2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)

On the Effectiveness of Emergent Task Allocation of Virtual Programmer Teams

Oskar Jarczyk*, Błażej Gruszka†, Leszek Bukowski‡ and Adam Wierzbicki§
Polish-Japanese Institute of Information Technology
Department of Social Informatics, Warsaw 02-008, Poland
*e-mail: oskar.jarczyk@pjwstk.edu.pl
†e-mail: blazej.gruszka@pjwstk.edu.pl
‡e-mail: bqpro@pjwstk.edu.pl
§e-mail: adamw@pjwstk.edu.pl

***Abstract*—*Virtual teams* of programmers are a popular form of collaboration both in Open Source, and commercial software projects. In *OSS (Open Source Software)* projects, programmers make their own decision which project to join, and, therefore, the process of task allocation among the project members is emergent. In this paper, we attempt to simulate such a process based on available data from *GitHub*. The simulation is used to test a hypothesis regarding the efficiency of emergent task allocation. In general, we find performance of coordinated methods to be better than emergent algorithms, which leads us to conclusion that controlling team membership through invitations and work assignment is a promising direction.**

I. Introduction

In recent years we have been witnessing a rise of the *OSSD (Open-Source Software Development)* model, in which developers cooperate without any formal authority or institution governing their work. An example of a portal enabling use of an open *OSSD* model is the *GitHub website* —an online social network for developers. Users of *GitHub* might create their own repositories and collaborate with other programmers on a variety of projects. Even such big companies as Twitter or Facebook have their own repositories on *GitHub*.

GitHub has no mechanism for supporting task allocation to users. Each user individually decides on what project to work on and how to allocate her contributions. Despite such a decentralized structure of developers' community there have been plenty of *GitHub* projects which have resulted in very good quality software. Nevertheless, it might be still possible to improve open collaboration on such platforms as *GitHub*. However if we wish to do that, we need to have a better insight into the process of task allocation among a decentralized community of developers working in *OSSD model*. In our paper we describe a simulation that is meant to improve our understanding of dependencies between strategies used by developers and effectiveness of their work.

Such investigations are important, since the number of *OSS* projects is growing and many companies are using solutions that have been developed in such models. In the future we might enhance the process of team formation or task allocation by intelligent task/developer recommendations. If we want to create such recommendation engines, we must know, for example, which strategies are most efficient in case of task allocation among developers —i.e. which of them are the quickest way of task solving.

We focus on the question of the *effectiveness of task allocation*. Are the autonomous behavioral strategies enough for efficient work on software projects? We define strategy efficiency as performing or functioning in the best possible manner with the least waste of *time*. Effectiveness is understood as a proper choice of the task allocation strategy and it may vary inside a group of agents (persons). We hypothesize the following:

Hypothesis 1 *Centralized task allocation strategies are more efficient than emergent task strategies.*

II. Related work

A. Scientific research regarding collaborative networks

In literature there are examples of how communities that are successful structure their work. O'Mahony, Ferraro (2007) conducted a multimethod study of one open source software community. They found that members developed a shared basis of formal authority but limited it with democratic mechanisms.[1] Other researchers (Anagnostopoulos, 2012) considered a setting modelled by a social network in which people have different skills and relationship among them.[2] Authors focused on proposing first online algorithms which assemble teams for the purpose of dealing with tasks. Balanced task assignment problem was introduced with online competitive algorithms. Scholars analysed previous work connected with team formation, i.e.: scheduling with load balancing[3] (scheduling of jobs on a set of machines with the goal of minimizing the maximum load on a machine), matching people to tasks (a matching problem for which several systems have recently been proposed, for instance easychair.org, linklings.com and softconf.com).[2] The problem of providing efficient solutions in the bidding model has been addressed by others Mehlhorn et.al. (2009).[4]

Team formation with coordination costs introduce the problem of team formation in social networks (Lappas et al., 2009).[5] The objective is to minimize the coordination cost, for example, in terms of diameter or weight of the minimum spanning tree for the team. This problem has been extended to cases in which potentially more than one member possessing each skill is required, and where density-based measures are used as objectives.[6][7]

Scholars Jin, Girvan et.al. (2001) in their paper "The structure of growing social networks" proposed two models of growth of social networks which have properties of fixed number of vertices, limited degree, clustering and decay of friendship. Simulation of those models shown emerging societies and existing clustering. Researchers reached a conclusion that complex and reasonable patterns of social networks, and their evolution, can emerge from simple rules. Furthermore, general form of those patterns is not influenced by micro details of the rules.[8] Models regarding social networks are considered by us for near-future research.

