Eric Schultz

szulc@umich.edu

SI 787: Matching Mechanisms, Final Report

Professor Chen
10/18/08

A Matching Mechanism for Allocating Attention to Tasks in Open Source Projects

Abstract

In this paper we discuss using the often celebrated top-trading cycles mechanism to increase participation in open source projects. Open source projects is typically characterized as voluntary activity where a small set of participant perform the majority of tasks. In addition, open source projects can substantially reduce the cost for business adopting the software. If we can increase participation in open source projects, we can potentially increase quality, time-to-market, reduce operational costs for businesses, and overall increase welfare for society.

Introduction

Matt Asay in a CNET article explains that open source projects can lead to substantial cost-savings for organizations (Asay 2001). Consider also that in "Open Source vs. Propriety Software: An Economic Perspective", Gök explains that 10% of open source developers comprise 74% of the work in open source projects. In addition, Gök describes that there a large disincentives for developers to engage in drudgery or "non-sexy" work. Admittedly, Gök describes that reputation systems help to ameliorate the disutility from drudgery work. It's not clear that reputation systems alone are the only mechanism by which we can increase participation. In other words, if we can design more or better mechanisms to increase participation in open source development, we can increase quality and time-to-market of projects, and, thus, reduce operational costs of businesses using open source software. Human resources in both open source projects and businesses can then be deployed to more valuable and non-redundant tasks.

Problem Statement

Let's assume that participants derive value from open source project only if is successful. For volunteers, the disutility from a particular task is outweighed by the value x the probability of the project succeeding. Projects are comprised of tasks, and the more tasks performed increases the likelihood of the project succeeding. We hypothesize that there are "would-be" participants for open source projects. They would be willing to participate but the P(success) is lower than the disutility of performing any task. Although, they would be willing to participate if a system could credibly ensure the commitment of another individual to perform a desired task. For instance, let's suppose we have two individuals Bob & Cathy. Bob & Cathy are interested in an open source project. Bob wants a bug fixed and is pretty good at UI fixes. Cathy wants a UI issue fixed and is pretty good at bug fixes. Bob is not terribly interested in fixing bugs and fixing a UI issue is time-consuming. Cathy faces the reverse situation. Let's formalize it by saying Bob values task A at 1 and task B at 5. Cathy values task A at 5 and task B at 1. Now, the cost for Bob of task A is 2 and task B is 6. The reverse is the case for Cathy. The utility to each is obviously negative. Let's consider how the situation looks formalized.

		Cathy		
		A	B	!AB
Bob	A	(-1, -1)	(4,4)	(-1,0)
	B	(0, 0)	(-1,-1)	(-1,0)
	!AB	(0,-1)	(0,-1)	(0,0)

Bob and Cathy don't know that each other exists in the open source project's market. Bob might as well assume that neither task A or B is going to done by someone else (i.e. Cathy's !AB), and vice-versa for Cathy. If that is so, Bob's best bet is to nothing. The same is true for Cathy. As we

can see, though, if Bob and Cathy have a credible commitment, (A,B) is pareto optimal. If both could trade tasks, they would be better off.

We don't necessarily have to confine our problem to bilateral trade. A mutual beneficial trading cycle with many participants could occur in which all members are better off by trading tasks.

We might also consider that Bob and Cathy's (or any trading cycle) tasks exist across different projects. This is an interesting dynamic that we have yet to see discussed. It offers the opportunity to distribute attention to valuable and complimentary, but possibly ignored, open source projects.

Lastly, take note that the above example is definitely a prisoner's dilemma. Without any commitment Bob and Cathy will both opt to do nothing. Our proposed mechanism assumes that commitment is can be credibly established while no authority can explicitly enforce allocation of tasks. From here on out, allocation should be understood as *suggested* allocation. At the end of the paper, we discuss, in brevity, how this mechanism might work with a reputation system.

Proposed Solution: TTCWC (Top-trading cycles without chains)

Literature Review

Our problem is characterized as one-sided matching mechanism. Agents have preferences over tasks while tasks do not have preferences. The most relevant literature on the subject is Abdulkadiroglu and Sömnez's "Housing Allocation with existing Tenants". In this model some agents have ownership over a desired resource (i.e. house), some do not, and there are some unowned resources. The housing allocation mechanism seeks to establish a matching such that existing tenants are guaranteed house at least as good as the house they currently own as well as being individual rational, pareto efficient, and strategy-proof (Abdulkadiroglu, Sömnez 1999). Similarly, our problem consists of agents (trading members) who have tasks they wish to trade, and some agents (volunteers) who do not have tasks to trade. Every agent has a preference over task they prefer to commit to.

