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. PEGs are important due to its application in surveillance, search and rescue, disaster robotics, boundary defense and so on. But, in general, PEG requires exponential time to compute the minimum number of pursuers to capture an evader. To mitigate this, we have design a parallel optimal algorithm to minimize the capture time in PEG. Given a discrete topology, this algorithm also outputs the minimum number of pursers to capture an evader. Our new algorithm enable us to investigate larger topologies. 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 also extend the algorithm to consider different speeds. we also extended the algorithm to increase evader survival in a game. We evaluate these algorithms for many different topologies.**

I. Introduction

Pac-Man is a popular maze arcade game developed and released in 1980 [1]. Any kid from the 80's and 90's would have some childhood memories associated with the game. Basically, the simple 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 the Pac-Man game at the start of the game. Technically speaking, 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 nature of the Pac-Man game can be studied by analyzing the associated PEG problem.

Pursuit-Evasion Games (PEGs) is a well studied topic in robotics literature [2]–[6]. The huge interest for PEGs in Robotics owes to its application in multi robot problems such as surveillance, search and rescue, boundary defense and many more. The aforementioned Pac-Man game 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 Issacs on differential games [7]. Since then, various versions of PEGs were introduced in robotics literature: continuous time PEGs [8], discrete time PEGs [3], discrete PEGs [9], to name a few. However, in this paper we focus on discrete PEGs based on the formulations presented in [5] and [9].

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 the a topological map of a domain representing the connectivity of various connected components in the domain of interest. Notably, various formulations of DPEGs have also been proposed based nature of search and definition of capture state of 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. Innately, the edges in the graph describes 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 arrested if one or more pursuers reside in the same node as the evader. Similar to [5], our work also aims at computing the stopping of the game (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

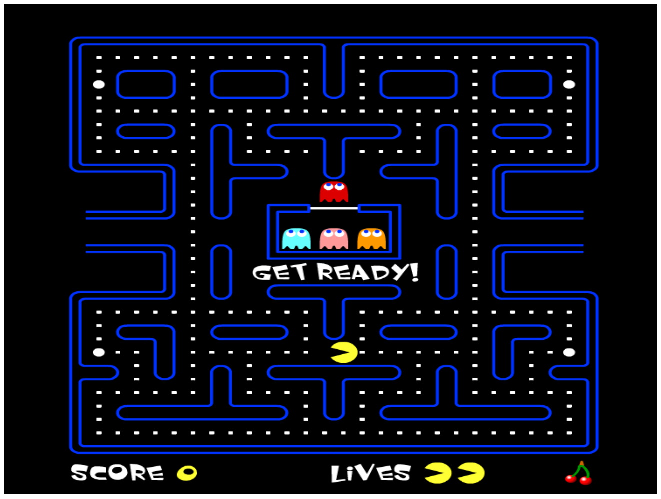


Fig. 1. A screenshot of the Pac-Man game. The yellow colored pie shaped object is the Pac-Man. The four entities at the center of the maze are the ghosts.

*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

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 optimal strategy. This algorithm also indicates the minimum number of pursuer 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 to compute in a feasible time due to our 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 maintain 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 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 optimal.

This paper initially focuses on the parallelization of the optimal strategy algorithm [5], to compute the shortest 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 velocities, 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). Topological graph is represented as an adjacency matrix, that is provided as input to the algorithm. Adjacency matrix is a square matrix of order n, where n is the number of regions (or occupied cells in discretized ambient), and each region denoted by matrix line i $(0 \leq i < n)$ can be linked or unlinked for each region (of the map) denoted by matrix column j $(0 \leq j < n)$. If two regions (i, j) are linked, then its value is 1, otherwise is 0.

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 occupy 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.

Abstractly, the game is played on a 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 take actions (move in our case) alternately and only one type of the participant take an action at a time step. In other words, each instance of a game play is composed of a pursuer's turn's to move at one game time step followed an evader's turn's to move at the subsequent game time step. Players move from current vertex u to an adjacent vertex v. If we use the boolean variable T (turn) to encode whether its 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 wins the game if they captures 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].

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.

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 motion commands for agents P and E. The goal is to minimize the maximum (worst-case) capture time of the evaders.

