Capítulo 9

Times Assíncronos

Jesmmer da Silveira Alves e Humberto José Longo*

Resumo: Times Assíncronos são organizações de *software* que visam a interação eficiente entre vários algoritmos para a resolução de problemas adequados à abordagem multi-algorítmica. Apesar do conceito simples e das evidentes vantagens deste método, é importante que o projetista escolha uma disposição de agentes e memórias adequada para melhor cooperação dos elementos no time e, com isto, melhor aproveitamento das estratégias utilizadas. Este capítulo descreve o conceito de Times Assíncronos, destacando suas principais características e vantagens, o projeto e a implementação de um *A-Team* mínimo.

Palavras-chave: Times assíncronos, *A-teams*, Multi-algorítmico.

Abstract: *Asynchronous Teams are software organizations aimed at allowing an efficient interaction among various algorithms. Despite its simple concept and its obvious advantages, it is important that the designer set an adequate agents and memories architecture for better cooperation of elements in the team and, thus, better use of the selected strategies. This chapter describes the concept of Asynchronous Teams, highlighting its key features and advantages, the project and implementation of a minimum A-Team.*

Keywords: *Asynchronous Teams, A-Teams, Multi-algorithmic.*

1. Introdução

Ao considerar-se a classe de problemas $\mathcal{NP}$-difíceis, observa-se que, para um grande espectro de problemas, são conhecidas heurísticas que fornecem soluções de boa qualidade para um certo conjunto de instâncias desses problemas. Mais ainda, quando o estudo concentra-se em casos particulares de um problema, algoritmos exatos e "rápidos" podem até existir. Por outro lado, estes algoritmos podem falhar, seja por não se adequarem à instância que se deseja resolver, seja pela dificuldade de se identificar que a instância corresponde a um caso particular.

Esses problemas, para os quais nenhum algoritmo conhecido é completamente satisfatório, têm motivado a proposição de métodos multi-algorítmicos de resolução. O princípio de tais métodos é que, para problemas com certo grau de dificuldade de resolução, algoritmos "simples" podem cooperar entre si de modo a gerar soluções de qualidade equivalente, ou mesmo superiores, à qualidade das soluções geradas por um algoritmo muito mais complexo. Esses algoritmos "simples", em geral, são heurísticas que implementam uma intuição sobre o problema e têm como característica básica um tempo reduzido de processamento.

Entretanto, a simples ideia de se combinar diferentes técnicas para a resolução de um problema não é suficiente para que se obtenha um algoritmo que garanta uma eficiência aceitável na resolução de qualquer instância do problema. A dificuldade está na interação entre diferentes algoritmos, pois nem sempre uma determinada combinação destes permite uma boa utilização do tempo disponível para o processamento, ou mesmo uma colaboração eficiente entre os algoritmos.

Esta dificuldade é reduzida quando utiliza-se uma arquitetura como a de Times Assíncronos (Souza, 1993; Talukdar, 1993). Estes são organizações de *software* que visam a interação eficiente entre vários algoritmos para a resolução de problemas adequados à abordagem multi-algorítmica. Um *A-Team* (do termo em inglês *Asynchronous Team*) é basicamente composto por agentes autônomos e repositórios ou memórias compartilhadas de soluções. Os agentes são algoritmos encapsulados com um protocolo de comunicação, mas que se comunicam de forma assíncrona, ou seja, um agente não troca informações diretamente com outro agente, o que ocorre exclusivamente através das memórias compartilhadas.

A característica mais atraente do paradigma de *A-Teams* está na sua flexibilidade. Os agentes em um *A-Team*, ao atuarem de forma autônoma, são suscetíveis de serem introduzidos ou retirados da organização a qualquer momento. Outro ponto importante é que a comunicação assíncrona entre os agentes permite que

*Autor para contato: longo@inf.ufg.br

estes sejam executados em paralelo, com a comunicação entre eles gerando um fluxo contínuo de modificações no conteúdo das memórias compartilhadas.

O objetivo deste Capítulo é mostrar importância de *A-Teams* na resolução de problemas complexos da pesquisa operacional. Assim, a Seção 2, além de caracterizar os *A-Teams*, descreve os seus principais componentes e uma representação gráfica usual para os mesmos; a Seção 3 descreve as principais etapas que devem ser observadas durante o projeto de um *A-Team*; a Seção 4 descreve os passos básicos para implementação de um *A-Team* através do padrão MPI e os elementos básicos de um *A-Team* Mínimo; a Seção 5 apresenta a aplicação deste paradigma aos mais diferentes problemas encontrados na literatura; e a Seção 6 expõe as considerações finais dos autores sobre a utilização de *A-Teams*.

2. Times Assíncronos

2.1 Conceitos básicos

Um Dado pode ser definido como qualquer elemento com algum significado. Quando um dado é manipulado, processado ou organizado, passa a ser considerado uma Informação. Aqui serão utilizados os termos Solução Viável para referenciar dados processados que satisfaçam requisitos pré-definidos de algum problema, Solução Ótima para referenciar uma solução viável que satisfaça da melhor forma possível todos os requisitos pré-definidos de um problema, Solução Ideal para referenciar uma solução ótima ou suficientemente próxima da ótima e Solução Promissora para referenciar uma solução com possibilidade de se tornar uma solução ideal.

O conjunto de soluções ideais para problemas multi-objetivos também é conhecido como Pareto Ótimo ou Não-Dominadas. O termo não-dominadas é devido ao fato de tais soluções possuírem critérios que não são dominados por nenhum outro critério presente em soluções no conjunto. Quando um conjunto de soluções possui critérios individuais que não podem ser otimizados sem o declínio de outros critérios, diz-se que tal conjunto pertence à fronteira do Pareto Ótimo.

2.2 Caracterização de *A-teams*

Times Assíncronos ou *A-Teams* são organizações de *software* compostas por Agentes, Memórias e Relações (entre memórias). Os agentes são responsáveis pela geração de novas soluções ou manipulação de soluções nas memórias (inserir, eliminar ou modificar uma solução). As memórias são repositórios de soluções e são também a única forma de comunicação entre os agentes. As relações definem como as soluções fluem no *A-Team* e a forma de acesso de agentes ao conjunto de memórias compartilhadas (fluxo de soluções).

A ideia básica por trás dos *A-Teams* é considerar um conjunto inicial de soluções (soluções candidatas) e efetuar transformações nestas soluções até se alcançar um conjunto de soluções ideais. O agente, o principal elemento neste processo, encapsula um algoritmo "simples" e possui habilidades que determinam que tipo de tarefas pode executar. A combinação de agentes de mesmo tipo (mas que podem executar tarefas diferentes) constituem organizações, que são responsáveis pela forma como os agentes vão trabalhar, interagir e cooperar uns com os outros.

As principais características deste tipo de organização de *software* são (Souza, 1993):

Autonomia: inexistência de agentes supervisores. Embora possam existir diferentes classes de agentes, não pode haver relacionamento hierárquico entre eles e todos os agentes devem atuar de forma autônoma;

Comunicação Assíncrona: nenhum tipo de sincronismo é permitido na execução dos agentes, ou seja, os agentes não podem esperar pela execução ou processamento de soluções por outros agentes. Além disso, um agente não troca informações diretamente com outro agente, isto ocorre exclusivamente através das memórias compartilhadas;

Fluxo de Dados Cíclico: as soluções geradas e/ou modificadas por um agente são depositadas nas memórias e disponibilizadas a outros agentes para serem novamente modificadas (os dados de saída de um agente são dados de entrada para outro agente);

Consenso Gradual: no início uma grande variedade de soluções é considerada, mas após algum tempo de processamento o consenso emerge, restando somente algumas alternativas que são disponibilizadas a todos os agentes. O bom desempenho de alguns agentes pode direcionar o trabalho dos demais a um conjunto promissor de soluções;

Sinergia: a combinação de algoritmos de maneira cooperativa produz melhores resultados que a execução dos mesmos individualmente; e

Eficiência em escala: a quantidade de agentes e memórias no time influencia na qualidade e na velocidade em que soluções ideais são geradas.

