Intelligent agents in Tileworld: report

Jakub Tomasek (N1308376B, Team United)

***Abstract*—The report describes our solution to the semestral project for CZ4046. We applied genetic programming (GP) to optimize behavior of agents in Tileworld, a multi-agent test-bead. The two resulting agents achieved on average scores 59, 726, and 19 when deployed in the environments number one, two, and three respectively.**

CONTENTS

I. INTRODUCTION

A. Introduction of Tileworld

Tileworld is an agent test-bead initially introduced by Pollack and Ringuette in 1990 [1]. Since then, the rules were slightly adjusted for recent needs to test more relevant issues in multiagent systems [2].

Tileworld is composed from cells arranged into a square grid. There are four types of objects which can be placed in the cells: tiles, holes, obstacles, and agents. Except agents, the objects appear and disappear dynamically. That is a statistical process; in each step, Gaussian distribution is sampled to determine number of objects of the given type created in the step. Each object has fixed lifetime after which the object disappears. To modify the density of the environment and basically change the nature of the problem we can adjust mean and variance of the Gaussian. Spacial distribution of objects in Tileworld is uniform.

An agent receives points for filling holes with tiles. To achieve that, the agent have six actions available: move up/down/left/right, pick up a tile, and drop a tile in a hole. Each action takes one time step of Tileworld. In the end of a run, performance of the agent is rated based on how many holes it filled over run of the agent given how many tiles were generated at the particular run.

There are several limitations. Agent deployed in Tileworld can carry up to three tiles at a time. Besides that, an agent is also limited by virtual fuel. One unit of fuel is consumed with each movement. Once the agent runs out of fuel it cannot move anymore. However, at any point, the agent can visit gas station located at [0,0] where it can refill back to the maximum level. That adds seventh action available to an agent: refueling. Additionally, an agent is also limited by its sensor range so that it can see only its neighbourhood.

Therefore, we can describe Tileworld as discrete, dynamic, episodic, deterministic, and not fully accessible environment.

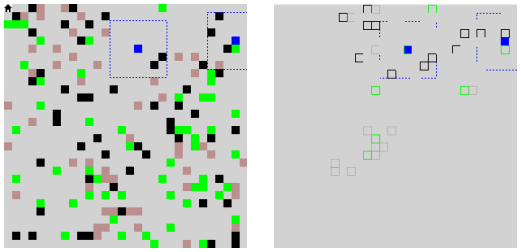
Tileworld is very flexible [2]. There are several versions of Tileworld with distinct features as each author alters the rules in order to test certain problems: for example, the value as well as the size of a hole may vary, holes and tiles may have different shape,

or agents may push tiles instead of picking them up and dropping them into holes.

B. Technical background

We were provided with a Tileworld system based on MASON. MASON is an open-source, multiagent, multi-domain, discrete-event simulation toolkit written in Java [3]. It was designed to facilitate large-scale simulations and it provides tools for visualization in 2D and even in 3D. Agents are regular Java objects extending TWEntity.

Figure I.1 shows the visualization of Tileworld in MASON. There, tiles are green, holes are purplish, obstacles are black, agents are blue, and gas station looks like a house. Dashed lines around agents signify the range of their sensors.



(a) (b)

Figure I.1: (a) Visualization of Tileworld and (b) Visualization of memory of a single agent. Tiles are green, holes are brownish, obstacles are black, agents are blue, refueling station is a house. Dashed lines around agents signify the range of their sensors.

C. Assignment

The aim of the exercise was to use the provided system to implement two agents which would cooperate when deployed in the environment in order to achieve highest score possible in 5,000 steps. Agents are initially deployed at coordinates [0,0] and [1,1]. Further, maximum level of fuel was 1,000 units and the reach of sensor was only three cells; i.e. the agent could access to 7×7 neighbourhood. Finally, the agents could exchange only 3 pieces of information per each step.

There are three reference configurations of the environment for testing, see Section Table I. In the table, $\mu_{objects}$, σ^2_{object} refer to the mean and variance of the Gaussian distribution describing number of objects created per each step.

