

Utilização do hardware reconfigurável para acelerar algoritmos evolutivos: o caso do problema do caixeiro viajante *

Iouliia Skliarova, António B. Ferrari

Resumo – Os algoritmos evolutivos (AE) revelaram-se uma abordagem efectiva no encontro de soluções sub-óptimas para os problemas de optimização combinatorial. Este artigo analisa a possibilidade de aceleração de AE para o caso do problema do caixeiro viajante (TSP) com a ajuda de hardware reconfigurável. Os resultados estimativos mostram que a combinação dum computador de uso geral e dos recursos duma FPGA permite incrementar significativamente o desempenho.

Abstract – Evolutionary algorithms (EA) have been shown to be an effective approach for finding near-optimum solutions to problems of combinatorial optimization. The paper analyzes a possibility of acceleration of EA for the traveling salesman problem (TSP) with the aid of reconfigurable hardware. The estimative results show that the combination of general-purpose computer and FPGA resources allows performance to be increased significantly.

I. INTRODUÇÃO

TSP é o problema dum caixeiro que começando da sua casa, quer viajar por um conjunto específico de cidades e voltar para casa no fim [1]. Cada cidade deve ser visitada uma só vez e, obviamente, o caixeiro precisa de encontrar o caminho mais curto. Em termos mais formais, o problema pode ser representado por um grafo interpretado $G=(V, E)$, onde $V=\{0, 1, \dots, n-1\}$ é o conjunto de vértices que correspondem às cidades, e E é o conjunto de arcos que identificam as estradas existentes entre as cidades. A cada arco $(i, j) \in E$ atribui-se um peso d_{ij} que especifica a distância entre as cidades i e j . Portanto o problema do caixeiro viajante consiste em encontrar o ciclo hamiltoniano mais curto num grafo interpretado. Neste artigo só abordamos o TSP simétrico para o qual $d_{ij}=d_{ji}$ para cada par de cidades.

TSP é um dos problemas de optimização combinatorial mais conhecidos que possui muitas aplicações práticas em áreas diversas tais como cristalografia de raios X [2], planeamento de tarefas [1], perfuração de placas de circuito impresso [3], mapeamento de ADN [4], etc. Embora seja bastante fácil formular o problema, é extremamente difícil resolvê-lo (TSP pertence à classe de

problemas NP-completos [5]). É por isso que se efectuam várias tentativas de encontrar soluções sub-óptimas o que é frequentemente suficiente para muitas aplicações práticas.

Dado que o TSP é um problema NP-completo, possui um espaço de soluções grande e a função de adequabilidade facilmente calculável, este serve bastante bem para os algoritmos evolutivos (AE). Neste artigo analisamos várias possibilidades de implementação de AE para o TSP.

O resto deste artigo está organizado da maneira seguinte. A secção II inclui uma breve introdução à computação evolutiva. Na secção III apresenta-se o algoritmo evolutivo que aplicámos para a solução do problema do caixeiro viajante. A seguir, na secção IV, descrevem-se os resultados da implementação do algoritmo evolutivo em software. A secção V considera a implementação parcial do AE em hardware reconfigurável. Na secção VI efectua-se a avaliação e a comparação dos resultados conseguidos com as duas implementações do AE, sendo uma baseada em software e outra em FPGA. Finalmente, as conclusões estão na secção VII.

II. COMPUTAÇÃO EVOLUTIVA

As técnicas de computação evolutiva são métodos de optimização probabilísticos que manipulam uma população de indivíduos (i.e. soluções potenciais) e baseiam-se na teoria de Darwin de selecção natural e evolução. As técnicas evolutivas são normalmente utilizadas para problemas de optimização e de pesquisa [6]. Na área de computação evolutiva destacam-se vários paradigmas tais como *algoritmos genéticos*, *estratégias evolutivas*, *programação evolutiva* e a *programação genética*. Os *algoritmos genéticos* tradicionais [7] manipulam *strings* binárias de comprimento fixo sobre as quais são definidos operadores de mutação e cruzamento (estes operam sem ter conhecimento algum sobre a interpretação das *strings*). No caso das *estratégias evolutivas*, a representação é um vector de valores reais que é manipulado pelos operadores de mutação [8]. Na *programação evolutiva* os indivíduos são representados por máquinas de estados finitos capazes de responder aos

* Trabalho financiado com a bolsa da FCT-PRAXIS XXI/BD/21353/99

estímulos ambientais [8]. Os operadores de variação (essencialmente, os de mutação) afectam a estrutura e o comportamento dos indivíduos. A *programação genética* é a técnica de computação evolutiva mais recente que estende o modelo genético ao espaço de programas [9]. Neste caso os indivíduos não são *strings* codificadas mas sim programas de computador. Portanto, a programação genética permite desenvolver (ou, evoluir) automaticamente programas que resolvem (exacta ou aproximadamente) um dado problema.

É de notar que com o decorrer do tempo, observa-se uma troca intensiva de ideias entre os investigadores que trabalham em cada uma das áreas mencionadas, o que resultou na redução de diferenças entre estes paradigmas. Portanto, descrevemos aqui características inerentes a todos os *algoritmos evolutivos*.

O diagrama básico de um algoritmo evolutivo está apresentado na fig. 1. Primeiro, deve-se construir uma população inicial de indivíduos. Normalmente, a população inicial é criada pela amostragem aleatória de soluções possíveis. Contudo, é também possível utilizar a população obtida com a ajuda de qualquer outro algoritmo ou pelos peritos humanos. A seguir cada solução é avaliada a fim de medir a sua adequabilidade. A partir deste momento o algoritmo entra em ciclo composto pelas operações de avaliação, variação e selecção. No fim de cada iteração (denominada geração) cria-se uma nova população de indivíduos.