Tais características lembram alguns sistemas naturais, como sociedades de insetos e comunidades celulares (Cicirello & Smith, 2001), muitas das quais apresentam estas mesmas características básicas ou uma boa parte delas. Nas colônias de formigas, por exemplo, a perda de um grupo pequeno de formigas não compromete a colônia, uma vez que todo o trabalho é feito de forma autônoma pelos membros da colônia. O trabalho em paralelo e independente de seus agentes (formigas) permite que a organização (colônia) atinja o seu objetivo (sobrevivência da colônia).

Algumas organizações de *software* podem apresentar um subconjunto destas características. Entretanto, nenhuma outra organização, além dos *A-Teams*, apresenta todas estas características ao mesmo tempo. Isto faz dos *A-Teams* uma importante ferramenta na busca de soluções de boa qualidade para problemas complexos, como problemas de otimização combinatória e problemas multi-objetivos, para os quais heurísticas aplicadas isoladamente não conseguem alcançar bons resultados.

2.3 Representação gráfica de *A-teams*

A representação gráfica mais usual para os elementos de um *A-Team* utiliza retângulos para memórias básicas e/ou composições destas e setas para os agentes. A direção das setas indica a direção do fluxo de soluções que são manipuladas pelos agentes.

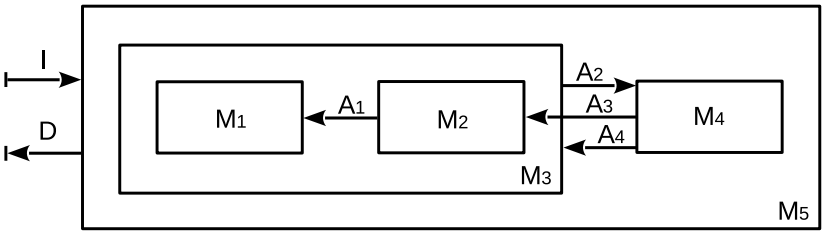


Figura 1. *A-Team* com 5 memórias e 6 agentes (setas representando agentes e retângulos representando memórias).

No exemplo da Figura 1 a memória M_3 é composta pelas memórias básicas M_1 e M_2, o agente A_1 lê da memória M_2 e escreve na memória M_1, A_2 lê de M_3 (de M_1 e/ou M_2) e escreve em M_4, A_3 lê de M_4 e escreve em M_2 e o agente A_4 lê de M_4 e escreve em M_3 (em M_1 e/ou M_2). Para alguns agentes com funções específicas não há necessidade de se representar a origem ou o destino das soluções e podem ser omitidos um desses ambientes, como os agentes I, que faz um preenchimento inicial das memórias, e D, que tem a função de eliminar soluções das memórias.

Esta representação gráfica evidencia os ambientes de entrada e saída dos agentes, ou seja, os conjuntos de memórias de onde e para onde os agentes podem ler e escrever soluções, respectivamente. Um mesmo ambiente pode servir como entrada e saída de soluções para um agente e ser um ambiente composto, como uma memória que possui outras memórias em sua estrutura. Neste caso, todos os agentes que têm acesso para leitura e/ou escrita a uma composição, também podem acessar todas as soluções em memórias internas à mesma. Tomando-se como exemplo o agente A_4, o seu ambiente de entrada é a memória M_4 e o seu ambiente de saída é composto pelas memórias M_1 e M_2.

Além dos agentes normalmente usados em um *A-Team*, como aqueles mostrados na Figura 1, há um conjunto de agentes utilizados durante a sua execução (agentes de manutenção) que, em geral, não são representados graficamente. O motivo é que esses agentes não são usados para resolver o problema em si, mas para realizar ações especificadas pelo projetista ou usuário durante a execução do *A-Team*, ou seja, ações como:

- Iniciar e parar o *A-Team*;
- Emitir relatórios sobre os elementos na estrutura; ou
- Ativar e/ou desativar elementos presentes no *A-Team*.

Uma representação gráfica para estes elementos é mostrada na Figura 2. Um retângulo tracejado é usado para representar todo o *A-Team*; setas unidirecionais são usadas para representar a aplicação de ações em um único sentido pelo projetista/usuário do *A-Team*; setas bi-direcionais representam ações nos dois sentidos, geralmente o projetista/usuário executa uma ação e aguarda o resultado do processamento da mesma pelo *A-Team*.

Figura 2. Representação de agentes de manutenção de um *A-Team*.

3. O Projeto de um *A-Team*

Alguns aspectos básicos, discutidos nas próximas seções, devem ser observados durante o projeto de um *A-Team* para permitir que diferentes formas de implementação, configuração e execução do mesmo sejam identificadas. Dentre esses aspectos, destacam-se a análise e decomposição do problema a ser resolvido, a definição dos parâmetros de configuração, a definição da topologia da rede de agentes e memórias, a garantia de fluxos cíclicos de dados entre os agentes e a sequência de execução dos agentes.

3.1 Análise e decomposição do problema

O primeiro passo no projeto de um *A-Team* é definir o problema a ser tratado. Nesta etapa são identificadas as características do problema, tais como suas variáveis, domínios e restrições. Depois disto, é necessário identificar os resultados esperados na execução do *A-Team*. Tais resultados não necessariamente devem ser ótimos, pois dependendo do tipo do problema o usuário pode ter que selecionar um conjunto de soluções que considerar ideais. Em problemas multi-objetivos, por exemplo, diversas funções objetivo devem ser satisfeitas e dificilmente se encontra uma única solução que satisfaça cada uma das funções objetivo.

Dada a especificação de um problema e a correta identificação do padrão de qualidade desejado, uma tarefa óbvia na tentativa de diminuir a complexidade é tentar decompor o problema original em subproblemas, ou relaxações do mesmo. A decomposição do problema e a diversificação dos algoritmos utilizados permite que projetistas manipulem propriedades que podem influenciar a eficácia do projeto em desenvolvimento.

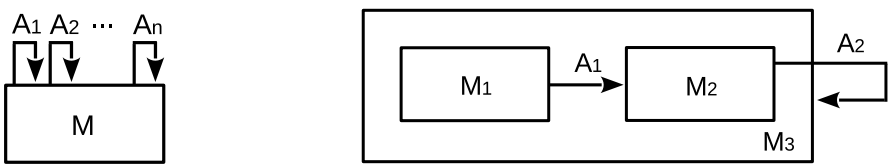
Esta é uma tarefa bastante específica e difícil de formalizar, pois depende muito das características do problema a ser resolvido. Cada decomposição pode ser associada a uma ou mais memórias, desta forma, projetistas podem fazer uso dos diversos métodos na literatura para resolver cada decomposição. Entretanto, dois critérios importantes devem ser observados, a habilidade em prover soluções e a satisfação das restrições especificadas por cada subdivisão do problema original.

3.2 Topologia do *A-team*

A topologia do *A-Team* é definida pela disposição dos agentes e memórias e pelo fluxo de dados entre esses elementos, e pode influenciar diretamente o desempenho do mesmo. É evidente que diferentes topologias permitem que diferentes sequências de agentes sejam aplicadas sobre as soluções nas memórias e, consequentemente, diferentes resultados podem ser alcançados. Ainda mais, o conjunto de agentes pode ser aplicado de diversas formas, dentre elas:

- Um time formado por várias cópias do mesmo algoritmo, possivelmente com diferentes configurações para os seus parâmetros;
- Um time formado por vários tipos de algoritmos. Neste caso, geralmente os algoritmos possuem funções distintas; e
- Uma terceira alternativa seria a combinação das duas opções anteriores, ou seja, vários tipos de algoritmos com várias cópias de cada tipo de algoritmo.

Uma topologia básica contém uma única memória e o conjunto de agentes trabalham sobre as soluções desta memória (Figura 3(a)). Esta topologia, embora simples, pode ser bastante útil e bem fácil de projetar. Entretanto, todo o conjunto de soluções nesta memória deve possuir a mesma representação (estrutura).



(a) (b)

Figura 3. Estrutura de um *A-Team* com (a) uma única memória e (b) mais de uma memória.