	Env. 1	Env. 2	Env. 3
Size	100×100	50×50	150×40
μ_{tile}	0.2	2	0.02
σ^2_{tile}	0.05	0.5	0.001
μ_{hole}	0.2	2	0.2
σ^2_{hole}	0.05	0.5	0.01
$\mu_{obstacle}$	0.2	2	0.5
$\sigma^2_{obstacle}$	0.05	0.5	0.1
Lifetime	100	30	120

Table I: Configuration of three predefined environments for testing.

II. Agent design

A. Introduction

From the lectures, we know that an intelligent agent should be reactive, pro-active, and social. That is, the agents should be capable of responding to the changes of Tileworld such as appearing and disappearing objects. They should be able to mover around the environment, pick up tiles and and drop them. Secondly, the agents should take initiatives; i.e. explore Tileworld in a clever way. Finally, the agents must communicate in order to align their behaviour and get better global information about the world. This briefly outlines problems needed to be solved to achieve good performance.

There were two basic approaches in our group for the design of an agent. First, probabilistic approach tries to maximize expected utility based on a probabilistic model of the world and uses altered path planning in order to maximize the explored area. Second approach applies genetic programming (GP) to find the optimal strategy for exploring the world.

My report focuses on description of the GP approach. Please refer to our group presentation and other reports for more information about the first approach. Both architectures are compared in experiments; see Section III-E. Motivation to additionally implement GP to solve the problem was partly selfish: there are only seldom opportunities to apply and learn this intriguing methods in practice.

1) Genetic programming and grammatical evolution: Given the simple structure of Tileworld it would be attractive to apply the simple reactive approach for the design of the agents. While this might be sufficient in some cases, there are many variables

and parameters and when combined with all possible actions it becomes intractable for a human to try optimizing the program. Moreover, while human intuition is often useful, it might be rather a burden in such case. Therefore, it becomes appealing to let the agents learn themselves the best strategy for exploring the world of tiles and learn how to cooperate. There is a number of methods used in multiagent systems for learning, for example neural networks, learning decision trees, classifiers, and genetic-based methods [4]. We utilized grammatical evolution (GE), smaller sister of more popular GP, as it is designed to develop algorithmic behaviour suitable for the problem [5].

GP became well-known tool of artificial intelligence. It belongs among evolutionary methods inspired by biological evolution. This direction in AI was mainly popularized by John Holland in early 1970s [6]. GP is one of the directions later introduced by Koza [7]. It automatically creates working computer programs only from given high-level program statement. Each program is represented by a tree and is assigned fitness according to its performance. In the first step, initial population of programs is randomly generated. Iteratively, to create offsprings we mix programs of each population together according to their fitness using *cross-over* operation while maintaining the elite candidates. Offsprings then become the next population. Further, operation like *mutation* are applied in order to avoid entrapment in local minima.

There is a number of nuances of GP which were introduced over time. It was shown that GP can perform actually better than a human in number of applications [8].

GE was presented as a more general alternative to GP [9]. While the intentions and principle behind GE and GP are similar, GE can evolve complete programs in an arbitrary language. It evolves programs according to rules of grammar defined by Backus-Naur form (BNF).

More recent works suggest that genetic network programming (GNP) should be more viable for the dynamic problems like Tileworld [10]. GNP represents program as a graph and thus it can create variety of loops functioniong like a memory over time similar to automaton. In [10] GNP performs better than GP. The main advantage is notably simpler representation of equivalent program.

B. General structure and the control loop

Agent performs its actions in classical feedback control loop consisting of four functions: *see*, *think*, *act*, and *communicate*. Function *see* fetches all objects in the neighbourhood and saves them into the memory. One of the problems here is the *memory* management as we must discard too old, irrelevant objects. That is discussed in Section II-D. Function *think* decides the next step of the agent and *act* then executes the outputted action. Finally, using function *communicate,* agent shares the most important results with its peers using Blackboard architecture. Communication is described in Section II-C.

Function *think* is the core of the decision-making. Here we decide the action taken based on the observation of Tileworld. Rough outline of the procedure in pseudo code is in Algorithm 1.