B. Works regarding strategies such as homophily, heterophily and preferential attachment

Most of the literature focuses on Internet social networks. Crandall et.al. (2010) convinces that people want to resemble their current friends due to social influence and also they tend to form new links to others who are already like them. McPherson et.al. (2001) stated that similarity breeds connection and the result is that people's personal networks are homogeneous with regard to many important social characteristics.[9] It was proven that prior collaborative ties have a profound effect on developers' project joining decisions.[10] Bisgin et.al. (2012) studied homophily in social media. Rogers et.al. (1995) defined homophily as the degree to which the innovator and the potential adopter are similar in certain attributes such as objectives, beliefs, norms, experience, and culture. Heterophily is the dual of homophily, and, according to Rogers et.al., "one of the most distinctive problems in the diffusion of innovations is that the participants are usually quite heterophilous." Next key understanding to many current collaboration models proposed to describe an evolution of complex networks, is the hypothesis that highly connected nodes increase their connectivity faster than their less connected peers - a phenomenon called preferential attachment.[11] When Wikipedia articles are treated as nodes, similarly to websites in the Internet, *Wikipedia* growth is proven to follow the behaviour of preferential attachment laws.[12]

III. GITHUB DATASET

Firstly, we identified the languages known to *GitHub*. There are 224 languages which *GitHub* can recognize through the *Linguist* library. Next, we analyzed repositories pull requests from the year 2013. Repos are sorted decreasing by an average of the number of forks, watchers, and stars. From the pushes —which are part of the pull requests —we use the 'language' value which is a number of files, inside a commit, identified by *GitHub* to be written in a particular programming language. Therefore, for a single repository, we have a set of languages and a number value which is the language use frequency. Later, we identify the *work left* value as a difference between the frequency and a maximum of the frequency for a particular language in the dataset. The next step is to analyze languages used by most active *GitHub* users, defined by Paul Miller (2013) to be users with biggest number of contributions and at least 252 followers. We take the 1000 top active users and download pushes made by them to analyze the frequency of languages used in the commits, analogically like in the previous point described above. Basing on those 1000 developers we create a pool of 1000 agents in the simulation.

IV. SIMULATION MODEL

Our simulation is meant to give us a better understanding of the following problem: say we do have a set of programming tasks (projects) to solve and there is a set of agents who are developers. In our simulation time is discrete —at each step of time each of the agents uses some strategy to choose on which task to work on at this point of time.

Each agent has some programming skills that is a set of languages at which he or she has some expertise. Each of task in the pool of our simulation also consists of set of skills that are needed to solve this task. Additionally, for any task there is some amount of work that has to be done before we may say that the task is solved. There are several considerations for our simulation model, we assess below:

A. Skills

Skill is a technical competency - a programming or scripting language, eventually a markup or data format. It is identified by its unique name. So we have a set of *skills* $\omega = \{S_1, S_2, S_3, ..., S_n\}$. In case of our simulation $n = 88$.

B. Tasks

We define some task T_i as a triplet: $T_i = \langle \omega_{T_i}, G_{T_i}, W_{T_i} \rangle$ – where ω_{T_i} is a set of *skills* in T_i, G_{T_i} is function that assigns to each element of ω_{T_i} a number of work units which have been completed in this *skill* and W_{T_i} is a function that assigns number of work units that has already been done in each element of ω_{T_i}. For example if we have $T_{abc} = \langle \{java, python\}, \{8,4\}, \{0,2\} \rangle$– it means that in the task named "abc" there are two *skills*, namely Java and Python, and $G_{T_{abc}}(java) = 8$ and $G_{T_{abc}}(python) = 4$, no work has already been done in java, but 2 work units has been done in python, that is $W_{T_{abc}}(python) = 2$.

C. Agents

In our simulation each agent is a triplet $A_i = \langle \omega_{A_i}, E_{A_i}, \xi_{A_i} \rangle$ – where ω_{A_i} is a set of *skills* that agent A_i posses, E_{A_i} is a function that assigns agent's experience (number of work units done) to each element of ω_{A_i} and ξ is a strategy that agent A_i uses for task selection. If, for example, we do have an agent $A_x = \langle \{java, python\}, \{2, 3\}, homophily \rangle$it means that agent A_x has two *skills* – Java and Python – A_x experience in Java is 2 and in Python is 3 and the strategy that agent A_x is using is homophily.

