2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

# Pac-Man is Overkill

Renato Fernando dos Santos[1,2], Ragesh K. Ramachandran[3], Marcos A. M. Vieira[2] and Gaurav S. Sukhatme[3]

***Abstract*— Pursuit-Evasion Game (PEG) consists of a team of pursuers trying to capture one or more evaders. PEG is important due to its application in surveillance, search and rescue, disaster robotics, boundary defense and so on. In general, PEG requires exponential time to compute the minimum number of pursuers to capture an evader. To mitigate this, we have designed a parallel optimal algorithm to minimize the capture time in PEG. Given a discrete topology, this algorithm also outputs the minimum number of pursuers to capture an evader. A classic example of PEG is the popular arcade game, Pac-Man. Although Pac-Man topology has almost 300 nodes, our algorithm can handle this. We show that Pac-Man is overkill, i.e., given the Pac-Man game topology, Pac-Man game contains more pursuers/ghosts (four) than it is necessary (two) to capture evader/Pac-man. We evaluate the proposed algorithm on many different topologies.**

## I. Introduction

Pac-Man is a popular maze arcade game developed and released in 1980 [1]. Basically, the game is all about controlling a "pie or pizza" shaped object to eat all the dots inside an enclosed maze without being apprehended by the four "ghosts" patrolling the maze. Figure 1 shows the screenshot of Pac-Man at the start of the game. Pac-Man is an instance of a Pursuit-Evasion Game (PEG) in which an evader (Pac-Man) is pursued by four pursuers (ghosts). Therefore, the evolution of the Pac-Man game can be studied by analyzing the associated PEG problem.

Pursuit-Evasion Game (PEG) is a well studied topic in the robotics literature [2]–[6]. The huge interest for PEGs in robotics stems from its application to multi robot problems such as surveillance, search and rescue, boundary defense etc. Pac-Man belongs to a subclass of PEGs commonly referred as the Multi-Pursuer Single-Evader (MPSE) [6] pursuit evasion problem. Common approaches for solving and analyzing PEGs are based on game theory, these approaches can be traced back to the seminal work of Isaacs on differential games [7]. Since then, various versions of PEGs were introduced in the robotics literature: continuous time PEGs [8], discrete time PEGs [3], discrete PEGs [9], etc. In this paper we focus on discrete PEGs based on the formulations presented in [5] and [9].

*This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001

[1]Instituto Federal de Educação, Ciência e Tecnologia de Mato Grosso do Sul, Rua Salime Tanure, s/n, Bairro Santa Tereza, Coxim, Mato Grosso do Sul, Brazil, CEP: 79400-000 renato.santos@ifms.edu.br

[2]Laboratory of Computer Vision & Robotics, Computer Science Department, Universidade Federal de Minas Gerais, Av. Antônio Carlos, 6627 - Prédio do ICEx Pampulha, Belo Horizonte, Minas Gerais, Brazil, CEP: 31270-901 mmvieira@dcc.ufmg.br

[3]Robotic Embedded Systems Laboratory, University of Southern California, Ronald Tutor Hall, RTH426, 3710 McClintock Ave, Los Angeles, CA 90089-9121 rageshku@usc.edu; gaurav@usc.edu

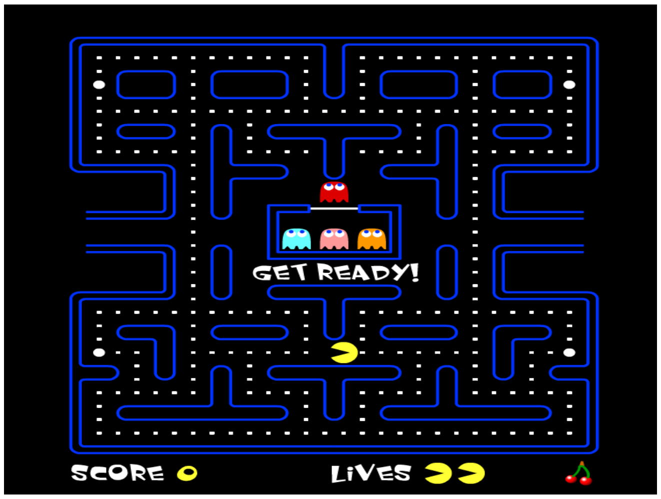

Fig. 1. A screenshot of the Pac-Man game. The yellow colored pie shaped object is the Pac-Man. The four entities at the center of the maze are the ghosts.