Algorithm 1 Pseudo code of procedure *think*

```
function TWAction think(){
    if(numberOfTiles()>0 AND
      isThisCellHole())  return PUTDOWN;
    else if(numberOfCarriedTiles()<3 AND
      isThisCellTile()) return PICKUP;
    else if(isThisFuelStation() AND state
      ==refueling) {
        state=exploring;
        return REFUEL;
    }
    else if(needToRefuel()) {
        state=refueling;
    }

    if(state==exploring) {
        BD= mostUnexploredSector();
        vector = decideDirection();
    }
    else {
        if(tileOnWay() AND
          numberOfCarriedTiles()<3) vector
          =tilePosition-currentPosition;
        else if(holeOnWay() AND
          numberOfCarriedTiles()>0) vector
          =holePosition-currentPosition;
        else vector=vectorInverse(
          currentPosition);
    }

    return (MOVE in getDirectionFromVector
      (vector));
}
```

Agent performs essential actions as an instinct without thinking. This is similar to a human; for example, one doesn't cautiously control breathing. Similarly to that, an agent picks up a tile or fills a hole when standing above one. These actions have priority.

There is virtually no situation when this wouldn't be the optimal action.

The agent has only two internal states: *exploring* and *refuelling*. Refuelling is initiated under specific conditions; the procedure is discussed in Section II-C3. When not refuelling, the agent is exploring Tileworld. The core function of the exploration is *decideDirection* which outputs a vector. The vector determines the direction in which the agent should move and is learned using GE described in Section II-E1.

C. Cooperation

It is questionable whether applying advanced techniques of cooperation like task sharing, forming coalitions, or voting to take decisions can be really beneficial in our case. The reason is that firstly there are only two agents and secondly the act of filling a hole with a tile is a simple task which agent can finish without forming a coalition with the other agent. Yet, such statement is not true in the version of Tileworld where agents push tiles [11]. There, agents can cooperate even on the level of moving one tile.

Nonetheless, agents cooperate on the global level. Firstly, in order to achieve maximum score, agents must cohere; i.e. align their behaviour. This basically means that they shouldn't waste their time exploring parts of the world other agent already explored. Secondly, they should update each other with relevant information to obtain more precise view of Tileworld.

1) Aligning behaviour: In the solution, there is no specific mechanism applied to achieve the coherence. That is, cooperation is not enforced externally to the *decideDirection* function. In fact, it was main lure to implement GP to observe whether such cooperation will emerge naturally. Therefore, one of the inputs into GE is the vector from the agent to the other agent.

2) Communication: Further, agents can use communication in order to get more precise view of Tileworld. Agents can ask other agents to send back location of tiles and holes on the other side. However, this becomes important only in certain situations, particularly when Tileworld is dense like in the second configuration, such communication is useless. It might be beneficial when the environment is very sparse. Besides that, the agent sends information about position and state.

We used blackboard communication architecture. Agents don't interact with each other directly but the communication between them is facilitated by a blackboard. Each agent is limited to three pieces of information uploaded to the blackboard per one step. Since agents are otherwise homogeneous, each agent is assigned with *id* to access the information on the blackboard.

Communication on the side of the agent is performed by function *communicate*. It is described in Algorithm 2. When agents determines that a percept is worth sharing it puts it into a queue. It selects important percepts like unpicked-up tiles or unfilled holes. Further, when the second agent asks for an object of certain type, it uses *priorityQueue* instead.

Algorithm 2 Pseudo code of procedure *communicate*

```
function communicate(){
  i = 3;
  if(timeOfSimulation MOD 5==0 AND i>0){
    uploadPosition(myId);
    i--;
  }
  if(changedState() AND i>0) {
    uploadState();
    i--;
  }
  if(numberOfCarriedTiles==3 AND
    dontSeeHole() AND !askedForHole AND i
    >0){
    uploadRequestForHole();
    i--;
  }
  else if(numberOfCarriedTiles==0 AND
    dontSeeTile() AND !askedForTile AND i
    >0){
    uploadRequestForTile();
    i--;
  }

  while(i>0 AND !isEmpty(PriorityQueue)) {
    uploadPercept(pop(PriorityQueue));
    i--;
  }
  while(i>0 AND !isEmpty(Queue)) {
    i--;
    uploadPercept(pop(Queue));
  }
}
```

3) Refuelling: The problem of refuelling is separated into two distinct subproblems. One is deciding when to refuel the agent and the second is the refuelling procedure itself.

Decision to refuel is based on two facts: firstly, the closer an agent is to the refuelling station the cheaper is to go to refuel. Secondly, agents can be efficient

while refuelling and explore the world on the way to the pit stop. Therefore it is not desirable that both agents refuel at the same time. As each agent must go refuel at least 5 times during one run we can sacrifice the surplus 1000 units in order to optimize the organization of refuelling.