In addition, Wang and Krishna's work on timeshare mechanisms is quite relevant. This research discusses mechanisms when scenarios are continuous in nature. They describe that the solution can be addressed by solving a static version of the problem and including a waiting queue for subsequent matches (Wang, Krishna 2006). Again, similarly, agents and tasks may enter or exit an open source project continuously over the life of a project(s). We include a waiting queue option to represent the idea that some agents and tasks are better off waiting until the next execution of the matching mechanism. We might expect that the matching mechanism could be executed weekly or possibly every time a task is added.

The Model

Let us formalize the situation. The problem consists of the following:

1) A finite set of tasks, T_i
2) A waiting queue option, {q}
3) A finite set of volunteer members, M_v
4) A finite set of trading members, M_t
5) A strict preference list $P = (P_i)_{i \in Mt \cup Mv}$

The entire set of members is $M = M_v \cup M_t$. All members have strict preference relation P_i on $T \cup \{q\}$ representing tasks an agent is willing to commit to. In addition, each M_t have ownership over a mutually exclusive subset of T. In other words, each trading member has one or more tasks that s/he wishes to trade. Tasks necessarily have to originate from a member, and by allowing members to submit more than one task gives us an excess of tasks. While at first glance it may appear that we are trying to allocate a finite set of tasks, realistically tasks act as a proxy for a member's attention, and ultimately attention is the scarce resource that we are really trying to optimize. Simply, there's a lot of tasks but not enough attention to devote to them.

We assume the set of tasks a member prefers to commit to and the set of tasks a member prefers another to commit to are exchangeable. For instance, m_0 prefers t_4, t_5, t_6 and owns t_0, t_1. Any combination in a matching is preferred to the waiting queue option. Another way of saying it is that we don't consider the scenarios like this: m_0 prefers t_4 for t_0 but does not prefer t_4 for t_1.

Whether a member is volunteer or trader is determined by whether or not s/he has submitted task(s) to be performed. In other words, if a member submitted task(s), s/he belongs to M_t. If s/he has not submitted a task, s/he belongs to M_v. We assume a member from M_v will accept any T_i in trade. To state simply, a volunteer is willing to perform his/her preferred tasks irrespective of which task is committed to. We also assume that a trading member will not commit to a task if no other member is willing commit to his/her submitted task.

This model is initially quite similar to Abdulkadiroglu and Sönmez's "Housing Allocation with existing Tenants" model (i.e. we have members with tasks to be performed and others who do not have tasks to be performed). There is one seemingly trivial difference that makes the problem extremely interesting and difficult. As noted before, trading members will not commit to a task unless someone commits to one of his/her tasks. To understand the oddity of this situation, let's phrase it in terms of housing allocation. An existing tenant will not move to another house *unless* someone is willing to move into his/her house. Similarly, a member is not willing to commit to a task unless someone is willing to commit to his/her task. As we will see in the mechanism, this leads to some very different considerations and approaches to resolving this oddity.

TTCWC Mechanism

Let's define exactly how the mechanism executes.

Initialization: A random set of priorities are generated for trading members and volunteers. A random set of priorities is generated for tasks.

1) Determine preferences
- Each member points to his/her preferred task or the waiting queue
- Every task points to it's associated member
    - IF a task does not have an associated member anymore, leave the task with no pointer
- IF a member points to q, remove the member and all associated tasks, determine preferences