In essence, a discrete pursuit evasion game (DPEG) amounts to solving a PEG on a graph. The graph in this setting can be interpreted as a topological map of a domain representing the connectivity of various components in the domain of interest. Notably, various formulations of DPEGs have been proposed based on the nature of search and the definition of the capture state of the evader. We refer the reader to the survey paper by T.H. Chung et al. [2] for an overview on the various formulations used in robotics.

Since our paper is anchored on DPEG, we envision the domain of DPEG to be a graph which models any complex bounded environment. The nodes of the underlying DPEG graph represent regions (e.g. rooms in buildings) in the environment under consideration. The edges in the graph describe the links among the various regions represented by nodes. In our formulation, the pursuers attempt to capture the evaders in the graph as a team. An evader is captured if one or more pursuers reside in the same node as the evader. Similar to [5], our work also aims at computing the minimum number of steps to capture all the evaders. However, in this work, we focus on the parallelization of the optimal strategy to compute the capture time delineated in [5]. In addition, our parallelized algorithm is shown to be effective in computing the capture time of the game, played by robots with different speeds.

The main contributions of our work are the following. First, we describe a parallel optimal capture time algorithm (Algorithm 1) to minimize the time of capture in a PEG, even if evader plays the optimal strategy. This algorithm also indicates the minimum number of pursuers necessary to capture an evader in a given topology. This parallelization is important since PEGs, in general, are EXPTIME-complete [10]. Second, we apply this algorithm to the Pac-Man game and show that the game only requires 2 ghosts, thus, Pac-Man is an overkill since it has 4 ghosts. To compute this result, we have to handle almost $10^8$ states, which is only possible in a time due to parallelization. Third, we extend this algorithm to the case when the evaders and pursuers have different speeds. Fourth, we extend the parallel algorithm (Subsection 1) to support pac-dots (Pac-dot Algorithm IV-A), that increases evader survival. Pac-dot Algorithm maintains the optimal properties of Algorithm 1, even if evader plays optimal strategy.

The remaining of this paper is structured as follows. In Section II, we present the assumptions, terminology and definitions. In Section III, we describe the parallel algorithm. In Section IV, we describe the PEG game with Pac-dots and present the Pac-dot algorithm. In Section V, we explain the implementation's details. In Section VI, we show the results for many different topologies, including the Pac-Man game. In Section VII, we conclude the paper.

## II. Definitions, taxonomy and assumptions

In this section, we describe the various assumptions, terminologies and definitions used in our paper. Firstly, we outline the framework used in the paper and specify the sensing, communication and computational capabilities of the robots used in our framework. In addition, we enumerate the resources which aid us in the improved scalability of the optimal strategy computation algorithm presented in [5]. Finally, we detail the terminologies and definitions used to describe our problem and its algorithmic solution.

We consider games that are played on domains composed of a considerable number of regions, such domains include but are not restricted to urban areas, indoor regions of a building, any disaster precinct. We assume that each participant (pursuer or evader) has access in real time to the current position of all other participants, including its own. That is to say, the participants are equipped with global vision, i.e. they have full knowledge of the game. This is the hardest case to capture evaders since evaders know before moving the exact positions of all pursuers on the map at that instant and this gives the opportunity for evaders to play optimally.

This paper initially focuses on the parallelization of the optimal strategy algorithm [5], to compute the least cost to PEG. In our problem setting, we identify the cost of PEG with the capture time of the evader. We propose an approach that uses parallelism to enable computing the required massive processing of the PEG. Furthermore, we extend our approach to incorporate heterogeneous robots. It is worth noting that, the pursuers can have distinct speeds, which contribute towards the reduction in capture time and/or the reduction in the number of pursuers necessary to catch an evader. Finally, we extend the parallel algorithm to consider the case of pac-dot.

As mentioned in the preceding section, the domain of a PEG is defined as a discrete space, bounded and mapped by a topological graph, discretized through a grid of the environment, where each node represents coarse-grained regions (grid cell) and edges/links connects neighbor regions (interconnected cells in the grid). A topological graph is represented as an adjacency matrix, that is provided as input to the algorithm.

