

Relatório Científico Parcial – Iniciação Científica

Processo FAPESP 2014/23564-0

Estudos de estruturas de dados eficientes para abordar o
problema de otimização U-curve

Beneficiário: Gustavo Estrela de Matos

Responsável: Marcelo da Silva Reis

Relatório referente aos trabalhos desenvolvidos entre 1 de janeiro e
10 de junho de 2015

Laboratório Especial de Toxinologia Aplicada, Instituto Butantan

São Paulo, 11 de Dezembro de 2016

Conteúdo

1 Resumo do Projeto Proposto

O problema de otimização U-curve pode ser utilizado para modelar problemas em diversas áreas; por exemplo, o problema de seleção de características em Reconhecimento de Padrões. Um algoritmo ótimo para abordar o problema U-curve é o U-Curve-Search (**UCS**). Esse algoritmo alcançou resultados promissores na solução desse problema, pois computa poucas vezes a função custo; porém, em sua atual implementação, **UCS** tem problemas de escalabilidade, o que se deve em grande medida à utilização de listas duplamente encadeadas para armazenar o controle do espaço de busca já percorrido pelo algoritmo. Dessa forma, propomos neste projeto de Iniciação Científica a investigação do uso de diagramas de decisão binária reduzidos e ordenados (ROBDDs) como solução de estrutura de dados para controlar o espaço de busca já percorrido pelo algoritmo **UCS**. Utilizamos ROBDDs para desenvolver o algoritmo **UCSR**, uma nova versão do **UCS** que foi implementada e testada utilizando o arcabouço featsel. Realizamos testes com instâncias artificiais e, em breve, utilizaremos também dados de problemas reais, tais como a etapa de seleção de características durante o desenho de W-operadores. Resultados iniciais mostraram que **UCSR** é mais eficiente do que **UCS** do ponto de vista de consumo de tempo computacional, o que sugere que o novo algoritmo é mais competitivo para resolver problemas práticos que possam ser descritos como um problema de otimização U-curve.

2 Atividades Realizadas

As seções a seguir apresentam as atividades realizadas nos seis primeiros meses de trabalho no projeto. Na seção ?? apresentamos conceitos fundamentais que foram estudados no início das atividades para a realização do projet. Na seção ?? mostramos as principais características da estrutura de dados ROBDD implementada no arcabouço `featse1`. Na seção ?? descrevemos as simplificações da busca em profundidade do UCS implementadas no UCSR3, UCSR4, UCSR5 e UCSR6. Na seção ?? explicamos o ajuste feito, entre tempo de execução e número de chamadas da função custo, implementados no algoritmo UCSR6. Na seção ?? apresentamos os resultados dos algoritmos implementados, comparando também com UCS e com Exhaustive Search (ES), uma implementação de busca exaustiva do espaço de busca.

2.1 Estudo do algoritmo UCS e de ROBDDs

Para criar novos algoritmos para solucionar o problema U-Curve, baseamo-nos na dinâmica do algoritmo **U-Curve-Search** (UCS), conforme apresentado em Reis [?]. Este algoritmo estrutura o espaço de busca como um recitulado Booleano e, através de uma busca em profundidade e com o auxílio de restrições ao espaço de busca, determina uma solução ótima para o problema.

No algoritmo UCS as restrições do espaço de busca são representadas por duas listas de restrições: inferior e superior. Essas listas armazenam vários elementos do espaço de busca, e cada elemento representa um intervalo que está restrito (i.e., um intervalo de elementos que foram removidos do espaço de busca corrente). Se um elemento X pertence a lista inferior (no caso dual, superior), temos que o intervalo $[\emptyset, X]$ ($[X, S]$) não está no espaço de busca. Estas listas foram implementadas com o uso de listas duplamente encadeadas, e para consultar se um elemento está no espaço de busca é necessário verificar se algum dos intervalos das listas cobre (i.e. contém) este elemento. Mais informações sobre o UCS podem ser encontradas em Reis [?], [?].