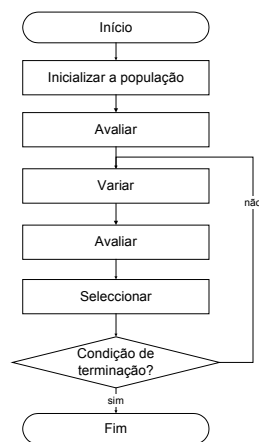


Fig. 1 – Esquema básico de um AE

Os operadores de variação (tais como os de mutação e cruzamento) são utilizados para gerar um conjunto novo de indivíduos. Um operador de mutação cria indivíduos novos ao efectuar certas modificações num só indivíduo, enquanto um operador de cruzamento cria indivíduos novos (descendentes) através de combinação de várias partes de dois ou mais indivíduos existentes (pais) [10]. A operação de selecção faz com que os indivíduos mais adequados sobrevivam e formem a geração seguinte. Este processo continua até que alguma condição de terminação seja atingida, tal como a obtenção de uma solução satisfatória, o esgotar do tempo disponível ou do número máximo de gerações permitidas.

Na maioria dos AE o tamanho da população é um valor fixo que é especificado como um parâmetro controlado pelo utilizador. No caso geral, o processo de selecção de indivíduos que passam para a geração seguinte, é executado uma vez por cada ciclo do algoritmo, permitindo deste modo primeiro criar todos os descendentes necessários e só depois realizar a selecção. Por outro lado, os algoritmos *steady state* invocam o processo de selecção cada vez que seja produzido um descendente a fim de manter o tamanho de população sempre constante.

As vantagens principais dos algoritmos evolutivos são as seguintes [11]:

- **Simplicidade conceptual.** Como descrevemos no início desta secção, o esquema básico dos algoritmos evolutivos é bastante simples e só é composto pelas fases de inicialização, avaliação, variação e selecção que são aplicadas em ciclo até que a população converge para a solução óptima (caso isto seja possível).

- **Aplicabilidade ampla.** Os algoritmos evolutivos são aplicáveis a qualquer problema que pode ser formulado como o de optimização. Para tal é necessário definir a estrutura de dados para representar as soluções, a função para avaliar a sua adequabilidade, os operadores de variação que permitem gerar soluções novas com base em soluções já existentes, e a função de selecção.

AE são aplicados com sucesso em áreas práticas diversas tais como a síntese lógica [12], criptografia [13], pilotagem automática de automóveis [14], engenharia mecânica [15], engenharia aérea, indústria nuclear e química [16], robótica [17], medicina [11], etc.

Uma área de investigação bastante recente é o *hardware evolutivo*. Este termo descreve abordagens diferentes utilizadas para desenvolver circuitos electrónicos com a ajuda de técnicas evolutivas [18]. Normalmente estas abordagens dividem-se em dois grupos de acordo com a área de aplicação. O primeiro grupo abrange a evolução directa em componentes existentes (geralmente, em FPGAs), enquanto o segundo grupo inclui as tentativas de simular a funcionalidade desejada a fim de a implementar posteriormente em componentes reais.

- **Os métodos evolutivos superam os clássicos em problemas reais.** Muitos problemas reais de optimização possuem características que não se coadunam com os requisitos dos métodos clássicos. Exemplos destas características são restrições não lineares, condições não estacionárias, etc. [11]. As superfícies de resposta em problemas reais são frequentemente multi-modais e os métodos baseados em gradiente convergem rapidamente para um óptimo local. Ao contrário disso, com as técnicas evolutivas é possível levar em conta todas estas características o que resulta em desempenho mais elevado.

- **Potencialidade de hibridação com outros métodos.** Os algoritmos evolutivos possuem uma

estrutura que facilita a incorporação relativamente natural de conhecimentos específicos ao problema em causa. Por exemplo, pode-se utilizar operadores de variação específicos, bem como uma função de adequabilidade específica. Para além disso, é possível e razoável combinar algoritmos evolutivos com as técnicas de optimização tradicionais. Existem várias potencialidades de fazer isto: correr os métodos baseados em gradiente a seguir à realização de pesquisa inicial com um AE; aplicar algoritmos diferentes em simultâneo (por exemplo, incorporar os métodos de pesquisa local no AE [19]); utilizar métodos rápidos (por exemplo, os algoritmos *greedy*) a fim de gerar a população inicial de indivíduos para o AE, etc.

▪ **Possibilidade de implementações paralelas e distribuídas.** É frequentemente possível realizar algumas partes de algoritmos evolutivos (por exemplo, a avaliação de indivíduos, a implementação de operadores de variação) em paralelo. Para além disso, pode-se explorar o paralelismo a nível de população. Neste caso criam-se grupos de indivíduos que se desenvolvem de uma maneira semi-independente, observando-se uma migração lenta de indivíduos entre os grupos [20]. É de notar também que os AE possuem paralelismo implícito pois são capazes de encontrar várias soluções de qualidade igual. Isto permite seleccionar a solução mais apropriada para o problema em causa.

▪ **Robustez às alterações dinâmicas.** Os métodos tradicionais de optimização não são robustos às alterações dinâmicas das condições do problema. Caso ocorra alguma alteração na especificação da tarefa, é frequentemente necessário executar de novo o algoritmo a fim de obter a solução. Ao contrário disso, os algoritmos evolutivos são capazes de adaptar soluções às circunstâncias correntes porque a população de indivíduos já obtidos pode servir de base ao aperfeiçoamento futuro.