Quando as soluções têm a mesma representação, é evidente a possibilidade de se usar uma única memória. Entretanto, a população de soluções pode ter diferentes representações e podem ser armazenadas em diferentes

memórias, dependendo da aplicação e dos algoritmos a serem utilizados. A opção por utilizar mais de um repositório de dados, na maioria das vezes, é escolhida através de um estudo das características tanto dos subproblemas e/ou relaxações do problema original, quanto das características das soluções produzidas pelas heurísticas disponíveis.

No exemplo da Figura 3(b), uma solução armazenada na memória M_1 somente estaria disponível ao agente A_2, através da memória M_2, após ser processada pelo agente A_1. Outra característica importante desta estrutura é que uma solução que acabou de ser trabalhada pelo agente A_1 não será mais selecionada pelo mesmo antes de ser trabalhada pelo agente A_2. Esta característica é bastante relevante quando se utiliza agentes de comportamento determinístico, pois evita o processamento de uma solução que acabou de ser gerada.

No exemplo da Figura 4, o deslocamento de uma solução ou conjunto de soluções da memória M_1 para a memória M_2, é resultante da seguinte sequência de transformações:

- zero ou mais transformações pelo agente A_1;
- uma transformação pelo agente A_2; e
- zero ou mais transformações pelo agente A_3.

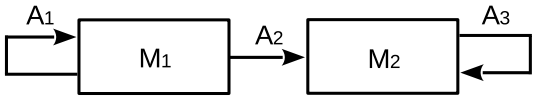


Figura 4. Controlando a sequência de transformações.

Outras estruturas diferentes, com características peculiares, podem ser utilizadas no projeto do *A-Team*. O projetista pode, por exemplo, usar a própria estrutura para restringir a leitura e/ou a escrita (de soluções) a um conjunto limitado de soluções (Figuras 5(a) e 5(b)).

(a) (b)

Figura 5. Restrição de leitura e/ou escrita a um conjunto limitado de soluções: (a) leitura de uma memória e (b) escrita em uma memória.

Em algumas situações o projetista pode ter a necessidade de garantir o processamento de uma determinada solução por um agente ou conjunto de agentes. No exemplo da Figura 6, suponha que M_4 trabalha com uma estrutura sequencial de organização das soluções (*first in - first out*). Neste caso, qualquer solução oriunda das memórias M_1 e/ou M_2, ao ser processada pelos agentes A_1 e A_2, deve ser processada pelo agente A_3 antes de novamente ser colocada a disposição dos dois primeiros agentes. Note-se que agora, diferentemente da situação mostrada na Figura 1, não há necessidade de atuação dos agentes I e D na memória M_4.

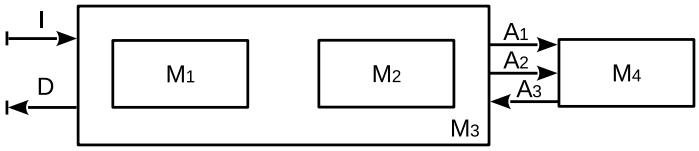


Figura 6. Garantia de processamento de uma solução por um agente ou conjunto de agentes.

O atendimento de vários requisitos previamente definidos, como a restrição de acesso a soluções pelos agentes ou a obrigatoriedade de se realizar uma sequência de transformações nas soluções armazenadas nas memórias, pode ser necessário ao se definir a topologia de um *A-Team*. O projetista pode usar as relações de processamento entre pares de agentes no atendimento de tais requisitos. A relação de processamento entre um par de agentes descreve qual dos agentes do par, possivelmente, trabalhará com soluções produzidas ou manipuladas pelo outro agente. O conjunto destas relações de processamento também define todas as possíveis sequências de transformações nas soluções.

Uma representação gráfica das relações de processamento (somente com agentes construtores e modificadores) do *A-Team* da Figura 1 pode ser visto na Figura 7. Cada seta representa um relacionamento entre dois agentes, sendo que a direção de uma seta determina uma possível sequência de processamento de uma solução.

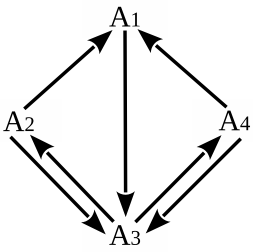


Figura 7. Exemplo de relações de processamento entre pares de agentes.

3.3 Fluxo de dados cíclico

A sequência com que as soluções fluem determina a sequência de transformações efetuadas pelos agentes e a forma como os agentes vão interagir e cooperar uns com os outros. Este fluxo de soluções das memórias para os agentes, e vice-versa, deve ser Cíclico, de modo que as alterações realizadas por um agente sejam depositadas nas memórias e novamente disponibilizadas a outros agentes do *A-Team*.

Um fluxo de dados é Fortemente Cíclico se todas as possíveis sequências de execução dos agentes formam laços (caminhos fechados). Garantir em um *A-Team* o fluxo de dados cíclico, ou fortemente cíclico, é permitir que todos os agentes troquem informações entre si e consequentemente cooperem uns com os outros, ou seja, a interação e cooperação são ações possíveis entre todos os agentes do time. Um dos modos de garantir a cooperação entre agentes é definir a topologia sempre se preocupando em construir memórias cíclicas. Uma memória M é uma Memória Cíclica se todos os agentes que podem escrever (depositar soluções) em M também podem ler soluções de M (Talukdar et al., 1998).

Claramente, memórias em que todos os agentes possuem o espaço de entrada igual ao espaço de saída são memórias cíclicas (ver Figura 3(a)). Ainda mais, para dois agentes, A_1 e A_2, e duas memórias, M_1 e M_2, se o espaço de entrada de A_1 (M_1) for igual ao espaço de saída de A_2, e reciprocamente, o espaço de entrada de A_2 (M_2) for igual ao espaço de saída de A_1, ambos os agentes formam um laço e ambas as memórias são memórias cíclicas (Figura 8).

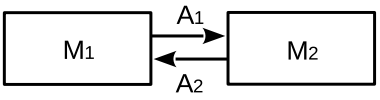
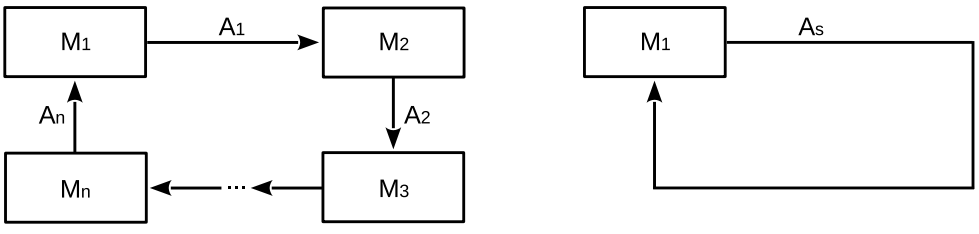


Figura 8. Memórias M_1 e M_2 cíclicas, com agentes A_1 e A_2 em um laço.

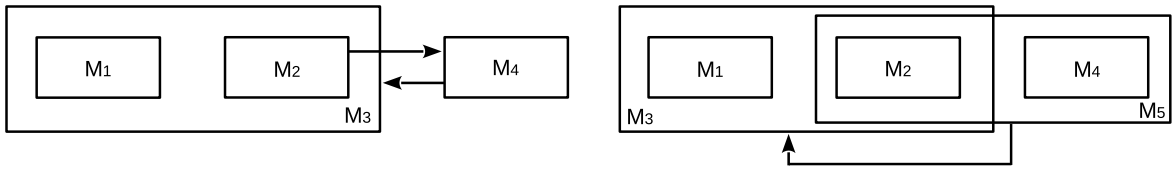
Em estruturas mais complexas, com um número considerável de memórias e agentes, o projetista pode garantir o fluxo de dados cíclico. Neste caso, para todo agente no fluxo de dados, deve existir um caminho $P(M, M)$ em que o espaço de entrada do primeiro agente no caminho é a memória M e o espaço de saída do último agente no caminho também é a memória M (ver Figura 9(a)). Se esta proposição for verdadeira para todos os agentes no fluxo de dados, a topologia construída corresponde a um fluxo de dados cíclico. A argumentação para isto é a seguinte: se para todo agente no fluxo de dados existe um caminho $P(M, M)$ que pode ser substituído por um super-agente hipotético A_s, que é a concatenação de todos os agentes no caminho, então a memória M é uma memória cíclica (ver Figura 9(b)).