2) Determine Cycles
- IF one cycle forms, assign tasks, and remove members, determine preferences
- IF multiple cycles form, start with the highest priority member, assign tasks, and remove members, determine preferences
    - IF a member had more than 1 task associated with him/her, leave those tasks with no pointers
- IF no cycle forms
    - Find the set of existing volunteers and tasks with no pointers
    - IF both exist
        - Start with the highest priority task and point it the highest priority volunteer, if a cycle exists, assign, remove members and tasks, determine preferences
        - IF no cycle exists iterate from each task over all volunteers. If still no cycle, decrement the highest priority task and repeat over all volunteers. If we find a cycle, assign, remove, and determine preferences.
        - IF no cycle still exists after iterating over all pointer-less tasks and volunteers, assign the highest priority volunteer the task s/he is pointing to and determine preferences.
    - IF pointer-less tasks do not exist but volunteers do
        - Assign the highest priority volunteer the task s/he is pointing to and determine preferences
    - IF volunteers don't exist (it doesn't matter whether pointer-less tasks exist or not)
        - Remove the highest priority pointer-less task, determine preferences
        - IF no pointer-less tasks exist, place all members and tasks in the queue
- IF no member exists goto step 3

3) End Algorithm

Description of Mechanism

One might have noticed that this algorithm is highly complicated and deviates substantially from matching mechanisms described in literature. Let's explore why this is so. First, let's imagine we have a trading member with an associated task and s/he is pointing to a preferred task, as follows:

t_0 -> m_0 -> t_5

m_0 will not commit to t_5 unless someone else will commit t_0. We might find another member who will commit to t0, but we continually encounter the same situation. For instance:

t_6 -> m_6 -> t_0 -> m_0 -> t_5

So, we essentially always have to find cycles and no chains, and, thus, the algorithm becomes more complicated.

In actuality, the only chains that would form are when a member points to q:

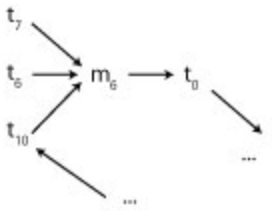
t_6 -> m_6 -> t_0 -> m_0 -> q

We might consider that if a volunteer pointed to t6, that we could execute the chain.

mv_8 -> t_6 -> m_6 -> t_0 -> m_0 -> q

But we can't allow this to occur in the algorithm; it's not strategy-proof. Notice m_0 gets his/her task performed for free. While that may not seems so bad, m_0 now has an incentive to truncate preferences in the hopes to create a q-chain. Hence, as soon as a member points to q, we immediately remove the member and associated tasks from the system. By remove we mean, simply, they are put into the queue to be included in a future execution at a later point.

When a cycle forms we assign tasks and remove members. Of course, as described beforehand, multiple tasks associated with a member are kept in the system as pointer-less tasks. For instance:

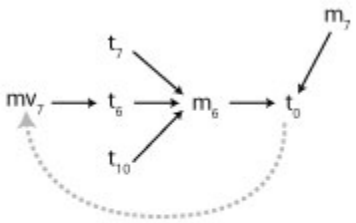


1. Cycle Forms

2. Assign tasks. Remove members.
Leave excess tasks without pointers

We can keep them in becomes we need not be concerned with strategic manipulation. The excess tasks could increase the welfare of other members (i.e. make cycles). While it's true that m_6 could potentially throw in a bunch of tasks that s/he is really not willing to trade for. If no one picks up those tasks (i.e. a cycle) then those tasks will be still associated with the m_6 at later executions of the algorithm. If another member commits to any of the excess tasks then we conclude that other members are better having the task remain in the system. In other words, we may potentially lose some strategy-proofness in trade for efficiency.

When we reach a point in the algorithm where no cycle forms, the process becomes quite involved. We start by finding cycles using the volunteers and pointer-less tasks. First, we should note that because volunteers have no associated tasks and given that we cannot accept chains, volunteers will not form a cycle without pointer-less tasks. The purpose of the exhaustive search with pointer-less tasks is to find a cycle where all members retain his/her preferred task. Having volunteers is interesting dynamic because they implicitly accept any task in trade. We run though the variations based on random priorities so that we can ameliorate any strategic manipulation. For example:



1. Exaustive cycle search finds t_0 connects to mv_7 first.

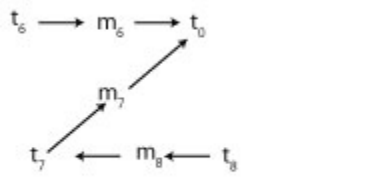
2. Assign tasks. Remove members.
Leave excess tasks without pointers

Alternatively, the exhaustive search could select the largest chain formed. The purpose here is to include as many trading members as possible since establishing commitment with them is so difficult. Another reason we might prefer to do this exhaustive volunteer cycle search is to ensure that volunteers receive some higher prioritized tasks. Otherwise, we might expect that volunteers

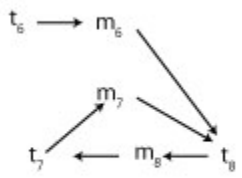
end up getting his/her least preferred task (if any). Accordingly, Ww might expect that any mechanism that consistently allocates volunteers with low preferred tasks (or worse the waiting queue) might over many executions give volunteers incentive to not participate.

