Internship Report

Solal Rapaport

June 2022

Summary

1 Introduction

There are two goals here, the first one is to build formulas that will allow robots, spread on a ring, to gather. We have k robots and we will use view vectors to build those formulas. The formulas will be an interpretation of the pseudo-code given in the research report [1].

The formulas we are building, will be used with formulas given in an other research report [2], and then will be tested in the acceleration algorithm using an interpolant [2]. Which leads us to the second goal, we want to implement and, if possible, improve this algorithm.

2 Logical Formulas

In this section, we will translate the algorithms given in the research report [1]. Some changes will have to be made because we can't literally translate an algorithm into a first-order logic formula.

Before each formula we will describe briefly their scope: when will they be true (or false). We won't present to you the implementation of those formulas in this report. There will be an annex available with the *Python* implementation that we use in order to test those formulas and to put them in the algorithm [2].

We have three strategies. Each of them allows a robot to move in a given direction based on its environment. They all have the same definition, they take one argument: the view vector (distance vector).

2.1 Configurations with single multiplicity

The strategy ϕ_{SM} is *true* if the given configuration has a single multiplicity and that the robot calling the strategy should move toward the robot at distance d_0:

$$\begin{gathered}\phi_{SM}(d_0, \dots, d_{k-1}) := \\ (\bigvee_{i=0}^{k-1}(d_i = 0 \bigwedge_{j=0\ \ j\neq i}^{k-1}(d_j > 0 \vee (d_j = 0 \wedge d_{j-1} = 0))))\wedge \\ (d_{k-1} \neq 0)\wedge \\ ((d_1 = 0 \wedge d_{k-2} = 0 \wedge d_0 \leq d_{k-1}) \vee (d_1 = 0 \wedge d_{k-2} \neq 0))\end{gathered}$$

In order to test our strategy we need a function that will initialize our first configuration and make it one with a single multiplicity without being already a winning one, we just thought it'd be a neat thing to do. Here is the formula $InitSM$ which is *true* if p, s and t form a configuration with a single multiplicity, the configuration is not a winning one, all p are initialized in the right scope, all t are initialized at 0 and all s are initialized at RLC (-1):

$$\begin{gathered}InitSM(p_0, \dots, p_{k-1}, s_0, \dots, s_{k-1}, t_0, \dots, t_{k-1}, size_{ring}) := \\ \bigvee_{i=0}^{k-1}(p_i \neq p_{i+1 \mod k-1})\wedge \\ (\bigwedge_{i=0}^{k-1}(p_i \geq 0 \wedge p_i < size_{ring} \wedge s_i = -1 \wedge t_i = 0))\wedge \\ (\bigvee_{i=0}^{k-1}(\bigvee_{j=0, j\neq i}^{k-1}(p_j = p_i \wedge \bigwedge_{h=0}^{k-1}(\bigwedge_{l=0, l\neq h}^{k-1}(p_h \neq p_l \vee p_h = p_i)))))\end{gathered}$$

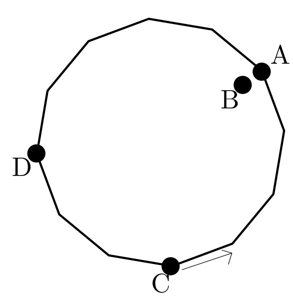


Figure 1: Single multiplicity configuration. Here, the view vector or C is (4, 0, 5, 3). Because there's only one 0 we know there is a single multiplicity. Because the 0 isn't the last int of the vector we know C is not on the multiplicity. There's only one free segment toward the multiplicity, hence C can move on this segment.

2.2 Gathering rigid configurations

Let d_{ij} be the value j of the view vector of the robot i, and ds_{ij} the value j of the symmetrical view of the robot i. The robot is calling the strategy ϕ_R.

Here are all the logic formulas used in order to build ϕ_R:

AllView is *true* if $d_{00},\dots,d_{k-1k-1}$ are all the views you can obtain from a single view vector $dist_0,\dots,dist_{k-1}$:

$$\begin{gathered}AllView(dist_0,\dots,dist_{k-1},d_{00},\dots,d_{k-1k-1}) := \\ (\bigwedge_{i=0}^{k-1}(\bigwedge_{j=0}^{k-1}(d_{ij} = dist_{(j+i) \mod k})))\end{gathered}$$

IsRigid is *true* if the given configuration is a rigid configuration. Meaning, all views are distinct, there is no multiplicity, and the configuration isn't symmetric nor periodic.

$$\begin{gathered}IsRigid(d_{00},\dots,d_{k-1k-1},ds_{00},\dots,ds_{k-1k-1}) := \\ \bigwedge_{i=0}^{k-1}(\bigwedge_{j=0}^{k-1} d_{ij} \neq 0)\wedge \\ \bigwedge_{i=0}^{k-1}(\bigwedge_{l=0\ l\neq i}^{k-1}((\bigvee_{j=0}^{k-1} d_{ij} \neq d_{lj}) \wedge (\bigvee_{j=0}^{k-1} d_{ij} \neq ds_{lj}) \\ \wedge(\bigvee_{j=0}^{k-1} ds_{ij} \neq d_{lj}) \wedge (\bigvee_{j=0}^{k-1} ds_{ij} \neq ds_{lj})))\end{gathered}$$

AllCode is *true* if (α'_r,β'_r) is the set of two natural numbers of the robot r such as α'_r and β'_r are codes of r's views, with $\alpha'_r < \beta'_r$. The process which leads us to obtain all view codes is defined in the research report [1].

$$\begin{gathered}AllCode(d_{00},\dots,d_{k-1k-1},ds_{00},\dots,ds_{k-1k-1},\alpha_0,\dots,\alpha_{k-1},\beta_0,\dots,\beta_{k-1}, \\ \alpha'_0,\dots,\alpha'_{k-1},\beta'_0,\dots,\beta'_{k-1}) := \\ \bigwedge_{i=0}^{k-1}(\alpha'_i < \beta'_i \wedge (\alpha'_i = \alpha_i \vee \alpha'_i = \beta_i) \wedge (\beta'_i = \alpha_i \vee \beta'_i = \beta_i)) \wedge \\ (\ (\alpha_0 < \alpha_1 < \dots < \alpha_{k-1} < \beta_0 < \dots < \beta_{k-1})\wedge \\ (\bigvee_{p=0}^{k-1}(\bigwedge_{q=0}^{p-1}(d_{0q} = d_{1q}) \wedge d_{0p} > d_{1p})) \wedge \dots \wedge \\ (\bigvee_{p=0}^{k-1}(\bigwedge_{q=0}^{p-1}(ds_{(k-2)q} = ds_{(k-1)q}) \wedge ds_{(k-2)p} > ds_{(k-1)p})) \\ \vee \\ ((\alpha_0 < \alpha_2 < \alpha_1 < \dots < \alpha_{k-1} < \beta_0 < \dots < \beta_{k-1}) \wedge \dots) \vee \dots\ )\end{gathered}$$

