STWM: A Solution to Self-adaptive Task-worker Matching in Software Crowdsourcing

--By Ying Fu

Outline

- Introduction
- Design of the solution
  - Framework
  - Meta-model for description
  - Match algorithm for individual worker
  - Team formation algorithm
- Simulation experiments
- Conclusion and future work

Introduction

- What is crowdsourcing ?
  - Outsourcing a task via open call
- Crowdsourcing websites
  - MTurk/ TopCoder/ Upwork/ CrowdFlower
  - 80,000 jobs, 5+ million workers
- Task-worker matching plays a crucial role
- How to describe task requirements and worker skills
- What criteria should be given in the description

Introduction

- Description with natural language (Taskcn)
  - Not machine-readable
  - Inefficient
  - subjective
- Tags (upwork)
  - Not sufficient to articulate task publishers' needs
    - task A(Java && Javascript)
    - task B(Java || Javascript)
  - Requirements are not exhaustive
  - Matching rules vary on skills
  - No single suitable worker for the task

A Solution – STWM

- Meta-model for description
  - Extensible/ Customized
- Self-adaptive task-worker matching algorithm
  - Efficient
  - Match tasks and workers according to the customized rules
- Team formation
  - Workers to form a team

Framework of STWM

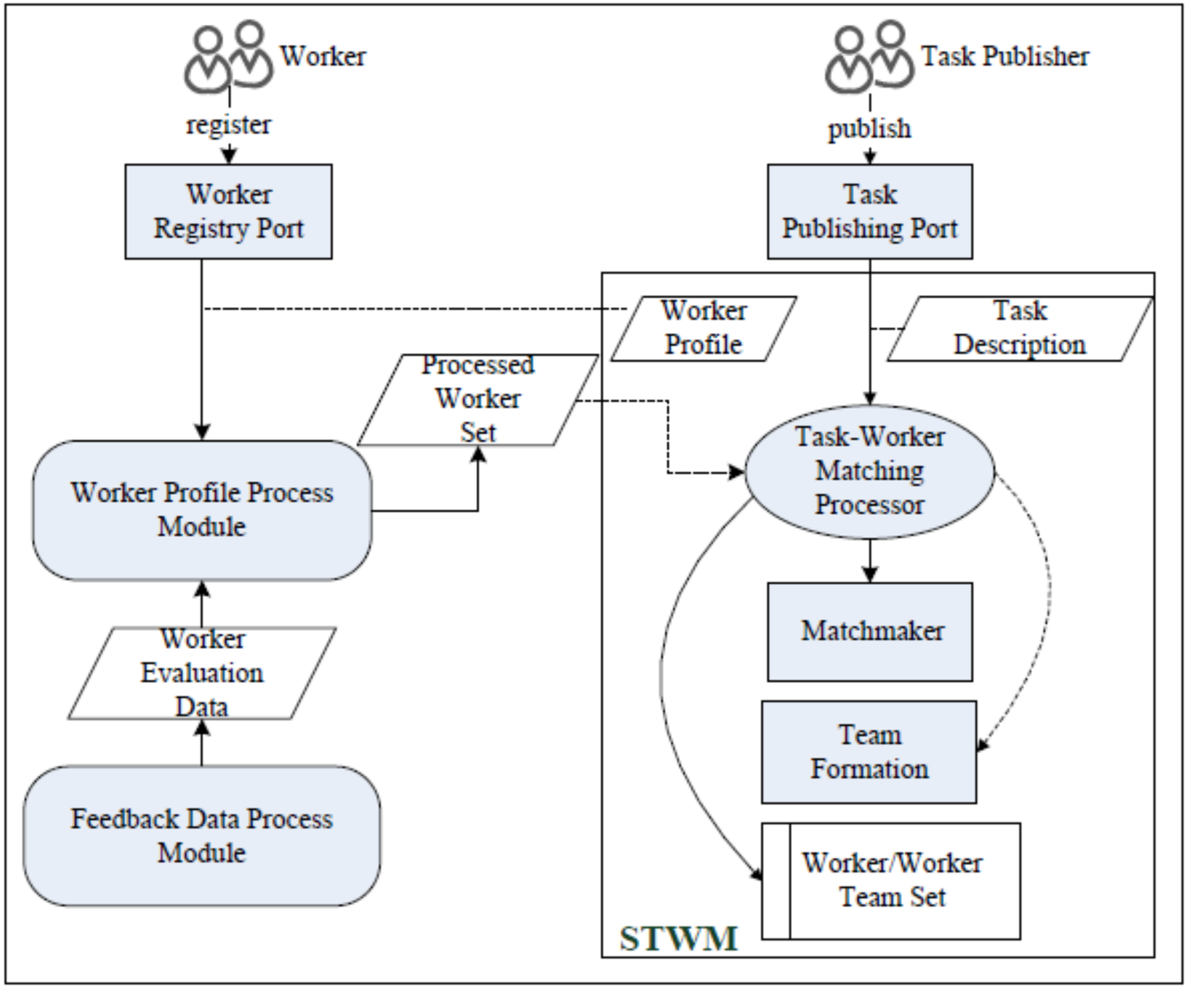

Fig.1. Structure of the task-worker matching solution

Meta model for description

```
Class PropertyMetaData{
    name: String
    type: <type, constraint>
    constraint: conts
    match: function
    composite: function
    domain: {time, space, skill, pay}
}
```

```
Class Property{
    value: <value, constraint>
    weight: w
    attribute: <type, value, constraint>
}
```

- constraint, a function defined in RDF
- match: =, within, $>$, $<$, $\neq$ … API provided
- composite: Max, Min, $\cup$, $\cap$… API provided

Meta model for description

<table>
<tr>
<td>Class languageMetaData{
name: language
type:<t,String || String[]>