Visando melhorar o controle de espaço de busca, desenvolvemos novos algoritmos para o problema U-Curve utilizando diagramas de decisão binária ordenadas e reduzidas (*Reduced Ordered Binary Decision Diagrams* – ROBDDs [?]). Esses diagramas representam, através de árvores, funções Booleanas $f : B^n \rightarrow B$ ($B = \{0, 1\}$); portanto, podemos controlar o espaço de busca utilizando um ROBDD tal que $f(X)$ vale 1 se X está fora do espaço de busca e 0 caso contrário. Além disso, consultar se um elemento pertence ao espaço de busca em um ROBDD é linear em relação a n (cardinalidade do conjunto de características) como pode ser visto na figura ??.

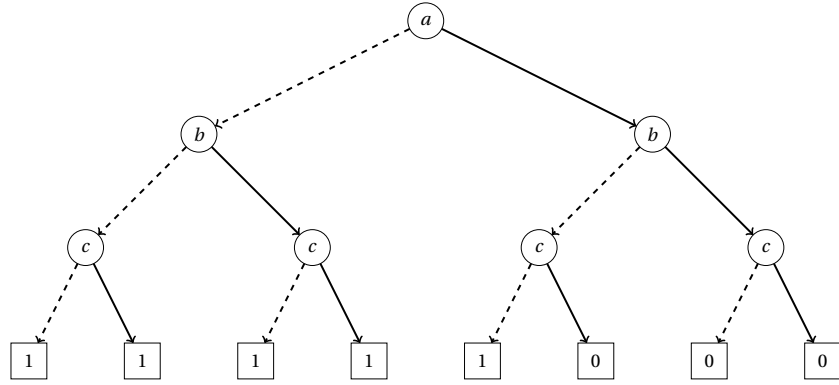
2.2 Implementação de ROBDDs no arcabouço `featsel`

A implementação do ROBDD foi feita utilizando o `featsel`, um arcabouço orientado a objetos desenvolvido em C++ que permite, entre outras funcionalidades, implementar algoritmos que utilizam elementos de um reticulado Booleano. A classe ROBDD possui métodos que permitem mudar o espaço de busca representado e é definida pela raiz da árvore, um objeto da classe `Vértice`.

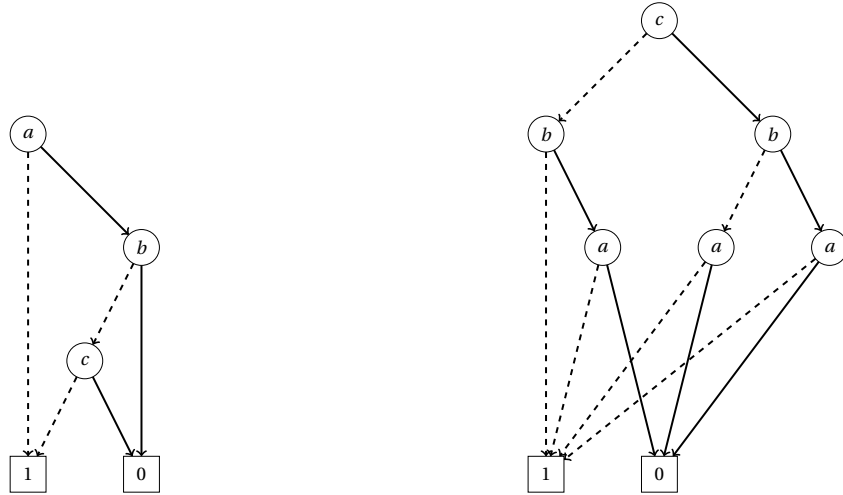
A classe `Vértice` possui uma estrutura que guarda um elemento do reticulado Booleano, para nós internos da árvore, ou um valor Booleano, para as folhas da árvore. Os elementos dos vértices são instâncias da classe `ElementSubset` do arcabouço `featsel`; os detalhes de implementação desta última podem ser encontradas em Reis [?]. Além disso, os vértices possuem duas referências para outros vértices permitindo a criação da árvore em si.

Para mudar a função que representa o espaço de busca atual no ROBDD, criamos o método `add_interval`, que recebe como parametro um elemento X e uma variável indicadora que decide qual intervalo será restrito, $[\emptyset, X]$ ou $[X, S]$. Neste método cria-se uma árvore temporária que representa o intervalo adicionado e une-se a ela o ROBDD original. Para realizar a união entre a árvore que descreve o espaço de busca corrente e a árvore temporária, implementamos o algoritmo proposto por Bryant e Randal [?].