▪ **Capacidade de auto-optimização.** Todas as técnicas de optimização requerem a definição adequada dos valores de variáveis exogéneas (tais como o tamanho de passo, etc.). Contudo, nos algoritmos evolutivos é possível optimizar estes parâmetros durante o processo de pesquisa enquanto os métodos tradicionais utilizam parâmetros definidos *a priori* pelos peritos humanos.

▪ **Possibilidade de resolver problemas que não possuem soluções conhecidas.** Esta é uma das maiores vantagens dos AE pois estes podem enfrentar problemas para os quais não se conhece solução.

III. ALGORITMO EVOLUTIVO PARA O PROBLEMA DO CAIXEIRO VIAJANTE

Descrevemos o algoritmo evolutivo que aplicámos para resolver o problema do caixeiro viajante. Para tal é necessário definir a representação de uma solução, a função de adequabilidade, os operadores de mutação e de cruzamento e os critérios de selecção.

A. Representação

No caso do TSP uma solução potencial é um caminho que começando numa cidade inicial percorre todas as cidades restantes numa determinada ordem. Representámos um caminho possível como um vector de números inteiros em que a cidade na posição i é visitada depois da cidade na posição $i-1$ e antes da cidade na posição $i+1$. Por exemplo, para o grafo mostrado na fig. 2 uma das soluções possíveis está marcada pelos arcos mais grossos e pode ser representada com o vector [0 1 4 2 3].

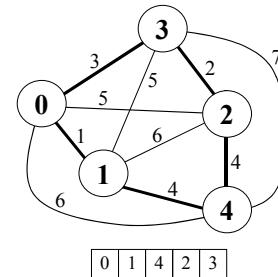


Fig. 2 – Representação dum caminho possível

B. Avaliação

Para o problema do caixeiro viajante a parte de avaliação do algoritmo é muito natural dado que a função de adequabilidade de um caminho corresponde ao seu comprimento. Para o exemplo da fig. 2 a adequabilidade da solução [0 1 4 2 3] é 14, e como se pode verificar esta é a solução óptima.

C. Operadores de variação

Os operadores de variação que aplicámos são os de mutação e cruzamento. Analisemo-los em mais detalhe.

C.1. Mutação

O operador de mutação selecciona aleatoriamente duas cidades numa solução, e inverte a ordem de todas as cidades que ficam entre as escolhidas. Como resultado, o operador de mutação tenta reparar o caminho que se intersecta a si próprio. O exemplo desta situação está ilustrado na fig. 3. Do lado esquerdo mostramos a sequência inicial de cidades. Caso seleccionemos as cidades número 0 e 4 e apliquemos o operador de mutação, obteremos o caminho apresentado do lado direito da fig. 3. Neste caso o comprimento do caminho resultante é mais curto do que o do caminho inicial, contudo nem sempre isto acontece.

C.2. Cruzamento

Aplicámos o operador de cruzamento PMX (*partially-mapped*) proposto em [21], que produz descendentes através da selecção de uma subsequência do caminho de um pai, preservando a ordem e a posição de um número máximo de cidades doutro pai. Este operador envolve dois

pais (p_1 e p_2) que produzem dois descendentes (o_1 e o_2). A subsequência de caminho que passa do pai ao filho é definida com a selecção de dois pontos de corte. Inicialmente, os segmentos que se encontram entre os pontos de corte são copiados do pai p_1 para o descendente o_2 , e do pai p_2 para o descendente o_1 . Estes segmentos definem também uma série de mapeamentos. A seguir todas as cidades que se encontram antes do primeiro ponto de corte e depois do segundo ponto de corte são copiados do pai p_1 para o descendente o_1 , e do pai p_2 para o descendente o_2 . É de notar que esta operação pode resultar num caminho inválido; por exemplo um descendente pode incluir cidades duplicadas. A fim de ultrapassar esta situação, utiliza-se a série de mapeamentos definida previamente que indica como permutar as cidades que se encontram em conflito.

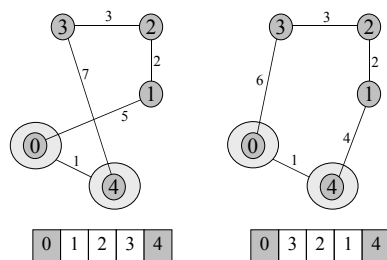


Fig. 3 – Aplicação do operador de mutação

Por exemplo, para os pais p_1 e p_2 com os pontos de corte marcados pelas linhas verticais:

$$p_1 = [3 | 0 | 1 | 2 | 4]$$

$$p_2 = [1 | 0 | 2 | 4 | 3]$$

o operador PMX define a seguinte série de mapeamentos:

$$(0 \leftrightarrow 0, 1 \leftrightarrow 2, 2 \leftrightarrow 4)$$

e preenche os segmentos dos descendentes entre os pontos de corte de maneira seguinte:

$$o_1 = [- | 0 | 2 | 4 | -]$$

$$o_2 = [- | 0 | 1 | 2 | -]$$

A seguir todas as cidades que ficam antes do primeiro e depois do segundo ponto de corte têm de ser transferidas dos pais para os filhos:

$$o_1 = [3 | 0 | 2 | 4 | 1]$$

$$o_2 = [1 | 0 | 1 | 2 | 3]$$

Como se pode ver entramos numa situação de conflito pois ambos os descendentes incluem cidades duplicadas (marcadas acima com os caracteres em negrito), o que está proibido pela definição do problema.