D. Agent's task and skill selection strategies

In our simulation we consider following the strategies: *random*, *preferential*, *heterophily*, *homophily* and *central assignment*. Each of these strategies, except central assignment, might be in one of the following methods: *most advanced*, *proportional time division*, *greatest experience* or *random*. Generally, agents use strategies to choose on which task to work and then the method is used to allocate agents' work units among *skills* in a chosen task. Now we describe all strategies in detail:

1) Homophily and heterophily: Homophily is an algorithm for assigning tasks which are best adjusted to an agent, while choice of most different *task* from the *agent's skills* is called *heterophily*. In other words, both of them use an algorithm for assigning tasks which are most (mis)matched to an agent. We sort the tasks according to the following scheme:

1) If *homophily* - take only those tasks in which the number of common agent's and task's skills is greater than 0. If a group of agent's and task's 'common tasks' is equal to 0, then choose a random task.
   If *heterophily* - consider all tasks.
2) For a group of tasks from step 1, for every task calculate: the number of agent's skills / number of common agent's skills and task's skills.
   a) If *homophily* - Select tasks with the lowest value. (Note that in case when all of the agent's skills are in the chosen task, then number of agent's skills / number of common agent's skills and task's skills are equal 1. If only some of agents skills are in the task then this is above 1.)
   b) If *heterophily* - If an agent has no common skills with the task, the number of common agent's skills and task's skills is divided by 0.1. Select tasks with the highest value.
3) For a group of tasks from step 2: If the number of selected tasks is equal to 1, then choose this task. If there is more than one task, in case of *homophily* select tasks with the fewest number of skills. For the *heterophily*, choose the task with the greatest number of skills.
4) For a group of tasks from step 3: If the number of selected tasks is equal to 1, then choose this task. If there is more than one task, select the task in which the average of the common agent's and task's skills is the highest (in *heterophily* - lowest). If number of selected tasks is 1, then choose this task. Otherwise, choose a random task from the last group.

2) Preferential: Strategy searches for the most advanced task in the simulation. It is calculated by average of work done within all *skills* inside a *task*. Such values is used to sort all tasks by their *general advancement* decreasing. General advancement for a task is calculated by an average, where a single value is an advancement in a *skill* in percentage. Preferential strategy works by letting agent choose a most advanced task that has at least one skill, which is also the agent's skill. Agent's experience in this skill plays here no role.

The heuristics follows the steps explained below:

1) Iterate through all of the agent skills:
   a) For this single skill, get a list of tasks which have the considered skill,
   b) In the list of tasks from step above, choose the most advanced task in general,
   c) If chosen task is more advanced than any previous choice, it becomes the result choice.
2) If the result choice is not null, work on the result task.

Algorithm 1 Homophily algorithm

```
Require: Agents, Tasks {Expected result - Chosen task}
  for $Agent\ agent : allAgents$ do
    $tasksSet = getCommonTasksBySkills($
    $agent, tasks)$
    if $count(tasksSet) = 0$ then
      $return\ random(tasksSet)$
    end if
    $tasksSet = getMaxClosestTasks($
    $agent, tasks)$
    $tasksSet = getTasksByMinNumberOfSkills($
    $agent, tasksSet)$
    if $count(tasksSet) = 1$ then
      $return\ firstTask(tasksSet)$
    end if
    $tasksSet = getTasksByMaxCommonSkillsExp($
    $agent, tasksSet)$
    if $count(tasksSet) = 1$ then
      $return\ firstTask(tasksSet)$
    end if
    $return\ random(tasksSet)$
  end for
```

Algorithm 2 Heterophily algorithm

```
Require: Agents, Tasks {Expected result - Chosen task}
  for $Agent\ agent : allAgents$ do
    $tasksSet = getMinClosestTasks($
    $agent, tasks)$
    $tasksSet = getTasksByMaxNumberOfSkills($
    $agent, tasksSet)$
    if $count(tasksSet) = 1$ then
      $return\ firstTask(tasksSet)$
    end if
    $tasksSet = getTasksByMinCommonSkillsExp($
    $agent, tasksSet)$
    if $count(tasksSet) = 1$ then
      $return\ firstTask(tasksSet)$
    end if
    $return\ random(tasksSet)$
  end for
```

Algorithm 3 Preferential algorithm

```
Require: Agents, Tasks sorted by advancement {Expected
  result - Chosen task}
  for $Agentagent : allAgents$ do
    for $SkillsingleSkill : agent.getSkills()$ do
      $Taskresult = PersistAdvancement$
      $.getMostAdvanced(singleSkill);$
    end for
  end for
```

3) Random: Agents are choosing completely randomly a task from the pool of unfinished tasks in simulation.

4) Strategy of central assignment: We also call it a 'strategy of central assignment', or a 'central planner' or 'central task allocation' strategy. Algorithm performs a multiple sorting operation. Simplifying the idea, it works by choosing a *task* with the least advanced *skill* inside and assign it to an agent which have highest experience in it. Before launching an iteration of the simulation, every agent has a *task* assigned to him or her (unless the number of tasks is smaller than the number of agents, then we can simply say that every task have an agent assigned to it).