(a) (b)

Figura 9. *A-Team* com memória cíclica representada por (a) caminho $P(M_1, M_1) = (A_1, A_2, \ldots, A_n)$ e (b) super-agente hipotético $A_s = A_1 + A_2 + \cdots + A_n$.

Apesar de, na maioria das vezes, não ser uma tarefa muito árdua verificar se todos os agentes do *A-Team* formam um laço, algumas representações podem dificultar a análise e até mesmo passar uma visualização "enganosa" para o projetista. Este é o caso de composições e intersecções entre memórias (na Figura 10(b), M_3 é a composição das memórias M_1 e M_2, M_2 é a intersecção das memórias M_3 e M_5), que podem dificultar a verificação do fluxo de dados cíclico. No exemplo da Figura 10(a) soluções nunca são lidas de M_1 e no exemplo da Figura 10(b) soluções nunca são lidas de M_1 ou escritas em M_4. Em ambos os casos, não se pode garantir o fluxo de dados cíclico no *A-Team*.



(a) (b)

Figura 10. Quebra do fluxo cíclico de soluções: (a) soluções nunca são lidas de M_1 e (b) soluções nunca são lidas de M_1 ou escritas em M_4.

3.4 Inicialização das memórias

A diversidade de soluções em uma memória é caracterizada pelas diferenças entre as mesmas, sendo que baixa diversidade representa um conjunto de soluções com poucas diferenças entre si e alta diversidade representa um conjunto de soluções bem distintas umas das outras. O percentual de preenchimento inicial das memórias é um atributo que pode influenciar muito o desempenho de um *A-Team* e está diretamente relacionado ao nível de diversidade das soluções.

Com o objetivo de aumentar a diversidade poderia se pensar em um preenchimento total da memória, mas, dependendo da política de exclusão/inclusão adotada, muitas soluções podem ser substituídas antes mesmo de serem processadas por algum agente. Entretanto, a inicialização com pouco uso do tamanho disponível da memória pode reduzir muito a diversidade das soluções e prejudicar o desempenho e a convergência do *A-Team*.

Uma opção aceitável é preencher inicialmente apenas um certo percentual do tamanho da memória e deixar o restante do espaço livre para a inserção de novas soluções durante o processamento do *A-Team*. Isto porque, após a inicialização, os agentes utilizam as soluções disponíveis nas memórias compartilhadas para produzir novas soluções, as quais são novamente colocadas a disposição dos agentes do *A-Team*, inclusive de quem as gerou. Um objetivo básico no projeto de um *A-Team* é estabelecer este percentual, de modo que se mantenha alta a diversidade na memória mas sem tornar lento o processo de convergência para soluções ideais.

3.5 Agentes

Os agentes em um *A-Team* são os elementos responsáveis por criar, manipular e remover soluções das memórias, podendo ser classificados da seguinte forma:

Iniciadores: agentes designados a fazer o preenchimento inicial das memórias no *A-Team* (ver agente I na Figura 1). Estes criam novas soluções a partir da definição do problema que está sendo resolvido, utilizando ou não informações localizadas em diferentes memórias;

Construtores: tais agentes têm a função de gerar novas soluções durante a execução do *A-Team*. Como estes têm essencialmente o mesmo comportamento que os agentes iniciadores, ambos podem até mesmo ser agrupados em uma única categoria;

Modificadores: tais agentes têm função específica de tentar melhorar a qualidade das soluções nas memórias. Uma solução processada por este tipo de agente é depositada novamente na memória, possivelmente substituindo a solução antiga; e

Destrutores: o conteúdo das memórias deve ser constantemente monitorado para evitar que as mesmas recebam uma quantidade indesejada de soluções, este controle é feito por agentes Destrutores (ver agente D na Figura 1). Este agente é usado para remover as soluções que não sejam promissoras na busca de soluções ideais, ou seja, decidir quando soluções devem ser eliminadas de um espaço de memória ou não.

O trabalho dos agentes iniciadores, se não for bem elaborado, pode ter um efeito negativo durante a execução do *A-Team*. Um preenchimento inicial deve evitar gerar soluções limitadas a um subconjunto do espaço de soluções, pois pode levar a uma baixa diversidade inicial. Um *A-Team* pode conter vários agentes iniciadores, construtores, modificadores e destrutores trabalhando em memórias distintas e manipulando soluções intermediárias. Todos os resultados produzidos pelos agentes, exceto os destrutores, são disponibilizados aos outros agentes via memórias.

A estrutura de um *A-Team* pode contar não só com memórias com soluções viáveis para o problema em questão, mas também com memórias que armazenam soluções parciais ou infactíveis, o que expande o número de possíveis algoritmos passíveis de serem utilizados. Dado que as possíveis soluções para determinado problema podem ser bem distintas umas das outras, o projetista pode, por exemplo, fazer uso de algoritmos de

consenso para aproveitar as similaridades ou as diferenças entre soluções, construindo novas soluções parciais ou infactíveis. Tais algoritmos podem usar duas ou mais soluções para gerar uma ou mais soluções novas, incorporando informações de todas as entradas (soluções lidas nas memórias). Os critérios para criar novas soluções podem ser baseados na intersecção, quando determinados atributos coincidem (assumem os mesmos valores) em várias soluções lidas (ver Figura 11(a)), e na diferença, quando os atributos diferem nas soluções lidas (Figura 11(b)).

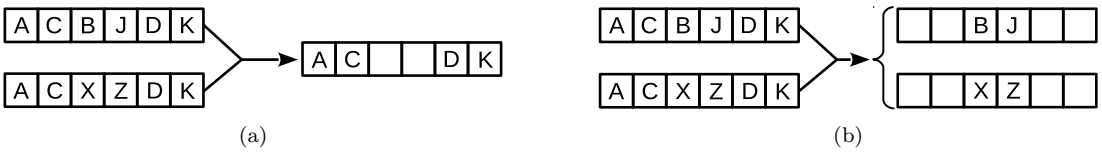


Figura 11. Novas soluções criadas a partir de alternativas de critérios de consenso: (a) intersecção e (b) diferença de soluções (Souza, 1993).

Uma possibilidade deste processo é mostrado na Figura 12. Agentes de consenso, *Deconstrutores*, lêem duas ou mais soluções factíveis na memória *Soluções Completas* para gerar soluções infactíveis, através das características em comum destas soluções, e então armazena o resultado na memória *Soluções Parciais*. Agentes *Construtores*, também baseados em algoritmos de consenso, fazem o processo inverso.

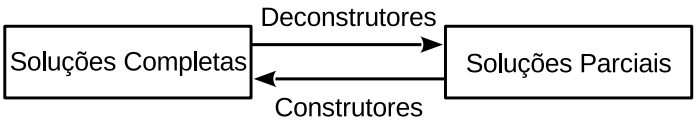


Figura 12. Exemplo de uso de agentes de consenso para decomposição de soluções viáveis em soluções parciais e vice-versa (Souza, 1993).

3.6 Políticas de seleção

Algumas características podem influenciar o desempenho do *A-Team* e devem ser consideradas, como a diversidade das soluções. Uma baixa diversidade pode acarretar em uma rápida convergência do processo, o que acontece quando nemhum melhoramento pode ser alcançado sobre as melhores soluções, e consequentemente prejudicar a busca pelo conjunto de soluções ideais. Entretanto, alta diversidade pode representar uma lenta convergência do processo, podendo levar um grande tempo antes que uma solução ideal seja descoberta.

Outra característica importante é a forma como os agentes se comportam ao selecionar novas soluções. Quando trabalhando com agentes de comportamento determinístico, deve-se evitar que uma solução que acabou de ser gerada seja selecionada novamente pelo mesmo agente que a gerou, pois tal solução não sofrerá mudança em sua estrutura (ver Figura 3(b), onde uma solução processada pelo agente A_1 não é mais selecionada pelo mesmo antes de ser processada pelo agente A_2).