We use a discrete-event simulation where the state of the system changes after each step of the game. In our context, this discrete sequence of events is a pursuer's turn to move followed by an evader's turn to move, this represents one step of the game or one time unit. At each step of the game, the participant pursuer or evader occupies one region and will move to an adjacent neighboring region. Based on the speed (hops), in a single game step, multi speed pursuers can move to multiple regions connected according to the underlying topological graph. We are interested in the class of games in which there exist enough pursers to guarantee game termination in topological graphs. To illustrate the effectiveness of our algorithm, we test it on graphs considerably larger than the ones used in [5]. Initially we consider that pursuers and evaders move at the same speed, one hop in the topology at each time step. We later consider the case where the pursuer can move multiple hops at each time step, moving faster than evader. Now we lay down the terminologies required for describing our problem.

The game is played on an undirected connected graph $G = (V,L)$, where a node $v \in V$ represent a region and a link between two regions $u$ and $v$ is represented using the edge $\{u,v\} \subset V \times V$. We consider two types of players in PEG: pursuers and evaders, indicated by $P$ and $E$ sets respectively. Let $P_i$ be the position of the $i_{th}$ pursuer on $G$. Similarly, $E_i$ gives the position of the $i_{th}$ evader. Now, we define a tuple $a =< P_1,P_2,...,P_n,E_1,E_2,...,E_m >$ containing the positions of all the participants in PEG. We assume that, during the evolution of the game, the pursuers and evaders make alternate moves and only one type of participant takes an action at a time step. Players move from current vertex $u$ to an adjacent vertex $v$. If we use the boolean variable $T$ (turn) to encode whether it's the pursuers turn or evaders turn to move, then the tuple $< a,T >$, can be used to represents a state of the game. The pursuers win the game if they capture the evader. Otherwise, if the evader can avoid indefinitely, then the evader wins the game. If the evader wins the game then it implies that the number of pursuers required to capture the evader is insufficient for the given graph. The minimum number of pursuers required to terminate a DPEG on $G$, is denoted by $c(G)$ in the literature [11].

We define the game as follows. The input is the initial position of the players (pursuers and evaders), and a topological graph of the environment. The output is the optimal sequence moves for all configurations of initial position of the players. The selected optimal sequence is the motion commands for agents $P$ and $E$. The goal is to minimize the maximum (worst-case) capture time of the evader. The game terminates when all the evaders are captured. An evader is captured when it resides on a vertex of the graph occupied by one or more pursuers. We refer to this state as the capture state of the evader.

From the above formulation, it is clear that a DPEG has at least $|V|^{|P|+|E|}$ states. Now, if we consider the fact that there are two turns (pursuers or evaders transition) associated with each state in the $|V|^{|P|+|E|}$ states, then state space of the game would contain $2*|V|^{|P|+|E|}$ states. Thus, the game's states are exponential in the number of players [10]. A sequence of transitions can represent the execution of a game.

## III. Optimal Algorithm

In this section, we describe our scalable algorithm to compute the optimal strategy for the pursuers and evaders participating in a PEG.

### A. Algorithm Strategy

In this section, we consider that the pursuers and evaders have the same speed.

The optimal strategy [5] is a zero-sum game, it uses a minimax algorithm [12] which minimizes the maximum possible loss for each player in the game. In a PEG, the pursuer's goal is to capture the evader as fast as possible whereas the evader's goal is to escape from the pursuers as long as possible.

We construct a game graph with the states as nodes and edges representing all possible transitions between the states (minimax tree). We refer to this graph as a *game graph*. The game starts with the pursuers turning to move and in sequence the evaders turn to move, both moves account one step at time. For each state is assigned a cost function $C(s)$ gives the minimum distance from $s$ to a capture state in a pursuer's move, and in an evader's move denotes the maximum distance from state $s$ to a capture state.

### B. Parallel Algorithm

Parallelization made the execution of the algorithm on large topological graphs tractable. As shown in Algorithm 1, original algorithm (it can be seen in [5]) has been parallelized in four key areas.