Behaviour of the central planner in a natural language follows steps explained below:

1) For i number of times, which is the count of agents, or if there are fewer tasks left than agents - count of the tasks, do the following:
   a) chose a task which has a skill with the biggest amount of work left,
   b) for the task from above, choose an agent which has the highest experience in the skill considered above,
   c) mark the task as taken, and the agent as busy.

E. Agent workflow

Steps of the simulation per one step at a time (for every A_j do following) are described in points below:

1) Agent A_j uses $A_j\{\xi\}$ strategy to choose a task T_i to work on in the current simulation tick
2) If $A_j\{\xi\}$ strategy returns no task, Agent A_j chooses a random task T_i
3) Agent A_j works on T_i
4) Use the strategy $A_j\{\xi\}$ method to chose a set of skills ω_x to work on.
   a) If strategy $A_j\{\xi\}$ have a *proportional time division* method - for every skill $\{S\}$ in ω_x, increment work done by $\frac{1}{n}$, where n is the cardinality of ω_x, and gain an experience
   b) If strategy $A_j\{\xi\}$ have one of: *most advanced*, *greatest experience*, or *random* methods - in a single skill $\{S\}$ increment work done by 1 and gain an experience
5) Tasks done leave the environment

F. Learning Process and Work Efficiency

Human nature is to learn new things through experience. Psychologists proved that a learning process can be described by a mathematical *Sigmoid function* (appendix figure no. 5Sigmoid functionfigure.caption.6). In simulation we propose a simplified *Sigmoid learning function* which makes for approximating the process of learning in humans. Different *internal task work distribution methods* (described in section no. IV-G) can significantly influence process efficiency of gaining experience by agents. Gaining experience and using it is indispensable for a person to resolve tasks faster and possibly move on to a new programming language.

That is the reason for the $\delta(S)$ function to be a sigmoid function (shaped *S-curve*) reflecting human learning progress calculated by an equation, and, finally, the function which accepts E (experience understood as amount of work done in the particular technology) as an argument is shown in figure no. 1.

Def. 1: Experience function equation

$$\delta(S_i^j) = \frac{1}{1 + e^{-E_i^j}}$$

.

1) Experience decay: lets consider the importance of experience decay due to inactivity. We are adding an aging effect of the user skills, which is decreasing the experience in a particular skill S_j by the value of β, in every iteration in which user does not work on the mentioned skill. Decrease value $\beta < 1$ is set to $5 \cdot 10^{-4}$, which means every single decrease moves experience 0.05% backwards on a sigmoid curve. If experience in a particular skill of an agent reaches *stupidity level* of 3%, experience won't be decreased anymore during the current tick.

2) Fully taught agents leave: as in previous chapters, an agent is motivated by becoming an expert in his / her fields of knowledge, by gaining experience through practical work. We speculate that agents, who are fluent in all of their skills, will avoid working anymore - by letting the tasks to be taken by somebody else and leaving the collaborative environment.

The behaviour can be described by two steps. Firstly, after every tick of the simulation, iterate through all agents and check if there are skills possessed by agents, which are on the top of the learning curve (have a value 1.0). If the agent does not have any skills with experience lesser than 1.0 (in other words, all his skills are fully taught), than remove this agent from the simulation.

Algorithm 4 'Fully taught agents leave' behaviour

```
Require: Agent {execute per step, last priority}
  for Agent agentInPool do
    for Skill singleSkill in allAgentSkills do
      if singleSkill.experience $\langle$ 1.0 then
        dontRemove = true;
      end if
    end for
    if !dontRemove; then
      simulation.removeAgentFromPool(agentInPool);
    end if
  end for
```

3) Mixing different behaviour in the simulation: behaviour of: *experience decay*, *fully taught agents leave*, and *granularity* can occur simultaneously in the simulation. Except the *central planning* (centrally-controlled strategy), where the central planner does not allow for granularity occurring in the simulation and this parameter must be turned off.

G. Internal Task Work Distribution Methods

Each of the agent strategies described in the section no. IV-C might be in one of the following methods:

1) Most advanced,
2) Proportional time division,
3) Greatest experience,
4) Random.

Those methods (also called modules) coexist with task choice strategies and they are used to choose a $skill$ inside the chosen $task$. We can additionally characterize the pair of task strategy and module as by their ability to add a new skill to agent's set of skills and opposite.

1) Most advanced method: this module selects a single most done skill inside a task. If there is no such skill (i.e. all skills have 'work done' at zero point), then it select randomly a single skill.

2) Proportional time division method: proportional module allows to work equally on multiple skills by diving time into particles, which makes for incrementing work done and experience by fractions of 1.