CodeMaker is *true* if the configuration is rigid and if $(a_0,\dots,a_{k-1},as_0,\dots,as_{k-1})$ are each code of each view passed as a parameter:

$$\begin{gathered}CodeMaker(d_{00},\dots,d_{k-1k-1},ds_{00},\dots,ds_{k-1k-1},a_0,\dots,a_{k-1}) := \\ IsRigid(d_{00},\dots,d_{k-1k-1},ds_{00},\dots,ds_{k-1k-1})\wedge \\ \exists\alpha_0,\dots,\alpha_{k-1},\beta_0,\dots,\beta_{k-1},\ \alpha'_0,\dots,\alpha'_{k-1},\beta'_0,\dots,\beta'_{k-1}, \\ AllCode(d_{00},\dots,d_{k-1k-1},ds_{00},\dots,ds_{k-1k-1},\alpha_0,\dots,\alpha_{k-1},\beta_0,\dots,\beta_{k-1}, \\ \alpha'_0,\dots,\alpha'_{k-1},\beta'_0,\dots,\beta'_{k-1}) \\ (\bigwedge_{i=0}^{k-1}(\bigwedge_{j=0,j\neq i}^{k-1}((a_i > a_j \wedge \alpha'_j > \alpha'_i) \vee (a_i < a_j \wedge \alpha'_j < \alpha'_i)))) \\ \bigwedge_{i=0}^{k-1}(\bigwedge_{j=0,j\neq i}^{k-1} a_i \neq a_j)\end{gathered}$$

FindMax is *true* if Max is the highest value of the view vector passed as a parameter:

$$\begin{gathered}FindMax(dist_0,\dots,dist_{k-1},Max) := \\ (\bigwedge_{i=0}^{k-1}(Max \geq dist_i) \wedge (\bigvee_{i=0}^{k-1}(Max = dist_i)))\end{gathered}$$

FindM is *true* if M is the index of the robot (index in the view vector) which has the largest code of view and a neighboring robot at distance Max:

$$\begin{gathered}FindM(d_{00},\dots,d_{k-1k-1},a_0,\dots,a_{k-1},Max,dM_0,\dots,dM_{k-1}) := \\ \bigvee_{m=0}^{k-1}((\bigwedge_{i=0}^{k-1}((a_m \geq a_i \wedge (d_{i0} = Max \vee d_{ik-1} = Max)) \\ \vee(d_{i0} < Max \wedge d_{ik-1} < Max))) \wedge M = m)\end{gathered}$$

FindN is *true* if N is the index of the robot (index in the view vector) with the largest code of view and M as a neighboring robot at distance Max:

$$
\begin{gathered}
FindN(d_{00},\dots,d_{k-1k-1},a_0,\dots,a_{k-1},Max,M,N) := \\
(d_{M0} = Max \wedge d_{Mk-1} = Max\wedge \\
((N = ((M+1) \mod k) \wedge a_{(M+1) \mod k} > a_{(M-1) \mod k})\vee \\
(N = ((M-1) \mod k) \wedge a_{(M-1) \mod k} > a_{(M+1) \mod k}))) \\
\vee \\
(d_{M0} = Max \wedge d_{Mk-1} \neq Max \wedge N = ((M+1) \mod k)) \\
\vee \\
(d_{M0} \neq Max \wedge d_{Mk-1} = Max \wedge N = ((M-1) \mod k))
\end{gathered}
$$

Since those formulas can't be implemented in *Python* because it is impossible to work around a variable index, we choose to build a new formula, $FindMN$ that will be *true* if both vectors dM and dN are the view vector of, respectively, M and N.

$$
\begin{gathered}
FindMN(d_{00},\dots,d_{k-1k-1},a_0,\dots,a_{k-1},Max,M,N, \\
dM_0,\dots,dM_{k-1},dN_0,\dots,dN_{k-1}) := \\
\bigvee_{m=0}^{k-1}(\quad (\bigwedge_{i=0}^{k-1}((a_m \geq a_i \wedge (d_{i0} = Max \vee d_{ik-1} = Max)) \\
\vee(d_{i0} < Max \wedge d_{ik-1} < Max))) \wedge M = m\wedge \\
(\quad (d_{m0} = Max \wedge d_{mk-1} = Max\wedge \\
((N = M+1 \mod k \wedge a_{(m+1) \mod k} > a_{(m-1) \mod k})\vee \\
(N = M-1 \mod k \wedge a_{(m-1) \mod k} > a_{(m+1) \mod k})))\vee \\
(d_{m0} = Max \wedge d_{mk-1} \neq Max \wedge N = M+1 \mod k)\vee \\
(d_{m0} \neq Max \wedge d_{mk-1} = Max \wedge N = M-1 \mod k) \quad)\wedge \\
((N = M-1 \mod k \wedge (\bigwedge_{l=0}^{k-1}(dN_l = d_{(m-1 \mod k)((k-1)-l)} \wedge dM_l = d_{ml})))\vee \\
(N = M+1 \mod k \wedge (\bigwedge_{l=0}^{k-1}(dN_l = d_{(m+1 \mod k)l} \wedge dM_l = d_{m((k-1)-l)})))) \quad)
\end{gathered}
$$

ϕ_R is *true* if the configuration is rigid, and if the robot is M and has a closest neighbor than N, or if the robot is N and has a closest neighbor than M.

$$
\begin{gathered}
\phi_R(dist_0,\dots,dist_{k-1}) := \\
\exists d_{00},\dots,d_{k-1k-1}, \ AllView(dist_0,\dots,dist_{k-1},d_{00},\dots,d_{k-1k-1})\wedge \\
\exists ds_{00},\dots,ds_{k-1k-1}, \bigwedge_{i=0}^{k-1}(ViewSym(d_{i0},\dots,d_{ik-1},ds_{i0},\dots,ds_{ik-1}))\wedge \\
\exists Max, a_0,\dots,a_{k-1}, dM_0,\dots,dM_{k-1}, dN_0,\dots,dN_{k-1}, \\
CodeMaker(d_{00},\dots,d_{k-1k-1},ds_{00},\dots,ds_{k-1k-1},a_0,\dots,a_{k-1})\wedge \\
FindMax(dist_0,\dots,dist_{k-1},Max)\wedge \\
FindMN(d_{00},\dots,d_{k-1k-1},a_0,\dots,a_{k-1},Max,dM_0,\dots,dM_{k-1},dN_0,\dots,dN_{k-1})\wedge \\
\exists dM2_0,\dots,dM2_{k-1},dN2_0,\dots,dN2_{k-1}, \\
((\bigwedge_{i=0}^{k-1}(dM2_i = dM_{i+1 \mod k})) \vee (\bigwedge_{i=0}^{k-1}(dM2_i = dM_{i-1 \mod k})))\wedge \\
(\bigvee_{i=0}^{k-1}(dM2_i \neq dN_i))\wedge \\
((\bigwedge_{i=0}^{k-1}(dN2_i = dN_{i+1 \mod k})) \vee (\bigwedge_{i=0}^{k-1}(dN2_i = dN_{i-1 \mod k})))\wedge \\
(\bigvee_{i=0}^{k-1}(dN2_i \neq dM_i))\wedge \\
\exists distM_0,\dots,distM_{k-1},distN_0,\dots,distN_{k-1}, \\
\bigwedge_{i=0}^{k-1}(distM_i = (\sum_{l=0}^{i} dM_l) \wedge distN_i = (\sum_{l=0}^{i} dN_l))\wedge \\
(\bigvee_{i=0}^{k-1} \quad (\quad (distM_i < distN_i \bigwedge_{q=0}^{i}(distM_q = distN_q) \bigwedge_{j=0}^{k-1}(dM_j = dist_j))\vee \\
(distM_i > distN_i \bigwedge_{q=0}^{i}(distM_q = distN_q) \bigwedge_{j=0}^{k-1}(dN_j = dist_j)) \quad) \quad)
\end{gathered}
$$

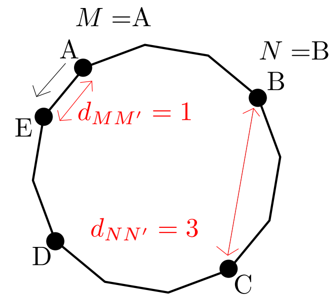


Figure 2: Rigid configuration. Here, A has the biggest code of view, and the only neighbor at distance Max that he has is B, hence A is M and B is N. As we can see with the distance in red, A will move toward E: this will create a single multiplicity faster than if B would have moved toward C

2.3 Gathering an odd number of robots

We are now building a strategy, ϕ_{ON}, that will gather an odd number of robots on a non-periodic configuration. It is the strategy with the lowest priority, meaning that the configuration won't be rigid and won't have any multiplicity.