A partir destas classes e também de outras existentes no arcabouço, criamos vários al-



(a)



(b)

(c)

Figura 1: exemplo do uso de ROBDDs para representar o espaço de busca corrente durante a execução do algoritmo UCS. Figura ?? : árvore binária que representa um espaço de busca arbitrário; um arco sólido (pontilhado) representa que o elemento da ponta superior está (não está) presente no subconjunto considerado. Uma folha tem valor 1 se um subconjunto está fora do espaço de busca corrente e 0 caso contrário. Figura ?? : ROBDD obtido através da redução da árvore de decisão binária da figura ?? . Figura ?? : ROBDD obtido através da redução de uma árvore de decisão binária cuja raiz é o elemento c ao invés de a ; a redução dessa árvore produz um ROBDD com mais elementos do que o exibido na figura ?? .

goritmos que utilizam ROBDDs para gerenciar o espaço de busca corrente: por exemplo, o UCSR2, que possui a mesma dinâmica que o algoritmo UCS, mas utiliza um ROBDD ao invés de duas listas de restrições. Também criamos os algoritmos UCSR3, UCSR4, UCSR5 e

UCSR6 que, além do uso de ROBDDs, têm em comum o fato de utilizarem uma dinâmica simplificada da busca em profundidade, que será explicada a seguir.

2.3 Simplificação da busca em profundidade (*DFS*) feita pelo algoritmo original

O algoritmo UCS possui uma busca em profundidade complexa que depende do uso de estruturas de dados para grafos. Essas estruturas armazenam elementos já visitados do espaço de busca mas que ainda não podem definir a ponta de um intervalo a ser removido do espaço de busca, sob o risco de perder o mínimo global. Tal risco ocorre em situações onde ocorrem empates no custo de dois elementos adjacentes entre si na cadeia do reticulado [?]. Todavia, o uso desse grafo resulta num consumo alto tanto de memória quanto de processamento.

Todavia, em instâncias de problemas de interesse prático que podem ser modelados como um problema U-curve, empates de custo entre elementos adjacentes é uma situação que raramente ocorre. Ademais, nas instâncias “difíceis” que utilizamos para realizar os testes, para cada cadeia do reticulado, é possível a ocorrência de no máximo um empate entre elementos adjacentes. Chamamos essa situação de “empate simples”. Por conta disso, passamos a investigar simplificações na busca em profundidade.

A primeira solução nova nos levou ao algoritmo UCSR3, que possui uma dinâmica bem parecida com a do UCS, porém sem guardar elementos visitados em um grafo, estratégia utilizada no UCS para determinar a sequência da busca em casos de empates. Neste novo algoritmo, no caso em que dois elementos $X \subset Y$ são adjacentes e possuem o mesmo custo, olhamos para um elemento $Z \subset X$ e se seu custo foi maior (menor) que o de X , restringimos o intervalo $[\emptyset, X]$ ($[X, S]$). O caso em que haja empate de custo entre Z e X (i.e., empate entre três ou mais elementos adjacentes dois a dois) não é tratado neste algoritmo.

Ainda com a hipótese de que há no máximo empates simples, elaboramos uma outra dinâmica, presente nos algoritmos UCSR4, UCSR5 e UCSR6. Nesta nova dinâmica fazemos um passeio pelo espaço de busca seguindo as seguintes regras, partindo de um elemento X do

espaço de busca, com Y adjacente a X :

- i. Se Y é adjacente superior (inferior) e tem custo menor que X , restringimos o intervalo $[\emptyset, X]$ ($[X, S]$) e passamos a avaliar Y ;
- ii. Se Y é adjacente superior (inferior) e tem custo maior que X , podemos restringir, ou não, como será discutido na próxima sessão, o intervalo $[Y, S]$ ($[\emptyset, Y]$);
- iii. Se Y é adjacente superior (inferior) e tem custo igual a X não fazemos nada;
- iv. Paramos quando não houverem mais elementos adjacentes a X .