O efeito negativo destas características pode ser minimizado com a escolha correta da política de seleção e devida associação aos agentes na estrutura. Uma política de seleção representa um conjunto de regras a serem consideradas quando é preciso selecionar soluções da memória, seja para processamento pelos agentes construtores e modificadores ou para serem eliminadas pelos agentes destrutores. Existem várias destas políticas, a maioria delas adaptadas a problemas específicos. As três principais são:

$\mathbf{P}_1$ – Seleção "gulosa": uma estratégia que sempre pega a melhor ou a pior solução candidata na memória;

$\mathbf{P}_2$ – Seleção com distribuição uniforme de probabilidade: neste caso todas as soluções na memória têm a mesma probabilidade de serem selecionadas, independente de quão boas sejam para o problema em questão; e

$\mathbf{P}_3$ – Seleção com distribuição linear de probabilidade: as soluções possuem uma probabilidade crescente de serem selecionadas, podendo ser da pior para a melhor ou da melhor para a pior.

Se o objetivo do uso de tais políticas é a seleção de soluções para processamento pelos agentes e a política usada é uma estratégia com maior probabilidade de selecionar as piores soluções, então soluções não promissoras podem ser processadas antes de uma possível eliminação. Esta estratégia, em relação a uma com maior probabilidade de selecionar as melhores soluções, pode reduzir o tempo de convergência para soluções ideais e manter maior diversidade nas memórias. Desta forma, o uso da política P_2 pode ser interessante quando não se sabe quais soluções candidatas na memória são consideradas promissoras;

O outro objetivo básico do uso de tais políticas é a seleção de soluções para serem eliminadas pelos agentes destrutores. Este objetivo é tão importante quanto o anterior, já que ambos devem ser considerados para o

balanceamento entre construção e destruição das soluções candidatas e manutenção da diversidade de soluções na memória. Entretanto, memórias que funcionam como estruturas simples de armazenamento não necessitam de uma política de seleção complexa. Uma organização simples do tipo *FIFO* (*First In, First Out*), em geral, é suficiente para controlar a quantidade de soluções neste tipo de memória.

A forma de implementação da política de seleção pode revelar características interessantes e oferecer ao projetista diferentes estratégias para selecionar soluções nas memórias. Para executar tal tarefa, o projetista pode escolher entre implementar:

- uma política para cada memória (todos os agentes que a acessam utilizam a mesma política); ou
- políticas diferentes para cada agente (diferentes políticas podem ser implementadas para diferentes agentes na mesma memória).

Quando o objetivo é selecionar uma solução a ser processada por agentes construtores ou modificadores, uma opção bastante usada é associar a cada agente uma política de seleção em particular. Entretanto, quando o objetivo é selecionar uma solução a ser eliminada, a maioria dos projetistas implementa uma única política para cada memória. Uma possível justificativa para este fato é que as memórias no *A-Team* geralmente trabalham com um único agente destrutor.

Um aspecto importante a ser considerado na implementação de políticas de seleção é a definição de quando uma nova solução deve ser incorporada ao conjunto de soluções factíveis (soluções promissoras). No caso de problemas multi-objetivos, por exemplo, dois critérios básicos para a inserção de novas soluções em uma memória são (Rodrigues, 1996):

- a solução original não domine a solução gerada a partir dela; ou
- a solução gerada domine a solução original.

No primeiro caso a solução deve pertencer ao mesmo conjunto de Pareto da solução original ou a uma camada de Pareto mais inferior. O segundo caso exige que os valores da solução gerada tenham necessariamente que ser melhores do que os valores da solução original para todas as funções objetivo, o que pode restringir muito a diversidade das soluções.

Outro aspecto importante é o controle do tamanho da população de soluções. Embora a maioria das pesquisas feitas com *A-Teams* determine que alguma estratégia deve ser usada para controlar a quantidade de soluções na memória quando novas soluções são geradas, este tipo de controle às vezes não é necessário. Ao invés de gerar uma nova solução e então encontrar uma posição na memória para adicioná-la, pode-se pensar em simplesmente substituir a última solução (solução selecionada pelo agente) pela nova solução gerada. Este tipo de estratégia torna desnecessária a preocupação com o tamanho da população de soluções e consequentemente não requer o uso de agentes destrutores (Salman et al., 2002).

4. A Implementação de *A-Teams*

Em uma implementação de *A-Teams* segundo uma arquitetura cliente-servidor, os agentes funcionam como clientes e as memórias como servidores de dados. Os clientes (agentes) requisitam informações a processos que atuam como servidores (memórias), através de chamadas a procedimentos remotos. Assim, usando-se este modelo de implementação, todos os algoritmos (clientes e servidores) podem ser executados concorrentemente em um computador, com um ou mais processadores, ou distribuídos em uma rede, com um computador alocado para cada algoritmo. Isto sem esforço adicional de programação (alteração de código). Alguns dos fatores que justificam a escolha desta estrutura são a inexistência de restrições quanto ao número de clientes e servidores; a garantia de integridade dos dados nas memórias, decorrente do tratamento, pelo servidor, das requisições dos clientes segundo a política FIFO (*First In, First Out*); e a existência de várias ferramentas de programação e suporte a esta abordagem.

Aplicações complexas, como uma resultante da implementação de um *A-Team*, em geral, exigem elevado grau de processamento, resultando num tempo de resposta muito grande quando colocadas para serem executadas em sistemas convencionais (um computador com poucos processadores). Desta forma, o ideal é que o código a ser desenvolvido para um *A-Team* utilize facilidades de programação e suporte a aplicações distribuídas oferecidas por diversas ferramentas atualmente disponíveis. Na Seção 4.1 é descrita, brevemente, uma das várias interfaces de programação que permitem a comunicação entre diversas partes de uma aplicação (processos) que estejam dispersas em um ambiente distribuído. Na Seção 4.2 são discutidos alguns dos aspectos da implementação de um *A-Team* usando-se esta ferramenta.

4.1 MPI (*Message Passing Interface*)

O padrão MPI é o resultado do esforço conjunto entre várias organizações (usuários e vendedores de sistemas paralelos) para o desenvolvimento de uma interface padrão de troca de mensagens. No padrão resultante destacam-se, dentre outros aspectos, a portabilidade de código fonte, a implementação eficaz em uma série de arquiteturas, a existência de conjunto extenso de funcionalidades, o suporte para arquiteturas paralelas heterogêneas, a existência de inúmeras implementações gratuitas e uma comunidade ativa de desenvolvedores em torno do padrão.

O padrão MPI foi desenvolvido a partir do paradigma *Message-Passing*, ou troca de mensagens, no qual várias instâncias de processadores, cada um com sua própria memória, são vistas conjuntamente e cooperam entre si na resolução de tarefas. Tal cooperação é conquistada com a comunicação entre os processos através do envio e recebimento de mensagens, invocando-se rotinas definidas no MPI para isso. Qualquer conceito de comunicação fora desta ideia, como processos acessando diretamente memórias de outros processos, não faz parte do modelo. Este padrão tem como base quatro conceitos principais:

Processo: cada parte, em execução, de um programa, as quais podem estar executando distribuídas em uma ou mais diferentes máquinas;

Grupo: um conjunto de processos. Inicialmente, todos os processos pertencem a um único grupo que está associado a um comunicador;

Rank: número inteiro que identifica unicamente um processo em um grupo. Esta identificação varia de zero até $N - 1$, onde N é o número de processos do grupo; e

Comunicador: escopo de uma operação de comunicação. Define o grupo de processos que podem se comunicar e viabiliza a troca de mensagens entre os mesmos.

De forma geral, um programa em MPI faz uso de rotinas de inicialização e finalização da biblioteca e também de rotinas de controle. Estas últimas são utilizadas para a obtenção de informações sobre o ambiente paralelo/distribuído que está em execução, tais como a identificação de um processo ou a quantidade de processos em um grupo.

Uma mensagem em MPI é composta de duas partes: dados (informações que se deseja enviar e receber) e envelope (informação da rota dos dados). Um dado é um conjunto de três parâmetros: endereço de sua localização (a posição de memória em que está alocado), número de elementos que serão enviados na mensagem e o seu tipo (o seu tipo de dado em MPI). Um envelope é formado pela identificação do processo remetente, a identificação do processo destinatário, um rótulo identificador da mensagem e o comunicador que irá viabilizar o procedimento.