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. Consequently, the game terminates if and only if all the evaders are captured. Thus, a non-terminating game implies that more pursuers are required to the capture all the evaders.

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 focus only on equipotent players. Multi speed players will be detailed later. We consider that the pursuers and evaders have the same speed, so they move one hop in the topological graph at each time step.

We remind about our assumption that, the positions of all participants are known for both pursuers and evaders at each time step. The optimal strategy for the pursuers and evaders is a zero-sum game, of the form where the gains or losses of the pursuers are exactly balanced by the losses or gains of the evaders. The optimal strategy proposed by [5], 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 evader as fast as possible whereas the evader's goal is to escape from the pursuers as long as possible.

Recall that, the evolution of our PEG occurs on a state space with $2*|V|^{|P|+|E|}$ states. We construct a game graph with the states as nodes and edges representing all possible transition between the states. It is easy to infer that the resulting graph is a connected and directed one. We refer to this graph as a *game graph* Please, do not confuse the *game graph*, which is a direct graph and represents all the player's moves in the game with the *topological map*, which is an undirect graph and represents the environment where the game is played. The start configuration of the game is the initial positions of pursuers and evaders. The game starts with the pursuers turn's to move and in sequence the evaders turn's to move, both moves account one step at time. In turn, there is an directed edge to all possible next states. The cost function $C(s)$ assign's a cost to every state in the state space. 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 a evader's move denotes the maximum distance from state s to a capture state.

For a pursuer, the algorithm determines the policy ρ that selects the neighboring state that has the smallest $C(s)$. On the contrary, the neighboring state with the largest $C(s)$ is chosen as the policy ε for an evader. The game is essentially a traversal on the game graph that ends, if the traversal ends in the capture states implying that the pursers won the game. Whereas, if some states recur constantly during the traversal of the game graph, then game is considered to be non-terminating, thereby favoring victory for the evaders.

Initially, Algorithm 1 generate all possible states that the pursuers and evaders position configurations can assume throughout the game. Next step is to assign an initial cost to each state and classify them into two sets: one containing the capture states, and the other containing non capture states. The initial cost assigned to the states is zero if is a capture state and infinite otherwise. After, during game graph traversing, costs of the states whose costs were initially infinite are updated to be the cost (distance) of each state to a capture state. There are two possibles results that one can obtain once the loops break: all states were processed, and the output is the shortest possible cost to capture or there are at least one state that remain with cost equal to ∞. The later means that the number of pursuers is insufficient for evaders' capture. More formally, we define the two cases mentioned above as:

$$C(s) = \min(N(s)) + 1 \tag{1}$$
$$s' = \begin{cases} C(s'), & \text{if } \exists N(s)(C(s') \neq \infty) \\ \infty, & \text{else} \end{cases}$$

$$C(s) = \max(N(s)) \tag{2}$$
$$s' = \begin{cases} C(s'), & \text{if } \forall N(s)(C(s') \neq \infty) \\ \infty, & \text{else} \end{cases}$$

Equation 1 calculates the state cost during a pursuer move. The current state and time is denoted by s. Let $N(s)$ be all possible transitions set of s, each transition s'. If the cost of s' is different from infinity, then this transition will result in a state for which the cost was previously updated. The cost of s' is maintained separately for the both pursuers and evaders. Finally the minimum cost is identified and added to one, obtaining the cost of s. The evader cost for a state can be evaluated in a similar manner by using Equation 2 instead of Equation 1. Moreover, the maximum value is chosen directly without any incrementation.

Each iteration of the repeat loop select the states with unaltered cost and having multiple transitions to marked states (states with modified cost or capture state). As mentioned earlier, pursuers and evaders take turns in executing their moves. When it is the pursuer's turn to move, the minimum cost state among all neighbor states is selected. We update the cost of current state by assigning it the cost of the selected neighbor incremented by one (transition cost). Similarly during an evader's move, the state gets the maximum cost among its neighbors.

All states that are marked in the i^{th} iteration are added to the set F_i. This means that F_0 is the set of all capture states, F_1 is the set of all states that can reach a capture state in a single pursuer move. Therefore inductively, F_n denotes the set of states that can transition to a capture state in n pursuer moves.