First we build the formula, $IsPeriodic$, that will return *true* if the configuration is periodic with an odd number of robots:

$$\begin{gathered} IsPeriodic(dist_0, \dots, dist_{k-1}) := \\ \exists p \in [1; \lfloor \tfrac{k}{3} \rfloor], (p+1) \mod 2 = 0 \wedge \\ \exists d'_0, \dots, d'_{p-1}, \bigwedge_{i=0}^{k-1} (d'_{i \mod p} = dist_i) \end{gathered}$$

Now, we build build ϕ_{OD}, the strategy returns *true* if the configuration is non-rigid, non-periodic, has no multiplicity and has an odd number of robots. If the robot is axial then it moves in order to create a multiplicity or a rigid configuration.

$$\begin{gathered} \phi_{ON}(dist_0, \dots, dist_{k-1}) := \\ \exists d_{00}, \dots, d_{k-1k-1},\ AllView(dist_0, \dots, dist_{k-1}, d_{00}, \dots, d_{k-1k-1}) \wedge \\ \exists ds_{00}, \dots, ds_{k-1k-1}, \bigwedge_{i=0}^{k-1} (ViewSym(d_{i0}, \dots, d_{ik-1}, ds_{i0}, \dots, ds_{ik-1})) \wedge \\ \neg IsRigid(d_{00}, \dots, d_{k-1k-1}, ds_{00}, \dots, ds_{k-1k-1}) \wedge \\ ((k+1) \mod 2 = 0) \wedge \\ \neg IsPeriodic(dist_0, \dots, dist_{k-1}) \wedge \\ (\bigwedge_{i=0}^{k-1} dist_i \neq 0) \wedge \\ (\bigwedge_{i=0}^{k-1} dist_i = ds_{0i}) \end{gathered}$$

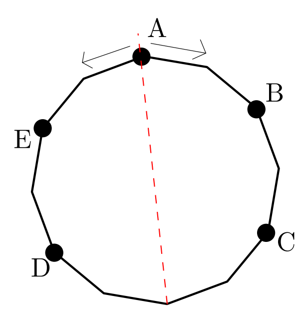


Figure 3: A symetrical, non-periodic configuration with an odd number of robots. Here, if the robot is axial it moves. In this case, A moves in order to create, eventually, a rigid configuration or a single multiplicity.

2.4 Combining formulas

Finally, we build the last formula, $\phi_{Ultimate}$, that will garanty us that, with an odd number of robot and no periodicity in the initial configuration, we can gather robots.

$$\begin{gathered}\phi_{Ultimate}(dist_0, \ldots, dist_{k-1}) := \\ \phi_{ON}(dist_0, \ldots, dist_{k-1}) \vee \\ \phi_{R}(dist_0, \ldots, dist_{k-1}) \vee \\ \phi_{SM}(dist_0, \ldots, dist_{k-1})\end{gathered}$$

2.5 Equivalence classes

2.5.1 Proof

In order to decrease the time it takes to find a loosing loop or to prove that there is none, we want to be able to detect if two configurations are in the same equivalence class.

First, we establish the properties that will define the three relations $\mathbb{R}_1$, $\mathbb{R}_2$ and $\mathbb{R}_3$ between two configurations. We will use the following notation to refer to those relations: with c and c', two configurations, c is related to c' through the relation $\mathbb{R}_i$, we write it $c \sim_{\mathbb{R}_i} c'$.

Little reminder: a configuration c is a vector $(p_0, \ldots, p_{k-1}, s_0, \ldots, s_{k-1}, t_0, \ldots, t_{k-1})$, or it can be subdivided in three vectors: the position one $(p_0, \ldots, p_{k-1})$, the status one $(s_0, \ldots, s_{k-1})$ and the equity one $(t_0, \ldots, t_{k-1})$.

Let's now established the property of each $\mathbb{R}_i$ relation:

1. $c \sim_{\mathbb{R}_1} c'$ if:
   $\forall i \in [0; k-1], \exists s, p'_i = p_i + s \mod size_{ring} \wedge t'_i = t_i + s \mod size_{ring} \wedge ((s_i = -1 \wedge s'_i = s_i) \vee (s_i \neq -1 \wedge s'_i = s_i + s \mod size_{ring}))$
   It is related here because positions have no name, we only give them a number in order to know the distance between 2 robots on different position. What matters is the view vector. If we do a rotation (i.e all robots keep the same status and equity bit and go right) then it is the same configuration than before. Likewise, status shouldn't have the number of the targetted position, it should have: left, right, same place and RLC. But because it has the number of the targetted position, if the configuration rotates then we update the status accordingly.

2. $c \sim_{\mathbb{R}_2} c'$ if:
   $\forall i \in [0; k-1], (p'_i = (size_{ring} - p_i) \mod size_{ring} \wedge t'_i = t_i \wedge ((s_i = -1 \wedge s'_i = s_i) \vee (s_i \neq -1 \wedge s'_i = (size_{ring} - s_i) \mod size_{ring})) \bigvee (p'_i = p_i \wedge s'_i = s_i \wedge t'_i = t_i))$
   Here, we are translating all robots in the mirror configuration. We built the symmetric configuration. Both configurations are related, because the view vector stays the same. Meaning the environment is identical, and robots will still move toward the same robot that kept the same distance.

3. $c \sim_{\mathbb{R}_3} c'$ if:
   For a given number of robots k there are $k!$ combinations of the configuration vector (i.e the position of a robot could be at the index 0 on the position vector or it could be somewhere else, neither robots or positions have an id). We define the vector o as the vector of index (exemple: with 3 robots o can be $(0,1,2)$ or $(0,2,1)$ or $(1,0,2)$ etc...)
   $\forall i \in [0; k-1], \exists (o_0, \ldots, o_{k-1}), p'_i = p_{o_i} \wedge s'_i = s_{o_i} \wedge t'_i = t_{o_i}$

We are now demonstrating that each $\sim_{\mathbb{R}_i}$ is an equivalence relation. Two configurations related with $\sim_{\mathbb{R}_i}$ are in the same equivalence class if and only if the relation is reflexive, symmetric and transitive. In order to demonstrate that, we're defining three configurations: $\exists c, c', c'' \in \mathbb{A}$, $\mathbb{A}$ being the set of all possible configurations.