constrain: conts
match: include
composite: ∪
domain: skill
}</td>
<td>Class timeMetaData{
name: time
type: <int, null>
constraint: conts
match: <
composite: Max
domain:time
}</td>
<td>Class payMetaData{
name: pay
type: <int, null >
constraint:conts
match: <
composite: Sum
domain: pay
}</td>
</tr>
</table>

Definitions of the metadata class for properties of time, pay and language skill

Meta model for description

A definition of class language and two instances this class

```
Class language {
   value: <{v}, inLanguageSet(v)> //inLanguageSet(v) is defined in RDF
   weight: w
   skill_level: <double[],constraint>
}
```

```
language language_of_task1 {
  value: <{java,C++,JavaScript}, null>
  weight: 0.9
  skill_level: <{3.0,3.0,3.0},{>,>,>}>
  constraint: java || C++ && JavaScript
}
```

```
language language_of_worker1{
  value:<{java, C++, sql}, null>
  weight: null
  skill_level: <{4.0,2.0,3.0},null>
}
```

Meta model for description

- Class definition for task and worker:

```
Class worker{
        Property[] skill_list;
        Property time;
        Property space;
        Property pay;
        double score;
}
```

```
Class task{
    Property[] skill_requirment;
    double skill_weight;
    Property time_requirment;
    Property space_requirment;
    Property pay_give;
}
```

score: a criterion used to sort matched workers

skill_weight: the weight of the skill_rerquirment among the Listed four property requirments in the task class

Matching algorithm for individual worker

- *Set<worker> FinalSet:* workers in this set meet all the requirements of the task T.
- *Set<worker> PreferCandidate:* workers in this set meet all the necessary requirements of the task T.
- *Set<worker> Candidate:* workers in this set meet part of the requirements of the task T.
- *Set<Property> M:* the set of all properties required by task T.
- *Set<Property> M':* the set of all necessary properties required by task T.

