attained, for instance, if the forth agent, a_4 is the last that is considered, when its favourite task has already been assigned. A swap can not occur since the contested task is the first of a list whose length is two for both a_3 and a_4. Moreover, they have the same number of feasible tasks (i.e. two). The right figure shows the best possible matching.

Randomized SMTI Revised

This algorithm is similar to the previous one. The only difference is when a swap can take place:

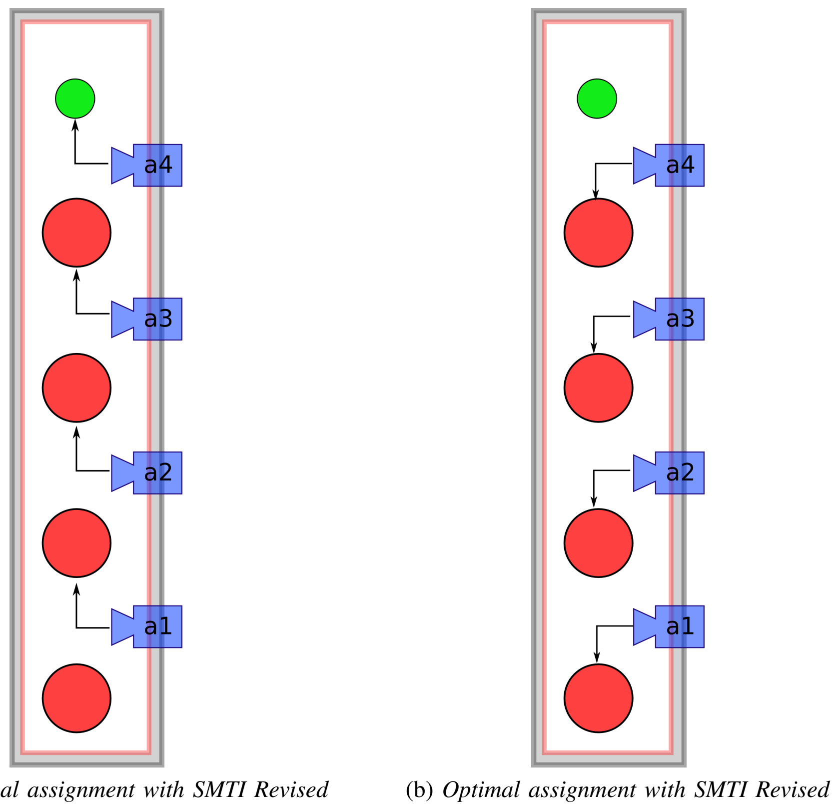
1) if the current agent can execute a lower number of tasks than its opposing agent;
2) according to the outcome of a coin toss, if the two agents can execute the same number of tasks.

The main risk of this algorithm is that it continues to repeat the same assignment procedure for ever, generating a deadlock. This is the reason we have introduced an upper limit to the number of iterations that can be executed.
Our intentions were to solve the bad situations of the previous algorithm, but we have found that it does not imply big advantages.



(a) *Non optimal assignment with SMTI Revised* (b) *Optimal assignment with SMTI Revised*

Figure 9: The swapping policy may not be sufficient to achieve the best assignment in SMTI Revised



(a) *Non optimal assignment with SMTI Revised* (b) *Optimal assignment with SMTI Revised*

Figure 10: Another example in which SMTI Revised does not guarantee optimality

4 SIMULATIONS

4.1 Task Simulator

We have built a task simulator in order to test the algorithms we have projected.
Two parameters can be chosen:

- the number of agents that monitor the area of interest
- the number of subzones the area is divided in, by intersecting the visual ranges of cameras, as explained in the previous sections.

It is defined a global structure called *pool*: it contains all tasks generated, until they are completed or dropped. We recall the definition of covering matrix $\mathbf{V}$ given in equation (6):

$$V_{ij} = \begin{cases} 1 & \text{if agent } i \text{ covers subzone } j \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

Tasks are generated in a simple way: we assume they are Poisson processes, that are processes whose interarrival times are i.i.d. random variables with exponential distribution. Each task structure contains:

- ID
- Instant when it occurs for the first time
- type: *patroling*, *automatic tracking*, *manual tracking* or *streaming*
- location or target agent (depending on the type) of interest
- service time, that is the required total time to complete its execution
- residual service time, that is the remaining time before its execution is completed
- (normalized) priority, according to the type
- a list of compatible agents, according to the covering matrix $\mathbf{V}$
- the agent that is executing it at the current time
- drop time: after this time, the task is removed from the *pool* even though it has not been completed, because it has got obsolete.