Consideremos o primeiro descendente o_1 . Neste caso é preciso substituir a cidade número 4 por qualquer outra cidade válida. Para tal utiliza-se a série de mapeamentos que indica que à cidade 4 corresponde a cidade 2 ($2 \leftrightarrow 4$). Contudo, a cidade 2 já está incluída no caminho o_1 . Portanto, recorre-se novamente à série de mapeamentos que mostra que à cidade 2 corresponde a 1 ($1 \leftrightarrow 2$). Em consequência, incluímos a cidade 1 no descendente o_1 .

O mesmo procedimento executa-se para o caso do segundo descendente. Finalmente, obtemos o resultado seguinte:

$$o_1 = [3 | 0 | 2 | 4 | 1]$$

$$o_2 = [4 | 0 | 1 | 2 | 3]$$

A aplicação do operador de cruzamento PMX está ilustrada na fig. 4. As figuras 4a) e 4b) representam os dois pais e as figuras 4c) e 4d) mostram os dois descendentes resultantes. Como se pode observar, a adequabilidade de um dos filhos é bastante menor que a dos ambos os pais, enquanto a do descendente o_2 é significativamente maior.

É de notar que o problema do caixeiro viajante possui a estrutura do “vale grande” do relevo de adequabilidade do espaço de pesquisa [22]. Isto é devido a uma forte correlação entre a adequabilidade de uma solução e a diferença existente entre esta solução e o óptimo global. Como resultado, todas as soluções boas possuem partes semelhantes. Portanto, para este problema o cruzamento é muito eficiente dado que permuta várias partes de soluções entre si.

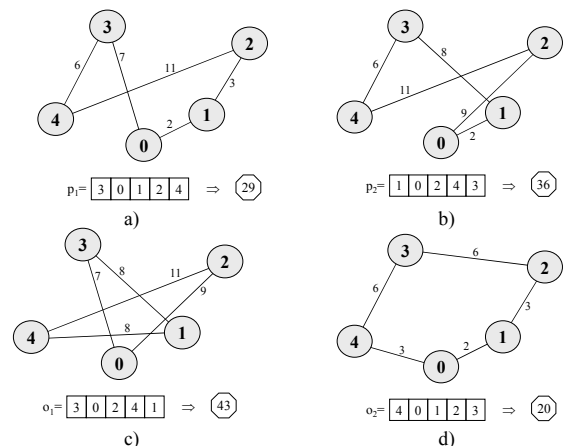


Fig. 4 – Aplicação do operador de cruzamento PMX

D. Selecção

A fim de escolher os pais para a produção de descendentes aplicamos a selecção proporcional à adequabilidade de indivíduos. Para tal utilizamos a abordagem da roda de roleta em que a cada indivíduo se atribui um sector cuja largura é proporcional à adequabilidade deste indivíduo. A roda é “activada” cada vez que se precisa de um pai.

Todos os descendentes criados formam a geração seguinte. Para além disso aplicamos a selecção *elitist* que garante a sobrevivência do melhor indivíduo encontrado durante a pesquisa.

IV. IMPLEMENTAÇÃO DO AE EM SOFTWARE

O algoritmo descrito foi implementado numa aplicação de software desenvolvida em C++. A seguir foi realizada uma série de experiências com instâncias de teste

disponíveis na TSPLIB [23]. As experiências foram efectuadas com diferentes probabilidades de cruzamento (nomeadamente, 10%, 25%, 50% e 100%). Para todas as instâncias o tamanho de população foi de 20 indivíduos e foram realizadas 1000 gerações. A probabilidade de mutação foi de 10%.

Os resultados são apresentados na tabela 1. A primeira coluna contém os nomes de várias instâncias. Os números incluídos em nomes especificam a quantidade de cidades na instância respectiva. As colunas t_{total} guardam o tempo total (em segundos) de resolução do problema num PentiumIII/800MHz/256MB com o sistema operativo Windows2000. As colunas t_{cros} indicam o tempo (em segundos) despendido a efectuar as operações de cruzamento. E finalmente, as colunas $\%_{cros}$ contêm a percentagem do tempo de realização de cruzamento comparando com o tempo total de execução do algoritmo.

Os melhores resultados em termos do desempenho do algoritmo foram atingidos com as probabilidades de cruzamento de 25% e 50%. As linhas que correspondem à melhor solução estão destacadas na tabela 1 com os caracteres em negrito. Por exemplo, para a instância *a280* o melhor resultado foi obtido com a probabilidade de cruzamento de 50%, para a instância *berlin52* – com a probabilidade de 10%, etc. Como se pode ver da tabela 1, uma percentagem significativa do tempo do processador é despendida a efectuar a operação de cruzamento. Obviamente, o valor $\%_{cros}$ é maior para a probabilidade de cruzamento de 100% (contudo, em algoritmos práticos a probabilidade de cruzamento é sempre bastante menor do que 100%). Mas mesmo para as probabilidades de cruzamento “melhores” (em termos dos resultados obtidos) o valor $\%_{cros}$ varia de 19% a 65%.

V. IMPLEMENTAÇÃO PARCIAL EM FPGA

Os algoritmos evolutivos possuem um certo grau de paralelismo em termos de pesquisa no espaço de soluções. Contudo, quando aplicados a problemas complexos, os AE requerem que a população seja grande e necessitam de um grande número de gerações, o que os torna bastante lentos. Têm sido efectuadas várias tentativas de acelerar algoritmos evolutivos com a ajuda do hardware reconfigurável [24-28]. Os autores respectivos

comunicam a obtenção dos resultados impressionantes em comparação com as implementações em software. Contudo, as realizações existentes baseadas em hardware dedicado são muito complexas e requerem bastantes recursos, ou, ao contrário, muito simplificadas o que não corresponde às necessidades de aplicações práticas. É por isso que decidimos analisar qual a parte do AE para o problema do caixeiro viajante é a mais crítica em termos do tempo que consome, e implementar só esta parte em FPGA.