B. Parallel Algorithm

One of the contributions of this work is a parallelized version of the algorithm to compute PEG optimally. Parallelization made the execution of the algorithm on large topological graphs tractable. As shown in Algorithm 1, original algorithm has been parallelized in three key areas.

In Algorithm 1, we assume that *Start Process* denote the procedure call to run a procedure in parallel. In Algorithm 1 there are 3 points of parallelization at lines 25, 27 and 35. A point of parallelization runs one procedure by call and it can run several procedures at a time.

The Line 22 of Algorithm 1 performs initialization/computation of some essential things to generate and execute the game, such as: process the graph adjacency matrix and generate the neighborhood for each vertex The Line 25 is the first parallelization point, that uses *Start Process* to start "simultaneously" several calls to *separate* procedure. *Separate* procedure (defined in lines 1-6) initialize the state's cost with 0 if it is a capture state or infinite if not.

In the next step (line 26), similar to what occurred in line 25, *Start Process* is triggered to run calls in parallel of the *genTransitions* for each state. Procedure *genTransitions* generate all possible transitions for pursuer's move and evader's move . Next step, variable i is initialized with zero and represents iteration number of the repeat structure. In the sequence, there is a repeat structure (lines 30-38) 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 structure, variable i is incremented by one and *change* is assigned with $false$. In line 33, it selects all unmarked states that have a transition to a marked state and assigned to U. Unmarked states are those still having cost equal to infinite while marked states is a capture state or a state that already have been selected in previous i iteration and thus its cost is the shortest distance to the capture state. At next line, a loop iterates over elements of U. For each state s, the parallelism is started by *calcCost* procedure call. i variable is the second positional argument to *calcCost* procedure. Defined in lines 7-20, *calcCost* procedure initially assigned $false$ to *change*. In next, if parameter s is a pursuer

Algorithm 1 Parallel algorithm for computing the game graph.

```
 1: procedure SEPARATE(s)
 2:     if $s$ is a capture state then
 3:         add $s$ to $F_0$
 4:         $C(s) \leftarrow 0$ {cost function}
 5:     else
 6:         $C(s) \leftarrow \infty$
 7: procedure CALCCOST($s$, $i$)
 8:     $change \leftarrow false$
 9:     if $s$ is a pursuer move then
10:         add $s$ to $F_i$ mark $s$
11:         $C(s) = \min(N(s)) + 1$
12:         add transition to $\rho$
13:         $change \leftarrow true$
14:     else{evader move}
15:         if all transition from $s$ reach a marked state then
16:             add $s$ to $F_i$ mark $s$
17:             $C(s) = \max(N(s))$
18:             add transition to $\varepsilon$
19:             $change \leftarrow true$
20:     return $change$
21:
22: {Initialization}
23: Generate all States
24: for all state $s$ do "in parallel"
25:     Start Process(SEPARATE(s))
26: for all no capture state $s$ do "in parallel"
27:     Start Process(GENTRANSIONS(s))
28:     {Procedure genTransitions omitted here.}
29: $i \leftarrow 0$
30: repeat
31:     $i \leftarrow i + 1$
32:     $change \leftarrow false$
33:     $U \leftarrow$ set of all unmarked states that have a transition
            to a marked state.
34:     for all $s$ in $U$ do "in parallel"
35:         $changed \leftarrow$ Start Process(CALCCOST($s$, $i$))
36:         if $changed$ then
37:             $change \leftarrow true$
38: until not $change$
```

movement, than it is marked and added to F_i set. At next line, the cost of s is defined according to Equation 1. In what follow, the transition (neighbor state of state s with minimum cost) is added to the police ρ, and $true$ is assigned to *change*, that is returned. If s is an evader move, then it checks the condition if all neighbors of s are marked and if they are not marked, *change* assigned $false$ is returned. If s has all neighbors marked, then s is marked and added to F_i. The function defined in Equation 2 assigns the maximum cost to s, the transition is added to police ε, $true$ is assigned to *change* that is returned.