Matching algorithm for individual worker

- necessary property:
  - if $p.domain \neq skill$ and $p.weight \geq baseline\_weight$
  - if $p.domain = skill$ and $p.weight \geq avg\_weight(skills)$
- calculation formula of worker.score
  - *w* is an instance of Class worker, *p'* is the property of the worker *w* with the same property name as *p*

$$w.score = \sum_{p \in M'} p.weight \times p'.skillLevelValue \mid for\ p' \in w\ and\ p'.domain = skill$$

Matching algorithm for individual worker

Algorithm1. Task-worker matching algorithm

Input: Set<worker> W; task T.

Output: Set<worker> W';

```
function matching(Set<worker> W, task T):
    FinalSet, PreferCandidate, Candidate, W' ← Ø;
    Set<worker> ℛ = Cluster(W, T);
    for each worker w in ℛ:
        if for ∀p ∈ M, p.match(p')=true then
            FinalSet = FinalSet ∪ w;
        else if for ∀p ∈ M', p.match(p')=true then
            PreferCandidate = PreferCandidate ∪ w;
        else if ∃p ∈ M, p.match(p')=true then
            Candidate = Candidate ∪ w;
        endif
    endfor
    if FinalSet ≠ Ø then
        for each worker w in FinalSet
            Calculate w.score;
        endfor
        W' = Sort(T, FinalSet)
    else if PreferCandidate ≠ Ø then
        for each worker w in PreferCandidate
            Calculate w.score;
        endfor
        W' = Sort(T, PreferCandidate)
    endif
    return W';
end function
```

Team formation algorithm

- 1) T: a task published by a client
- 2) W: a set of workers
  - n Number of workers
- 3) I: a set of required properties of the task T
  - m Number of required properties
- 4) W^j : jth worker in W
- 5) I^i: ith property of the task T
- 6) Q: team assigned to the task T
- 7) q: property profile of the team Q

Team formation algorithm

- Worker W
  - $a_{ij} = 1$ the jth worker W^j has the ith property of I (I^i)
  - $a_{ij} = 0$ otherwise
- Team Q
  - $q_i = 1$ the ith property of I (I^i) is covered by the team Q

$$x_j = \begin{cases} 1, \text{if jth worker belongs to team Q} \\ 0, \text{otherwise} \end{cases} \quad \text{for } W^j \in W$$

$$q_i = \min\{\textstyle\sum_{W^j \in W} a_{ij} x_j , 1\} \quad i = 1,2,3 \ldots m$$

Team formation algorithm

- the team formation problem can be formally formulated as a binary integer program as follows, where c_j represents the cost of choosing the worker w^j.

$$\text{Minimize } \sum_{W^j \in W} c_j x_j$$

$$\text{Subject to } \sum_{W^j \in W} a_{ij} x_j \geq 1 \quad \text{for } I^i \in I$$

$$\text{and } x_j = 0 \text{ or } 1 \quad \text{for } W^j \in W$$

- Meta-RaPS-SCP-Construction
  - a feasible solution for a SCP instance
  - randomness

Team formation algorithm

Algoritm 2.Team formation

Input: Set<worker> W; task T; int preferCount; int maxLoops.

*Output: Set<team> **Qs**; //a team is a set of workers*

```
function teamFormation(Set<worker> W, task T, int preferCount,int maxLoops):
    Qs ← Ø; Set<Property> PSet ← Ø; int i = 0;
    While Qs.size < preferCount && i ≤ maxLoops:
            Boolean isFeasible = true;
            team Q = Meta-RaPS-SCP-Construction
                    (T, W, %priority, %restriction);
            for each property p of task T:
                    PSet = Ø;
                    for all workers in team Q add p' to PSet;
    //p' is the property of the worker in team Q with the same property name as p
                    Property pt = p.composite(PSet);
    //the composite function is given in the definition of p as shown in meta-model
                    if p.match(pt) = false then
                            isFeasible = false;
                            break;
                    endif
            endfor
            if isFeasible = true then
                    Qs = Qs ∪ Q;
            endif
            i ++;
    end while
    return Qs;
end function
```