3) Greatest experience method: method selects only one skill in which an agent is most experienced. If the agent is equally well-experienced in more than one skill, then selects randomly one skill from this set.

4) Random method: this module selects one single random skill to work on. Firstly, the strategy evaluates the intersection of agents skills and tasks required skills. If this intersection is empty, the strategy continues to choose between any random skill left.

In case of most advanced, proportional or greatest experience method the following rule holds: check for intersection between task skills and agent skills, and use it to point skills on which the agent can work on. If the intersection is empty, it means agent does not have common skills with task, and will choose one or more task skills to create experience from scratch. In most advanced method, if the intersection is empty, the agent will choose a single random skill.

H. Granularity of agent behaviour within work choice

In real life, programmers tend to stick with one task for longer than one bit of time, therefore we propose a behaviour of granulating task choice for longer than 1 step of simulation.

- *The quit event*—the quit event happens when the agent decides to move on to an other task.
- Granularity of *task* only—the agent condenses his or her work within one particular *task*, but has a choice to work on different *skills*. In other words, while the agent sticks within Ti task, for every step of simulation skill choice strategy apply. Granularity rules apply until the quit event happens.
- Granularity of *task* and *skill*—the agent condenses his / her work within one particular *task* and *skill* inside this *task*. The agent won't change the task and skill he / she works on until the quit event happens.
- *The quit momentum*—a decision made by the agent to quit working on a particular task and/or skill and reconsider choosing new work from the pool of available tasks. The quit momentum happens with a probability of $P(A) = 20\%$.

V. Mixing strategies for finding the optimum

We implemented a way of finding an optimum, starting from an initial naive mix of strategies. We tried to identify an approximate strategy distribution which we can use in a hypothetical optimal model. It should be "close" (in the divergence-sense) to good strategies and "far away" from bad strategies.[13]

Before enabling evolution, strategies were assigned to agents static, but now we allow agents to change strategy. We introduce agent utility function (def no. 2) which will determine the evolution process. Table no. I shows our evolution plans (scenarios) and they differ only by beginning strategy allocation.

TABLE I: Initial mix of task-choice strategies

Plan	Homphily	Heterophily	Preferential
S1	80%	20%	0%
S2	80%	0%	20%
S3	20%	80%	0%
S4	0%	80%	20%
S5	20%	0%	80%
S6	0%	20%	80%
S7	33,(3)%	33,(3)%	33,(3)%

After every 10 ticks of simulation every agent get a new task-choice strategy according to the implemented well-known genetic algorithm called *Stochastic Universal Sampling (SUS)* method. *SUS* is a fitness proportionate selection method which uses a single random value to sample all solutions by choosing them at evenly spaced intervals. This gives weaker members of the population (according to the fitness function) a chance to be chosen.

Def. 2: Agent utility function

$$utility = min\{\delta_j^i\} + \epsilon \sum_j \delta_j^i$$

$$\delta_j^i - agent\ experience\ in\ specific\ skill$$

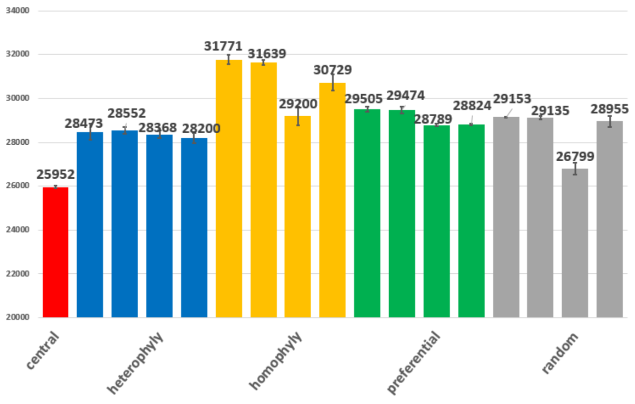
$$\epsilon = \frac{1}{20}$$

Utility function considers lowest experience in a single skill plus $\frac{1}{20}$ part of an average experience in all the skills. In next chapter we won't present detailed results for evolution (including convergence) because boundaries are dynamic and it is impossible to find an equilibrium for this evolutionary game - mostly because of a limited set of tasks in simulation. We want to continue research on this problem by enabling unlimited flow of tasks, exactly as they appeared on GitHub.

VI. Analysis of Results

We constructed a pivot table to group the result set, depending on the parameters of the simulation, which we use as filters (sets of *scenarios*, behavioral features - *experience decay*, *etc*). For purpose of this chapter, we call 'behavioral features' parameters: *experience decay*, *granularity*, and *fully* *taught agents leave*. Their impact on results we analyse in next subchapters. Despite the fact, that the data can be aggregated into one general performance benchmark, (as seen on figure no. 3), it won't allow to answer research questions directly without ungrouping them into a couple of particular scenarios - the main reason of this are extensive thresholds and various sensitivity of the strategies to different workloads. The lowest value of tick count is somewhere at 2038, while its peak reaches the value of 80837.