Because all members own tasks and point to preferred tasks, we should always end up with a cycle in the first round. If there are no pointer-less tasks after removing the cycle, then we are essentially done. If there are pointer-less tasks left, then we potentially can form a cycle with volunteers.

Eventually, we might run into the case where no volunteers are present and the no cycles are formed. First, we start removing pointer-less tasks since they cannot point and form a cycle with anyone anymore. For instance:



1. Remove t_0, it can't point to anyone.



2. Now m_7 prefers t_8 and cycle forms

If we still end up with no cycles, we conclude that all members and tasks are better off waiting until the next execution of the algorithm. We certainly could try to find cycles by asking members to point to his/her next preferred task. First, this would require us to go through many different possible combinations. Second, we assume that members don't have stringent time preferences. In that, we mean they are willing to wait until the next execution of the algorithm if it means that they may get their preferred task choice. Lastly, we are not so much concerned with trying to comprehensively match every member in one execution of the algorithm. We are simply concerned with getting *more* tasks than usual accomplished in a algorithm execution.

Example of Mechanism

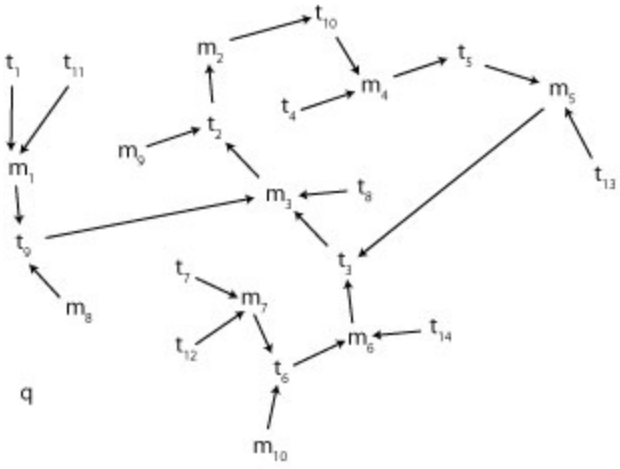
Let's assume we have 10 members (7 trading members and 3 volunteers). Each members preferences are as follows denoted as such member{(tasks committing to), (tasks wanting others to commit to)}:

m_1 {(t_9, t_{10}, t_2), (t_1, t_{11})};
m2 {(t_{10}, t_3, t_5, t_6), (t_2)};
m3 {(t_2, t_4, t_5, t_6), (t_3, t_8, t_9)};
m4 {(t_5, t_9, t_1, t_8, t_{11}), (t_4, t_{10})};

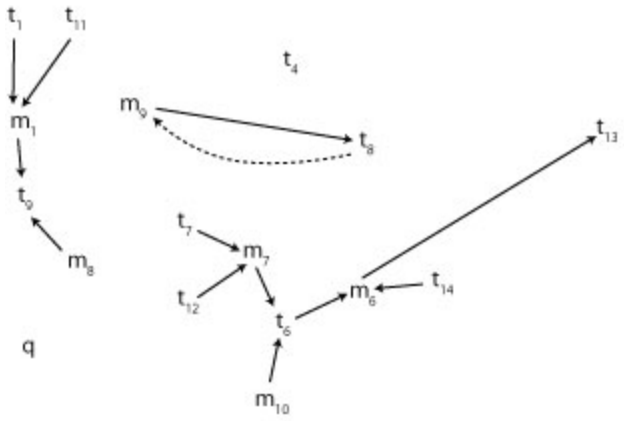
m5 {(t_3, t_8, t_{12}, t_{14}), (t_5, t_{13})};
m6 {(t_3, t_{13}, t_7), (t_6, t_{14})};
m7 {(t_6, t_1, t_{13}, t_{11}), (t_7, t_{12})};
m8 {(t_9, t_{14}, t_{11}), (any)};
m9 {(t_2, t_8, t_{11}), (any)};
m10 {(t_6, t_{13}, t_1), (any)};

Under a purely volunteer regime, the outcome would be (m_8, t_9), (m_9, t_2), (m_{10}, t_6). In other words, only 3 tasks would be performed in one execution of the algorithm. If execute the algorithm 3 times, t_1, t_2, t_6, t_8, t_9, t_{11}, t_{13}, t_{14} (eight tasks) would be performed (assuming that volunteers want to keep performing tasks in subsequent executions). In addition, t_{11} would be performed twice.