Agent commences refuelling under two circumstances:

- It reaches emergency level of fuel (distanceToFuelingStation+20)
- No other agent is refuelling and fuel reaches trigger level defined by function f of Manhattan distance to the gas station $dist$:

$$f(dist) = -\frac{400}{x_{max} + y_{max}} dist + 400,$$

where x_{max} and y_{max} are dimensions of Tileworld. The procedure of refuelling is straightforward. We set the direction to the gas station and allow the agent to pick up/drop tiles only on the way; i.e. with angle in range $\langle \frac{\pi}{2}, \pi \rangle$ relative to the agent. This is already obvious from function *think* in Algorithm 1.

D. Memory management

Each agent is equiped with memory. In each step, all detected objects are simply pushed into the memory with time stamp of detection. In Tileworld objects disappear after certain time given by parameter *lifeTime*. The main problem here is estimate when it is optimal to discard an object from the memory. Even if we knew the *lifeTime* precisely, it wouldn't be enough to know that object disappeared without seeing the object disappear.

Therefore, we try to guess *lifeTime* only roughly from first object observed to disappear. There is high probability that such object appeared in the early stage of the run. Time of simulation when the object disappears gives us biased estimate of the *lifeTime* higher than the real *lifeTime*. This is then used to forget objects. The estimate is improved over time; when agent keep discovering that objects are not where they were assumed to be according to memory *lifeTime* estimate is decreased.

Additionally, I added feature of sectors in the memory. The grid is divided into sectors with size 10×10 cells. When an agent visits a sector we decrease the value of the sector by constant; by testing I determined 300 units is a reasonable value. Visit of a sector by other agent decreases the value twice. In each step the value of each sector increases back up by one unit until it reaches back the original maximum value. This way, we can easily retrieve the most unexplored region of grid from memory of the agent.

E. Genetic programming for Tileworld

1) Exploration of Tileworld: Our work is partly inspired by work of Luke [5] who successfully applied GP to evolve team of soccer bots with ultimate application in RoboCup Soccer. While the domains are different, there are many similarities. Tileworld is obviously simpler as soccer is continuous environment with larger set of actions. Also number of agents is only 2 instead of 11 in soccer. Further, [11] proposed similar approach to the one taken here as well.

The evolved program performs vector and scalar operations with input parameters and it outputs a vector. The vector is then converted into direction using simple function *getDirectionFromVector*. Basically, when vector points to the right the agent will go to the right, when vector points to the left the agent will try to go left. If there is an obstacle, alternative direction is taken.

Obviously, this is not optimal as the agent can get stuck on clustered obstacles. This wasn't a problem for the given environments but it might be a problem if the environment was denser. Then, it would be beneficial to apply A* search; i.e. we would use A* to determine the shortest path between the agent and the position where the vector points to taking the first step of the path. However, A* search is very expensive for applying when using GP and it is, in fact, not necessary as I show in experiments (Section III-D). Therefore, this remained an opened option.

I devised set of 12 vector and scalar operations. They are all listed in Table II. Besides that, there is around 22 different input parameters retrieved from memory or from the blackboard; see Table III. The operations and parameters can be nested according to the rules of grammar defined using BNF:

```
1 <opVector> ::= biggerThan(<opScalar>,<
    opScalar>,<opVector>,<opVector>) |
    vectorAdd(<opVector>,<opVector>)|
    scalarProduct(<opScalar>,<opVector>) |
    scalarProduct(-1,<opVector>) | rotate90
    (<opVector>) |  zeroVector(<opVector>,<
    opVector>,<opVector>) | <vector>
```

```
<opScalar> ::= norm(<opVector>) |
  dotProduct(<opVector>,<opVector>) |
  scalarAdd(<opScalar>,<opScalar>)|
  scalarTimes(<opScalar>,<opScalar>) | <
  scalar>
<vector>   ::= T1 | H1 | A1 | A2 | BD |
  previousDirection | randomVector()
<scalar>   ::= 0 | 1 | -1 | 2 | 1/2 | nOfT
   | dA2 | A2r | randomScalar() | paramT1
  [<i>] | paramH1[<i>]
<i>        ::= 0 | 1 | 2
```

Both agents deployed into Tileworld are homogeneous. While [5] reported improvement with heterogeneous population, our problem differs as as there is no cooperation on the basic level. This was discussed in Section II-C.