- Reflexivity

  $$- \ c \sim_{\mathbb{R}_1} c$$

  If $s = 0$ then $\forall i \in [0; k-1] p_i = p_i + 0 \mod size_{ring} \wedge t_i = t_i + 0 \mod size_{ring} \wedge ((s_i = -1 \wedge s_i = -1) \vee (s_i \neq -1 \wedge s_i = s_i + 0 \mod size_{ring}))$. We know that $p_i \in [0; size_{ring} - 1]$ hence, we have: $\forall i \in [0; k-1] p_i = p_i \wedge t_i = t_i \wedge s_i = s_i$ meaning $c = c$, c is equivalent to c.

$- \ c \sim_{\mathbb{R}_2} c$

We have $p_i = p_i \wedge s_i = s_i \wedge t_i = t_i$: the property is verified and c is related to c.

$- \ c \sim_{\mathbb{R}_3} c$

If $o_0, \ldots, o_{k-1}$ is the vector of the original index order, then: $p_i = p_i \wedge s_i = s_i \wedge t_i = t_i$ and the property is verified and c is related to c.

- Symmetry

$- \ c \sim_{\mathbb{R}_1} c'$ if and only if $c' \sim_{\mathbb{R}_1} c$

Let's assume that $c \sim_{\mathbb{R}_1} c'$ and that $c' \nsim_{\mathbb{R}_1} c$
Because $c \sim_{\mathbb{R}_1} c'$ then $\forall i \in [0; k-1], \exists s, p'_i = p_i + s \mod size_{ring} \wedge t'_i = t_i + s \mod size_{ring} \wedge ((s_i = -1 \wedge s'_i = s_i) \vee (s_i \neq -1 \wedge s'_i = s_i + s \mod size_{ring}))$.
And because $c' \nsim_{\mathbb{R}_1} c$ there must be: $\exists i \in [0; k-1], \forall s, p_i \neq p'_i + s \mod size_{ring} \vee t_i \neq t'_i + s \mod size_{ring} \vee ((s'_i \neq -1 \vee s_i \neq s'_i) \wedge (s'_i = -1 \vee s_i \neq s'_i + s \mod size_{ring}))$.
However, because $p'_i = p_i + s \mod size_{ring}$ then $p_i = p'_i - s \mod size_{ring}$ and, with the $\mod size_{ring}$, $+$ and $-$ operators mean the same with natural integers. We can define $\exists s' \in [0; size_{ring} - 1], p_i = p'_i + s' \mod size_{ring}$.
We demonstrate that $t_i = t'_i + s' \mod size_{ring}$ the same way.
Finally, if $s'_i = -1$ then $s_i = s'_i$ which would make false the following assertion: $(s'_i \neq -1 \vee s_i \neq s'_i) \wedge (s'_i = -1 \vee s_i \neq s'_i + s \mod size_{ring})$. If $s'_i \neq -1$ then $s'_i = s_i + s \mod size_{ring}$ and with the previous demonstration we show that $s_i = s'_i + s' \mod size_{ring}$.
We face a contradiction. We can't have $c' \nsim_{\mathbb{R}_1} c$ if $c \sim_{\mathbb{R}_1} c'$.

$- \ c \sim_{\mathbb{R}_2} c'$ if and only if $c' \sim_{\mathbb{R}_2} c$

We, here, can only proove symmetry because the property includes $c = c'$ ($p'_i = p_i \wedge s'_i = s_i \wedge t'_i = t_i$), then it's trivial that $c \sim_{\mathbb{R}_2} c'$ if and only if $c' \sim_{\mathbb{R}_2} c$.

$- \ c \sim_{\mathbb{R}_3} c'$ if and only if $c' \sim_{\mathbb{R}_3} c$

Let's assume that $c \sim_{\mathbb{R}_3} c'$ and that $c' \nsim_{\mathbb{R}_1} c$
Because $c \sim_{\mathbb{R}_3} c'$ then $\forall i \in [0; k-1], \exists (o_0, \ldots, o_{k-1}), p_i = p'_{o_i} \wedge s_i = s'_{o_i} \wedge t_i = t'_{o_i}$, and $c' \nsim_{\mathbb{R}_1} c$ implies that there must be: $\exists i \in [0; k-1], \forall (o'_0, \ldots, o'_{k-1}), p'_i \neq p_{o'_i} \vee s'_i \neq s_{o'_i} \vee t'_i \neq t_{o'_i}$
That would mean that we have: $\exists i \in [0; k-1], \forall (o'_0, \ldots, o'_{k-1}), p'_{o_i} \neq p_{o'_i} \vee s'_{o_i} \neq s_{o'_i} \vee t'_{o_i} \neq t_{o'_i}$. However, $\exists (o_0, \ldots, o_{k-1}) \in \forall (o'_0, \ldots, o'_{k-1})$
We face a contradiction. We can't have $c' \nsim_{\mathbb{R}_1} c$ if $c \sim_{\mathbb{R}_1} c'$.

- Transitivity

$-$ If $c \sim_{\mathbb{R}_1} c'$ and $c' \sim_{\mathbb{R}_1} c''$ then $c \sim_{\mathbb{R}_1} c''$

Because $c \sim_{\mathbb{R}_1} c'$ then we have $\forall i \in [0; k-1], \exists s_1, p'_i = p_i + s_1 \mod size_{ring} \wedge t'_i = t_i + s_1 \mod size_{ring} \wedge ((s_i = -1 \wedge s'_i = s_i) \vee (s_i \neq -1 \wedge s'_i = s_i + s_1 \mod size_{ring}))$.
Because $c' \sim_{\mathbb{R}_1} c''$ then we have $\forall i \in [0; k-1], \exists s_2, p''_i = p'_i + s_2 \mod size_{ring} \wedge t''_i = t'_i + s_2 \mod size_{ring} \wedge ((s'_i = -1 \wedge s''_i = s'_i) \vee (s'_i \neq -1 \wedge s''_i = s'_i + s_2 \mod size_{ring}))$.
If we substitute all c' values from $c \sim_{\mathbb{R}_1} c'$ in $c' \sim_{\mathbb{R}_1} c''$ we find:
$\forall i \in [0; k-1], \exists s_2, p''_i = p_i + s_1 + s_2 \mod size_{ring} \wedge t''_i = t_i + s_1 + s_2 \mod size_{ring} \wedge ((s_i = -1 \wedge s''_i = s_i) \vee (s_i \neq -1 \wedge s''_i = s_i + s_1 + s_2 \mod size_{ring}))$
With $s_1 \in \mathbb{N}$ and $s_2 \in \mathbb{N}$ we define $s_3 \in \mathbb{N}, s_3 = s_1 + s_2$. Substituing s_1 or s_2 with s_3 still fits the relation property thanks to the $\mod$. Also, if $s'_i = -1$ then $s'_i = s_i = -1$ we can substitute those assertions in the formula. The same goes if $s'_i \neq -1$ then $s_i \neq -1$. Hence $c \sim_{\mathbb{R}_1} c''$.