No algoritmo **UCSR4**, dado um nó X e o conjunto de características $S = \{x_1, x_2, \dots, x_n\}$, o i -ésimo elemento adjacente a X visitado será Y_i tal que $(Y_i \cup X) \setminus (Y_i \cap X) = x_i$. Por exemplo, se $X = 00101$ então os vizinhos de X visitados (assumindo que todos serão visitados) serão, na ordem: 10101, 01101, 00001, 00111 e 00100. Note que, dependendo da composição de X , podemos visitar muitos elementos adjacentes superiores sem visitar nenhum elemento adjacente inferior, o que pode causar mais cálculos do que o necessário para visitar um elemento adjacente de menor custo.

Visando melhorar o percorrimento de elementos adjacentes, formulamos o **UCSR5**, que percorre os elementos adjacentes intercalando as visitas entre vizinhos superiores e inferiores, enquanto possível. Por exemplo, se $X = 00100$, uma possível sequência de vizinhos visitados (assumindo que todos serão visitados) é, na ordem: 10100, 00000, 00110, 01100 e 00101. Podemos ver na tabela ??, essa mudança no percorrimento de vizinhos trouxe melhorias no tempo de execução em relação ao **UCSR4**.

2.4 Ajuste entre tempo de execução e número de chamadas da função custo

Como vimos na seção ??, a nova dinâmica não especifica se devemos restringir um elemento Y adjacente a X e com custo maior. Se não restringirmos esse elemento corremos

o risco de ter que recomputar seu custo, pois, diferentemente do UCS, os algoritmos UCSR4, UCSR5 e UCSR6 não possuem estrutura que guarda o custo dos elementos visitados. Porém, o elemento Y pode estar contido em algum intervalo que será restrito em futuras iterações, tornando desnecessário restringir Y logo no momento que o visitamos.

Portanto, temos um *tradeoff* entre atualizações do ROBDD e cálculo de função de custo. Uma atualização nas restrições implica na construção de um ROBDD temporário, que representa o novo intervalo restrito, e sua união com o ROBDD que representa as novas restrições. Como visto em Randal e Bryant [?], essa operação tem complexidade $O(|R_1| * |R_2|)$, em que $|R_1|$ e $|R_2|$ são as quantidades de nós das árvores que representam os ROBDDs atual e temporário, respectivamente. Por outro lado, não restringir um elemento de custo maior pode fazer com que calculemos o custo de um mesmo nó do reticulado mais de uma vez, o que, dependendo da função de custo, pode levar a um grande número de cálculos desnecessários.

Com o intuito de comparar as duas escolhas possíveis, desenvolvemos os algoritmos UCSR5, que não restringe todo elemento adjacente com custo maior visitado, e o UCSR6, que restringe esses elementos. Como podemos verificar na tabela ?? o algoritmo UCSR5 teve desempenho em tempo um pouco melhor do que o UCSR6, que por sua vez teve menos cálculos da função custo, fato que podemos constatar na tabela ??.

2.5 Resultados dos experimentos

Para testar os algoritmos criados neste projeto, tendo como *benchmarking* os algoritmos UCS original e ES, utilizamos um programa em Perl que foi desenvolvido para esse fim. Nesse programa, para cada tamanho de S , diversas instâncias de problemas de tamanho $|S|$ foram geradas aleatoriamente a partir de reduções polinomiais do problema subset sum, que é NP-difícil [?]. Para cada instância e para cada algoritmo, o arcabouço *featsel* era chamado, guardando ao final da execução o tempo computacional que foi exigido e o número de chamadas da função custo. Os resultados foram então agrupados por tamanho de instância e tiveram suas médias calculadas. Nas tabelas ?? e ?? mostramos os resultados em termos de

tempo e de número de chamadas da função custo, respectivamente..

