Representação de um Problema de Alocação de Tarefas em Sistemas Multi-agentes como um Problema de Programação Linear

Anny G. N. Figueira, Tulio L. Basegio e Rafael H. Bordini

[1]Faculdade de Informática – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brazil

{anny.figueira, tulio.basegio}@acad.pucrs.br, rafael.bordini@pucrs.br

Resumo. *Este trabalho tem como objetivo apresentar uma implementação em programação linear do problema de alocação de tarefas em sistemas multi-agentes apresentado no artigo "An Algorithm for Allocating Structured Tasks in Multi-Robot Scenarios"[Basegio and Bordini 2017]. Esta implementação tem como objetivo oferecer uma comparação justa de resultados entre a solução proposta no trabalho de Tulio L. Basegio, descentralizada, e uma solução ótima centralizada, obtida através de programação linear.*

1. Introdução

O trabalho de Tulio L. Basegio [Basegio and Bordini 2017], o qual este trabalho tomou como base e também referido como "trabalho original" ao longo do documento, propõe um algoritmo para resolver o problema de alocação dinâmica de tarefas de forma descentralizada, considerando conjuntos de robôs heterogêneos e diferentes tipos de tarefas. Tais características – heterogeneidade dos robôs e a definição de diferentes tipos de tarefas – são levadas em consideração por serem importantes em cenários reais de desastres, tais como cenários de enchentes e alagamentos [Scerri et al. 2012] e esta é a motivação do trabalho original.

Para a avaliação do trabalho, o autor apresentou em seus experimentos uma comparação entre sua solução descentralizada e uma implementação centralizada desenvolvida através de programação linear. Porém, nem todas restrições originais do problema foram representadas nesta implementação, de modo que ela abstraía alguns passos realizados no processo de alocação das tarefas. Com o objetivo de complementar o trabalho original e viabilizar uma comparação mais justa dos resultados, o presente trabalho propõe mudanças na representação do problema em programação linear, adicionando as restrições faltantes.

A Seção 2 apresenta uma descrição do problema tratado no trabalho original, bem como aspectos relevantes da implementação a fim de contextualizar as mudanças realizadas para esta contribuição, apresentadas na Seção 3. A Seção 4 trata da conclusão e de trabalhos futuros.

2. Apresentação do problema e implementação

Nesta seção é descrito o problema de alocação de tarefas e são apresentados os detalhes da modelagem e implementação relevantes. Todas as informações aqui apresentadas se referem ao trabalho original de Tulio L. Basegio.

2.1. Problema de alocação de tarefas

O problema de alocação de tarefas entre múltiplos robôs consiste em identificar quais robôs deveriam executar quais tarefas de forma a atingir um objetivo global da melhor maneira possível. Quando uma tarefa é executada por um robô, ela contribui com algum valor para o objetivo global, e este valor é referido como valor de utilidade.

Com o objetivo de representar diversos problemas envolvidos em cenários reais, são utilizados três tipos de tarefas:

- **Tarefa atômica *(Atomic task - AT)*:** tarefa que não pode ser decomposta em subtarefas.
- **Tarefa simples decomponível *(Decomposable simple task - DS)*:** tarefa que pode ser decomposta em um conjunto de subtarefas atômicas ou outras DSs desde que tais partes decompostas sejam executadas por um mesmo robô.
- **Tarefa composta *(Compound task - CT)*:** tarefa que pode ser decomposta em um conjunto de subtarefas atômicas ou subtarefas compostas. Quando cada uma das subtarefas precisa ser alocada para diferentes robôs, a tarefa é chamada de CN (N subtarefas que necessitam exatamente de N robôs). Quando não há tais restrições e as subtarefas podem ser alocadas para de um a M robôs, a tarefa é chamada de CM, onde M é o número de subtarefas.