$-$ If $c \sim_{\mathbb{R}_2} c'$ and $c' \sim_{\mathbb{R}_2} c''$ then $c \sim_{\mathbb{R}_2} c''$

Because $c \sim_{\mathbb{R}_2} c'$ then wa have $\forall i \in [0; k-1], (p'_i = (size_{ring} - p_i) \mod size_{ring} \wedge t'_i = t_i \wedge ((s_i = -1 \wedge s'_i = s_i) \vee (s_i \neq -1 \wedge s'_i = (size_{ring} - s_i) \mod size_{ring})) \bigvee (p'_i = p_i \wedge s'_i = s_i \wedge t'_i = t_i))$.
Because $c' \sim_{\mathbb{R}_2} c''$ then we have $\forall i \in [0; k-1], (p''_i = (size_{ring} - p'_i) \mod size_{ring} \wedge t''_i = t'_i \wedge ((s'_i = -1 \wedge s''_i = s_i) \vee (s'_i \neq -1 \wedge s''_i = (size_{ring} - s'_i) \mod size_{ring})) \bigvee (p''_i = p'_i \wedge s''_i = s'_i \wedge t''_i = t'_i))$.
If we substitute all c' values from $c \sim_{\mathbb{R}_2} c'$ in $c' \sim_{\mathbb{R}_2} c''$ we find:
$\forall i \in [0; k-1], (p''_i = (size_{ring} - size_{ring} + p_i) \mod size_{ring} \wedge t''_i = t_i \wedge ((s_i =$

$-1 \wedge s''_i = s_i) \vee (s_i \neq -1 \wedge s''_i = (size_{ring} - size_{ring} + s_i) \mod size_{ring})) \bigvee (p''_i = p_i \wedge s''_i = s_i \wedge t''_i = t_i))$
With that, we can establish that $c = c''$, c is equivalent to c'', $c \sim_{\mathbb{R}_2} c''$.

– If $c \sim_{\mathbb{R}_3} c'$ and $c' \sim_{\mathbb{R}_3} c''$ then $c \sim_{\mathbb{R}_3} c''$

Because $c \sim_{\mathbb{R}_3} c'$ then we have $\forall i \in [0; k-1], \exists(o_0, \dots, o_{k-1}), p'_i = p_{o_i} \wedge s'_i = s_{o_i} \wedge t'_i = t_{o_i}$
Because $c' \sim_{\mathbb{R}_3} c''$ then we have $\forall i \in [0; k-1], \exists(o'_0, \dots, o'_{k-1}), p''_i = p'_{o'_i} \wedge s''_i = s'_{o'_i} \wedge t''_i = t'_{o'_i}$
By simply substituing c' in $c' \sim_{\mathbb{R}_3} c''$ with its values from $c \sim_{\mathbb{R}_3} c'$ we find :
$\forall i \in [0; k-1], \exists(o_0, \dots, o_{k-1}), p''_i = p_{o_i} \wedge s''_i = s_{o_i} \wedge t''_i = t_{o_i}$
And just like that we've shown that $c \sim_{\mathbb{R}_3} c''$ because substituing c' makes it fit the $\sim_{\mathbb{R}_3}$ property.

Now we show the bisimulation property of our set of configurations. We show that, for a given configuration, its successor will have, in his equivalence class, a successor of an equivalent configuration that the given one.

A configuration is a successor from another configuration if the status vector or the position vector has changed. Obviously the equity vector always changes from a configuration to its successor.

A successor can't be in the equivalence class of its parent, because we either modify the status or the position and the status vector (status get back to RLC), and it would require to modify only position, or, status and position but without setting status to RLC, in order to, maybe, have an equivalent configuration.

Let's assume that we have four configurations c_1, c'_1, c_2 and c'_2, such as $c_1 \sim_{\mathbb{R}} c'_1$, c_2 is a successor of c_1 and c'_2 is a successor of c'_1. We now demonstrate that $c_2 \sim_{\mathbb{R}} c'_2$.

2.5.2 Formulas

The formula *SameClassRot* is *true* if one configuration is a rotation of the other.

$$\begin{gathered} SameClassRot(p_0, \dots, p_{k-1}, s_0, \dots, s_{k-1}, t_0, \dots, t_{k-1}, p'_0, \dots, \\ p'_{k-1}, s'_0, \dots, s'_{k-1}, t'_0, \dots, t'_{k-1}, size_{ring}) := \\ (\bigvee_{s=0}^{size_{ring}-1} (\bigwedge_{i=0}^{k-1} (p'_i = p_i + s \mod size_{ring} \wedge t'_i = t_i + s \mod size_{ring} \wedge \\ ((s_i = -1 \wedge s'_i = s_i) \vee (s_i \neq -1 \wedge s'_i = s_i + s \mod size_{ring}))))) \end{gathered}$$

The formula *SameClassMirror* is *true* if one configuration is the mirror of the other.

$$\begin{gathered} SameClassMirror(p_0, \dots, p_{k-1}, s_0, \dots, s_{k-1}, t_0, \dots, t_{k-1}, p'_0, \dots, \\ p'_{k-1}, s'_0, \dots, s'_{k-1}, t'_0, \dots, t'_{k-1}, size_{ring}) := \\ \bigwedge_{i=0}^{k-1} (p'_i = (size_{ring} - p_i) \mod size_{ring} \wedge t'_i = t_i \wedge \\ ((s_i = -1 \wedge s'_i = s_i) \vee (s_i \neq -1 \wedge s'_i = (size_{ring} - s_i) \mod size_{ring}))) \end{gathered}$$

The formula *SameClassOrder* is *true* if one configuration is the same with different index on the configuration vectors than the other configuration.

$$\begin{gathered} SameClassOrder(p_0, \dots, p_{k-1}, s_0, \dots, s_{k-1}, t_0, \dots, t_{k-1}, p'_0, \dots, \\ p'_{k-1}, s'_0, \dots, s'_{k-1}, t'_0, \dots, t'_{k-1}) := \\ \exists o_{0_1}, \dots, o_{0_{k-1}}, \dots, o_{k!-1_0}, \dots, o_{k!-1_{k-1}}, \\ (o_{0_1} = 0 \wedge \dots \wedge o_{0_{k-1}} = k-1 \wedge \dots \wedge o_{k!-1_0} = k-1 \wedge \dots \wedge o_{k!-1_{k-1}} = 0) \wedge \\ (\bigvee_{i=0}^{k!-1} (\bigwedge_{j=0}^{k-1} (p'_j = p_{o_{i_j}} \wedge s'_j = s_{o_{i_j}} \wedge t'_j = t_{o_{i_j}}))) \end{gathered}$$

Finally we build *SameClass* which is *true* if both configuration are in the same equivalence class.

$$\begin{gathered} SameClass(p_0, \dots, p_{k-1}, s_0, \dots, s_{k-1}, t_0, \dots, t_{k-1}, p'_0, \dots, \\ p'_{k-1}, s'_0, \dots, s'_{k-1}, t'_0, \dots, t'_{k-1}, size_{ring}) := \\ \exists pr_0, \dots, pr_{k-1}, sr_0, \dots, sr_{k-1}, tr_0, \dots, tr_{k-1}, pm_0, \dots, pm_{k-1}, sm_0, \dots, sm_{k-1}, tm_0, \dots, tm_{k-1}, \\ (SameClassRot(p_0, \dots, p_{k-1}, s_0, \dots, s_{k-1}, t_0, \dots, t_{k-1}, \\ pr_0, \dots, pr_{k-1}, sr_0, \dots, sr_{k-1}, tr_0, \dots, tr_{k-1}) \wedge \\ SameClassOrder(pr_0, \dots, pr_{k-1}, sr_0, \dots, sr_{k-1}, tr_0, \dots, tr_{k-1}, \end{gathered}$$

$$p'_0, \ldots, p'_{k-1}, s'_0, \ldots, s'_{k-1}, t'_0, \ldots, t'_{k-1})) \bigvee$$
$$(SameClassRot(p_0, \ldots, p_{k-1}, s_0, \ldots, s_{k-1}, t_0, \ldots, t_{k-1},$$
$$pr_0, \ldots, pr_{k-1}, sr_0, \ldots, sr_{k-1}, tr_0, \ldots, tr_{k-1}) \wedge$$
$$SameClassMirror(pr_0, \ldots, pr_{k-1}, sr_0, \ldots, sr_{k-1}, tr_0, \ldots, tr_{k-1},$$
$$pm_0, \ldots, pm_{k-1}, sm_0, \ldots, sm_{k-1}, tm_0, \ldots, tm_{k-1}) \wedge$$
$$SameClassOrder(pm_0, \ldots, pm_{k-1}, sm_0, \ldots, sm_{k-1}, tm_0, \ldots, tm_{k-1},$$
$$p'_0, \ldots, p'_{k-1}, s'_0, \ldots, s'_{k-1}, t'_0, \ldots, t'_{k-1}))$$

