Relatório Científico Final – Iniciação Científica

Processo FAPESP 2016/25959-7

Projeto de algoritmos baseados em florestas de posets para o problema de otimização U-curve

Beneficiário: Gustavo Estrela de Matos

Responsável: Marcelo da Silva Reis

Relatório referente aos trabalhos desenvolvidos entre 1 de maio e 31 de dezembro de 2017

Laboratório Especial de Toxinologia Aplicada, Instituto Butantan

São Paulo, 7 de Janeiro de 2018

Conteúdo

1 Resumo do Projeto Proposto

O problema U-curve é uma formulação de um problema de otimização que pode ser utilizado na etapa de seleção de características em Aprendizado de Máquina, com aplicações em desenho de modelos computacionais de sistemas biológicos. Não obstante, soluções propostas até o presente momento para atacar esse problema têm limitações do ponto de vista de consumo de tempo computacional e/ou de memória, o que implica na necessidade do desenvolvimento de novos algoritmos. Nesse sentido, em 2012 foi proposto o algoritmo `Poset--Forest-Search` (`PFS`) [1], que organiza o espaço de busca em florestas de posets. Esse algoritmo foi implementado e testado, com resultados promissores; todavia, novos melhoramentos são necessários para que o `PFS` se torne uma alternativa competitiva para resolver o problema U-curve. Neste projeto propomos modificações ao `PFS` na escolha de caminhos de percorrimento da floresta de busca, e na estrutura de dados utilizada para armazenar este grafo, com o uso de diagramas de decisão binária ordenados (OBDDs) [3]; também propomos a criação de uma versão paralela e escalável do algoritmo `PFS`. Além disso, propomos a criação de um algoritmo baseado no `PFS` que tenha características de um algoritmo de aproximação, no qual o critério de aproximação da solução ótima se baseie no teorema da navalha de Ockham. Os algoritmos desenvolvidos serão implementados no arcabouço *featsel* [2] e testados com instâncias artificiais e também reais, com conjuntos de dados de aprendizado de máquina retirados do University of California Irvine (UCI) Machine Learning Repository.

2 Atividades Realizadas

Uma implementação do `PFS`, feita por Reis, está disponível no arcabouço *featsel*. Usamos esta implementação como base para estudar as modificações feitas ao `PFS`.

2.1 Estudo de algoritmos baseados em florestas

O algoritmo `Poset-Forest-Search` (`PFS`) é um algoritmo ótimo para resolver o problema de otimização U-Curve e serviu de base para a criação da maioria dos algoritmos elaborados neste trabalho. O `PFS` é uma generalização de um outro algoritmo mais simples, o `U-curve-Branch-and-Bound` (`UBB`), que é um algoritmo *branch-and-bound* ótimo que decompõe o espaço de busca em uma árvore, e acha o mínimo global do problema fazendo ramificações e podas nesta árvore.

A árvore de busca do `UBB` permite que a procura pelo mínimo ocorra de maneira parecida com uma busca em profundidade, que percorre cadeias do reticulado Booleano de subconjuntos menores para maiores. Sempre que o custo de um subconjunto X_i aumenta em comparação ao anterior X_j no percorrimento, a hipótese de que a função de custo é decomponível em curvas em U garante que a subárvore que começa em X_i pode ser removida do espaço de busca; chamamos este procedimento de poda. O algoritmo `UBB` tem, entretanto, uma limitação, pois quando a função de custo do problema é monótona não-crescente, a condição de poda nunca é verdadeira e o espaço de busca inteiro é visitado, o que compromete a escalabilidade do algoritmo.

O `PFS` enfrenta esta limitação ao fazer percorrimentos de cadeias do espaço de busca em duas direções, de conjuntos menores para maiores (como faz o `UBB`) e também o contrário. Para fazer isso, este algoritmo decompõe o espaço de busca em duas árvores, uma para cada direção de percorrimento. Com a criação de duas estruturas para representar o mesmo espaço de busca, torna-se necessário a atualização de uma estrutura sempre que a outra sofrer mudanças, e isto implica na utilização de florestas ao invés de árvores para representar o espaço de busca no `PFS`. Resumidamente, uma iteração do deste algoritmo deve escolher uma direção de percorrimento; fazer o percorrimento com poda (de maneira similar ao `UBB`); e, finalmente, atualizar a floresta dual a que foi percorrida.