Instância		Tempo (segundos)						
$ S $	$2^{ S }$	UCSR6	UCSR5	UCSR4	UCSR3	UCSR2	UCS	ES
1	2	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	4	0.00	0.00	0.00	0.00	0.00	0.00	0.00
3	8	0.00	0.00	0.00	0.00	0.00	0.00	0.00
4	16	0.01	0.01	0.01	0.01	0.01	0.01	0.00
5	32	0.01	0.01	0.01	0.01	0.01	0.01	0.01
6	64	0.01	0.01	0.01	0.01	0.01	0.01	0.01
7	128	0.01	0.01	0.01	0.01	0.02	0.02	0.01
8	256	0.02	0.02	0.02	0.02	0.03	0.03	0.02
9	512	0.04	0.05	0.04	0.04	0.06	0.07	0.04
10	1024	0.08	0.09	0.08	0.08	0.13	0.13	0.09
11	2048	0.14	0.15	0.14	0.16	0.27	0.27	0.18
12	4096	0.44	0.47	0.44	0.61	0.99	0.71	0.36
13	8192	0.70	0.74	0.75	1.25	2.03	1.44	0.73
14	16384	2.42	2.37	2.47	4.60	6.90	4.82	1.52
15	32768	3.27	3.23	3.17	6.94	11.07	7.06	3.11
16	65536	27.11	25.75	26.73	57.83	83.44	44.56	6.46
17	131072	61.17	56.54	56.64	142.34	174.26	108.07	13.49
18	262144	293.86	262.42	273.24	699.01	740.60	366.96	27.93
19	524288	645.57	578.95	589.50	1335.98	1632.08	907.94	58.80

Tabela 1: resultado de tempo dos testes dos algoritmos desenvolvidos neste projeto para instâncias difíceis do problema U-curve. Cada linha i representa o tempo médio obtido rodando cada algoritmo sobre 20 instâncias diferentes de tamanho 2^i .

Como podemos observar nas tabelas ?? e ??, o algoritmo UCSR2 possui desempenho próximo ao UCS, provavelmente por apresentarem dinâmica muito similar entre si. Por conta do UCS ter um controle muito complicado, o UCSR2 não teve bons resultados de tempo e foi mais lento que o UCS na maioria dos testes. Esse fato provavelmente se deu devido ao *overhead* trazido pelo uso do ROBDD para gerenciar o espaço de busca corrente; o mesmo aconteceu com o algoritmo UCSR3, que continuando com uma dinâmica muito parecida com o UCS, apenas foi melhor do que o UCSR2, provavelmente porque o primeiro dispensa o uso do grafo para controlar os elementos visitados na busca em profundidade. O primeiro algoritmo que apresentou melhores resultados de tempo que o UCS foi o algoritmo UCSR4; porém, como foi apresentado na seção ??, esse algoritmo tinha possibilidades evidentes de melhorias. Ao

Instância		Número de chamadas da função custo						
$ S $	$2^{ S }$	UCSR6	UCSR5	UCSR4	UCSR3	UCSR2	UCS	ES
1	2	2.00	2.00	2.00	2.00	2.00	2.00	2
2	4	3.80	4.00	4.20	3.80	3.80	3.90	4
3	8	6.50	7.30	7.30	6.50	6.45	6.50	8
4	16	12.85	13.70	14.05	12.80	12.60	12.85	16
5	32	19.95	21.30	21.40	21.00	19.75	19.20	32
6	64	31.25	33.40	33.35	32.50	31.25	36.40	64
7	128	50.70	58.15	62.15	53.90	50.85	56.00	128
8	256	86.50	94.85	101.70	92.90	89.60	94.45	256
9	512	170.60	191.00	203.60	179.75	171.25	174.50	512
10	1024	273.75	322.25	336.75	286.10	272.70	280.05	1024
11	2048	425.15	500.40	532.10	440.80	430.90	441.20	2048
12	4096	1119.00	1271.80	1362.35	1121.55	1096.65	1077.45	4096
13	8192	1437.10	1637.35	1692.30	1524.75	1437.00	1412.90	8192
14	16384	2710.20	3203.10	3317.95	2874.50	2791.55	2702.25	16384
15	32768	3181.60	3522.90	3764.50	3251.45	3147.50	3088.65	32768
16	65536	9637.20	11544.65	11603.25	9923.35	9613.25	9380.85	65536
17	131072	10327.90	12122.05	12301.70	10680.95	10370.95	9962.50	131072
18	262144	21838.55	25847.15	26551.45	22676.30	21562.70	20920.90	262144
19	524288	27931.55	32460.70	34011.60	28795.00	27823.70	27099.90	524288