Thus, we can compute the cost $C(s)$ for each s state at

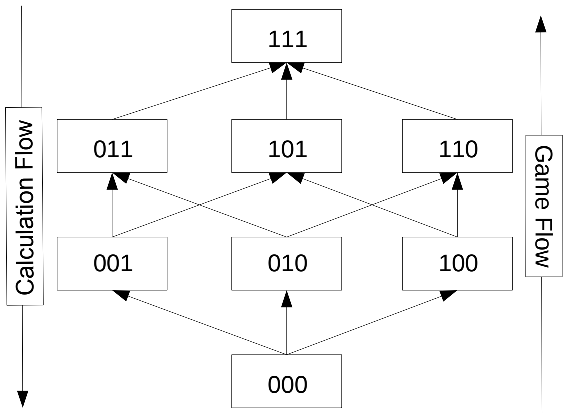


Fig. 2. The sequence of pac-dots can be reached in a game with $k = 3$. Side arrows indicate the Game flow and inverting arrows the Calculation flow.

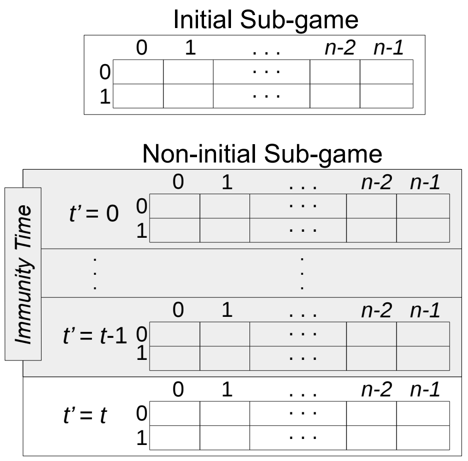


Fig. 3. Initial sub-game and non-initial sub-game.

time i in parallel. If at least one calcCost procedure called return $true$, repeat structure will be running at least one more time, as described in the algorithm.

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 topological graph there is a path of length v_i exist 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. For example, if P_x has velocity $v_x = 3$, then P_x can moving for any adjacent path with length between [0..3] from its current position.

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 largest pursuer speed value. After generation states step, all transitions are generated considering the speed of each pursuer, that is, all nodes that it can be reached with distance lesser or equal than their speed. Algorithm sequence has been unchanged.

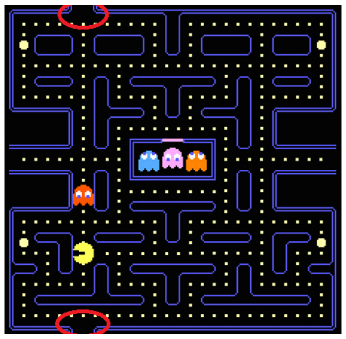
Time complexity of both Breadth-First Search (BFS) and Depth-First Search (DFS) are $O(|V|+|L|)$. Both algorithms starts at a vertex u and search a path to any vertex v in a connected graph. In our case, all paths of length equal to the largest speed value has to be computed. Considering all these factors, the worst case complexity of our approach can be shown to be $O(|V|)O(|V|+|E|) = O(|V|^2 + |V||E|)$.

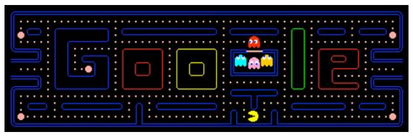
IV. PEG WITH PAC-DOT

Pac-dot is an item (pill) present in the Pac-Man game that when taken by the evader, allows the evader to be immune of 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, evader can reach another pac-dot and the immunity time restarts as explained above. If a pac-dot is not reached by evader or there is no more pac-dots, pursuers pursuit evader until capture, and the game terminates. The locations of the pac-dots in the topological map are known before the game starts.

Let k be the number of pac-dots in a game. The k pac-dots can be reached in $k!$ distinct orders. Initially, any of k pac-dots can be reached and the second pac-dot will be any of the $k-1$ remaining, and so on. Instead of having to compute $k!$ distinct games, we take a dynamic programming approach to re-utilize some computation. We generate 2^k sub-games to compute all $k!$ distinct orders to reach the pac-dots. Figure 2 shows a topological sorting representing the $k!$ distinct orders for $k = 3$. Each rectangle is a sub-game and arrows indicate the game flow. In the pre-process step of the algorithm, 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. The game always starts from the initial sub-game. If a pac-dot is reached by the evader, then matching bit is inverted to one, and followed 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 other sub-games that makes up a game with pac-dots. Each sub-game of this type has t frames, whose $t' = [0..t-1]$ is referred to immunity time and $t' = t$ is the frame after immunity time. In this state the game terminates or a new pac-dot can be reached, and the sub-games flow follow. The behavior similar for all non-initial sub-games. A sub-game has $t * 2 * |V|^{|P|+|E|}$ states