2.2 Modificações do PFS na escolha de raízes

Para se fazer um percorrimento no espaço de busca, o algoritmo `PFS` deve escolher uma direção de percorrimento, isto é, uma das duas florestas que representam o espaço de busca, e também uma raiz da floresta escolhida. Esta escolha é feita de maneira arbitrária, e por conta disso, investigamos como diferentes escolhas podem afetar o desempenho do algoritmo.

Na implementação de Reis, a estrutura de dados utilizada para armazenar raízes é a *map* do `C++`. Escolhe-se nesta implementação o primeiro ou último elemento da estrutura, o que coincide com a primeira ou última raiz quando estas estão ordenadas lexicograficamente por seus vetores característicos. Em nosso trabalho, experimentamos duas modificações para esta escolha:

- de maneira aleatória e uniformemente provável;
- e de maneira determinística, com a raiz de maior sub-árvore completa. Neste caso, o tamanho da sub-árvore completa é o tamanho deste grafo quando nenhuma poda foi feita.

A justificativa para se fazer uma escolha uniforme entre as raízes é ter um algoritmo que não possui viés na escolha de percorrimentos, assim podemos investigar se o viés da escolha arbitrária feita na implementação de Reis compromete a execução do algoritmo. Para fazer a implementação da escolha aleatória e uniforme fizemos uma pequena modificação ao código de Reis. Dado uma floresta $\mathcal{F} = \{r_1, r_2, \ldots, r_l\}$ sorteia-se um número pseudo-aleatório a entre 1 e l e escolhe-se a raiz r_a para o percorrimento.

A escolha de raiz com maior sub-árvore foi feita com a intuição de que percorrimentos em árvores maiores implicariam em maiores podas e consequentemente menos nós visitados no espaço de busca. A decomposição do espaço de busca em árvore feito por Reis faz com que ordenar as raízes por tamanho decrescente de sub-árvores completas coincida com uma ordenação lexicográfica da direita para a esquerda dos vetores característicos das raízes. Para fazer isto, basta modificar a ordenação feita pela estrutura *map* do `C++`.

Tabela 1: Comparação entre os algoritmos `PFS` e `PFS_RAND`. O tempo de execução do segundo é maior do que o primeiro enquanto a quantidade de chamadas da função custo é parecida em ambos.

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$\|S\|$	$2^{\|S\|}$	`PFS`	`PFS_RAND`	`PFS`	`PFS_RAND`
10	1024	0.013 ± 0.003	0.014 ± 0.003	590.8 ± 198.5	599.5 ± 177.5
11	2048	0.019 ± 0.004	0.022 ± 0.007	1114.8 ± 331.3	1090.1 ± 350.3
12	4096	0.029 ± 0.008	0.036 ± 0.013	1848.6 ± 600.8	1835.7 ± 683.0
13	8192	0.060 ± 0.018	0.090 ± 0.039	4314.4 ± 1496.4	4201.1 ± 1580.7
14	16384	0.100 ± 0.041	0.191 ± 0.110	7323.4 ± 3318.9	7333.8 ± 3161.0
15	32768	0.180 ± 0.076	0.453 ± 0.311	12958.1 ± 5654.0	12807.5 ± 5753.7
16	65536	0.406 ± 0.185	1.715 ± 1.400	27573.8 ± 12459.5	27036.9 ± 12687.5
17	131072	0.717 ± 0.397	5.416 ± 5.266	48176.2 ± 26938.3	47852.1 ± 26427.6
18	262144	1.325 ± 0.754	15.890 ± 17.726	84417.9 ± 48587.7	84025.0 ± 48882.4
19	524288	2.771 ± 1.603	69.600 ± 82.342	167659.1 ± 99686.7	164612.1 ± 102018.3

Chamamos o algoritmo que faz a escolha uniforme das raízes de `PFS_RAND`, e os resultados de tempo de execução e número de chamadas da função de custo são apresentados na tabela 1. Podemos observar que esta modificação ao `PFS` não foi benéfica, pois comparando com a implementação de Reis o tempo de execução médio aumentou, e o número médio de chamadas da função de custo continua parecido. O fato do número de chamadas da função de custo não ter diferenças significativas implica que esta escolha de raiz não trouxe mudanças a dinâmica do algoritmo original, logo o tempo a mais de execução veio do código que faz a escolha da raiz. Isto é feito com o percorrimento da estrutura de *map*, o que adiciona tempo linear (sobre a quantidade de raízes) a cada escolha de raiz.