Each agent structure is defined by:

- ID
- the task it is executing at the current time
- the set of subzones it covers
- the scores of compatible tasks
- a preference list containing compatible tasks, ordered by a desecending scores
- the position of *pool* where compatible tasks are placed
- a counter of the number of iterations when the agent is IDLE.

4.2 Results

In this section we analyse the simulation results, in order to compare the previously described algorithms. The Randomized SMTI Revised algorithm will not be dealt, since it does not exhibit significant advantages with regard to the SMTI Revised. The simulations were performed by means of Mathworks MATLAB™R2009a (7.8.304). Unless differently specified, the attained results are a mean over ten random realizations of the pool, with the same instance parameters:

- N = 8 (number of agents);
- M = 9 (number of monitored zones);
- Covering matrix:

$$\mathbf{V} = \begin{bmatrix} 1&1&0&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0&0 \\ 0&1&0&1&0&1&0&0&0 \\ 0&1&0&0&1&0&1&0&0 \\ 0&0&1&0&1&0&0&1&0 \\ 0&0&1&1&0&0&0&0&1 \\ 0&0&0&1&0&1&0&0&1 \\ 0&0&0&0&1&0&1&1&0 \end{bmatrix}$$

- $T^{\text{OP}}_{\text{occ}} = T^{\text{TRK}}_{\text{occ}} = T^{\text{STR}}_{\text{occ}}$ = 2 s.
- $T^{\text{OP}}_{\text{serv}} = T^{\text{TRK}}_{\text{serv}} = T^{\text{STR}}_{\text{serv}}$ = 3 s.
- Simulation time: $T = 500$ s.
- Load factor $C \approx 0.48$.

The considered environment is described in figure 11.

To begin with, we take into consideration the performances in terms of dropped tasks. The plot in figure 12 shows the importance of the choice of the parameter T_{drop}. As predictable, the longer it is, the smaller is the dropping rate. The maximum value it can assume is determined by the practise implementation, as it describes the maximum allowed duration of a task life. After T_{drop}, a task is to be considered obsolete, so it is removed from the pool.

Still considering the dropping rate, it is interesting to compare the behaviour of the different

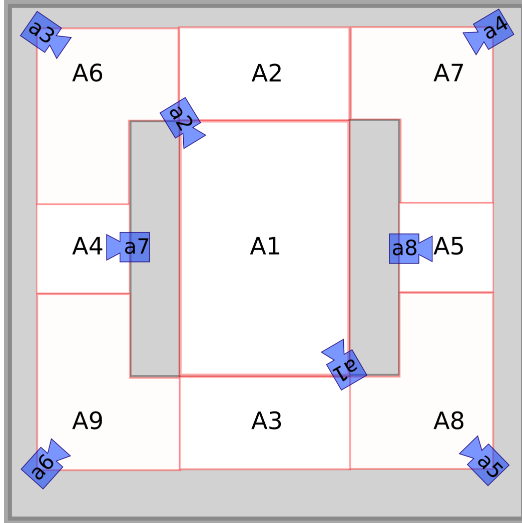


Figure 11: Simulation environment

types of tasks, depending on the values of parameters α and γ.

Figure 13a shows how the highest priority tasks are preferred when $\alpha = 1$. This occurs because α weighs the intrinsic priority of the tasks. On the one hand, since this is maximum for tasks of manual tracking -the so called "OP" tasks- PLI, SMTI Revised and Greedy (that make use of the utility function described in equation (22) show extremely low dropping rate for this kind of tasks. On the other hand, lower intrinsic priority tasks are penalized, as illustrated in figures 13b and 13c. Streaming tasks, that have the smallest priority degree, show the worst dropping rate for high values of α.