As mensagens em MPI podem ser enviadas de um processo para outro, na chamada comunicação ponto a ponto, ou de um processo para todos os outros do seu grupo, representando a comunicação coletiva. Quando um grupo de processos é iniciado, um comunicador predefinido é criado (inter-comunicador) e a troca de mensagens naquele grupo passa a ser permitida.

O MPI permite a gestão dinâmica de processos (criação, inserção em um comunicador e comunicação com o restante dos grupos). Para viabilizar tal procedimento, são disponíveis rotinas para iniciar um conjunto de processos de um único programa e para iniciar conjuntos de processos de diversos programas. Durante a execução de uma aplicação MPI, quando um grupo cria um outro grupo de processos, ele é considerado o grupo pai e o grupo iniciado dinamicamente é considerado o grupo filho. Um novo comunicador passa a existir (inter-comunicador) e, assim, a troca de mensagens entre pai e filho é viabilizada. Entretanto, em alguns casos, é necessário mais do que a comunicação entre grupo pai e grupo filho. Tal situação pode ser ilustrada quando dois grupos filhos precisam trocar mensagens entre si. Neste cenário, apesar da existência de um mesmo pai, não existe nenhum comunicador entre eles.

Para tais situações o MPI oferece, como alternativa de estabelecimento de comunicação entre processos, rotinas baseadas no modelo cliente-servidor, ou seja, processos aceitando e iniciando conexões com outros processos. Este procedimento acontece da seguinte forma: os processos escolhidos para serem servidores abrem uma porta e publicam um nome para ela, podendo, a partir daí, serem encontrados. Os processos clientes procuram por um nome de servidor, obtêm a porta correspondente e iniciam uma conexão com a mesma. O resultado desta sequência de ações é um novo inter-comunicador, que permite a troca de dados até que a conexão seja desativada. A troca de mensagens continua acontecendo da mesma forma, ou seja, através das rotinas de envio e recebimento.

4.2 Um *A-team* mínimo

Na Figura 13 é mostrado um exemplo de um *A-Team* mínimo, formado pelos componentes `iniM`, `M`, `agM`, `paraAteam` e `listaM`. Este pequeno conjunto de componentes é bastante relevante, pois contém todos os elementos (memória, agente e iniciador de memória) de um *A-Team*. Além disso, ele implementa a estrutura básica de qualquer *A-Team*, ou seja, uma memória que recebe, armazena e envia soluções para os agentes (`M`), um agente que requisita, modifica e envia soluções para a memória (`agM`), e agentes responsáveis por iniciar (`iniM`) e parar (`paraAteam`) o *A-Team* e listar o conteúdo corrente da memória (`listaM`).

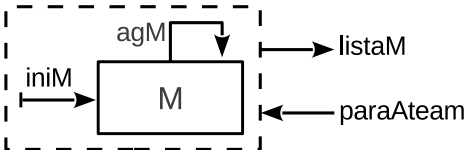


Figura 13. *A-Team* mínimo.

Neste caso, a memória `M` é iniciada com uma quantidade determinada de soluções e o agente `agM` (uma ou mais cópias dele) tenta melhorar a qualidade das soluções em `M`. Dado que os agentes implementados são modificadores, é dispensável o uso de políticas de destruição e agentes destrutores para o controle da quantidade de soluções armazenadas na memória.

Adicionalmente, uma interface gráfica poderia ser incluída no projeto, para proporcionar ganhos como a alteração dos parâmetros de configuração do *A-Team* em tempo de execução, um maior controle sobre os processos relativos aos agentes distribuídos na rede ou uma melhor forma de visualização dos resultados obtidos. Além disso, a interface pode ser encarregada de iniciar dinamicamente cada agente e cada memória.

A Figura 14 mostra um modelo de arquitetura cliente/servidor, para o exemplo mínimo da Figura 13, já com uma interface gráfica. Para a implementação desta arquitetura, a interface gráfica deve contar com um comunicador direto definido para a troca de mensagens com todos os demais componentes do *A-Team*, uma vez que a mesma será responsável, durante a execução do *A-Team*, por iniciar dinamicamente cada agente e cada memória. Por outro lado, a comunicação entre os agentes e a memória necessita do estabelecimento de uma conexão, já que eles fazem parte de grupos de processos diferentes.

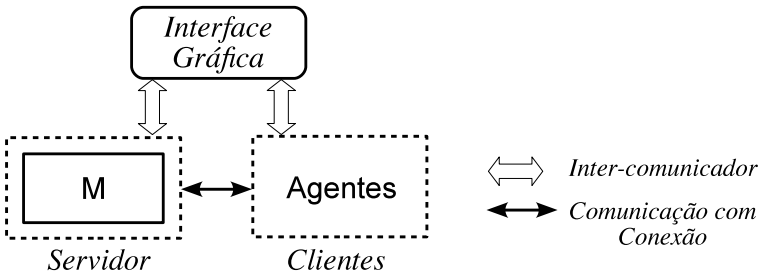


Figura 14. Arquitetura cliente/servidor com interface gráfica para um *A-Team* mínimo.

O servidor deve ficar sempre à espera de requisições de conexões. Além disso, o servidor deve ser *multithread*, para que vários agentes possam ter suas requisições atendidas ao mesmo tempo, e deve ser incluído algum mecanismo de controle de concorrência, para garantir a integridade dos dados armazenados na memória. Desta forma, cada *thread* que precise executar a operação de escrita na memória entra em uma fila de espera para obter o acesso a tal rotina e, só após realizar a atividade, ela retorna para seu fluxo de execução.

Para finalização do execução do *A-Team*, o agente `paraAteam` deve enviar uma mensagem de parada para ao servidor (tal tarefa pode ser realizada a partir da interface gráfica). Recebida a mensagem, o servidor a envia para os agentes e continua ativo. Os agentes, por sua vez, finalizam sua execução, desconectam-se do servidor e enviam uma mensagem de aviso de paralização para o agente `paraAteam`. Este, após ser notificado de que todos os agentes já finalizaram, envia uma última mensagem para o servidor, de finalização imediata.

Agora descreveremos como ocorre em MPI a comunicação com conexão entre componentes do *A-Team* (troca de mensagens entre um emissor e um receptor). No exemplo a seguir é demonstrado o empacotamento/envio de uma mensagem com o conteúdo `INIT_VARS` por um emissor e o seu recebimento/desempacotamento pelo receptor.

Inicialmente, a mensagem é empacotada em um *buffer* de envio. A chamada à rotina `MPI_Pack` empacota o dado contido na variável *method*, ou seja, `INIT_VARS`. Além deste dado, consta também a quantidade de dados que será empacotada, o tipo destes dados, o *buffer* de envio, o tamanho em *bytes* do *buffer* de envio, a posição no *buffer* em que o dado será empacotado e o comunicador para a troca da mensagem:

```
int method = INIT_VARS;
char buffer[1];
position = 0;
MPI_Comm commServerMD;
MPI_Pack(method, 1, MPI_INT, buffer, 1, position, commServerM);
```

O envio da mensagem para o servidor M, através do comunicador commServerM, pode ser feita com a chamada à rotina MPI_Send. Na chamada a esta rotina os parâmetros são os dados a serem enviados, o número de elementos contidos no *buffer*, os tipos dos dados, o *rank* do processo que irá receber a mensagem, o identificador da mensagem e o comunicador para a troca da mensagem:

```
MPI_Send(buffer, position, MPI_PACKED, 0, 1, commServerM);
```

A mensagem é recebida pelo receptor através da rotina MPI_Recv. Os seus parâmetros são o endereço inicial do *buffer* de entrada, o tamanho do *buffer* de entrada, o tipo de dado recebido, o *rank* do processo que enviou a mensagem, o identificador da mensagem, o comunicador por onde a mensagem chegará e uma variável que armazena dados da situação de recebimento:

```
char buffer[1];
MPI_Comm newCommClient;
MPI_Status status;
MPI_Recv(buffer, 1, MPI_PACKED, MPI_ANY_SOURCE, MPI_ANY_TAG,
         newCommClient, status);
```

Uma vez recebida a mensagem, o receptor deve desempacotá-la para para ter acesso ao valor realmente enviado (INIT_VARS). A rotina MPI_Unpack desempacota os dados contidos na mensagem e a variável *method* passa a conter o valor inteiro INIT_VARS, completando a troca de mensagens. Os seus parâmetros são o endereço inicial da mensagem a ser desempacotada, o tamanho em bytes da mensagem, a posição em *bytes* que se deve desempacotar, o local para o armazenamento do valor desempacotado, a quantidade de dados que deve ser desempacotada, o tipo do dado que será desempacotado e o comunicador por onde a mensagem chegou:

```
int position = 0;
int method = 0;
MPI_Unpack(buffer, 1, position, method, 1,
           MPI_INT, newCommClient);
```

No caso de um *A-Team* com mais componentes, grande parte do trabalho constitui-se apenas na replicação da solução adotada neste subconjunto de elementos para os demais, considerando-se os ascpectos de projeto destacados na Seção 3.