2.1.1. Descrição do problema

A descrição do problema, em linguagem natural, dá-se da seguinte forma: há uma quantidade de robôs, de tarefas e de subtarefas, sendo que cada tarefa possui uma ou mais subtarefas. Há tipos de tarefas cada subtarefa pertence a somente um tipo. Cada tipo de tarefa possui um número mínimo e máximo de subtarefas que um robô precisa pegar quando for fazer a alocação de subtarefas desse tipo. Com estes valores mínimos e máximos é feita a representação das tarefas DS, CN e CM. Por exemplo: um robô tentando alocar um tarefa DS precisa pegar todas as subtarefas dela, sendo o mínimo e máximo iguais.

Além disso, também há a relação entre as capacidades, papéis e subtarefas. Esta relação se dá da seguinte maneira: cada robô possui um conjunto de capacidades, cada papel é composto por um conjunto de capacidades específicas que um robô precisa possuir para desempenhá-lo e cada subtarefa precisa ser executada por um robô capaz de desempenhar um papel associado a ela.

O objetivo do problema de otimização é, então, encontrar uma alocação que maximize a soma das utilidades de todos robôs, satisfazendo todas estas restrições.

2.2. Abordagem proposta centralizada

Para a implementação utiliza-se uma **organização** que é responsável por anunciar em um *blackboard* as tarefas que precisam ser executadas, de forma visível a todos os **agentes**. Além disso, o **ambiente** é o local onde os agentes executam as tarefas.

A partir disso os agentes iniciam um sistema de leilão entre si, a fim de que cada um aloque o maior número possível de tarefas que pode desempenhar. Os algoritmos são executados em cada agente, caracterizando um sistema descentralizado, de modo que cada agente está ciente de suas capacidades e, por consequência, das tarefas que pode executar.

2.3. Implementação

Os agentes se comunicam através de Jason[1], um interpretador para uma versão estendida de AgentSpeak.

A implementação foi feita através de um projeto JaCaMo[Boissier et al. 2013], um framework para programação multi-agentes. Ele oferece suporte a todos artefatos apresentados na modelagem, sendo eles:

- **Ja**son, para a programação dos agentes;
- **CA**rtAgO[2], para programação de artefatos de ambiente (*environment* na qual os agentes estão inseridos); e
- **Mo**ise, para a programação de organizações multi-agentes[3].

Além disso, foi implementado um gerador de entradas com o intuito de gerar conjuntos de entradas com diversas características de forma automática. Este gerador providencia entradas com configuração desejada no formato utilizado pelo JaCaMo, bem como no formato para execução no GLPK(*GNU Linear Programming Kit*)[4], ferramenta utilizada para programação linear. Além disso, ele também possui um *parser* que lê e consolida os resultados do GLPK.

Para as entradas no GLPK, no entanto, a especificação do problema é feita já possuindo as informações de quais tarefas cada robô pode desempenhar, abstraindo completamente o passo em que os robôs descobrem quais papéis podem desempenhar com base em suas capacidades para, então, descobrirem quais tarefas podem executar. Esta diferença pode impactar no tempo de execução do GLPK e, por isso, neste trabalho é apresentada uma mudança na implementação deste gerador de modo a refletir todas as restrições do problema original também na especificação do problema no GLPK.

2.4. Gerador

O gerador de entradas (*data generator*) é escrito em Java e possui os seguintes parâmetros para criação de dados para experimentos:

- ***nSimulacoes***: quantidade de conjuntos de entradas que serão geradas com essa configuração;
- ***nAgents*****:** quantidade de agentes;
- ***ntasksCompL*****:** quantidade de tarefas CM;
- ***ntasksCompN*****:** quantidade de tarefas CN;
- ***ntasksCompN*****:** quantidade de tarefas DS;
- ***limitAgents*****:** máximo de tarefas por agente;
- ***aijMax*****:** *payoff* máximo por tarefa;
- ***nroles*****:** quantidade de papéis;
- ***ncapabilities*****:** quantidade de capacidades;
- ***ncapRole*****:** máximo de capacidades por papel;
- ***nsubTasksL*****:** máximo de subtarefas de uma tarefa CM;
- ***nsubTasksN*****:** máximo de subtarefas de uma tarefa CN;