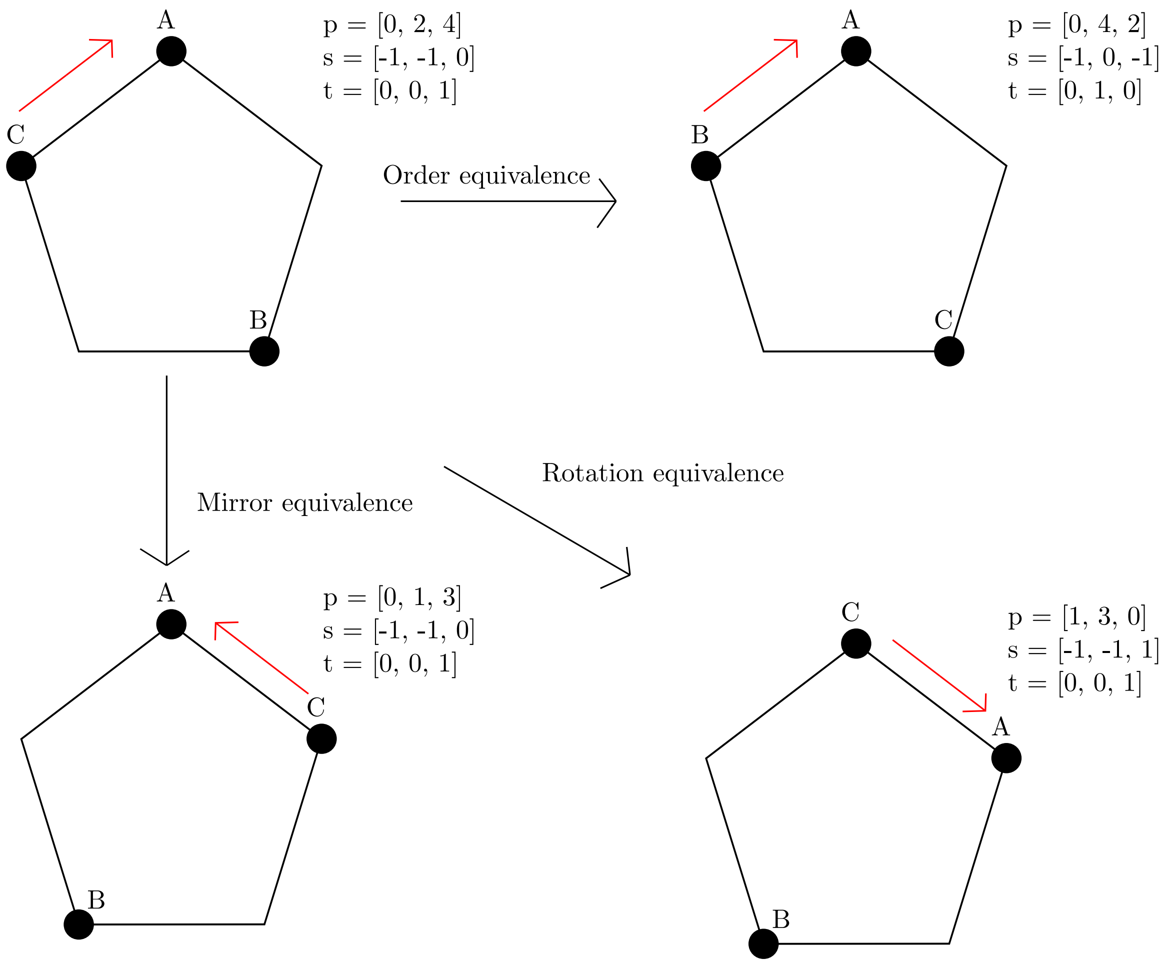


Figure 4: An example of four configurations that are in the same equivalence class

3 Algorithms

Now that we have done all of our logical formulas, we need to test those in the acceleration algorithm using an interpolant [2] and in an alternate version of that same algorithm.

We needed to create an alternate version because of the way the formula, *BouclePerdante*, is done. Two ways it can be done:

1. we can try to create a loosing loop by trying to add as many *AsyncPost* as needed (increase the size of the loop if it's not a loosing one) with a maximum of the size of the graph of all possible configurations

2. or we can try to create a loop that comes back to a previous configuration with only one *AsyncPost*

The first possibility has been implemented in the acceleration algorithm using an interpolant [2]. In order to implement the second possibility we needed to change the algorithm because the winning condition wasn't good anymore.

First we will try to prove that the alternate version of the algorithm works.

3.1 Alternative Version Algorithm

```
foreach synchronous winning strategy f do
    k = 1;
    while true do
        I(c) = Init(c);
        continue = true;
        while continue do
            if MaybeThisSize ≠ null then
                NotThisSizeBis =[i for i in range (k) and i ∉ MaybeThisSize];
                if Init(c) ∧ Post(c, c1), Post(c1, c2) ∧ · · · ∧ Post(c_{k−1}, c_k) ∧
                 BouclePerdante(c_k, NotThisSizeBis)   SAT then
                    exit;                                   /* Loosing Strategy */
                end
            end
            if I(c) ∧ Post(c, c1), Post(c1, c2) ∧ · · · ∧ Post(c_{k−1}, c_k) ∧
             BouclePerdante(c_k, NotThisSize)   SAT then
                if I = Init then
                    exit;                                   /* Loosing Strategy */
                else
                    MaybeThisSize.append(k);
                    k = k + 1;
                    continue = false;
                end
            else
                I′ = Interpolant(I(c) ∧ Post(c, c1), Post(c1, c2) ∧ · · · ∧ Post(c_{k−1}, c_k) ∧
                 BouclePerdante(c_k, NotThisSize));
                if I′ ⟹ I then
                    if k = size_max then
                        exit;                               /* Winning Strategy */
                    else
                        NotThisSize.append(k);
                        k = k + 1;
                        continue = false;
                    end
                else
                    I = I ∨ I′;
                end
            end
        end
    end
end
```

3.2 Proof

First let's talk about the termination of the algorithm:

- The list of synchronous winning strategy is finished
- We can exit the "**while** *true*" (l.3) loop with *exit* instructions that we find at line 10, 15 and 25.
  - We find a loosing loop without the interpolant and then we enter the *exit* at line 10 or the one at line 15 if I is still equal to $Init$
  - We find a loosing loop with the interpolant and then we increase k, we exit the "**while** *continue*" loop (l.6) which allows us to reinitialize I and test if a loosing loop exists for a higher k or for this k without the interpolant.
  - We don't find any loosing loop, then, eventually, the interpolant will stop growing and $(I \vee I') \implies I$, likewise, k will reach $size_{max}$ and we will enter the *exit* at line 25. k will always reach $size_{max}$ if there is no loosing loop, because if the condition line 13, which checks if there is a loosing loop, is false, then if $k < size_{max}$ we reach line 28 and we increase k. Also, the interpolant will eventually stop growing because the graph of all possible configurations is finished and the interpolant won't create new variables.
- To summarize, we can't have more than $size_{max}$ failure at finding a loosing loop and if we find one we either exit if $Init = I$ or we keep trying until we find none or one where $Init = I$.