The intuition behind the parallel Algorithm, it consists of a similarity between the four functions (Lines 3, 5, 10 and 13) in one aspect, that is lock-free (lockless) programming features. Lock-free programming is a technique for multi-threaded programs without using locks or mutex [13], [14]. A multi-thread implementation of our algorithm is lock-free because it satisfies three properties [13], [14]: it is multi-thread; threads access shared memory, and; threads can't block each other.

In Line 1 of Algorithm 1 generate all states of pursuers and all states of evaders. In the Lines 2-3 initialization function is called to assign the initial cost of each state. The transitions from each state to an adjacent state are computed by genTransition function call (Lines 4-5). In the sequence, there is a repeat structure (lines 6-14) that run a loop until there are no new cost of the states to be updated (the negation of *change* boolean variable becomes *true*). Within the repeat

**Algorithm 1** Parallel algorithm.

```
 1: Generate all States
 2: for all state $s$ do "in parallel"
 3:     INITIALIZATION(s)
 4: for all no capture state $s$ do "in parallel"
 5:     GENTRANSITIONS(s)
 6: repeat
 7:     $change \leftarrow false$
 8:     $U_p \leftarrow$ set of all unmarked pursuers states
 9:     for all $s$ in $U_p$ do "in parallel"
10:         $change \leftarrow change$ or PURSUERCALCCOST($s$)
11:     $U_e \leftarrow$ set of all unmarked evader states
12:     for all $s$ in $U_e$ do "in parallel"
13:         $change \leftarrow change$ or EVADERCALCCOST($s$)
14: until not $change$
```

**Algorithm 2** Parallel algorithm functions.

```
 1: function INITIALIZATION($s$)
 2:     if $s$ is a capture state then
 3:         $C(s) \leftarrow 0$ {cost function}
 4:     else
 5:         $C(s) \leftarrow \infty$
 6: function PURSUERCALCCOST($s$)
 7:     if $s$ has one or more transition to a marked state then
 8:         $C(s) \leftarrow \min_{\forall s' \in N(s)} (C(s')) + 1$
 9:         add transition to $\rho$
10:         return $true$
11:     return $false$
12: function EVADERCALCCOST($s$)
13:     if all transition from $s$ reach a marked state then
14:         $C(s) \leftarrow \max_{\forall s' \in N(s)} (C(s'))$
15:         add transition to $\varepsilon$
16:         return $true$
17:     return $false$
```

structure, *change* is assigned with *false*. In Line 8, all unmarked pursuers states are selected and added to collection $U_p$. At next line, a loop (Lines 9-10) iterates over elements of $U_p$. Each pursuer state $s$ is passed as an argument to pursuerCalcCost function that returns *true* if state cost has been updated or *false* otherwise. Similarly, at Lines 11-13 the same reasoning is used for evaders' states.

In Algorithm 2 there is the definition of called functions in Algorithm 1. GenTrantitions was omitted here. Initialization function (1-5) checks if the argument $s$ is a capture state, initializing it with zero if *true* or $\infty$ otherwise. At Lines 6-11, is defined as the pursuerCalcCost function, with $s$ parameter. On the next line, is checked if $s$ has at least one adjacent state that has been its cost updated, if the statement is *true*, then the cost of $s$ be calculated and updated in the next lines. In Line 6, $s'$ denotes an element of set $N(s)$, that denotes all adjacent states of the state $s$, then for all adjacent state its cost is checked by $C(s')$ function, min function select the adjacent

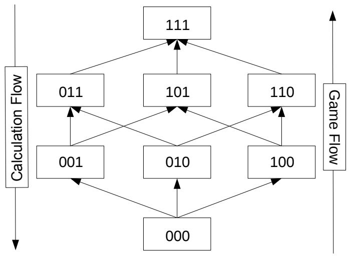

Fig. 2. A Power Set for $k = 3$, that represents the sequence of pac-dot can be reached in a game.