[1]Mais informações: `http://jason.sourceforge.net/`
[2]Mais informações: `http://cartago.sourceforge.net/`
[3]Mais informações: `http://moise.sourceforge.net/`
[4]Mais informações: `https://www.gnu.org/software/glpk/`

- ***nsubTasksS*:** máximo de subtarefas de uma tarefa DS;
- ***forceAgentCap*:** *flag* que indica se deseja-se forçar todas capacidades nos agentes;
- ***subtasksFix*:** *flag* que indica se a quantidade de subtarefas será igual ao limite máximo estabelecido.

Após determinadas as configurações, são criadas as seguintes listas:

- ***agents*:** lista de *Strings* que armazena os nomes dos agentes;
- ***roles*:** lista de *Strings* que armazena os nomes dos papéis;
- ***capabilities*:** lista de *Strings* que armazena os nomes das capacidades;
- ***roleCapabilities*:** lista de *roles* que armazena os papéis por capacidades;
- ***agentCapabilities*:** lista de *roles* que armazena as capacidades dos agentes;
- ***agentsList*:** lista de *agents* que armazena os próprios agentes;
- ***taskList*:** lista de *taks* que armazena as tarefas propriamente ditas;
- ***tasksCompN*:** lista de *Strings* que armazena os nomes das tarefas CN;
- ***tasksCompL*:** lista de *Strings* que armazena os nomes das tarefas CM;
- ***tasksCompS*:** lista de *Strings* que armazena os nomes das tarefas DS.

O gerador inicia populando a lista $agents$, passando por todos valores de $a = 1$ até $a = nAgents$ e criando uma *String* "ag" $+ a$ e adicionando na lista. Ele faz o mesmo processo para as listas $roles$ e $capabilities$, em função de $nroles$ e $ncapabilities$ com as *Strings* iniciando em "$role$" e "cap", respectivamente.

É então populada a lista $roleCapabilities$, percorrendo cada *String* em $roles$ e criando uma $role$ com este nome. Este processo se dá da seguinte maneira: para cada $role$ criada, é sorteada uma quantidade aleatória de capacidades entre 1 e $ncapRole$. São então sorteadas capacidades existentes na lista $capabilities$ e adicionadas nesta nova $role$ criada, tantas vezes quanto a quantidade gerada no passo anterior. Por fim, a $role$ criada é adicionada na lista $roleCapabilities$.

Após isso, a lista $agentsList$ é populada. Caso a *flag* $forceAgentCap$ estiver atribuida com $true$, são então atribuídas todas as capacidades a todos agentes. Caso contrário, são sorteadas capacidades aleatoriamente para os agentes. Este processo é feito da seguinte maneira: é percorrida a lista $agents$ e criado um $agent$ para cada *String* existente nesta lista. Para cada capacidade existente na lista $capabilities$, é sorteado um valor booleano que determina se este agente possuirá ou não tal capacidade. Caso ao final deste processo o agente acabe por não ter nenhuma capacidade atribuída, são-lhe dadas todas capacidades. Ao final, o $agent$ criado é adicionado na $agentsList$.

São então criadas as tarefas. Dado cada tipo de tarefa, é gerada uma quantidade de $tasks$ de acordo com os valores de $ntasksCompN$, $ntasksCompL$ e $ntasksCompS$. Caso a *flag* $subtasksFix$ esteja atribuída com $false$, é sorteado um valor aleatório de subtarefas entre 1 e $nsubTasksN$, $nsubTasksL$, ou $nsubTaskS$, de acordo com o tipo de tarefa. Caso contrário, a quantidade de subtarefas corresponde ao valor total de $nsubTasksN$, $nsubTasksL$ ou $nsubTaskS$. Cada subtarefa é então instanciada como uma $task$ contendo seu próprio nome (formado de $[nome\ da\ task]$ + "*st*" + $[número\ da\ subtask]$), nome da tarefa a qual faz parte, tipo e um papel sorteado aleatoriamente dentre os papéis presentes em $roles$.