Fig. 3: Aggregated benchmarks for *task allocation strategies*



from left: *proportional*, *random*, *most advanced*, *greatest experience*

On the figure no. 3, we show results for the calculated 4 scenarios, *granularity*, *experience decay* and *fully taught agents leave* options are disabled. The exact numbers, with an average for all workloads, are given below in the table no. II.

TABLE II: Result set for different scenarios, all behavioral features turned off

strategy	**method**	$a = 10$	$a = 25$	$a = 50$	$a = 120$	**average**	avg(c)
central	-	72 190	23 270	3 151	5 197	25 952	1.87%
heterophily	proportional	77 193	29 787	2 575	4 336	28 473	2.88%
heterophily	random	77 322	30 073	2 557	4 256	28 552	14.83%
heterophily	most-adv	77 143	30 001	2 111	4 217	28 368	4.44%
heterophily	greatest-exp	77 224	29 397	2 038	4 141	28 200	8.41%
homophily	proportional	80 862	32 690	5 282	8 251	31 771	1.25%
homophily	random	80 837	32 701	4 958	8 061	31 639	1.68%
homophily	most-adv	76 401	29 980	3 796	6 624	29 200	2.49%
homophily	greatest-exp	79 056	31 569	4 534	7 756	30 729	4.11%
preferential	proportional	78 387	30 585	3 681	5 369	29 505	0.89%
preferential	random	78 248	30 818	3 524	5 306	29 474	3.41%
preferential	most-adv	77 506	29 762	2 924	4 965	28 789	1.19%
preferential	greatest-exp	77 502	29 901	2 909	4 986	28 824	1.52%
random	proportional	77 762	30 475	3 471	4 906	29 091	0.94%
random	random	77 719	30 544	3 149	5 127	30 121	9.22%
random	most-adv	72 673	27 458	2 523	4 544	23 958	4.23%
random	greatest-exp	77 161	29 881	2 916	5 864	33 892	4.33%

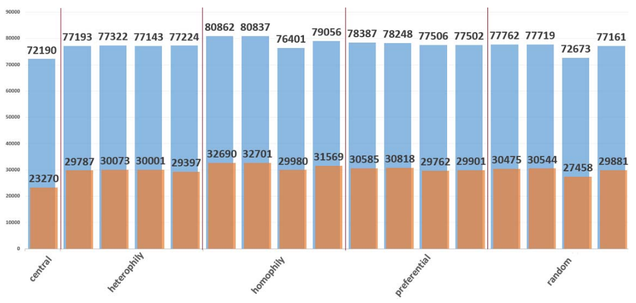
a = {N} means subset of *N-agents*, c = *confidence level*

A. *Effectiveness of Emergent Task Allocation Strategies*

In the scenario with a low number of agents (10 *agents* - 120 *tasks*), the *heterophily strategy* (with average 77221) is more efficient than the *homophyly* (average 79289) (except for *homophyly* with internal *most advanced method*). *Random strategy* with *most advanced method* seems to be the better of the two options considered above. Best strategy here is the *central assignment* - 72190 (results presented below on figure no. 4).

Lets take a look at the opposite scenario - with a big number of agents (120) and a smaller number of tasks (50), which is presented by figure no. 5. *Heterophily strategy* (4238) is better than the *central strategy* (5197), and *homophily strategy* (7673) is the worst. *Preferential* and *random* are somewhere the same and having similar efficiency level as the *central planner*.

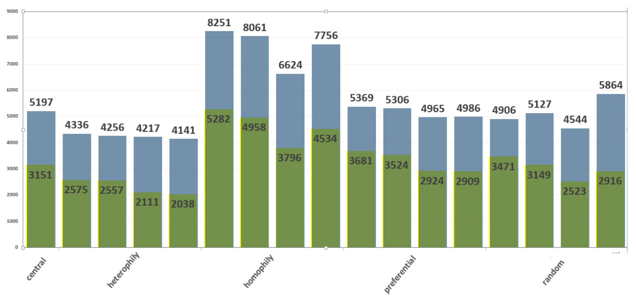
Fig. 4: Benchmarks for sets $(10, 120)$, $(25, 100)$ of agents and tasks



from left: *proportional*, *random*, *most advanced*, *greatest experience*

When it comes to the $(25, 100)$ and $(50, 15)$ scenarios, in the first case results are highly similar to each other, in the second case - benchmarks are comparable to the $(120, 50)$ set. It means for both $(50, 15)$ and $(120, 50)$ workload, so where there is a big number of agents and fewer tasks, *heterophyly* wins over *central planner* and *preferential*, and the *homophyly strategy* is the worst.