Tabela 2: resultado de número de chamadas da função custo dos testes dos algoritmos desenvolvidos neste projeto para instâncias difíceis do problema U-curve. Cada linha i representa o número médio de chamadas da função custo obtido rodando cada algoritmo sobre 20 instâncias diferentes de tamanho 2^i .

aplicarmos a primeira dessas melhorias, criando o algoritmo **UCSR5**, obtivemos o melhor consumo de tempo entre os algoritmos desenvolvidos nesse projeto; porém, **UCSR5** calcula muitas vezes a função custo. Como apresentamos na seção ??, **UCSR6** possui desempenho de tempo um pouco pior do que o **UCSR5**, mas calcula menos vezes a função custo; portanto, **UCSR6** pode ser considerado o melhor algoritmo desenvolvido até aqui neste projeto.

3 Plano de atividades para o próximo período

Os algoritmos desenvolvidos até agora no projeto utilizam uma ordem fixa de variáveis no ROBDD. Como vimos nas figuras ?? e ??, diferentes ordens de variáveis podem resultar em ROBDDs de tamanhos distintos; conseqüentemente, escolher uma ordem que minimize o tamanho do ROBDD redundaria em uma economia de tempo e de memória. Porém, achar uma ordenação ótima para um ROBDD é um problema NP-difícil [?]. Portanto, atualmente estamos estudando o uso de heurísticas para determinar ordenações de variáveis do ROBDD, considerando como primeira tentativa algoritmos genéticos [?, ?].

Após implementação desses algoritmos, faremos testes com a melhor versão desenvolvida com instâncias de interesse prático, como, por exemplo, em projeto de W-operadores [?]. Os resultados serão incluídos em um manuscrito científico que será preparado em breve.

Estas atividades estão planejadas para serem concluídas até o final de julho, pois o aluno foi selecionado para o programa "Ciência Sem Fronteiras" e estará fora do Brasil por um ano realizando intercâmbio supervisionado na Texas A&M University, College Station, Estados Unidos. Solicitaremos em breve uma interrupção da bolsa atual a partir de 1 de agosto; caso a FAPESP aprove o pedido, ao retornar de seu intercâmbio o aluno estudará a aplicação dos algoritmos desenvolvidos em generalizações do problema de otimização U-curve: por exemplo, em problemas nos quais o reticulado que organiza o espaço de busca não seja Booleano.

Referências

- [1] Reis, Marcelo S. “Minimization of decomposable in U-shaped curves functions defined on poset chains—algorithms and applications.” PhD thesis, Institute of Mathematics and Statistics, University of São Paulo, Brazil, (2012).
- [2] Reis, Marcelo S., Carlos E. Ferreira, and Junior Barrera. “The U-curve optimization problem: improvements on the original algorithm and time complexity analysis.” arXiv preprint arXiv:1407.6067, (2014).
- [3] Bryant, Randal E. “Graph-based algorithms for boolean function manipulation.” IEEE Transactions on Computers, 100.8 (1986): 677-691.
- [4] Brace, Karl S., Richard L. Rudell, and Randal E. Bryant. “Efficient implementation of a BDD package.” Proceedings of the 27th ACM/IEEE design automation conference, (1991).
- [5] Rice, Michael and Sanjay Kulhari. “A survey of static variable ordering heuristics for efficient BDD/MDD construction.” University of California, Tech. Rep., (2008).
- [6] Bollig, Beate and Wegener, Ingo. “Improving the variable ordering of OBDDs is NP-complete.” IEEE Transactions on Computers, 45.9 (1996): 993–1002.
- [7] Rehan, Sanisha and Bansal, Manu. “Performance Comparison among different Evolutionary Algorithms in terms of Node Count Reduction in BDDs.” International Journal of VLSI and Embedded Systems, (2013) 491–496.
- [8] Mathew, Tom V. “Genetic algorithm.” Department of Civil Engineering, Indian Institute of Technology, Bombay (2005).