Após isso, são definidos os papéis que cada agente pode desempenhar. Isto é feito através do método $checkRolesCapacities()$ da classe $agent$ e, para tal, são utilizadas as listas $roles$ e $rolesCapabilities$ populadas anteriormente.

É então feito um equilíbrio da quantidade de papéis e capacidades dos agentes e das tarefas e gerados os *payoffs* dos agentes para cada tarefa, de modo aleatório. Para cada tarefa, é gerado um número aleatório entre 1 e $aijMax$ e este *payoff* é adicionado ao agente informando a qual subtarefa ele se refere e qual o papel necessário para executá-la.

Por fim, são gerados os arquivos que serão usados como entrada para o alocador. São gerados os seguintes arquivos:

- Arquivos .asl de cada agente, contendo suas crenças iniciais: número de agentes no total, próprias capacidades, limite de agentes por tarefa e próprio *payoff* para cada tarefa;
- Arquivo .asl da organização contendo os anúncios das capacidades necessárias para desempenhar cada papel e das subtarefas pertencentes a cada tarefa, seus tipos e papéis necessários para executá-las;
- Arquivo .pddl que descreve o problema de otimização, apesentando todas as restrições e o objetivo.
- Arquivo .mod que é usado como entrada para o GLPK, contendo: o objetivo a ser maximizado, as restrições de subtarefas e de agentes e os *payoffs* que cada agente possui para cada subtarefa.

Conforme mencionado na Seção 2.3, o arquivo que é usado como entrada para o GLPK não possui as restrições relacionadas a papéis e capacidades, apenas as referentes a tarefas, subtarefas e agentes. Portanto, a contribuição proposta neste trabalho visa acrescentar as restrições restantes ao gerador, de modo que os problemas representados no GLPK reflitam de forma mais fidedigna as características do problema.

3. Contribuição

Esta seção apresenta a alteração realizada no gerador de entradas de modo a refletir de forma mais fiel a formalização do problema apresentada no trabalho original, bem como os testes realizados desta implementação.

3.1. Implementação

Primeiramente, foram adicionadas as seguintes restrições no gerador a serem escritas no arquivo do GLPK:

- `C7{y in ROLES}: sum{x in CAPABILITIES} G[y,x] <= C:`
  para cada papel, o somatório das capacidades requeridas por ele não pode ser maior do que o total de capacidades existentes;
- `C8{i in ROBOTS}: sum{x in CAPABILITIES} V[i,x] <= C:`
  para cada agente, o somatório de capacidades que ele possui não pode ser maior do que o total de capacidades existentes;
- `C9{i in ROBOTS}: sum{y in ROLES} Z[i,y] <= R:`
  para cada agente, o somatório de papéis que ele pode desempenhar não pode ser maior do que o total de papéis existentes;
- `C10{k in SUBTASKS}: sum{y in ROLES} H[k,y] <= R:`
  para cada subtarefa, o total de papéis que ela requer não pode ser maior do que o total de papéis existentes;

- `C11{i in ROBOTS, y in ROLES}: sum{x in CAPABILITIES} G[y,x]*V[i,x]*Z[i,y] = sum{x in CAPABILITIES} G[y,x]*Z[i,y]:` para cada agente e cada papel, se o agente possui as capacidades que este papel requer, então ele pode desempenhar tal papel;
- `C12{i in ROBOTS, k in SUBTASKS}: sum{y in ROLES} H[k,y]*Z[i,y]*f[i,k] = sum{y in ROLES} H[k,y]*f[i,k]:` para cada agente e cada subtarefa, se o agente pode desempenhar os papéis que esta subtarefa requer, então ele pode alocá-la.