Fig. 5: Benchmarks for sets $(50, 15)$, $(120, 50)$ of agents and tasks



from left: *proportional*, *random*, *most advanced*, *greatest experience*

B. *Effectiveness of Centralized Task Allocation Strategy*

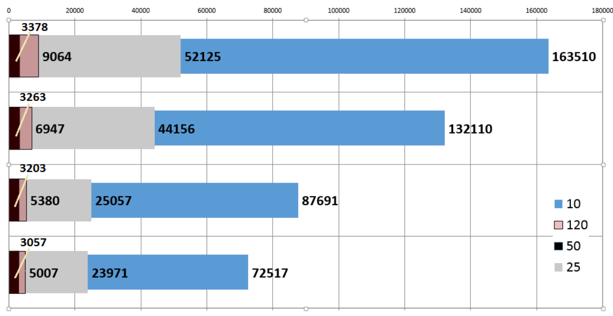
In most cases, *central task allocation* strategy is best in the task resolution. *Central task allocation* strategy can not implement granularity behaviour (as in section IV-F3), neither use an internal task work distribution method. Table no. III shows results through all the scenarios. Central planner has very good benchmarks for small workloads (10 and 25 agents), while in the bigger scenarios it is often outrun by *heterophily*.

TABLE III: Result set for central strategy through all parameters

ed	**ftl**	$a = 10$	**c**	$a = 25$	**c**	**average**
false	**false**	72 517	0.23%	23 971	1.78%	26 138
most efficient?		yes - central		yes - central		yes
false	**true**	132 110	0.15%	44 156	2.08%	46 619
most efficient?		preferential		preferential		no
true	**false**	87 691	1.46%	25 057	2.32%	30 333
most efficient?		yes - central		yes - central		yes
true	**true**	163 510	2.18%	52 125	2.08%	57 019
most efficient?		homophily		heterophily		no
ed	**ftl**	$a = 50$	**c**	$a = 120$	**c**	**average**
false	**false**	3 057	2.47%	5 007	5.79%	26 138
most efficient?		heterophily		heterophily		yes
false	**true**	3 263	2.03%	6 947	0.94%	46 619
most efficient?		yes		preferential		no
true	**false**	3 245	1.50%	5 525	10.32%	30 333
most efficient?		heterophily		yes		yes
true	**true**	3 378	4.04%	9 064	4.10%	57 019
most efficient?		yes		preferential		no

ed = *experience decay*, ftl = *fully taught agents leave*,
c = *confidence level*

Fig. 6: Benchmarks for *central assignment* strategy



from the top (expDec,ftl): (Y, Y), (N, Y), (Y, N), (N, N)

C. *Impact of Internal Task Work Distribution Methods*

In all mentioned task strategies, proportional time division strategy is faster than greedy strategy. Random skill choice strategy is close to the middle between proportional time division and greedy strategies.

D. *Influence of 'task granularity'*

We find *granular behavior* to have a small impact on the simulation results. Results shown on table IV explain the gain in time when granularity is enabled, for the scenario $(120, 50)$, and in most cases - when granularity enabled - the strategies resolve tasks quicker.

TABLE IV: Impact of turning on task granularity

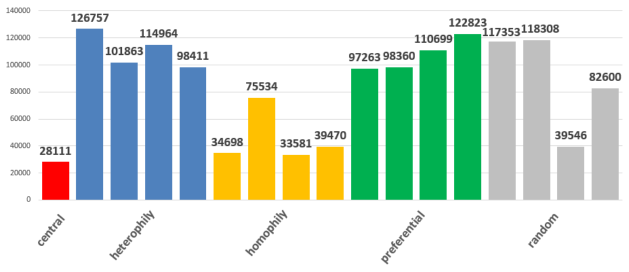
strategy	**granularity**	**granularity**	**delta(δ)**
-	disabled	enabled	-
central	5 197	4 865	333
heterophily			
proportional	4 336	4 177	160
random	4 256	4 007	250
most advanced	4 217	4 020	197
greatest experience	4 141	4 192	-52
homophily			
proportional	8 251	7 012	1 238
random	8 061	7 170	891
most advanced	6 624	5 348	1 277
greatest experience	7 756	6 695	1 062
preferential			
proportional	5 369	5 373	-4
random	5 306	5 349	-43
most advanced	4 965	4 960	5
greatest experience	4 986	4 979	8
random			
proportional	4 906	4 618	288
random	5 127	4 933	195
most advanced	4 544	4 120	424
greatest experience	5 864	5 599	265

δ is a gain in efficiency after enabling granularity

E. *Influence of 'experience decay'*

Intuition suggests that the forgetting process will impact agents who use the *heterophily* strategy. Results proved that for the scenario with the small number of agents, enabling forgetting makes for *homophily* strategy to be 3 times more efficient than the *heterophily* strategy.