2) Technical implementation of GP: For implementation of GP I used framework EpochX [12]. It allows breeding of program populations according to defined grammar and it also implements all genetic operations necessary.

We must be able to evaluate the performance of an agent with the generated program in string in Tileworld. There were two options: using an interpreter or compiling separate class. While using interpreter was a straightforward solution, it was very inefficient. Since there are two agents, the program is ran 10,000 times only in one initiation of Tileworld. Considering we have 500 programs in a population and 200 generations, the time is crucial. Additionally, I make average over two runs since there are stochastic operations. Thus finally the second, more complicated solution, was the only plausible. With each initiation of Tileword new class is compiled based on the generated program. That significantly improved speed and made the application feasible.

III. Results and Discussion

A. Agent breeding

After several experiments I heuristically chose parameters of GP: population size was limited to 500 while the evolution stopped after 100 generations. Further, only operations *crossover* and *mutation* were applied with probability 0.7 and 0.3 respectively. Since the programs were stochastic, I also allowed 5 elite programs to be preserved to the next generation. Also, I limited initial depth of the tree to 14 since deeper program than 14 often couldn't be compiled because of its length. It is caused by Java as it limits size of bytecode of a single method to 64kB.

Generally, the fitness rocketed up in few first generation and then was trapped in a local maximum. Even high probability of mutation didn't really help. The problem at this point was that the breed of programs converged towards a simpler solution while, in my opinion, the optimal solution lay in more structured and deeper program. The final elite programs had usually depth 5-8. Notably, the elite solution for the third environment was extremely simple. The program directed the agent towards the first tile it saw or the first tile it received information about. Yet, the performance was remarkable. The agents were able to locate and place more than 20% of tiles on average; see Table IV. Despite all that, the solutions were generally better than the simple random exploration even for the over-fitted programs applied on other configurations.

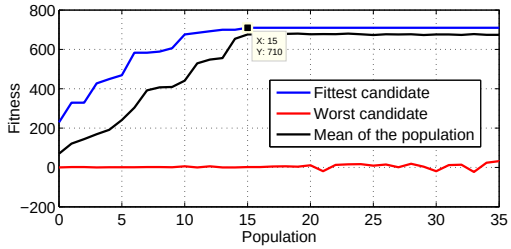


Figure III.1: Evolution in second environment. The fitness rockets up and then levels off at local maximum.

B. Experiments

Each program mentioned here in results was ran 100 times in order to obtain mean and variance. Further, variables of the configuration were controlled in order to demonstrate certain features of the environments and test the sensitivity to the changes.

C. Elite agents for separate environments

At first, I trained the agents using each environment configuration separately; please refer to Table I for the configuration parameters. Table IV compares the performance of the final agents from each configuration. The results were rather over-fitted programs as none of them didn't really excel in all three environments.

Function	# of parameters	returns	Description
biggerThan(s1,s2,v1,v2)	4	v	if s1>s2 return v1, otherwise return v2
vectorAdd(v1,v2)	2	v	adds two vectors
scalarProduct(s1,v1)	2	v	performs product of vector with scalar
inverse(v1)	1	v	returns inverse of the vector v1
rotate90(v1)	1	v	returns vector perpendicular to v1
zeroVector(v1,v2,v3)	3	v	if v1 is [0;0], returns v2, otherwise v3
randomVector()	0	v	returns normalized random vector
norm(v1)	1	s	returns l^2-norm of the vector
dotProduct(v1,v2)	2	s	performs dot product between v1 and v2
scalarAdd(s1,s2)	2	s	returns s1+s2
scalarTimes(s1,s2)	2	s	returns s1*s2
randomScalar()	0	s	returns random scalar on range $\langle 0; 5\rangle$

Table II: Functions of GP

Terminals	Type	Description
T1,T2	v	first/second nearest tile
H1,H2	v	first/second nearest hole
prevDirection	v	previous output vector
A1	v	position of the agent
A2	v	vector to the other agent
BD	v	vector to unexplored sector
numberOfTiles	s	number of carried tiles of the agent
dA2	s	distance to the second agent
A2r	s	second agent refuelling
paramT1[i]	s	number of obstacles/tiles/holes around T1
paramH1[i]	s	number of obstacles/tiles/holes around H1
0,1,-1,2,1/2	s	constants