Simulation experiments

Exp1: experiment for task-worker matching with comparison

- Same skill requirements, different preference

```
task A{
    skill_requirement = {langOfA,dbOfA};
    skill_weight = 0.70;
    time_requirement = timeOfA;
    space_requirement = null;
    pay_give = payOfA;
}
```

```
task B{
    skill_requirement = {langOfB,dbOfB};
    skill_weight = 0.70;
    time_requirement = timeOfB;
    space_requirement = null;
    pay_give = payOfB;
}
```

```
langOfA{
    value:<{Java,JavaScript},null>
    weight: 0.80
    skill_level:<{3.0,3.0},{>,>}>
    constraint: Java && JavaScript
    }
```

```
langOfB{
    value:<{Java,JavaScript},null>
    weight: 0.20
    skill_level:<{3.0,3.0},{>,>}>
    constraint: Java && JavaScript
}
```

Simulation experiments

- **Exp1: experiment for task-worker matching with comparison**

```
databaseMetaData{
    name:database
    type:<t,String || String[]>
    constraint: null
    match: include
    composite: ∪
    domain: skill
}
```

```
database{
  value: <{v}, null>
  weight: w
  skill_level: <double[],constraint>
}
```

```
dbOfA{
  value:<{mysql},null>
  weight: 0.20
  skill_level: <{3.0},{>}>
  constraint: null
}
```

```
dbOfB{
  value:<{mysql},null>
  weight: 0.80
  skill_level: <{3.0},{>}>
  constraint: null
}
```

Definition for property database

Simulation experiments

- **Exp1: experiment for task-worker matching with comparison**

Table 1-a. workers recommended for task A

properties / workers	language		database	worker.score
	Java	JavaScript		
A1	4.96	4.92	3.12	6.92
A2	4.90	4.80	4.20	6.87
A3	4.83	4.76	3.14	6.71

Table 1-b. workers recommended for task B

properties / workers	language		database	worker.score
	Java	JavaScript		
B1	4.03	3.21	4.98	3.49
B2	3.26	4.13	4.78	3.35
B3	3.58	4.47	4.72	3.30

Simulation experiments

Exp2: experiment for team formation

- set preferCount = 2, maxLoops = 100, %priority = 80% and %restriction = 60% .

```
task C{
    skill_requirement = {langOfC,dbOfC};
    skill_weight = 0.70;
    time_requirement = timeOfC;
    space_requirement = null;
    pay_give = payOfC;
}
```

```
langOfC{
  value:<{Java,C++,JavaScript,Ruby},null>
  weight:0.60
  skill_level: <{3.0,3.0,3.0,3.0},{>,>,>,>}>
  constraint: Java || C++ && JavaScript
                && Ruby
}
```

```
dbOfC{
  value:<{mysql,Oracle},null>
  weight:0.40
  skill_level: <{3.0,3.0},{>,>}>
  constraint: mysql || Oracle
}
```

```
timeOfC{
  value:<100,inHours>
  weight:0.10
  constraint: null
}
```

```
payOfC{
  value:<500,inTotal>
  weight:0.20
  constraint: null
}
```

Simulation experiments

Exp2: experiment for team formation

Table 2. teams recommended for task C

		language	database	time	pay
team1	worker1-1	Java(4.52),Ruby(4.26)	mysql(4.83)	90	321
	worker1-2	C++(3.59),JavaScript(3.62)	mysql(3.61)	95	150
team2	worker2-1	Java(4.17),JavaScript(3.21)	Oracle(3.77)	100	270
	worker2-2	Ruby(3.87),JavaScript(3.58)	none	76	162

Exp3: no suitable team found for task T

- langOfC.value = {Java, JavaScript,Ruby, Html5}
- constraint: Java && JavaScript && Ruby && Html5
- payOfC.value = 50
- the task should be re-described

Conclusion and future work

- An extensible meta-model for the descrip-tion of task requirements and worker skills
- A self-adaptive matching algorithm
- Dynamically team formation

- Conduct some experiments in a real crowdsourcing workplace to evaluate the practicability of our solution.
- Improve our meta model to support more complex constraints on task requirements such as the dependency between tasks.