As regards the average waiting time of the executed tasks, the overall effect of choosing high values of α is to reduce it. This happens because γ weighs the task life span, giving preference to tasks that are close to be dropped. As a consequence, the time spent in queue by the executed tasks results longer, in average. However, observing in detail the task behaviour, it can be found that streaming tasks do not follow the general trend (as shown in figure 14). In order to achieve a better comprehension of these results, it is useful to think about the role of α and γ. Considering the array of active tasks (both being executed or waiting), sorted by occurence time, we can say that T_{drop} defines the length of this structure. The time interval $\mathcal{I}$ in which the probability of selecting a task is higher, is linked to γ. The smaller it is, the wider $\mathcal{I}$ is. Viceversa, for higher values of γ, $\mathcal{I}$ shrinks to its fixed extreme $t_{\text{current}} - T_{\text{drop}}$. Dually, α plays the same role with regard to priority: higher values of α imply bigger probability of selecting only the highest priority tasks (see figure 17 for a graphic representation of the role of α and γ).

In spite of the fact that high values of α generally cause the waiting time to decrease, streaming tasks behave differently. This is because they have the lowest intrinsic priority, so with $\alpha \approx 1$ agents tend to neglect them. Obviously, longer T_{drop} allows the queue to grow, so waiting time are increased.

We have already stated that we are interested in a good trade-off between continuity and optimality. We now consider the first one, comparing the proposed algorithms (figure 15).

By discontinuity we mean that a task is left by an agent before it comes to complete execution. As predictable, the PLI algorithm does not exhibit the desired continuity. This happens because this approach is memoryless: it does not take into account the previous matching in order to update the assignment when a new task occurs or is completed. Greedy and NBf show the best results, since they are designed to be extremely conservative in matching tasks and agents. The best performances are exhibited by the Greedy algorithm when $\alpha \approx 0$, because in such a case agents evaluate tasks considering only their occurence time. As a consequence the oldest tasks are always at the top of the agents'lists. SMTI Revised achieves good performances, thanks to the fact that swaps are limited by restrictive conditions.

Another meaningful aspect that has to be inspected is the presence of undesired IDLE agents (figure 16). Under our assumptions we can exclude an instantaneous optimal solution in which some agents are unloaded (as we have already stated in proposition 1). As matter of fact, PLI never shows IDLE agents. SMTI Revised achieves good performances since it willfully tries to match each agent. In order to do it, in comparing agents to be assigned to a task, it