Table III: Terminals/Input parameters of GP

Env. E	μ_{e1}	σ^2_{e1}	μ_{e2}	σ^2_{e2}	μ_{e3}	σ^2_{e3}
Env. 1	**89**	**0.93**	670	35.70	14	1.05
Env. 2	59	6.93	**726**	**61.23**	19	0.36
Env. 3	17	0.45	109	87.60	**20**	**1.84**

Table IV: Scores of final agents optimized in respective environment configuration E. μ_e and σ^2_e refer to mean and variance of their average score achieved in configuration e.

D. Candidate agent and discussion

The ultimate goal was to find an agent which would perform well universally in as many configurations as possible. I attempted to redesign fitness function so that it could be based on all three environments. This actually didn't resulted in better agents than those described in the previous section. Therefore, having choice from three programs, the elite pair of agents bred in the second environment achieved most universal results and thus it is my final choice.

I further performed "sensitivity" analysis, i.e. I examined how the performance changes when parameters are slightly changed for the second environment. Figure III.2 shows the change of score when lifetime of objects is controlled. The score rises until the lifetime reaches 60. Then obstacles are so dense that the agents start to have troubles to reach the fuel

point. I predicted this when designing the agents. The solution would be implementation of any search method, for example A* which guarantees optimal path.

Additionally, Figure III.3 shows similar trend for changing of the density for each object separately and for changing the total density. Interestingly, the score at the agents at first decreases. That might be caused by the internal structure of the program. Naturally, when density of holes and tiles increases with agent achieves better score. This effect levels off more quickly when only number of either holes or tiles is constant. The pleasant result is that increasing obstacle density doesn't have significant influence on the performance. Thus simplified vector approach taken in this work seem to be sufficient.

E. Comparison

The advantage of having two separate architectures is that we can compare between the solutions and verify whether the direction was correct. Table V compares the probabilistic agent introduced in the beginning with the agent described in this report. The agent optimized using GP clearly outperforms the probabilistic agent in the configuration two and three. Further, in configuration one the agent gives more consistent results than the probabilistic agent. Moreover, if the probabilistic agent was compared against the agent bred in configuration one it would get significant lead while it would be comparable in the other environments.

	μ_{e1}	σ^2_{e1}	μ_{e2}	σ^2_{e2}	μ_{e3}	σ^2_{e3}
Probability	**70.86**	**26.60**	698	159.00	16	11.93
GP	59	6.93	**726**	**61.23**	**19**	**0.36**

Table V: Comparison of scores between my candidate agent and probability based agent .

IV. Conclusion

To sum up the foregoing, in the project we used two methods to design agents for exploring Tileworld. One is based on probabilistic approach and the second used GP to optimize agents in the sense of natural selection. This report mainly focused on the latter. Agents running the elite code were bred in the second environment where it performed very well. Unfortunately, we weren't able to avoid the

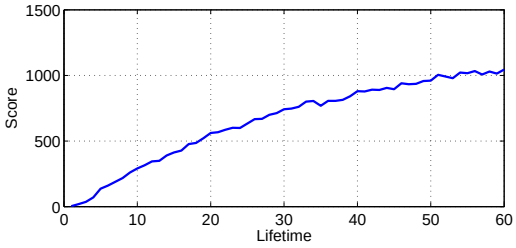


Figure III.2: Candidate program in the second environment when lifetime is controlled.

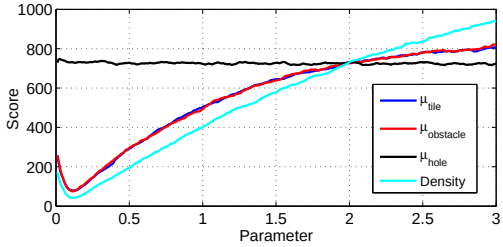


Figure III.3: Score of the candidate program in the second environment when μ_{tile}, $\mu_{obstacle}$, μ_{hole} and total density (i.e. all parameters together) are adjusted on interval $\langle 0; 3\rangle$ (x-axis).

eternal problem of optimization, trapping in the local maxima. Programs converged to local maxima despite high degree of "annealing" from mutations applied. These solutions may have been oversimplified and I believe that better solution could have been achieved. Quality of evolved solution is bound to the quality of input functions and parameters. That determines the nature of the program space and subsequently the achievable results. Most likely, there is more appropriate approach than using simple vectors to get better results. As mentioned earlier, more recent literature proposes Genetic Network Programming as a superior tool for dynamic applications like Tileworld.