Now, let's see if the algorithm returns what we need:

- There is no object returned here, what we are showing is that the algorithm exits at the proper instruction in the proper circumstances.
- Let's try a proof by contradiction:
  - First, we assume that all the formulas are right and do what they are supposed to do.
  - Let's say we exit the algorithm line 10, and that there is, in fact, no loosing loop. Then the condition line 9 must have been SAT in order to execute the instruction line 10 but because there is no loosing loop then the condition line 9 is $UNSAT$ and we face a contradiction.
  - Likewise, let's assume we exit the algorithm line 15 and that the strategy has no loosing loop. It is possible that the condition line 13 is SAT but because we know there is no loosing loop then I has been modified by the interpolant, creating new configurations, including some that aren't reachable (otherwise the strategy has a loosing loop). Then if I has been modified, the condition line 14 is $false$ and we never execute the instruction line 15. In the other hand, if I hasn't been modified then the condition line 13 is $UNSAT$ because there is no loosing loop and we never execute the set of instructions between line 14 and 19 and we don't exit line 15. We face a contradiction.
  - Finally, let's say we exit the algorithm at line 25 and that there is a loosing loop. Two things: we have reached $k = size_{max}$ and the interpolant can't grow anymore $(I' \implies I)$, meaning that, for every loop size and for all configurations we can't find a loosing loop. Because there is a loosing loop either the interpolant find it (l.13) and we add the size to $MaybeThisSize$ and then we confirm the size of the loosing loop line 9, either we find the loop when $I = Init$ at line 13. We only increase k by one for each iteration. k can't reach $size_{max}$ without reaching first the size of the loosing loop that will be added to $MaybeThisSize$ or will lead directly to the *exit* line 15. We face a contradiction.

4 Tests

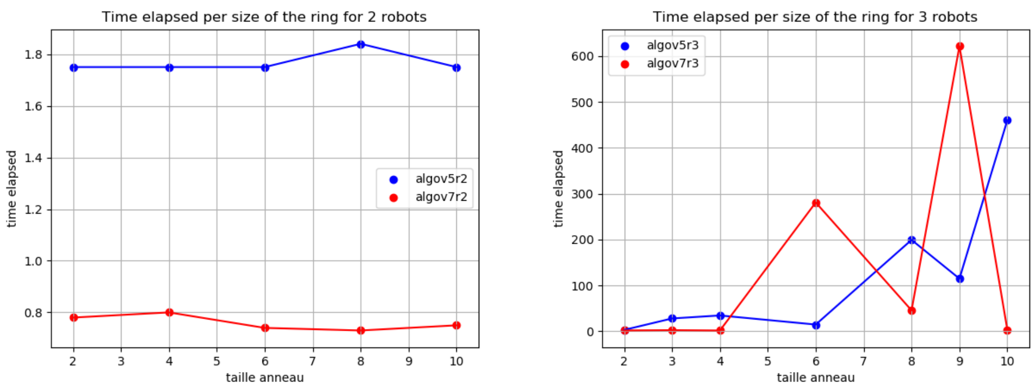
We are now comparing both algorithms through some tests. We will put an initialized configuration with no other conditions than: no winning configuration, all s at -1 and all t at 0. And with that configuration, one of the strategy written above. We mesure the time it takes the algorithm to find a loosing loop. We will change the number of robots and the size of the ring from a test to another. There is a timeout of 24h (86400s), and all tests are performed on the same computer.

Here 'algov5' is the algorithm from the report [2], and 'algov7' is the alternate version of the algorithm, presented above.

We will use one graph per number of robots. Each graph will show the time it tooked to find a loosing loop per size of the ring for a given number of robots.

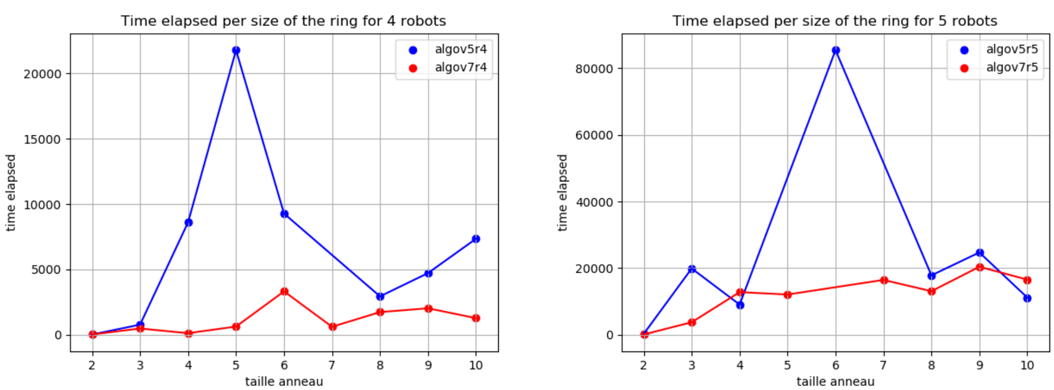
4.1 Test ϕ_{Simple}

First, we test ϕ_{Simple}, a simple strategy that moves the robot toward its closest neighbor.



(a) (b)

Figure 5: Results for 2 (a) & 3 (b) robots

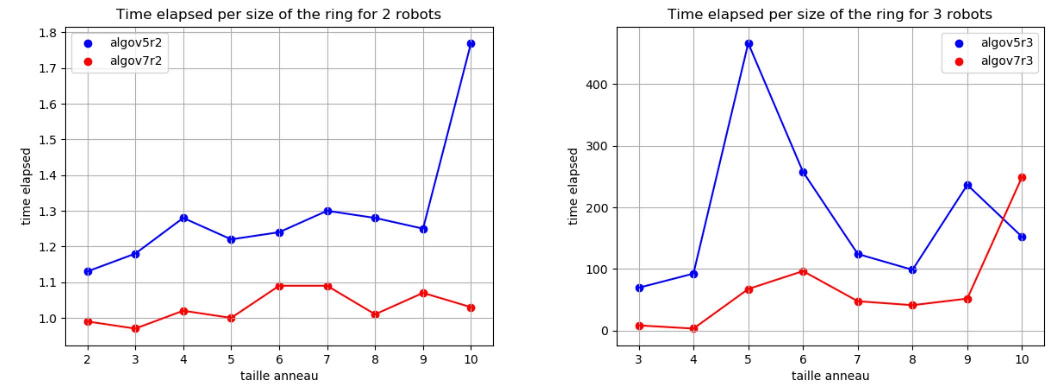


(a) (b)