Os resultados apresentados na secção IV mostraram que a parte mais crítica do algoritmo considerado é a operação de cruzamento. Portanto, o aumento da eficiência desta operação vai influenciar significativamente o desempenho de todo o algoritmo. Por isso sugerimos a implementação da operação de cruzamento em FPGA [29].

A arquitectura do circuito respectivo está representada na fig. 5. Esta inclui uma unidade de controlo central que activa, na ordem necessária todos os passos da operação de cruzamento. A versão da arquitectura implementada é capaz de processar caminhos compostos por 1024 cidades no máximo. Por isso existem quatro blocos de memória do tamanho $2^{10} \times 10$ bits que servem para guardar os dois caminhos-pais (“Pai 1” e “Pai 2”) e os dois caminhos-descendentes resultantes (“Filho 1” e “Filho 2”). Um caminho é representado de maneira seguinte. Em cada endereço de memória está escrito o número duma cidade. A cidade na posição de memória i é visitada depois da cidade que se encontra no endereço $i-1$ e antes da cidade que está na posição $i+1$. Uma das aplicações práticas possíveis do circuito proposto é descrita em [29] para a qual umas 64-128 cidades são suficientes. Portanto, pode-se diminuir significativamente o tamanho dos blocos de memória.

Os pontos de corte utilizados no cruzamento são escolhidos aleatoriamente pela aplicação de software e escritos em dois registos especiais de 10 bits (“Primeiro ponto de corte” e “Segundo ponto de corte” na fig. 5). O registo “Max” guarda o comprimento actual do caminho. De acordo com este valor a unidade de controlo só vai forçar o processamento da área adequada dos blocos de memória que contêm os dois pais e os dois descendentes. Isto permite acelerar o cruzamento de caminhos curtos.

Instância	10%			25%			50%			100%		
	t_{total}	t_{cros}	$\%_{cros}$	t_{total}	t_{cros}	$\%_{cros}$	t_{total}	t_{cros}	$\%_{cros}$	t_{total}	t_{cros}	$\%_{cros}$
<i>a280</i>	4.99	0.72	14.4	5.88	1.49	25.3	8.39	3.66	43.6	12.51	7.22	57.7
<i>berlin52</i>	1.08	0.71	65.7	1.27	0.32	25.2	1.79	0.75	41.9	2.64	1.42	53.8
<i>bier127</i>	2.36	0.36	15.2	2.75	0.68	24.7	3.92	1.68	42.9	5.84	3.29	56.3
<i>d657</i>	12.82	2.45	19.1	15.18	4.62	30.4	23.33	11.92	51.1	35.66	23.21	65.1
<i>eil51</i>	1.06	0.16	15.1	1.22	0.31	25.4	1.76	0.74	39.2	2.58	1.38	53.5
<i>fl417</i>	7.77	1.34	17.2	9.30	2.62	28.2	13.56	6.36	46.9	20.50	12.59	61.4
<i>rat575</i>	10.88	1.89	17.4	12.97	3.78	29.1	19.46	9.53	48.9	30.74	19.39	63.1
<i>u724</i>	14.27	2.69	18.9	17.02	5.36	31.5	25.60	13.19	51.5	40.55	26.74	65.9
<i>vm1084</i>	22.83	5.01	21.9	28.09	9.69	34.5	43.63	24.13	55.3	71.12	49.20	69.2

Tabela 1 – Resultados das experiências em software

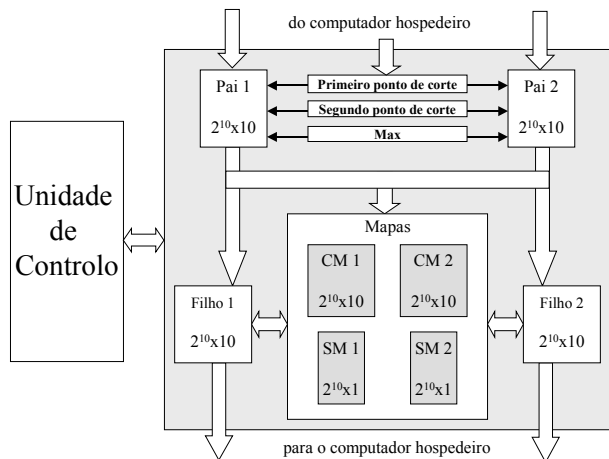


Fig. 5 – Arquitectura proposta para a realização da operação de cruzamento PMX

Para além disso há quatro blocos de memória adicionais que ajudam na realização do cruzamento. Estes blocos são “CM1” e “CM2” do tamanho de $2^{10} \times 10$ bits (vamos referenciá-los por *mapas complexos*), e “SM1” e “SM2” do tamanho de $2^{10} \times 1$ bits (referenciados por *mapas simples*).

A fim de executar o cruzamento PMX é necessário realizar a seguinte sequência de operações. Primeiro, os valores dos pontos de corte e do número de cidades para uma dada instância do problema devem ser carregados na FPGA. Depois, os caminhos-pais são transferidos do computador hospedeiro para os blocos de memória “Pai 1” e “Pai 2”. Cada vez que o número de uma cidade se escreve num bloco de memória dos pais, a posição do mapa simples respectivo com o mesmo endereço inicializa-se com o valor “0” (o que permite efectuar o *reset* dos mapas simples).