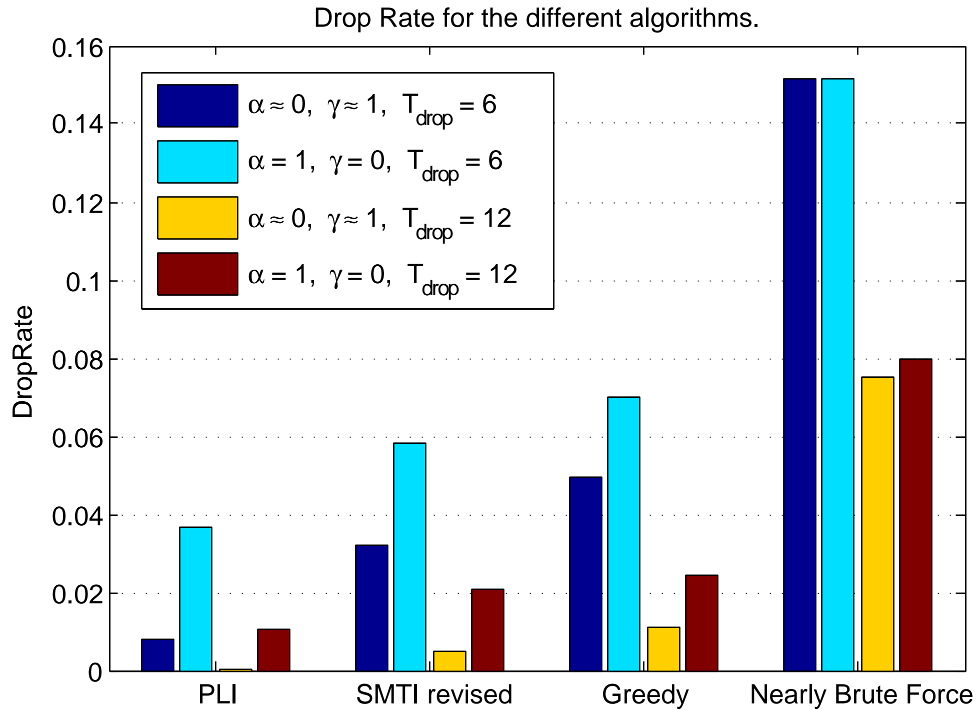
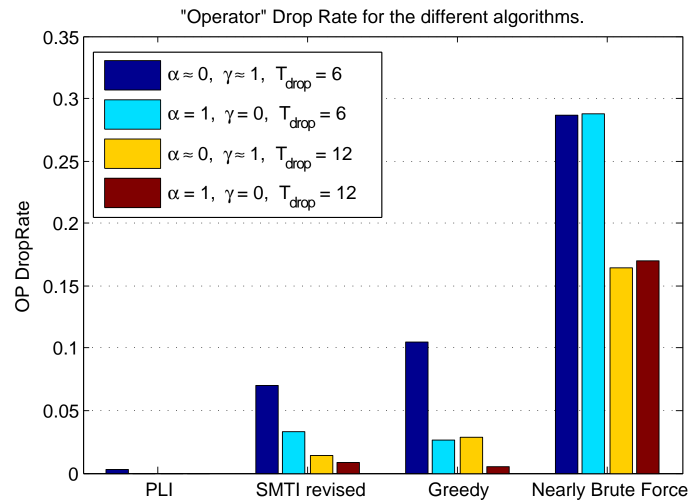


Figure 12: Drop rate

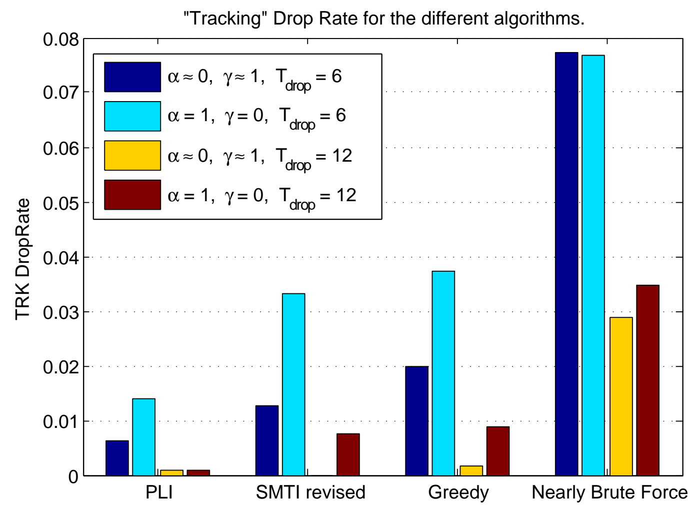
weighs the shortness of their list of unexplored feasible tasks. As already described in example of figure (5), Greedy and NBf do not guarantee the IDLE avoidance.

As a general trend, when working with a longer T_{drop}, the