(a) Pac-Man Toroidal


(b) Google Pac-Man

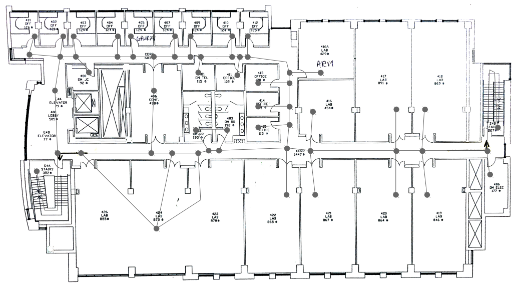

(c) Building

Fig. 4. Evaluated topologies.

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	22
Reduced Toroidal Pac-Man	3	102	1061208	13803796	4	9	13	21	23

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	22
Reduced Pac-Man	3	102	36081072	472402512	2	10	orange/green	39	22
Reduced Toroidal Pac-Man	3	102	12734496	168394616	1	10	green	36	23
Reduced Toroidal Pac-Man	3	102	36081072	477578800	2	10	cyan/green	50	23

and the game with k pac-dots has $[(2^k-1)*t+1]*|V|^{|P|+|E|}$ states and the space states is $[(2^k-1)*t+1]*2*|V|^{|P|+|E|}$.

Let $s=<a',a,T>$ be the new state, where $a'=<b_1,..,b_k,t>$. Each b_i is a boolean that represents the status of corresponding pac-dot, assigned with $false$ if not taken and $true$ otherwise.

A. Pac-dot Algorithm

This algorithm follows the same definitions and terminologies engaged for the parallel algorithm, except that all agents are homogeneous and they 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, α and ∞, where 0 is initial value of capture states, ∞ is initial value of states that evader position is equal to pac-dot position, and; common states are initialized with α. The values of α and ∞ represents two big integer values as follows: $0 \ll \alpha < \infty$, where ∞ is an integer bigger than α that attracts the evader to a pac-dot position. In Equation 1 the if and else sentences are replaced by if $\exists N(s)(C(s')<\alpha$ and else s'. Similarly, in Equation 2 the if and else sentences are replaced by if $\forall N(s)(C(s')<\alpha$ and else s'. 2) transitions: if sub-game is the initial sub-game or no initial sub-game and time $t'=t$, then states of pursuers turn to move that has value equal ∞, it will be the corresponding boolean of b_i set to true, that is, this is a transition of sub-games. If any non-initial sub-game and $0 \leq t' < t$, then for all pursuers turn move of t' to $t'+1$. Still under the last conditional sentence, all states where one or more pursuer occupy the same evader position, then this pursuers are penalized with their shift to a position defined a priori. 3) The *repeat* structure lines, 30-38, it was redefined with the inclusion of two for structures. The most external for in non crescent order iterates over the sub-games. The most internal for, too in non crescent order, iterates over time t. If is the initial sub-game, unique case that sub-game value is zero/$false$ for all b_i, the internal for breaks after one iteration.

V. IMPLEMENTATION

In this section, we present the details of implementation of the algorithm and simulator and, we also describe the computers 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 [13]. Besides the native resources of Python language, we use the Joblib package [14]. 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

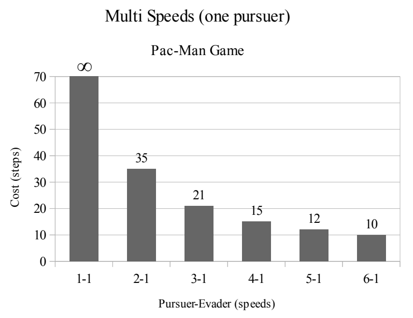


Fig. 5. Pac-Man game multi speed evaluated with one pursuer and one evader.