A seguir, os segmentos entre os pontos de corte devem ser transferidos do “Pai 1” para o “Filho 2” e do “Pai 2” para o “Filho 1”. Cada vez que se transfere a cidade $c1$ do bloco “Pai 1” para o bloco “Filho 2”, o valor “1” deve ser escrito no endereço $c1$ do mapa simples “SM2”. A mesma coisa acontece com o outro pai, i.e. quando copiamos a cidade $c2$ do bloco “Pai 2” para o bloco “Filho 1”, o valor “1” deve também ser escrito no endereço $c2$ do mapa simples “SM1”. Ao mesmo tempo o valor $c1$ é guardado no endereço $c2$ no mapa complexo “CM1”, e o valor $c2$ é guardado no endereço $c1$ no mapa complexo “CM2”. Para o exemplo considerado na secção III (subsecção C.2.), todos os blocos devem ser preenchidos da maneira apresentada na fig. 6.

O passo seguinte da operação de cruzamento pressupõe que todas as cidades que se encontram antes do primeiro ponto de corte e depois do segundo ponto de corte devem ser copiadas do bloco “Pai 1” para o “Filho 1” e do bloco “Pai 2” para o “Filho 2”. Para além disso todos os conflitos têm de ser resolvidos.

Para isso aplicámos a estratégia apresentada na fig. 7. Analisemos aqui o caso do preenchimento do bloco “Filho 1” antes do primeiro ponto de corte, pois para os casos restantes os passos a realizar são semelhantes. Primeiro, o

número da cidade c é lido do bloco “Pai 1”. Caso no endereço c do bloco “SM1” esteja escrito o valor “0”, isto indica que a cidade c pode seguramente ser copiada para o bloco “Filho 1”. Caso contrário, se no endereço c do bloco “SM1” estiver escrito o valor “1”, isto significa que a cidade c já foi incluída no caminho “Filho 1”. Isso implica que esta deve ser substituída por qualquer outra cidade. Para tal utiliza-se o mapa complexo “CM1” da maneira ilustrada na fig. 7.

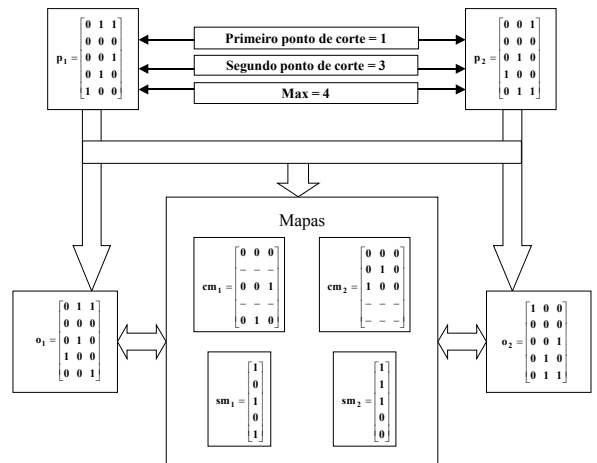


Fig. 6 – Conteúdo dos blocos de memória e dos registos para o exemplo considerado na subsecção C.2. da secção III

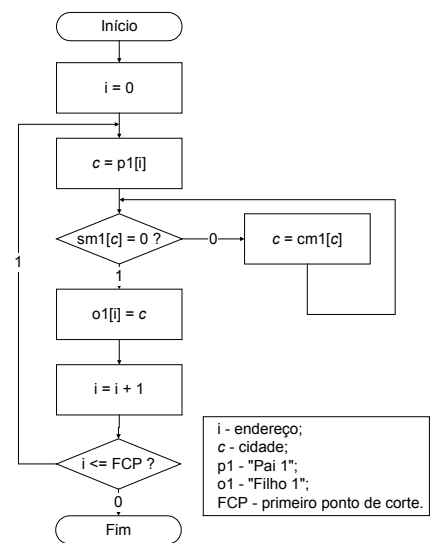


Fig. 7 – Algoritmo que controla o preenchimento do bloco “Filho 1” antes do primeiro ponto de corte

Para compor o segundo caminho-descendente são executadas as mesmas operações com a única diferença de que são utilizados os mapas “SM2” e “CM2”. Por fim, os caminhos-descendentes resultantes são transferidos para o computador hospedeiro.

VI. AVALIAÇÃO DOS RESULTADOS

Para as experiências foi utilizada a placa com interface PCI ADM-XRC da AlphaData [30] que contém uma

FPGA da família Virtex-EM da Xilinx (nomeadamente, a XCV812E) [31]. Esta FPGA incorpora uma memória adicional de blocos SelectRAM organizados em colunas e inseridos em cada quarta coluna de CLBs. No total há mais de um milhão de bits disponíveis que complementam os 294 Kbits de RAM distribuída. Portanto, esta FPGA serve muito bem para a arquitectura proposta porque podemos aproveitar a grande quantidade da memória disponível para guardar os caminhos-pai e os descendentes bem como os mapas necessários. A interacção com a FPGA é feita com a ajuda da biblioteca de interface com a placa ADM-XRC que suporta a inicialização e configuração da FPGA, transferência de dados, processamento de interrupções e erros, gestão de relógio, etc.

A tabela 2 contém informação acerca da área ocupada pelo circuito implementado e da sua frequência de relógio. A área está expressada em número de *slices* da Virtex, sendo cada CLB composto por dois *slices*. Entre parêntesis está indicada a percentagem dos recursos da XCV812E que o circuito consome.