A restrições C11 e C12 são as mais relevantes, uma vez que afirmam a relação entre as capacidades atreladas a um papel, as capacidades que o agente possui e os papéis que ele pode desempenhar e a relação entre os papéis requeridos por uma subtarefa, os papéis que um agente pode desempenhar e a relação de alocação do agente com esta subtarefa.
Para que estas restrições funcionem, foram adicionados também os seguintes parâmetros:

- `param G{ROLES,CAPABILITIES}, binary:` quais capacidades cada papel requer;
- `param V{ROBOTS,CAPABILITIES}, binary:` quais capacidades cada agente possui;
- `param Z{ROBOTS,ROLES}, binary:` quais papéis cada agente pode performar;
- `param H{SUBTASKS,ROLES}, binary:` quais papéis cada subtarefa requer;
- `param C, integer:` número total de capacidades;
- `param R, integer:` número total de papéis;

Além disso, foi necessário modificar o gerador de modo que ele preencha os valores destes parâmetros. Estas informações já eram armazenadas nele, uma vez que tais relações são repassadas para os arquivos referentes aos agentes, organização e alocador.

3.2. Testes

Para a execução de testes, o gerador foi modificado de modo a gerar em paralelo, para cada configuração, um arquivo de entrada apenas com as restrições já presentes anteriormente e um com as novas restrições adicionadas. Foram então gerados 10 conjuntos de entradas com as respectivas configurações:

agents	*tasksL*	*tasksN*	*tasksS*	*limit*	*aijMax*	*roles*	*capabilities*	*capRole*
3	3	3	3	6	6	4	4	3

Todas as entradas geradas foram então executadas no GLPK e seus resultados analisados.

Inicialmente, foram gerados novos arquivos adicionando-se apenas as restrições C7, C8, C9 e C10 e, para este conjunto, ambos arquivos nos modelos novo e antigo apresentaram os mesmos resultados. Este resultado era esperado, uma vez que a implementação anterior já possuía informação de quais agentes podem realizar quais subtarefas, através do valor de utilidade destas.

A seguir, foram adicionadas também as restrições C11 e C12, consideradas mais relevantes. Para este conjunto de testes, os arquivos novos contendo tais restrições obtiveram um resultado diferente dos arquivos no modelo antigo, apresentando valores menores

em seus objetivos máximos. Tal resultado não era esperado, uma vez que supunha-se que o conjunto de restrições anterior era suficiente para representar a informação de quais agentes podem executar quais tarefas. Este resultado precisa ser, portanto, melhor investigado e comparado com o resultado obtido pela solução proposta descentralizada, executada para a mesma configuração de entrada.

4. Conclusão e Trabalhos Futuros

As restrições presentes na formalização do problema no trabalho original se mostraram fáceis de transcrever para a sintaxe do GLPK. Os resultados para o problema de maximização dos valores utilidade precisam ser melhor investigados, a fim de averiguar se as restrições anteriores estavam adequadas, se a solução proposta apresenta resultados semelhantes e se as restrições C11 e C12 foram descritas corretamente na formalização do problema apresentada no trabalho original.

Referências

Basegio, T. L. and Bordini, R. H. (2017). *An Algorithm for Allocating Structured Tasks in Multi-Robot Scenarios*, pages 99–109. Springer International Publishing, Cham.

Boissier, O., Bordini, R. H., Hübner, J. F., Ricci, A., and Santi, A. (2013). Multi-agent oriented programming with jacamo. *Sci. Comput. Program.*, 78(6):747–761.

Scerri, P., Kannan, B., Velagapudi, P., Macarthur, K., Stone, P., Taylor, M., Dolan, J., Farinelli, A., Chapman, A., Dias, B., and Kantor, G. (2012). Flood disaster mitigation: A real-world challenge problem for multi-agent unmanned surface vehicles. In *Proceedings of the 10th International Conference on Advanced Agent Technology*, AAMAS'11, pages 252–269, Berlin, Heidelberg. Springer-Verlag.