Chamamos de `PFS_LEFTMOST` o algoritmo que faz a escolha da raiz com maior sub-árvore completa, e a tabela 2 mostra resultados de tempo de execução e número de chamadas da função de custo desta variante comparado a implementação de Reis. Podemos observar que esta modificação também não foi benéfica pois, além do maior tempo de execução, o número de chamadas da função de custo aumentou. Isto significa que o comportamento do algoritmo foi o contrário a intuição que motivou a modificação, pois mais nós do espaço de busca foram visitados.

Tabela 2: Comparação entre os algoritmos `PFS` e `PFS_LEFTMOST`. O tempo de execução e também o número de chamadas da função custo é maior para o `PFS_LEFTMOST`.

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$\|S\|$	$2^{\|S\|}$	PFS	PFS_LEFTMOST	PFS	PFS_LEFTMOST
10	1024	0.013 ± 0.002	0.023 ± 0.004	606.1 ± 133.5	665.0 ± 165.8
11	2048	0.020 ± 0.004	0.042 ± 0.010	1122.1 ± 351.2	1316.6 ± 382.2
12	4096	0.032 ± 0.008	0.078 ± 0.024	2183.7 ± 733.2	2515.8 ± 871.3
13	8192	0.054 ± 0.017	0.160 ± 0.061	3887.7 ± 1389.9	4716.8 ± 1777.8
14	16384	0.107 ± 0.034	0.345 ± 0.133	7851.2 ± 2793.0	9506.8 ± 3673.9
15	32768	0.196 ± 0.085	0.672 ± 0.274	13780.3 ± 6049.9	17071.6 ± 7005.1
16	65536	0.348 ± 0.189	1.271 ± 0.661	24106.5 ± 13159.9	30055.6 ± 15363.6
17	131072	0.785 ± 0.361	3.137 ± 1.476	52369.0 ± 24751.2	67585.6 ± 30978.4
18	262144	1.445 ± 0.657	6.146 ± 3.032	92095.9 ± 42566.6	120635.7 ± 58039.0
19	524288	3.298 ± 1.883	13.881 ± 7.595	199151.0 ± 112167.8	256078.6 ± 135958.4

2.3 Mudificação do PFS no armazenamento de raízes

A quantidade de raízes durante a execução do `PFS` pode ser grande, e por isso a estrutura utilizada para o seu armazenamento deve ser eficiente para operações como inserções, remoções e consultas. Motivados por resultados de um trabalho de iniciação científica anterior (FAPESP processo #2014/23564-0), experimentamos o uso de diagramas de decisão binária ordenados (OBDDs) para o armazenamento de raízes.

A estrutura de OBDD é um diagrama capaz de representar funções Booleanas $f : \{0,1\}^n \to \{0,1\}$. Para utilizarmos esta estrutura no armazenamento de raízes, criamos uma OBDD que representa a função que mapeia para valores 1 os subconjuntos que são raízes da floresta e para 0 os demais. Para de fato armazenar as raízes tivemos que modificar a estrutura usual de OBDD, então ao invés de usar folhas com valores 0 ou 1, usamos folhas que armazenam nós do espaço de busca ou um ponteiro nulo. A figura 1 mostra um exemplo de floresta do espaço de busca e a correspondente representação em nossa estrutura de OBDD.

Como a estrutura de dados que armazena as raízes foi modificada, é necessário definir o abordagem para escolha de raiz na etapa de percorrimento. Utilizamos então a abordagem de escolha aleatória e uniforme, pois esta mostrou uma dinâmica similar a implementação de Reis.