pool contains more elements and makes unlikely that an agent has no tasks to do.

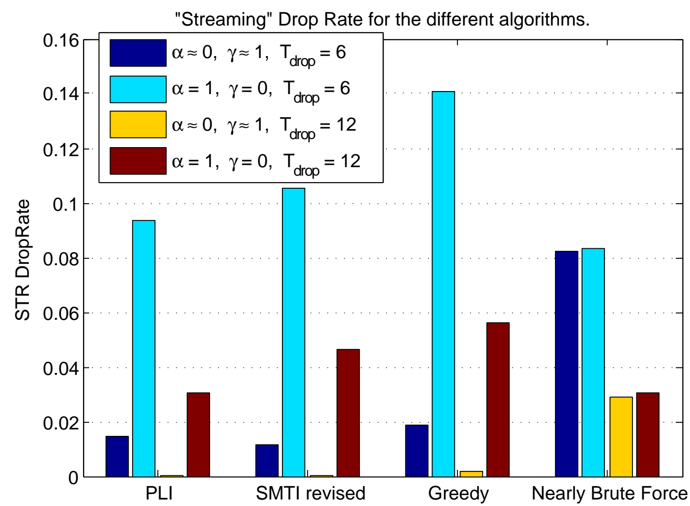
Referring to optimality, a remarkable aspect is the average sum of the intrinsic priorities of assigned tasks (shown in figure 18). The best results are obtained by PLI and SMTI Revised: there are not significant variations with different values of α.