5. Aplicações de *A-Teams*

O modelo de *A-Teams* tem sido aplicado na resolução de uma grande variedade de problemas oriundos da Pesquisa Operacional. Além disso, vários desses estudos procuraram validar aspectos e/ou características importantes do modelo. Um dos estudos pioneiros foi conduzido por Souza (1993) sobre a resolução do problema do Caixeiro Viajante (*TSP - Traveling Salesman Problem*). Este trabalho identificou a importância da correta definição da estrutura dos times na busca pela eficiência em escala. Rodrigues (1996) mostrou que o modelo de *A-Teams* é eficiente na detecção do pareto ótimo ou quase ótimo de instâncias de uma generalização do *TSP*, o Problema do Caixeiro Viajante com Várias Matrizes de Distância (*MDTSP - Muti-Distance Traveling Salesman Problem*).

Outro problema clássico é o de escalonamento de tarefas (*FSP - Flow Shop Problem*). Neste caso, o objetivo é encontrar uma sequência de execução de tarefas (*jobs*) em um conjunto de máquinas, de forma a minimizar o tempo entre o início do processamento do primeiro *job*, na primeira máquina, e o término do último *job* na última máquina. Peixoto (1995) apresentou uma especificação de *A-Teams* para o *FSP* que destacou-se por encontrar eficiência em escala e *speed up* próximo do linear. Além disso, obteve resultados iguais ou melhores aos já conhecidos para o problema. No *LCSP* (*Labor Constraint Satisfaction Problem*) uma tarefa está sujeita a restrições de precedência e restrições de escala de trabalhadores. Cada tarefa tem um tempo de processamento especificado e um perfil de trabalho (que denota o número de trabalhadores necessários para executar a tarefa) que pode variar de acordo com o processamento das tarefas. Cavalcante et al. (2002) encontraram soluções de alta qualidade e identificaram a presença de sinergia na execução paralela dos algoritmos usados no time desenvolvido para este problema.

Longo (1995), em estudos sobre a aplicação de *A-Teams* ao Problema de Recobrimento de um Conjunto (*SCP - Set Covering Problem*), identificou a sensibilidade de uma implementação de *A-Teams* em relação à variação de alguns de seus parâmetros. O objetivo deste *A-Team* foi encontrar, conjuntamente, limites inferiores e superiores para os valores das soluções ótimas de instâncias do problema em questão. Esta abordagem é descrita na Seção 5.2 por ser um bom exemplo da utilização de diferentes heurísticas na composição do time. Antes, o *SCP* é detalhado na Seção 5.1.

5.1 *SCP*

O Problema de Recobrimento de um Conjunto (*SCP - Set Covering Problem*) é equivalente à busca pelo menor número de subconjuntos, de um determinado conjunto, que unidos geram este conjunto principal. Formalmente, dado um conjunto $I = \{1, \ldots, m\}$ e uma família $F = \{I_1, \ldots, I_n\}$ de subconjuntos de I e $J = \{1, \ldots, n\}$, qualquer subconjunto $J^* \subseteq J$ define um recobrimento de I, se $\bigcup_{j \in J^*} I_j = I$. Ao se associar a cada conjunto j da família F um custo $c_j > 0$, o objetivo passa a ser encontrar um recobrimento J^* cujo custo total $\sum_{j \in J^*} c_j$ seja mínimo. Uma formulação como um problema de programação inteira 0-1 é:

$$(SCP) \begin{cases} \min \sum_{j \in J} c_j x_j, & & & \\ \text{s.a.} \sum_{j \in J} a_{ij} x_j & \geq & 1, & i \in I, \\ x_j & \in & \{0,1\}, & j \in J, \end{cases}$$

onde a_{ij} tem valor 1 se $i \in I_j$ e 0 caso contrário.

O Problema de Recobrimento de um Conjunto é computacionalmente difícil de ser resolvido e claramente pertencente à classe dos problemas NP-Completos. Garey & Johnson (1979) observam que, mesmo se $|I_j| \leq 3, I_j \subseteq I$, o problema continua pertencendo à classe NP-Difícil. Só é possível garantir que seja resolvido em tempo polinomial se $|I_j| \leq 2, \forall\, I_j \subseteq I$.

A importância do *SCP* na Pesquisa Operacional vem do seu modelo teórico, em particular suas interconexões com outros ramos da matemática discreta, tais como hipergrafos, funções boolenas ou satisfatibilidade; e do grande número de situações reais que podem ser modeladas com o problema. Pode-se definir apropriadamente um conjunto finito I que represente ações a serem executadas, decisões a serem tomadas ou alguma outra modelagem de uma situação real; uma família F de subconjuntos de I, onde cada subconjunto represente diferentes meios, recursos ou métodos para atingir, total ou parcialmente, o objetivo desejado; e um custo associado a cada membro de F. A busca da melhor solução para a situação real reduz-se então a encontrar um conjunto de membros de F que seja de custo mínimo e recubra o conjunto I.

Tomando-se, por exemplo, o problema de recuperação de informações em n arquivos I_j, onde $c_j = |I_j|, j = 1, \ldots, n$, é o tamanho de cada arquivo. Cada unidade de informação i é armazenada em pelo menos um arquivo I_j ($i \in I_j$). Supondo-se existirem m requisições de informações, então um recobrimento ótimo fornece um subconjunto de arquivos, minimizando o volume total de informações que necessita ser pesquisado para garantir a recuperação das informações requisitadas.

Outras situações que podem ser modeladas com este problema são definições de escalas para tripulações de linhas aéreas, roteamento de veículos, recuperação de informações, investimento de capital, projeto de circuitos, coloração de mapas, análise PERT/CPM e lógica simbólica, dentre outras.

5.2 Um *A-team* para o *SCP*

Na figura 15 é mostrada uma proposta de *A-Team* para resolução do *SCP*. A abordagem primal-dual do *A-Team* desenvolve, conjuntamente, limites superiores e limites inferiores (que correspondem a valores inferiores ou iguais ao da relaxação linear do modelo (*SCP*) descrito na Seção 5.1) para o valor da solução ótima.

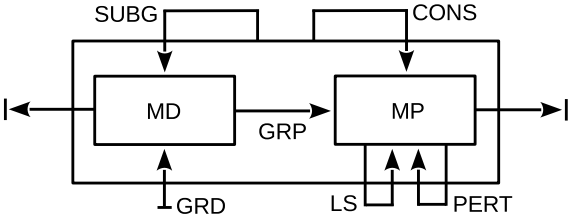


Figura 15. *A-Team* primal-dual para resolução do *SCP*.

A estrutura do *A-Team* é composta de duas memórias compartilhadas e seis agentes. Uma das memórias (**MP**) armazena soluções factíveis para a instância do *SCP* que está sendo tratada (limites superiores). A outra memória (**MD**) armazena soluções para o dual da relaxação linear da instância (limites inferiores). Os

agentes descritos a seguir são compostos de algoritmos que trabalham em cima destas populações de soluções, melhorando-as e acrescentando novos elementos às memórias:

GRD : Construtor de soluções duais. A função deste agente é a construção de soluções duais e o envio das mesmas para a memória correspondente. Este agente utiliza um procedimento baseado em heurísticas gulosas duais, propostas por Balas & Ho (1980) e Fisher & Kedia (1990)), para gerar as soluções.