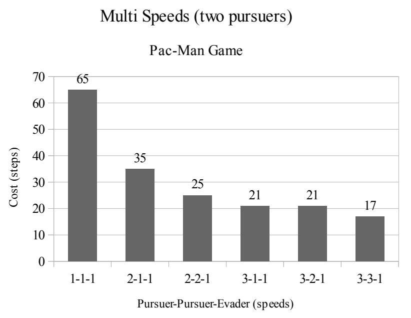


Fig. 6. Pac-Man game multi speed evaluated with two pursuers and one evader.

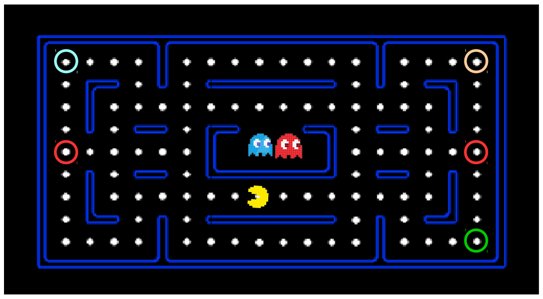


Fig. 7. Reduced Pac-Man. Reduced Toroidal Pac-Man has as an edge that links the two red circled nodes.

RAM, 1TB HD. All simulations of the Pac-Man game run in Computer 1. Simulation of 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 a 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 evader). 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 topology 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.

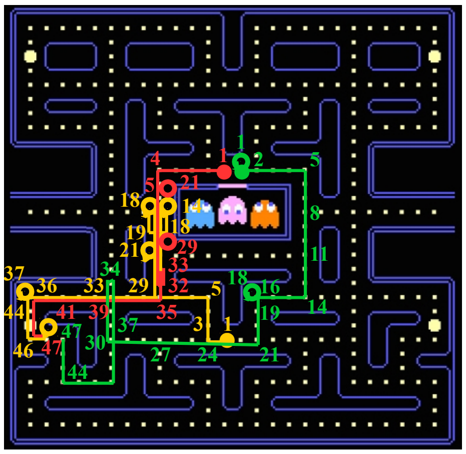


Fig. 8. Trace execution Pac-Man with two pursuers (speed = 1) and one evader (speed = 1), from start (initial position of original Pac-Man game) to capture.

Next, we evaluate the Pac-Man game with different pursuer's speeds. Figure 5 presents the number of steps for each configuration. Each configuration (labels on X-axis) has two numbers that represents 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*.

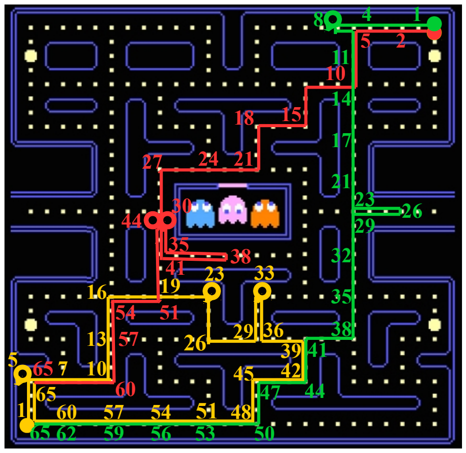


Fig. 9. Trace execution Pac-Man with two pursuers (speed = 1) and one evader (speed = 1), from start to capture, in the worst case.

Figure 8 and Figure 9 illustrate the trace execution with 2 pursuers and 1 evader for the cases where the players start in the initial position of original Pac-Man game and when they start in the worst case execution (1st line of Table I). All players have the same speed. The red and green lines indicates pursuer's paths and yellow line is the evader's path.

Table II shows the results of the PEG with pac-dot simulation for Reduced Pac-Man and Reduced Toroidal Pac-Man topologies. This topologies was simulated on computer 1 and computer 2. Columns are defined from left to right. Columns 1-5 have the same meaning as in Table II. 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 shows the time to calculate the cost. Column 8 presents the total execution time (Column 6 + Column 7). Column 9 have the same meaning as in Table II, 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 billions of states and allows to compute PEG in larger topologies. For instance, we showed that 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. [Online]. Available: https://doi.org/10.1007/s13235-019-00319-x

[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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397595800266

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

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