(a)

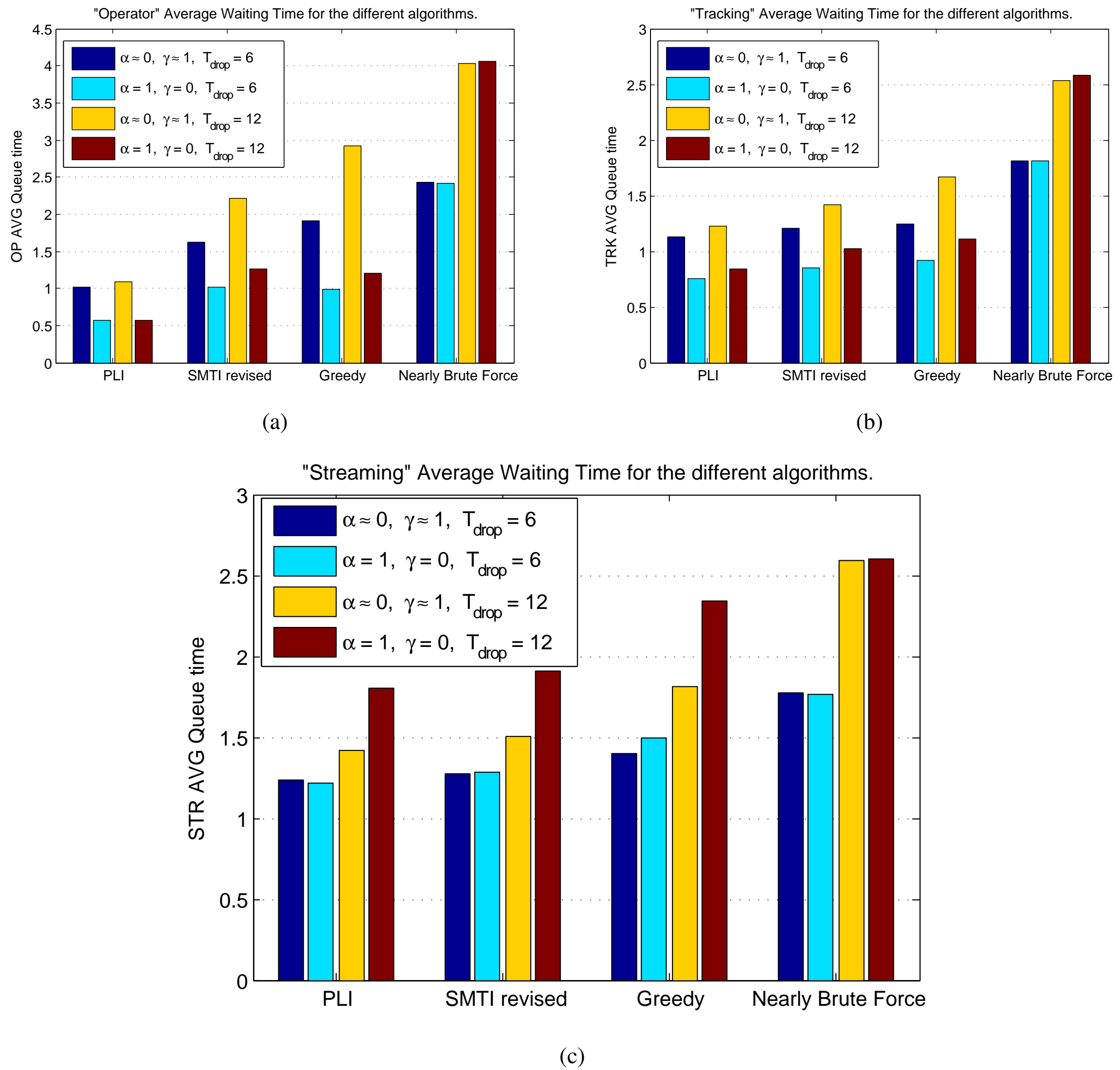


(b)