Figure 6: Results for 4 (a) & 5 (b) robots

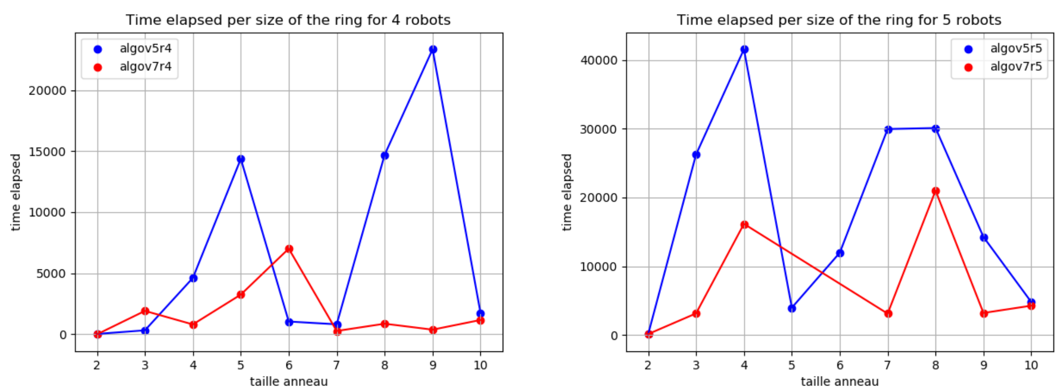
4.2 Test ϕ_{SM}

Now we test ϕ_{SM}, a strategy a bit more elaborate that should be harder to solve and more relevant.



(a) (b)

Figure 7: Results for 2 (a) & 3 (b) robots

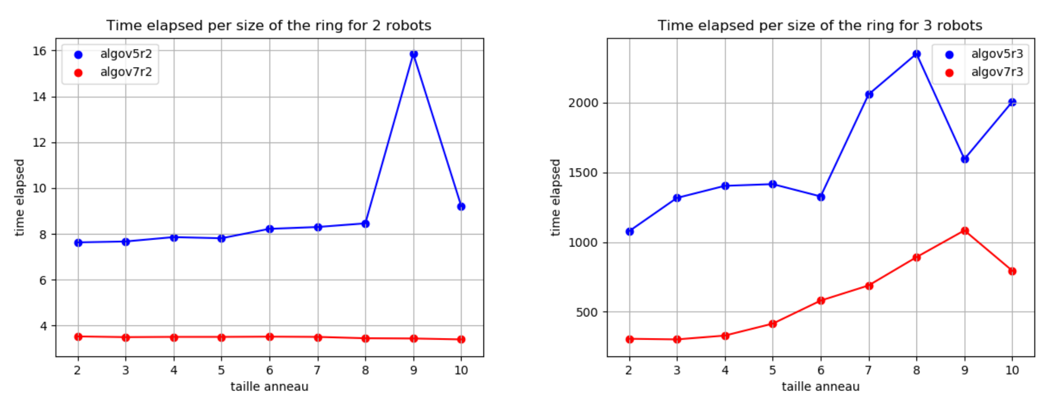


(a) (b)

Figure 8: Results for 4 (a) & 5 (b) robots

4.3 Test ϕ_R

We, now, test ϕ_R the most complex strategy that we have so far.



(a) (b)

Figure 9: Results for 2 (a) & 3 (b) robots

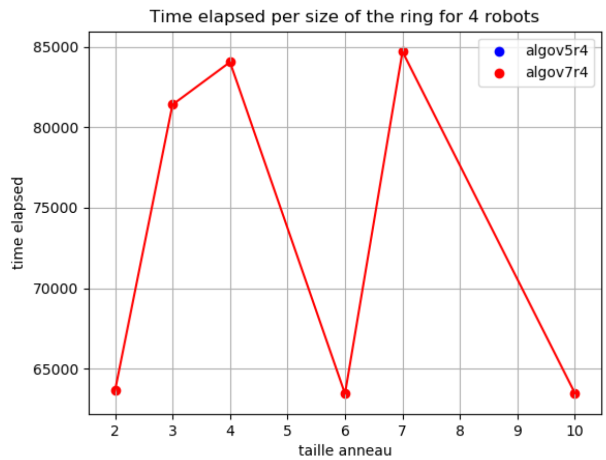
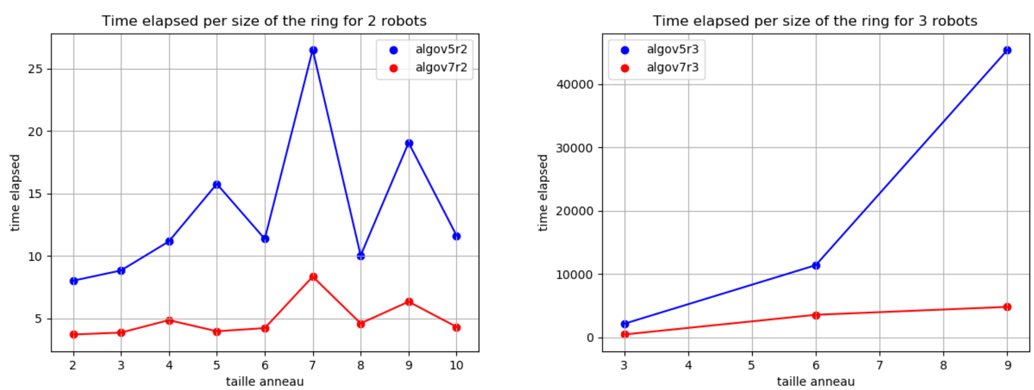


Figure 10: Results for 4 robots

4.4 Test $\phi_{Ultimate}$

Finally, we test the "Ultimate" strategy, the one that assembles all the strategies above: ϕ_{SM}, ϕ_R and ϕ_{ON}.



(a) (b)

Figure 11: Results for 2 (a) & 3 (b) robots

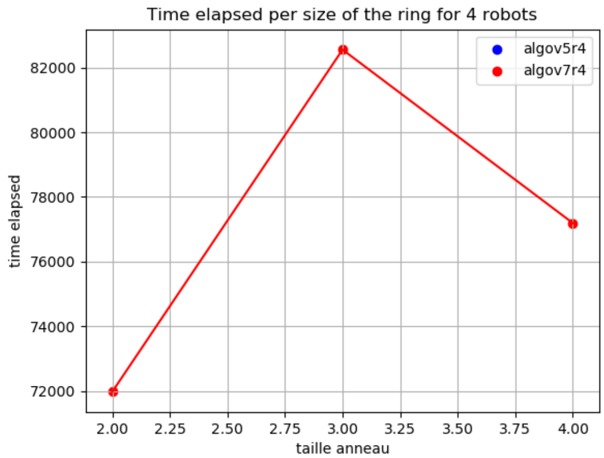


Figure 12: Results for 4 robots

5 Conclusion

//TODO

References

[1] Ralf Klasing, Euripides Markou, and Andrzej Pelc. *Gathering asynchronous oblivious mobile robots in a ring.* Tech. rep. RR-1422-07. UMR 5800 - Université Bordeaux 1, 351, cours de la Libération, 33405 Talence CEDEX, France: Laboratoire Bordelais de Recherche en Informatique, Jan. 2007.

[2] Nathalie Sznajder and Souheib Baarir. *Algorithme d'accélération par interpolants. (French) [Acceleration Algorithm using an interpolant].* Tech. rep. Laboratoire Informatique de Paris 6 (LIP6), Feb. 2022.