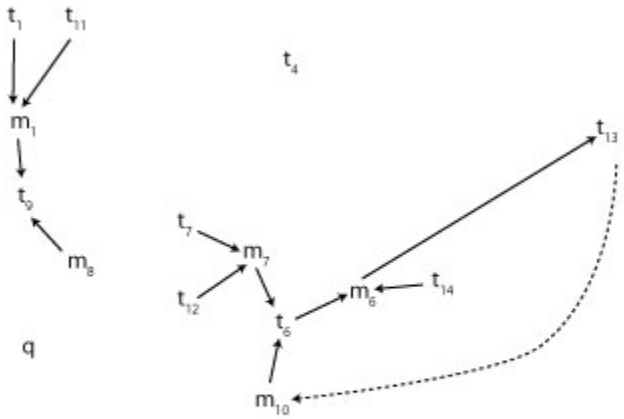
The proposed mechanism improves upon this by allocating more tasks to be performed in less executions of the algorithm. Here's how the above scenario plays out:



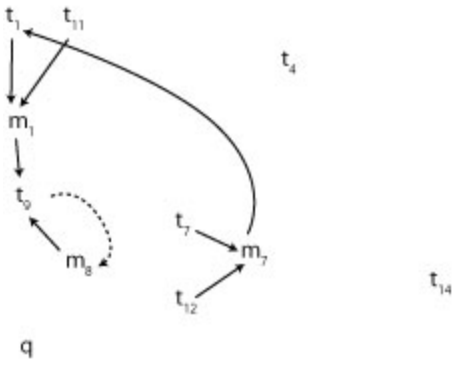
1. Determine pref Strences, determine if a cycle exists.
Indeed, m_2->t_{10}->m_4->t_5->m_5->t_3->m_3->t_2->m_2.
Assign tasks and remove members.



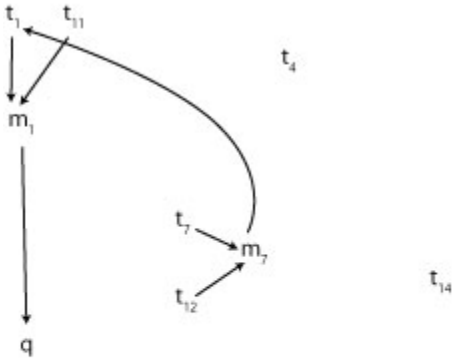
2. No cycle forms. Start with highest priority pointer-less task to highest priorty volunteer.
t_4 -> m_8 (no cycle), t_4 -> m_9 (no cycle), t_4 -> m_{10} (no cycle)
t_8 -> m_8 (no cycle), t_8 -> m_9 (CYCLE FOUND)
Assign task, remove members: m_9->t_8



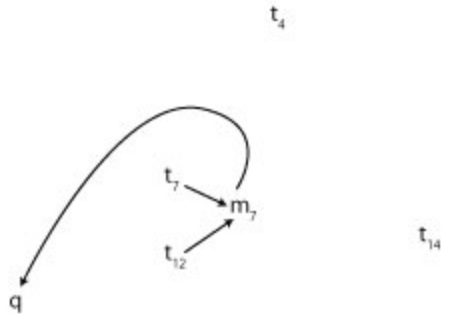
3. No cycle forms. Start with highest priority pointer-less task to highest priorty volunteer.
t_4 -> m_8 (no cycle), t_4 -> m_{10} (no cycle)
t_{13} -> m_8 (no cycle), t_{13} -> m_{10} (CYCLE FOUND) Assign task, remove members: m_{10}->t_6->m_6->t_{13}