(c)

Figure 13: Drop rate



(a)

(b)

(c)

Figure 14: Average waiting time

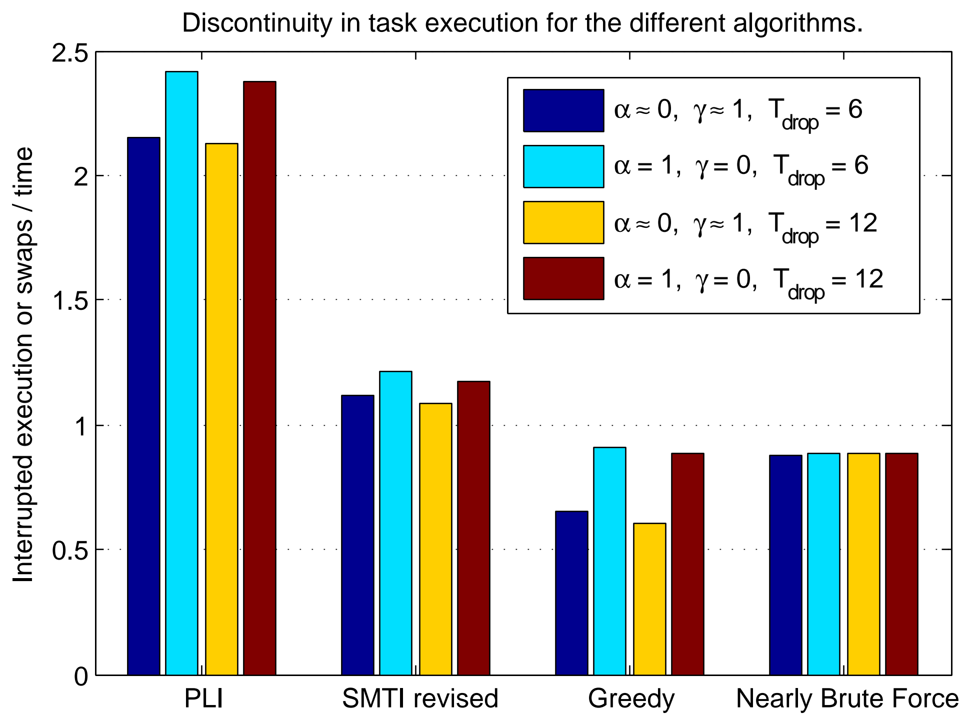


Figure 15: Discontinuity rate

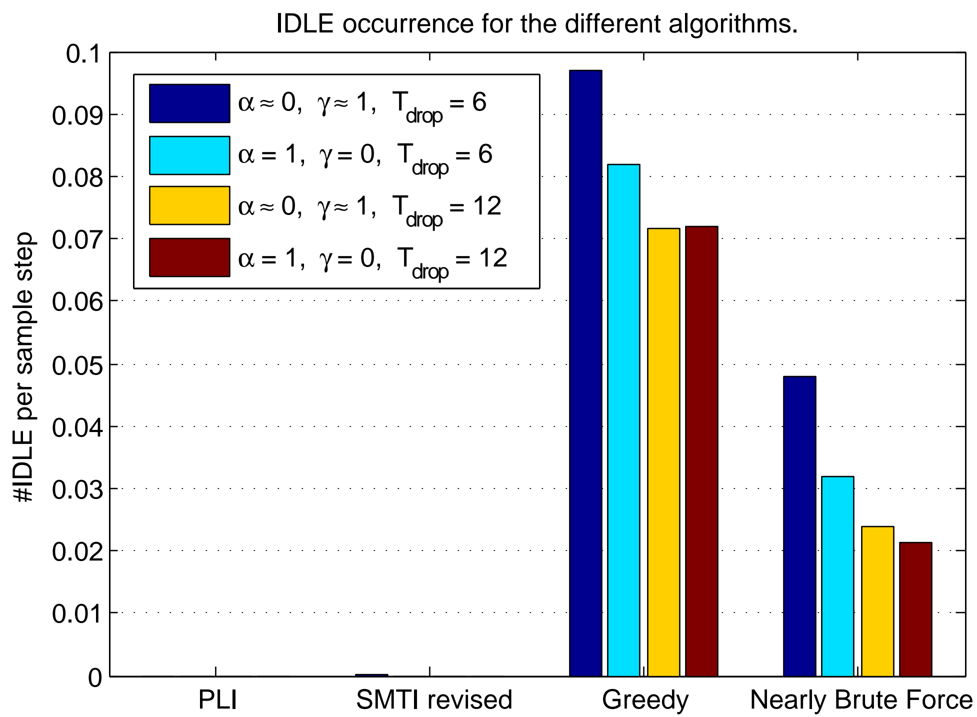


Figure 16: IDLE occurrence

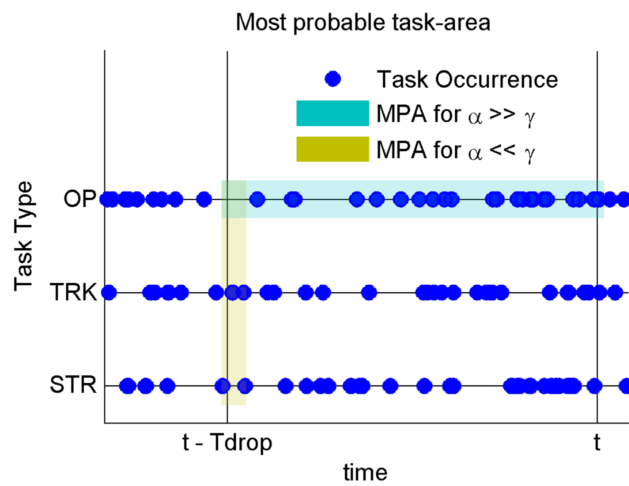


Figure 17: A graphic representation of the role played by α and γ in determining the most probable task to be selected