Fig. 7: Benchmarks for *exp-decay* on, 10 *agents* and 120 *tasks*



from left: *proportional*, *random*, *most advanced*, *greatest experience*

F. *Influence of 'fully taught agents leave'*

Enabling behavior where fully taught agents leave simulation always worsens results of all of the strategies. Obviously, smaller number of agents in the simulation makes for weaker collective work force. It also decreases performance of *central planner* and in the same time benefits *preferential* strategy.

G. Results of the evolution

TABLE V: Domination of particular task-choice strategies, all behavioral features turned off

Plan	Dominator
S1	Heterophily
S2	Preferential
S3	Heterophily
S4	Preferential
S5	Preferential
S6	Preferential
S7	Preferential

According to the results shown in the table V, the most dominant strategy is *preferential*, which won over other strategies in all of the plans where it was used. Interesting regularity occurred in the plan $S7$ (as seen in appendix graph 13An example of the strategy domination (plan S7)figure.caption.16) where *homophily* dominates more than *heterophily*, and in direct confrontation in plan $S3$ (appendix 12An example of the strategy domination (plan S3)figure.caption.15) *homophily* lost.

VII. Conclusions and Future Work

This paper answered questions regarding best strategies for an *efficient task resolution*, basing on real-life data parsed from the active repositories and users of the *GitHub* portal. We got results which support the view of other researchers [1] that successful communities are governed by non-autonomous decisions. Grouped results shown that **centralized task assignment** strategies are are more efficient than emergent task strategies. Despite the fact there were scenarios which shown hypothesis no.1 to be questionable (VI-B), they don't necessarily reflect a typical workload in *OSSD* model. For grouped results the hypothesis regarding superiority of centralized strategies over emergent strategies is proven to be true. Agents were working on a discrete countable set of tasks - their adaptation to the *collaborative environment* through the personal evolution was limited to the countable number of tasks left in the simulation universe. In the near future we want to simulate an environment of tasks incoming to the simulator through a stream - in a similar way repositories on *GitHub* had an activity through their timeline. We will also introduce more sophisticated behavioral model of a user - implementing i.e. *experience cut-point* (a question - can a person reach the maximum of experience in his field).

Acknowledgment

This work is supported by Polish National Science Centre grant 2012/05/B/ST6/03364

References

[1] Siobhn O'Mahony and Fabrizio Ferraro. The emergence of governance in an open source community. *Academy of Management Journal*, 50(5):1079–1106, 2007.

[2] Aris Anagnostopoulos, Luca Becchetti et.al. Online team formation in social networks. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, New York, NY, USA, 2012. ACM, pages 839–848.

[3] R. L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring), New York, NY, USA, 1972. ACM, pages 205–217.

[4] Kurt Mehlhorn. Assigning papers to referees. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming: Part I*, ICALP '09, Berlin, Heidelberg, 2009. Springer-Verlag, pages 1–2.

[5] Theodoros Lappas, Kun Liu et.al. Finding a team of experts in social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, New York, NY, USA, 2009. ACM, pages 467–476.

[6] Amita Gajewar and Atish Das Sarma. Multi-skill collaborative teams based on densest subgraphs. In *SDM*. SIAM / Omnipress, 2012, pages 165–176

[7] Cheng-Te Li and Man-Kwan Shan. Team formation for generalized tasks in expertise social networks. In *Social Computing (SocialCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pages 9–16.

[8] Emily M Jin, Michelle Girvan et.al. Structure of growing social networks. *Physical review E*, 64(4):046132, 2001.

[9] Miller McPherson, Lynn Smith-Lovin et.al. Birds of a feather: Homophily in social networks. *Annual review of sociology*, 2001, pages 415–444.

[10] Jungpil Hahn, Jae Yun Moon et.al. Emergence of new project teams from open source software developer networks: Impact of prior collaboration ties, 2006.

[11] H. Jeong, Z. Nda, and A. L. Barabsi. Measuring preferential attachment in evolving networks. *EPL (Europhysics Letters)*, 61(4):567, 2003.

[12] A. Capocci, V. D. P. Servedio et.al. Preferential attachment in the growth of social networks: The internet encyclopedia wikipedia. *Phys. Rev. E*, 74:036116, Sep 2006.

[13] Christopher Mattern. Mixing strategies in data compression. In James A. Storer and Michael W. Marcellin, editors, *DCC*. IEEE Computer Society, 2012, pages 337–346.