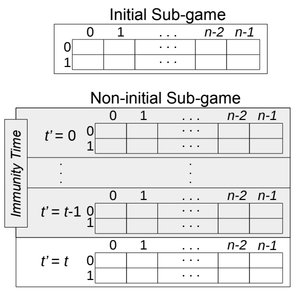

Fig. 3. Initial sub-game and non-initial sub-game.

state with lowest cost that is incremented by one and assigned to $C(s)$, updating $s$ cost. In sequence, the transition from $s$ to $s'$ (selected by min function) is added to $\rho$ policy. $true$ is returned. If the conditional structure is not satisfied, then $false$ is returned at Line 17. The evaderCalcCost function (Line 12-17) on the if structure check if all adjacent states of $s$ has been its cost updated, returning $false$ at Line 17 or case $true$ performed the calculate and update cost of $s$ in sequence. At Line 14, max function selects the maximum cost between the cost of all adjacent states of $s$ and updates its cost. In sequence, the transition is added to $\varepsilon$ policy and $true$ is returned.

### C. *Multi Speeds Pursuers*

In this section, we explain the approach for heterogeneous pursuers where each pursuer in $P$ can move with a different speed (in terms of the hops). More concretely, let a pursuer $P_i \in P$ is occupy a vertex $s$ and if it can move with a maximum speed $v_i$, then when it's turn comes, $P_i$ can reach any node in the topological graph there if there is a path of length $v_i$ between the node and $s$. In other words, the velocity of a pursuer is its capacity of how many hops it can move in a game step.

*1) Solution approach:* First is selected the largest speed between all pursuers. In next, the adjacency matrix of the topological graph is used to perform a graph search (Breadth-First Search or Depth-First Search) for each node to compute all paths, whose path length be lesser or equal than the largest pursuer speed value. After generation states step, all transitions are generated considering the speed of each pursuer, that is, all nodes that can be reached with distance lesser or equal than their speed.

## IV. PEG with Pac-dot

Pac-dot is an item (pill) present in the Pac-Man game that when taken by the evader, allows the evader to be immune to being captured for an immunity time of $t$ time units. During this immunity time $t$, the game swaps the roles of pursuers and evaders so that if the evader captures the pursuers, the pursuers can be penalized. In this case, the pursuer is moved to a previously specified location. After the immunity time, pursuers come back to the game and two cases can happen: if the game still has one or more pac-dot, evaders can reach another pac-dot and the immunity time restarts as explained above. If a pac-dot is not reached by the evader or there are no more pac-dots, pursuers pursue the evader until they capture it, and the game terminates. The locations of the pac-dots in the topological map have been known before the game starts.

Let $k$ be the number of pac-dot in a game. The number and structure of sub-games generated is modeled as a Power Set [15]. We generate $2^k$ sub-games as shown in Figure 2 for $k = 3$. Each rectangle is a sub-game, arrow Calculation Flow indicates the computation of state costs game flow, and Game Flow indicates the followed flow to obtain the optimal path, after state costs computation. For each pac-dot is assigned a boolean bit, to identify the taken flow. If all bits are equal zero, this represents the initial game sub-game. If a pac-dot is reached by the evader, then the matching bit is inverted to one, and follows the arrow to the new corresponding sub-game, and so on, until the game terminates.

In the Figure 3, we show the two types of sub-games. Initial sub-game is the sub-game where the game starts, it has only a frame and it does not have immunity time. A frame is composed of a set of states that corresponding to $2 * |V|^{|P|+|E|}$. The game can start and terminate in the initial sub-game. Non-initial sub-games are all subsequent sub-games that has $t$ frames, where $t-1$ frames are relative to immunity time duration, and the last frame $t$ is the frame after the immunity time. In the last sub-game frame the game continues until it terminates or a new pac-dot is reached, and the sub-games flow follow ahead. For all other non-initial sub-games the behavior is similar. A non-initial sub-game has $t * 2 * |V|^{|P|+|E|}$ states. A game with $k$ pac-dots has $[(2^k - 1) * t + 1] * |V|^{|P|+|E|}$ states, and the space states it's the double. Let $s =< a', a, T >$ be the new state, where $a' =< b_1, .., b_k, t >$.

### A. *Pac-dot Algorithm*

This algorithm follows the same definitions and terminologies engaged for the parallel algorithm (Algorithm 1), and all agents have speed equal one.