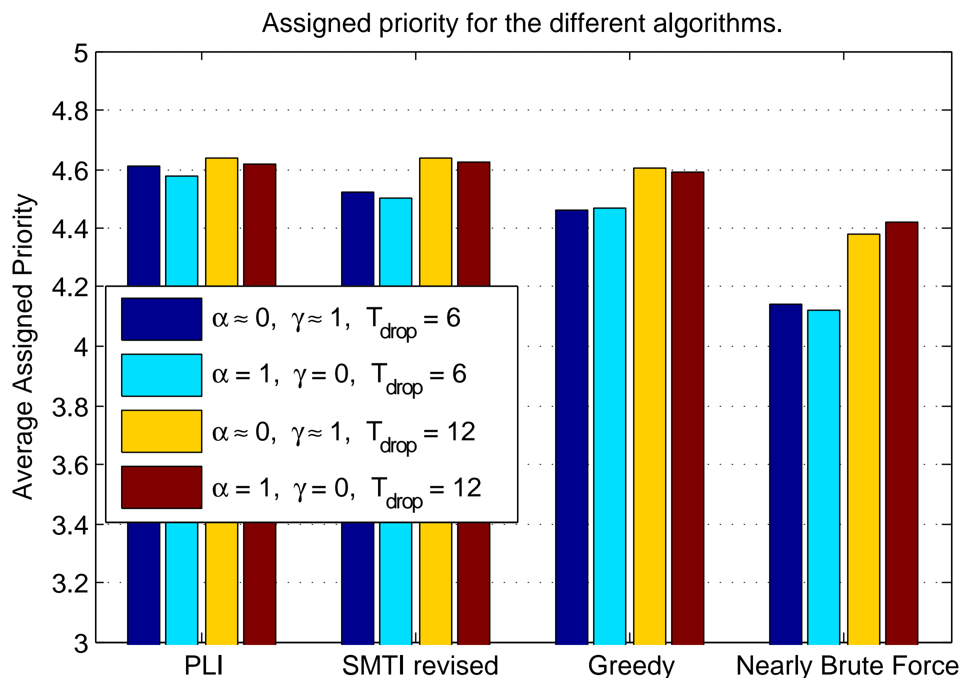


Figure 18: Assigned priorities

5 CONCLUSIONS

The proposed SMTI Revised algorithm shows good performances. In maximizing assigned priorities, avoiding IDLE agents and reducing both drop rate and average waiting time, it is similar to the centralized assignment (PLI). Moreover, it is better in terms of continuity. As precedently discussed, SMTI Revised can be easily turned into a distributed algorithm, even though the consequent costs in terms of both communication and code implementation should be furtherly analysed. As regards the PLI algorithm, it performs best over almost every index, but it is hardly reliable over a real scenario like ours, due to the bad scalability and the unacceptable discontinuity in task execution. In such a situation, it can be wiser to turn to Greedy or Nearly Brute-force algorithms. They are simpler to implement, scalable and cheaper. In particular, Greedy algorithm allows the designer to tune the value of α, improving control on the assignment policy. In addition, it shows acceptable performances, except for IDLE agents avoidance.

6 FUTURE WORK

A point that has not been taken into account is the problem of keeping the pool up to date when algorithms are implemented in distributed form. In this paper we have assumed that not only the pool of tasks, but also all the variables which describe state of the system and instantaneous associations between tasks and agents, are always disposable and consistent for each agent. In addition, cameras are considered sequentially (even if in a random order) and *one for* iteration in task selection algorithm. Hence the generalization to a real system of independent agents rises many problem in the field of communications and consensus between actors:

- Keeping the pool consistent is the most critical point. A deep analysis of consequences derived from different versions of data stored in the cameras memory is required. Also, it is important to find out a consensus mechanism for determining the most trustworthy version and develop merging algorithms, if necessary.
- Another question is to ponder the effects of completely asynchronous acting cameras and to implement a management of task assignment collision. Consensus algorithms, deterministic or randomized are likely the best manner to proceed. Alternatively it can be defined a *leader* agent (hierarchical logic) that directly decides the best solution when such a collision happens.

It may be interesting to implement the assignment policy following the market based approach. This would imply to define communication costs properly. Moreover, the swapping mechanism should be furtherly explored. As a matter of fact, it could avoid reiterating unnecessary auctions, that are expensive in terms of synchronization and communication costs.

A completely different problem is an extension to PTZ cameras. It is possible to fraction areas in sub-areas corresponding to discrete camera angles of view and still apply the algorithms considered in this paper (with minimal adjustments). Continuous angles are not defined if the present framework holds.

References

[1] Lagesse B. A game-theoretical model for task assignment in project management. In *IEEE International Conference on Management of Innovation and Technology*, 2006.
[2] Moore B.J. and Passino K.M. Distributed task assignment for mobile agents. *IEEE Transactions on automatic control, VOL. 52, NO. 4*, April 2007.
[3] Luc Brunet. Consensus-based auctions for decentralized task assignment. Master's thesis, Massachusetts Institute of Technology, 2008.
[4] Manlovea D.F., Irvinga R.W., Iwama K., Miyazaki S., and Morita Y. Hard variants of stable marriage. *Theoretical Computer Science 276, 261-279*, 2002.
[5] Myung Joo Ham1 and Gul Agha1. Study of coordinated dynamic market-based task assignment in massively multi-agent systems. 2007.
[6] Feiler M.J. On distributed search in an uncertain environment. In *Proceedings of the 1st IFAC Workshop on Estimation and Control of Networked Systems*, Venice, Italy, September 4-26 2009.
[7] Michael N., Zavlanos M. M., Kumar ., and Pappas G.J. Distributed multi-robot task assignment and formation control. In *ICRA 2008. IEEE International Conference on Robotics and Automation*, 2008.
[8] Gale D.; Shapley L. S. College admissions and the stability of marriage. *The American Mathematical Monthly, Vol. 69, No. 1., pp. 9-15.*, 1962.
[9] Hunt W. The stable marriage problem. http://www.csee.wvu.edu/~ksmani/courses/fa01/random/lecnotes/lecture5.pdf.