Área (em <i>slices</i>)	Número dos blocos de RAM	Frequência de relógio máxima (MHz)
149 (1%)	20 (7%)	102.659

Tabela 2 – Parâmetros do circuito implementado

A comparação entre as implementações baseadas em software e em hardware do operador de cruzamento PMX é apresentada na tabela 3. A primeira coluna contém o número de cidades no problema respectivo. A coluna t_{soft} guarda o tempo de realização do cruzamento em software. A versão de software baseia-se na linguagem C++ e foi executada num PentiumIII/800MHz/256MB com o sistema operativo Windows2000. A coluna t_{hard} contém o tempo de realização do cruzamento em hardware. A versão de hardware foi executada em FPGA XCV812E sob a frequência de relógio de 40 MHz. Para facilitar a comparação, os pontos de corte foram seleccionados iguais em ambas as implementações. A aceleração conseguida é dada pela expressão $t_{\text{soft}}/t_{\text{hard}}$. Os resultados das experiências mostraram que o cruzamento PMX é executado em FPGA 10-50 vezes mais rapidamente do que em software.

É de notar que o tempo do cruzamento realizado em FPGA depende fortemente dos pontos de corte seleccionados. Isto explica-se pelo facto da primeira parte do cruzamento PMX (i.e. a troca dos segmentos entre os pontos de corte) ser executada em paralelo para ambos os pares pai-descendente. Para além disso, esta parte só envolve a leitura simples do bloco de memória que guarda o caminho-pai, e a escrita sequencial para o bloco de memória correspondente ao caminho-descendente, sem executar qualquer outra operação. Como resultado, esta fase é muito rápida. Por outro lado, a segunda parte do cruzamento (i.e. o preenchimento dos descendentes antes do primeiro e depois do segundo pontos de corte) é

executada sequencialmente para cada par pai-descendente. Para além da realização de operações de leitura/escrita na memória, devem ser efectuadas algumas verificações que garantem a construção de caminhos válidos. Em consequência, quanto maior for a distância entre os pontos de corte, mais rápido será concluído o cruzamento em FPGA. A mesma tendência observa-se em versão de software embora a primeira parte do cruzamento seja realizada de modo sequencial.

A aceleração conseguida em FPGA em comparação com a implementação em software explica-se pelas razões seguintes. Primeiro, foi aplicada a técnica de processamento paralelo, i.e. a parte da operação de cruzamento PMX é executada em paralelo em FPGA. Segundo, a organização dos blocos de memória é otimizada para os tamanhos de dados necessários. Ahamos que é possível conseguir uma aceleração maior através da realização da segunda parte do cruzamento também em paralelo. Para tal basta efectuar modificações simples na unidade de controlo.

Número de cidades	t_{soft} (ms)	t_{hard} (ms)	Aceleração
280	1.0896	0.08443	12.9
52	0.4293	0.01872	22.9
127	0.6203	0.04040	15.4
657	3.838	0.19486	19.7
51	0.2151	0.01806	11.9
417	2.0605	0.12524	16.5
575	2.7703	0.17199	16.1
724	10.2145	0.20443	49.9

Tabela 3 – Comparação entre as implementações baseadas em software e em hardware do operador de cruzamento PMX

É de notar que uma outra parte de AE que, para muitas aplicações, é o entrave, é a tarefa de avaliação de indivíduos. Para aliviar este efeito usam-se métodos que estimam a valor de adequabilidade de uma solução em vez de o calcular exactamente [32].

VII. CONCLUSÕES

Neste artigo apresentámos uma descrição breve de algoritmos evolutivos e mostrámos que estes podem ser eficientemente utilizados para resolução de problemas de optimização combinatória. Como um estudo de caso, analisámos o problema do caixeiro viajante.

Para além disso apresentámos a implementação de um dos operadores genéticos (nomeadamente, o de cruzamento) com base em hardware reconfigurável. O circuito proposto é bastante simples, consome poucos recursos da FPGA e é capaz de funcionar com a frequência de 100 MHz. Os resultados das experiências mostraram que a implementação do operador de cruzamento em FPGA tem influência significativa no desempenho global do algoritmo.

Existem várias ideias interessantes para investigação futura. Por exemplo, em [33, 34] foi mostrado que a

combinação de algoritmos evolutivos e de pesquisa local permite obter soluções de melhor qualidade, especialmente para problemas de grande dimensão. Contudo, a pesquisa local consome muito tempo de computação, o que torna a ideia da sua implementação em hardware reconfigurável bastante promissora.