To be able to take into account the pac-dot during the game, we made three modifications between parallel algorithm and Pac-dot Algorithm: 1) the initialization values are 0, $\alpha$ and $\infty$. States of the initial sub-game or states of the last frame of any non-initial sub-game are initialized with zero if a capture state, $\infty$ if a state whose evader occupies a pac-dot position, and $\alpha$ remaining states (case 1). Otherwise,

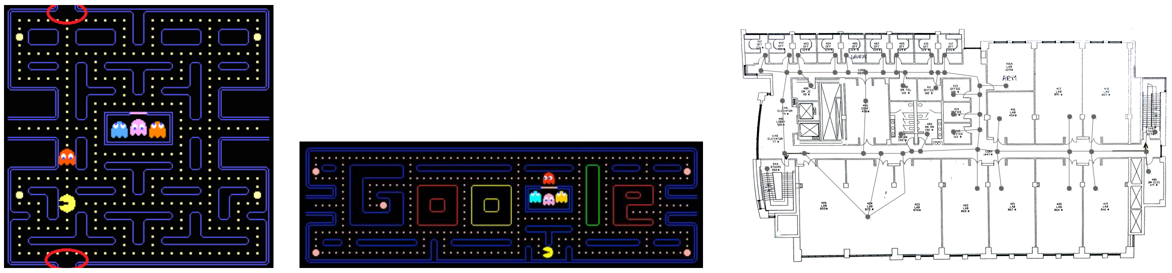
(a) Pac-Man Toroidal (b) Google Pac-Man (c) Building

Fig. 4. Evaluated topologies.

capture states are initialized with ∞ and remaining states are initialized with $\alpha$ (case 2). The two cases of initialization can be seen in Figure 3. 2) transitions: consider the two cases mentioned above, in the first case the transition occur locally (inside same frame), but if evader position is a pac-dot in pursuers states, then this transition be to states in the next sub-game, according with the pac-dot (see Figure 2 under Game Flow perspective). In the second case, all pursuers state transition be from $t'$ frame to $t'+1$ frame, and the evaders state transition be in $t$ frame. 3) The *repeat* structure in Algorithm 1 was redefined with the inclusion of two $for$ structures at lines 8-13. The most external $for$ in non crescent order iterates over the sub-games. The most internal $for$ also in non crescent order, iterates over time (frame) $t$.

## V. Implementation

In this section, we present the details of implementation of the algorithm and simulator. We also describe the computer's configuration that we used to run the experiments.

We develop a discrete simulator to evaluate PEGs in different topologies. The algorithm and simulator were encoded in Python 3 language [16]. Besides the native resources of Python language, we use the Joblib package [17]. More specifically, we utilized the class *joblib.Parallel* to implement the parallel functions.

We use two computers to simulate the games: 1) Processor Intel XEON E5-2630 v3 2.4GHz 32 cores, 128GB RAM and 12TB HD; 2) Intel Core i7-6800K 3.4GHz 12 cores, 64GB RAM, 1TB HD. All simulations of the Pac-Man game run in Computer 1 and other PEGs run in computer 2.

## VI. Simulations and Results

This section presents the results of the experiments performed using the parallel Algorithm 1 and the pac-dot algorithm (subsection IV-A). We evaluated the following topologies: Pac-Man game (Figure 1), a Pac-Man topology in a torus (Figure 4a) where we add a north-south connection (marked in red ellipse), Pac-Man version from Google (Figure 4b), a two-story building (Figure 4c), a Reduced Pac-Man version (Figure 7) where the red circles should be disregarded, and Reduced Pac-Man topology in a torus (Figure 7) where we add an east-west connection (marked in red circle).

Table I shows the results of the PEG simulation for previous described topologies. The set of topologies was simulated on computer 1. Columns are defined from left to right. Column 1 identifies the topology. Column 2 indicates the number of players (pursuers and evaders). Column 3 shows the number of vertices for each topology. Column 4 and 5 displays the number of states and transitions needed to generate the game graph. Column 6 depicts the time to generate the game graph (all states and transitions). Column 7 shows the time to calculate the cost. Column 8 presents the total execution time (Column 6 + Column 7). Column 9 shows the number of steps (cost) to capture the evader in the worst case. Column 10 presents the diameter of the topological graph. The difference between cost and diameter is that the diameter is an invariable continuous path from beginning to end and the cost may vary due to evader's attempts to prolong capture with each movement. For example, if the evader does not move during the entire game, the pursuer, in the worst scenario, would have to transverse the diameter of the graph, and the cost would be the same as the diameter.