Besides that, the quality of "side" functions such as refueling, communication, and memory management turned out to have crucial influence on the performance. For example, when I disabled exchanging percepts, the performance of the agent three halved. Also the added functionality of unexplored sectors was often used in the generated programs. When one observes the behaviour of agent 2 in visualization, it is clearly used whenever there is no tile or hole to follow.

Finally, two homogeneous agents deployed in environment 1, 2, and 3 were able to fill on average 59,

726, and 19 holes with tiles.

REFERENCES

[1] M. E. Pollack and M. Ringuette, *Introducing the Tileworld: Experimentally evaluating agent architectures*. Defense Technical Information Center, 1990. [Online]. Available: http://www.aaai.org/Papers/AAAI/1990/AAAI90-028.pdf 1

[2] M. Lees, "A history of the tileworld agent testbed," *School of Computer Science and Information Technology, University of Nottingham, Nottingham*, 2002. [Online]. Available: http://www.cs.nott.ac.uk/WP/2002/2002-1.pdf 1

[3] S. Luke, C. Cioffi-Revilla, L. Panait, and K. Sullivan, "Mason: A new multi-agent simulation toolkit," in *Proceedings of the 2004 SwarmFest Workshop*, vol. 8, 2004. [Online]. Available: http://cobweb.cs.uga.edu/~maria/pads/papers/mason-SwarmFest04.pdf 2

[4] D. L. Poole and A. K. Mackworth, *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010. [Online]. Available: http://www.google.com/books?hl=en&lr=&id=eALhh_tkpv4C&oi=fnd&pg=PR7&dq=poole+artificial+intelligence+foundations+of+&ots=NrjVegZbh2&sig=HV93x-lgAPJeRrOOcY-5I57HvYY 3

[5] S. Luke, C. Hohn, J. Farris, G. Jackson, and J. Hendler, "Co-evolving soccer softbot team coordination with genetic programming," in *RoboCup-97: Robot soccer world cup I*. Springer, 1998, p. 398–411. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-64473-3_76 3, 5, 6

[6] J. H. Holland, *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. Oxford, England: U Michigan Press, 1975, vol. viii. 3

[7] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1. [Online]. Available: http://www.google.com/books?hl=en&lr=&id=Bhtxo60BV0EC&oi=fnd&pg=PP17&dq=genetic+programming+koza&ots=9ojUkqj1LU&sig=qxGAqJYiWmQNEUw0CLtZp7Cd-Ks 3

[8] ——, "Human-competitive results produced by genetic programming," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 251–284, Sep. 2010. [Online]. Available: http://link.springer.com/article/10.1007/s10710-010-9112-3 3

[9] C. Ryan, J. J. Collins, and M. O. Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *Genetic Programming*. Springer, 1998, p. 83–96. [Online]. Available: http://link.springer.com/chapter/10.1007/BFb0055930 3

[10] B. Li, S. Mabu, and K. Hirasawa, "Tile-world #x2014; a case study of genetic network programming with automatic program generation," in *2010 IEEE International Conference on Systems Man and Cybernetics (SMC)*, Oct. 2010, pp. 2708–2715. 3

[11] H. Iba, "Emergent cooperation for multiple agents using genetic programming," in *Parallel Problem Solving from Nature — PPSN IV*, ser. Lecture Notes in Computer Science, H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, Eds. Springer Berlin Heidelberg, Jan. 1996, no. 1141, pp. 32–41. [Online]. Available: http://link.springer.com.ezlibproxy1.ntu.edu.sg/chapter/10.1007/3-540-61723-X_967 4, 5

[12] F. Otero, T. Castle, and C. Johnson, "Epochx: Genetic programming in java with statistics and event monitoring," in *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*. ACM, 2012, p. 93–100. [Online]. Available: http://dl.acm.org/citation.cfm?id=2330800 6