SUBG : Modificador de soluções duais. Este agente utiliza o método de otimização por subgradientes (Camerini et al. (1975); Held et al. (1974)) para a resolução de uma relaxação Lagrangeana associada ao problema. O agente busca informações, necessárias ao método dos subgradientes, nas duas memórias: um limite superior no valor da solução ótima do SCP é obtido na memória de soluções primais e da memória de soluções duais é obtida uma solução, cujos valores das variáveis duais funcionam como os multiplicadores Lagrangeanos. A nova solução dual, obtida com a resolução da relaxação Lagrangeana, é enviada pelo agente para a memória de soluções duais.

GRP : Construtor de soluções primais. Este agente utiliza um conjunto de heurísticas gulosas. O critério guloso utilizado pode considerar somente os dados que definem a instância do SCP (c_j, I_j) como utilizada por Balas & Ho (1980) ou utiliza também informações da solução dual como em Fisher & Kedia (1990).

LS : Modificador de soluções primais. Este agente busca uma solução na memória de soluções primais e tenta melhorá-la com o uso de uma heurística de busca local baseada em busca tabu. A vizinhança da solução corrente é o subconjunto de soluções que podem ser obtidas, a partir da corrente, com a inversão de valores de duas das variáveis. O "tabu" é fixado às trocas de valores de variáveis que levam a valores da função objetivo não menores do que o da solução corrente.

PERT : Modificador de soluções primais. O processo de aprimoramento de uma solução primal baseia-se na perturbação da mesma, ou seja, a transformação de uma solução em outra redundante (adicionando-se novas variáveis a solução), e o posterior descarte do máximo de variáveis, desta solução perturbada, de modo que o recobrimento não seja afetado. Este processo é repetido um certo número de vezes, para uma mesma solução inicial, e a melhor solução obtida nas várias iterações é armazenada na memória de soluções primais.

CONS : Modificador/Construtor de soluções primais. A ação deste agente tem por base algoritmos genéticos de consenso e consiste na determinação da interseção (variável a variável) entre três soluções armazenadas na memória de soluções primais. Considerando-se que x^1 é a melhor solução ali armazenada, x^2 e x^3 duas outras quaisquer, e x^0 a nova solução a ser gerada, então o critério de consenso utilizado pelo agente é: $x_j^0 = 1$ se $x_j^1 = x_j^2 = 1$ ou $x_j^1 = x_j^3 = 1$, caso contrário $x_j^0 = 0$ $(j \in J)$. Esta solução parcial pode não corresponder a um recobrimento. Neste caso é completada com as mesmas heurísticas gulosas utilizadas no agente GRP.

6. Considerações Finais

O modelo de *A-Teams* tem sido aplicado com sucesso a uma grande variedade de problemas de otimização, comprovando que são melhores que qualquer estratégia isolada na busca por soluções ótimas globais para problemas complexos.O sucesso da aplicação deste paradigma deve-se em muito à simplicidade da sua arquitetura. Esta permite que agentes utilizarem diferentes estratégias para resolver problemas específicos e até mesmo combinar a velocidade e a perícia de agentes de *software* com o conhecimento e as habilidades de humanos na busca por melhores resultados. Além disso, memórias, como estruturas simples de armazenamento, podem facilmente armazenar informações referentes a configuração de agentes e/ou outras memórias.

Apesar do conceito simples e evidentes vantagens deste paradigma, o projetista de um *A-Team* pode enfrentar vários desafios ao projetar os seus elementos. Estes, se não forem corretamente definidos, podem prejudicar o desempenho de toda a organização e consequentemente influenciar na qualidade e velocidade em que os resultados são alcançados. As informações mantidas nas memórias compartilhadas, por exemplo, sofrem uma série de modificações durante o processo de execução. Para tais modificações serem benéficas ao desempenho do time, é necessário um controle preciso do conteúdo das memórias. Este controle se estende desde a inicialização, quando é feito um preenchimento inicial das memórias, até o momento de decidir quais delas não devem mais fazer parte do processamento do time.

Referências

Balas, E. & Ho, A., Set covering algorithms using cutting planes, heuristics and subgradient optimization: A computational study. In: Padberg, M.W. (Ed.), *Combinatorial Optimization*. New York, USA: Elsevier North-Holland, v. 12 de *Mathematical Programming Studies*, p. 37–60, 1980.

Camerini, P.M.; Fratta, L. & Maffioli, F., On Improving Relaxation Methods by Modified Gradiente Techniques. v. 3 de *Mathematical Programming Studies*, p. 26–34, 1975.

Cavalcante, C.C.B.; Cavalcante, V.F.; Ribeiro, C.C. & de Souza, C.C., Parallel cooperative approaches for the labor constrained scheduling problem. In: Ribeiro, C.C. & Hansen, P. (Eds.), *Essays and Surveys in Metaheuristics*. Norwell, USA: Kluwer Academic, p. 201–225, 2002.

Cicirello, V.A. & Smith, S.F., Insect societies and manufacturing. In: *Proceedings of IJCAI-01 Workshop on Artificial Intelligence and Manufacturing: New AI Paradigms for Manufacturing*. San Francisco, USA: Morgan Kaufmann, p. 328–329, 2001.

Fisher, M.L. & Kedia, P., Optimal solution of set covering/partitioning problems using dual heuristics. *Management Science*, 36(6):674–688, 1990.

Garey, M.R. & Johnson, D.S., *Computers and Intractability. A Guide to the Theory of NP-Completeness*. New York, USA: W. H. Freeman, 1979.

Held, M.; Wolfe, P. & Crowder, H.P., Validation of subgradient optimization. *Mathematical Programming*, 6(1):62–88, 1974.

Longo, H.J., *Aplicação de A-Teams ao Problema de Recobrimento de um Conjunto*. Dissertação de Mestrado, Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Computação Científica, Campinas, SP, 1995.

Peixoto, H.P., *Uma Metodologia de Especificação de Times Assíncronos para Problemas de Otimização Combinatória*. Dissertação de Mestrado, Universidade Estadual de Campinas, Instituto de Computação, Campinas, SP, 1995.

Rodrigues, R.F., *Times Assíncronos para a Resolução de Problemas de Otimização Combinatória com Múltiplas Funções Objetivo*. Dissertação de Mestrado, Universidade Estadual de Campinas, Instituto de Computação, Campinas, SP, 1996.

Salman, F.S.; Kalagnanam, J.R.; Murthy, S. & Davenport, A., Cooperative strategies for solving the bicriteria sparse multiple knapsack problem. *Journal of Heuristics*, 8(2):215–239, 2002.

Souza, P.S., *Asynchronous Organizations for Multi-Algorithm Problems*. PhD Thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, USA, 1993.

Talukdar, S., Asynchronous Teams. Technical Report 207, Carnegie Mellon University. Engineering Design Research Center. Department of Electrical and Computer Engineering, 1993.

Talukdar, S.; Baerentzen, L.; Gove, A. & Souza, P.S., Asynchronous teams: Cooperation schemes for autonomous agents. *Journal of Heuristics*, 4(4):295–321, 1998.

Notas Biográficas

Jesmmer da Silveira Alves é graduado em Tecnologia em Processamento de Dados (Faculdades Unidas de Itumbiara, 2000), especialista em Redes de Computadores (Universidade Salgado de Oliveira, 2003) e mestre em Ciência da Computação (Universidade Federal de Goiás, 2009). Atualmente é professor de ensino técnico e tecnológico no Instituto Federal de Educação, Ciência e Tecnologia Goiano. Principais áreas de interesse: desenvolvimento *Web* e análise de algoritmos.

Humberto José Longo é graduado em Ciência da Computação (Universidade Federal de Goiás, 1990), mestre em Ciência da Computação (Universidade Estadual de Campinas, 1995) e doutor em Informática (Pontifícia Universidade Católica do Rio de Janeiro, 2004). Atualmente é professor adjunto no Instituto de Informática da Universidade Federal de Goiás. Principais áreas de interesse: Algoritmos e Otimização.