Pac-man topology (first line of Table I) requires 65 steps to capture the evader and only needs 2 pursuers.

Next, we evaluate the Pac-Man game for heterogeneous pursuers with different speeds. Figure 5 presents the number of steps for each configuration. Each configuration (labels on X-axis) has two numbers that represent the speeds of *pursuer-evader*. The evader's speed is always 1. If the cost is infinite, it means that the number of pursuers is insufficient to capture the evader. As the speed of the pursuer increases, the number of steps to capture decreases since the pursuer can travel more hops in one time step.

Figure 6 shows the costs of the game with different speeds for games with 2 pursuers. The labels on X-axis have three numbers to indicate the speeds of *pursuer-pursuer-evader*.

Table II shows the results of the PEG with pac-dot simulation for Reduced Pac-Man and Reduced Toroidal Pac-Man topologies. These topologies were simulated on computer 1 and computer 2. Columns are defined from left to right. Columns 1-5 have the same meaning as in Table I. Column 6 shows the number of pac-dots of the simulation. Column 7 presents the evader immunity time duration. Column 8 depicts the position of each pac-dot in a simulation, differentiated by circle color (Figure 7). Columns 9 and 10 show the time to calculate the cost. Column 8 presents the total execution time (Column 6 + Column 7).

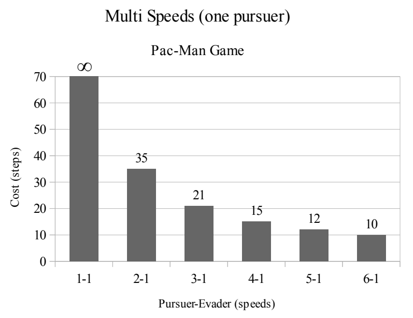

Fig. 5. Pac-Man game multi speed evaluated with one pursuer and one evader.

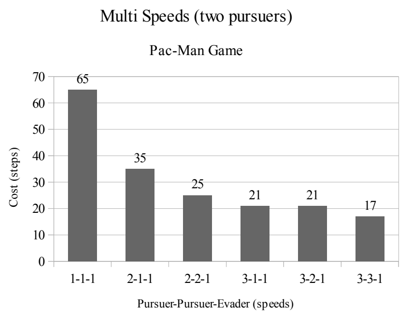

Fig. 6. Pac-Man game multi speed evaluated with two pursuers and one evader.

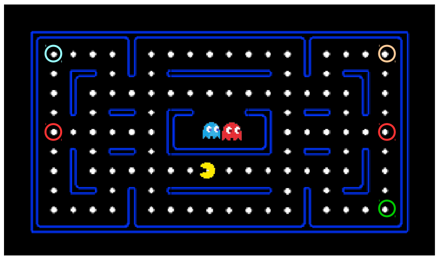

Fig. 7. Reduced Pac-Man. Reduced Toroidal Pac-Man has as an edge that links the two red circled nodes.

TABLE I

Topologies and simulation results - Parallel Algorithm with out pac-dot

| Topology | Players | Vertices | States | Transitions | Time to generate Game Graph (min) | Cost Calculation Time (min) | Total Execution Time (min) | Cost/Steps | Diameter |
|---|---|---|---|---|---|---|---|---|---|
| Pac-Man | 3 | 290 | 24389000 | 313285590 | 16 | 272 | 288 | 65 | 51 |
| Pac-Man Toroidal | 3 | 290 | 24389000 | 402542566 | 23 | 355 | 378 | 73 | 38 |
| Pac-Man Google | 3 | 317 | 31855013 | 402542566 | 29 | 431 | 460 | 64 | 49 |
| Building 2-Floors | 3 | 118 | 1643032 | 20847208 | 1 | 7 | 10 | 17 | 17 |
| Reduced Pac-Man | 3 | 102 | 1061208 | 13654176 | 4 | 9 | 13 | 21 | 21 |
| Reduced Toroidal Pac-Man | 3 | 102 | 1061208 | 13803796 | 4 | 9 | 13 | 21 | 21 |