4. No cycle forms. Iterating over pointer-less tasks yields nothing.
Assign m_8->t_9, remove from system



5. m_1 -> q, remove m_1 and associated tasks t_1, t_{11}



3. No more cycles can be formed. Put everything into q.

Final matching is:
(m_2, t_{10}), (m_4, t_5), (m_5, t_3), (m_3, t_2), (m_9, t_8), (m_{10}, t_6), (m_6, t_{13}), (m_8, t_9)

Unmatched: (m_7, m_1) & (t_7, t_4, $t1_4$, t_{12}, t_1, t_{11})

We have 8 tasks being performed in only one execution of the algorithm. Most volunteers received a high prioritized task. m_9 would have preferred t_2, but if we assume that members prefer having more tasks performed quicker to a 1st, 2nd or nth choice, we can expect that m_9 won't want to deviate.

Theoretical Properties

The mechanism should credibly satisfy several criteria: individual rationality, strategy-proofness, and pareto efficiency.

Individual rationality
A member that participates (given truthful preferences) cannot due worse upon participation. The mechanism never assigns a task to a member that s/he did not prefer less than the queue option.

Pareto efficiency
In every step each member points to his/her most preferred task for trade or to perform voluntarily. A member is only assigned a less preferred task in the event that a cycle forms with that task. A cycle did not form with the more preferred task (if it did s/he would have been

assigned the more desired task). If a cycle did not form, the tasks would not have been committed to; one member would have defected starting a chain reaction where no one wanted to do his/her assigned tasks.

Strategy-proof
No member can gain by truncating his/her preferences. Upon truncation, a cycle forms which would have formed irrespective of truncation. Or the member is assigned {q} and none of his/her tasks she wanted performed could be committed to. If a member manipulates preferences, s/he will only end up with a less preferred task than what would have otherwise been. Lastly, a member *could* add several addition tasks s/he is not willing to trade for. A member might do this in the hopes of a volunteer starting a cycle that accomplishes an untradeable task. It's a risky strategy since it's also likely that a chain could form around the undesirable task, and s/he will want to defect after assignment. So, the mechanism can be considered mostly strategy-proof.

Discussion & Future Research

We presented a mechanism that is individually rational, pareto efficient, and arguably strategy-proof. This mechanism ideally can increase participation in open source projects while reducing the time to develop the project. In addition, it is more effective at allocating attention while lightening the burden on volunteers to accomplish all tasks. Lastly, we never explicitly modeled tasks from different projects. This is intentional. As we can see, the mechanism irrespective of what project a task came from. We have yet to see any discussion on how to allocate attention across complimentary projects. A mechanism such as this can be established such that it acts as a task clearinghouse for any open source project. With that, members can engage in trades across different projects.

Admittedly, this is simple model of a complex system. There are many other considerations. For instance:

- What if members have preferences over tasks they do not want to be a part of the project?
- What if a member just wants to submit a task (e.g. a bug report) but has no desire in participating in the project?
- What if we want members to simultaneously work on the same task? We might want this in order to discover different solutions to a task.
- What if a member is willing to perform two or more tasks in order to get his/her task committed to?

Lastly, we need to strongly note that allocation in the mechanism can only be a suggestion. There is no authority that can coerce a member into performing a task. It's quite possible that members will say that they will commit to a task but inevitably defect once his/her submitted task is completed (or simply they prefer not to perform the task anymore). This mechanism alone

cannot solve this problem, and it certainly is outside the scope of this paper. In general, we expect that reputation systems could help ameliorate the problem by establishing repercussions for members that defect on a previously stated commitments. This mechanism will undoubtedly be more successful if coupled with a reputation system.

References

Abdulkadiroglu, A., Sönmez, T., 1999. House Allocation with Existing Tenants. *Journal of Economic Theory*, Elsevier, vol. 88(2), pages 233-260, October.

Gök, A., 2004. *Free Software & Open Source Days / 2004 Workshop*. Presented February 27-29, 2004.

Wang, Y. and Krishna, A. 2006. Timeshare Exchange Mechanisms. *Manage. Sci.* 52, 8 (Aug. 2006), 1223-1237. DOI= http://dx.doi.org/10.1287/mnsc.1060.0513.

Asay, M. Enterprises saving millions of dollars with open source., 2001. Retrieved October 1, 2008, from http://news.cnet.com/8301-13505_3-9770468-16.html.