REFERÊNCIAS

- [1] B. L. Golden, B. K. Kaku, "Difficult Routing and Assignment Problems", in "Handbook of Discrete and Combinatorial Mathematics", K. H. Rosen, et al., editors, CRC Press, pp. 692-705, 2000.
- [2] M. Junger, et al., "The traveling salesman problem", in "Network Models", M. Ball, et al., editors, pp. 225-330, 1995.
- [3] J. D. Litke, "An Improved Solution to the Traveling Salesman Problem with Thousands of Nodes", Communications of the ACM, vol. 27, No. 12, pp. 1227-1236, 1984.
- [4] R. K. Ahuja, et al., "Applications of network optimization", in "Network Models", M. Ball, et al., editors, pp. 1-83, 1995.
- [5] M. Garey, D. Johnson, "Computers and Intractability: A Guide to the Theory of NP-completeness", Freeman, 1979.
- [6] M. Gen, R. Cheng, "Genetic Algorithms & Engineering Optimization", John Wiley & Sons, Inc., 2000.
- [7] D. E. Goldberg, "Genetic Algorithms in Search, Optimization & Machine Learning", Addison-Wesley, 1989.
- [8] K. De Jong, "Evolutionary Computation: Recent Developments and Open Issues", in "Evolutionary Algorithms in Engineering and Computer Science", K. Miettinen, et al., editors, pp. 43-54, 1999.
- [9] J. R. Koza et al., "Genetic Programming III. Darwinian Invention and Problem Solving", Morgan Kaufmann, 1999.
- [10] Z. Michalewicz, D. B. Fogel, "How to Solve It: Modern Heuristics", Springer, 2000.
- [11] D. B. Fogel, "An Introduction to Evolutionary Computation and Some Applications", in "Evolutionary Algorithms in Engineering and Computer Science", K. Miettinen, et al., editors, pp. 23-41, 1999.
- [12] V. Sklyarov, "An Evolutionary Algorithm for the Synthesis of RAM-Based FSMs", in "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali, editors, pp. 108-118, 2002.
- [13] N. Nedjah, L. de Macedo Mourelle, "Minimal Addition Chain for Efficient Modular Exponentiation Using Genetic Algorithms", in "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali, editors, pp. 88-98, 2002.
- [14] N. Laumanns, et al., "Evolutionary Multi-objective Integer Programming for the Design of Adaptive Cruise Control Systems", in "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali, editors, pp. 200-210, 2002.
- [15] J. T. Alander, J. Lampinen, "Cam Shape Optimization by Genetic Algorithm", in "Genetic Algorithms and Evolution Strategies in Engineering and Computer Science", D. Quagliarella, et al., editors, pp. 153-174, 1998.
- [16] T. Bäck, et al., "Evolutionary Algorithms: Applications at The Informatik Center Dortmund", in "Genetic Algorithms and Evolution Strategies in Engineering and Computer Science", D. Quagliarella, et al., editors, pp. 153-174, 1998.
- [17] G. N. Nyakoe, et al., "Optimization of Pulse Pattern for a Multi-robot Sonar System Using Genetic Algorithm", in "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali, editors, pp. 179-189, 2002.
- [18] J. F. Miller, et al., "Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study", in "Genetic Algorithms and Evolution Strategies in Engineering and Computer Science", D. Quagliarella, et al., editors, pp. 105-131, 1998.
- [19] C. R. Reeves, T. Yamada, "Embedded Path Tracing and Neighbourhood Search Techniques in Genetic Algorithms", in "Evolutionary Algorithms in Engineering and Computer Science", K. Miettinen et al., editors, pp. 95-111, 1999.
- [20] M. Tomassini, "Parallel and Distributed Evolutionary Algorithms: A Review", in "Evolutionary Algorithms in Engineering and Computer Science", K. Miettinen, et al., editors, pp. 113-133, 1999.
- [21] D. E. Goldberg, R. Lingle, "Alleles, Loci, and the Traveling Salesman Problem", Proc. 1st Int. Conf. on Genetic Algorithms, pp. 154-159, 1985.
- [22] P. J. Darwen, "Evolving a Schedule with Batching, Precedence Constraints, and Sequence-Dependent Setup Times: Crossover Needs Building Blocks", in "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali, editors, pp. 525-535, 2002.
- [23] <http://www.informatik.uni-heidelberg.de/groups/comopt/software/TSPLIB95/index.html>
- [24] M. Salami, T. Hendtlass, "A Fast Evolutionary Algorithm for Image Compression in Hardware", in "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali, editors, pp. 241-252, 2002.
- [25] P. Graham, B. Nelson, "Genetic Algorithms in Software and in Hardware – A Performance Analysis of Workstation and Custom Computing Machine Implementation", Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines, 1996.
- [26] C. Aporntewan and P. Chongstitvatana, "A Hardware Implementation of the Compact Genetic Algorithm", Proc. IEEE Congress on Evolutionary Computation, Korea, pp. 624-629, May 2001.
- [27] B. Shackelford, G. Snider, R. J. Carter, E. Okushi, M. Yasuda, K. Seo and H. Yasuura, "A High-Performance, Pipelined, FPGA-Based Genetic Algorithm Machine", Genetic Programming and Evolvable Machines vol. 2, No. 1, Kluwer Academic Publishers, pp. 33-60, March 2001.
- [28] P. Graham and B. Nelson, "A Hardware Genetic Algorithm for the Traveling Salesman Problem on SPLASH 2", Proc. 5th Int. Workshop on Field Programmable Logic and Applications, pp. 352-361, 1995.
- [29] I. Skliarova, A. B. Ferrari, "FPGA-based Implementation of Genetic Algorithm for the Traveling Salesman Problem and its Industrial Application", in "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali, editors, pp. 77-87, 2002.
- [30] <http://www.alphadata.co.uk>
- [31] Xilinx, "The programmable logic data book", Xilinx, San Jose, 2000.
- [32] M. Salami, T. Hendtlass, "A Fitness Estimation Strategy for Genetic Algorithms", in "Developments in Applied Artificial Intelligence", T. Hendtlass, M. Ali, editors, pp. 502-513, 2002.
- [33] P. Merz, B. Freisleben, "Genetic local search for the TSP: New results", Proc. IEEE Int. Conf. On Evolutionary Computation, ICEC'97, pp. 159-164, 1997.
- [34] K. Bhatia, "Genetic Algorithms and the Traveling Salesman Problem", Computer Science and Engineering CSE292: New Age Algorithms, University of California at San Diego, 1994.