TABLE II

Topologies and simulation results - Parallel Algorithm with pac-dot.

| Topology | Players | Vertices | States | Transitions | Number of Pac-dot | Immunity Time | Pac-dot Position | Cost/Steps | Diameter |
|---|---|---|---|---|---|---|---|---|---|
| Reduced Pac-Man | 3 | 102 | 12734496 | 166569444 | 1 | 10 | green | 34 | 21 |
| Reduced Pac-Man | 3 | 102 | 36081072 | 472402512 | 2 | 10 | orange/green | 39 | 21 |
| Reduced Toroidal Pac-Man | 3 | 102 | 12734496 | 168394616 | 1 | 10 | green | 36 | 21 |
| Reduced Toroidal Pac-Man | 3 | 102 | 36081072 | 477578800 | 2 | 10 | cyan/green | 50 | 21 |

Column 9 has the same meaning as in Table I, as mentioned earlier.

## VII. Conclusions

In this work, we presented a parallel algorithm to compute the optimal policy to capture an evader in a PEG. This algorithm can handle millions of states and allows us to compute PEG in larger topologies. For instance, we showed that the Pac-Man game is an overkill since it only needs 2 ghosts instead of 4. We also evaluated PEGs in many other topologies. We also extended the algorithm for different speeds. Finally, we also extended the algorithm to increase evader survival in a game.

For future work, we intend to integrate these algorithms into a control framework.

## References

[1] "Pac-man," https://www.thoughtco.com/pac-man-game-1779412, July 2019.

[2] T. H. Chung, G. A. Hollinger, and V. Isler, "Search and pursuit-evasion in mobile robotics," *Autonomous robots*, vol. 31, no. 4, p. 299, 2011.

[3] S. D. Bopardikar, F. Bullo, and J. P. Hespanha, "On discrete-time pursuit-evasion games with sensing limitations," *IEEE Transactions on Robotics*, vol. 24, no. 6, pp. 1429–1439, 2008.

[4] S. Pan, H. Huang, J. Ding, W. Zhang, C. J. Tomlin, *et al.*, "Pursuit, evasion and defense in the plane," in *2012 American Control Conference (ACC)*. IEEE, 2012, pp. 4167–4173.

[5] M. A. M. Vieira, R. Govindan, and G. S. Sukhatme, "Scalable and practical pursuit-evasion with networked robots," *Intelligent Service Robotics*, vol. 2, no. 4, p. 247, Sep 2009. [Online]. Available: https://doi.org/10.1007/s11370-009-0050-y

[6] V. R. Makkapati and P. Tsiotras, "Optimal evading strategies and task allocation in multi-player pursuit–evasion problems," *Dynamic Games and Applications*, Jul 2019.

[7] R. Isaacs, *Differential Games: A Mathematical Theory with Applications to Warfare and Pursuit, Control and Optimization*, ser. Dover books on mathematics. Dover Publications, 1999. [Online]. Available: https://books.google.com/books?id=XIxmMyIQgm0C

[8] H. Yamaguchi, "A cooperative hunting behavior by mobile-robot troops," *the International Journal of robotics Research*, vol. 18, no. 9, pp. 931–940, 1999.

[9] T. D. Parsons, "Pursuit-evasion in a graph," in *Theory and applications of graphs*. Springer, 1978, pp. 426–441.

[10] A. S. Goldstein and E. M. Reingold, "The complexity of pursuit on a graph," *Theoretical Computer Science*, vol. 143, no. 1, pp. 93 – 112, 1995.

[11] A. Berarducci and B. Intrigila, "On the cop number of a graph," *Advances in Applied Mathematics*, vol. 14, no. 4, pp. 389–403, 1993.

[12] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009.

[13] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 73–82.

[14] M. Herlihy, "A methodology for implementing highly concurrent data objects," *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 5, p. 745–770, Nov. 1993.

[15] K. J. Devlin, *Fundamentals of contemporary set theory*. Springer, Berlin, 1979.

[16] P. S. Foundation. (2019) Python programming language. [Online]. Available: https://www.python.org

[17] J. Developers. (2008) Joblib running python functions as pipeline jobs. [Online]. Available: https://joblib.readthedocs.io/en/latest/index.html