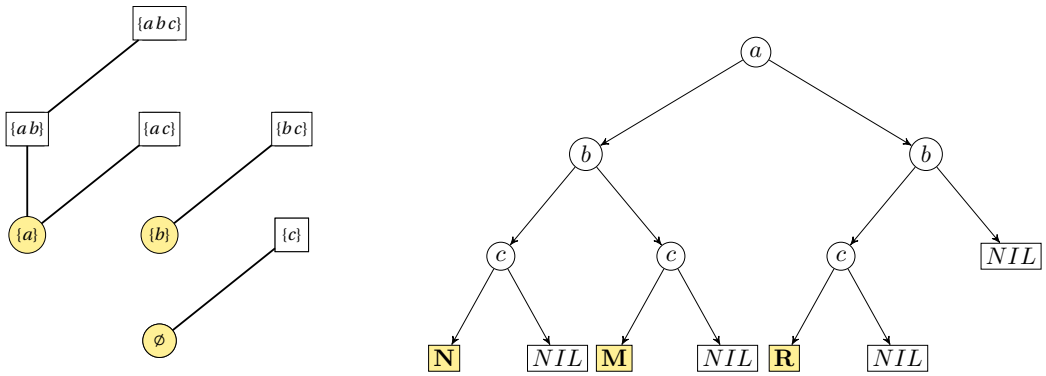

Figura 1: Exemplo de OBDD que representa uma floresta do `PFS`*. Esta OBDD contém os nós* **N**, **M** *e* **R**, *que representam respectivamente os subconjuntos* 000, 010 *e* 100. *As folhas* NIL *indicam que os subconjuntos de tal caminho na OBDD não são raízes na floresta, como por exemplo os subconjuntos* $11X$

Tabela 3: Comparação entre os algoritmos `PFS` *e* `OPFS`*. O número de chamadas médio da função custo é parecido enquanto o tempo de execução é maior para o* `OBDD`*.*

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$\|S\|$	$2^{\|S\|}$	`PFS`	`OPFS`	`PFS`	`OPFS`
10	1024	0.013 ± 0.003	0.018 ± 0.003	598.0 ± 192.8	635.5 ± 171.9
11	2048	0.020 ± 0.004	0.029 ± 0.007	1152.1 ± 314.7	1117.9 ± 336.4
12	4096	0.031 ± 0.010	0.049 ± 0.013	2024.1 ± 751.6	2048.2 ± 700.9
13	8192	0.057 ± 0.017	0.097 ± 0.033	3996.3 ± 1431.6	3973.4 ± 1462.6
14	16384	0.094 ± 0.038	0.171 ± 0.063	6634.8 ± 2944.0	6906.5 ± 2786.5
15	32768	0.182 ± 0.079	0.323 ± 0.156	13140.1 ± 6020.6	12711.2 ± 6319.7
16	65536	0.370 ± 0.169	0.660 ± 0.314	25658.2 ± 11606.7	25303.4 ± 12169.5
17	131072	0.819 ± 0.370	1.480 ± 0.665	53344.9 ± 24350.4	53217.2 ± 24154.5
18	262144	1.515 ± 0.905	2.736 ± 1.626	94677.6 ± 54496.3	94079.4 ± 55435.6
19	524288	2.612 ± 1.869	4.818 ± 3.355	156150.5 ± 107369.8	156021.8 ± 107516.8
20	1048576	6.085 ± 3.900	11.550 ± 7.661	344144.1 ± 212627.1	343229.2 ± 212624.4

Chamamos o algoritmo que usa OBDDs para armazenamento de raízes de `OPFS`. A tabela 3 mostra uma comparação de tempo de execução e número de cálculos da função de custo entre o algoritmo original e esta modificação. Observamos que o número de chamadas da função custo é similar enquanto o tempo de execução é maior para o algoritmo `OPFS`, portanto esta modificação não trouxe melhoras ao algoritmo `PFS`.

2.4 Paralelização do PFS

A decomposição do espaço de busca em uma floresta feita pelo algoritmo `PFS` separa tal espaço em partes disjuntas que são árvores. Estruturas disjuntas como estas podem ser percorridas de maneira paralela sem ou com poucas interferências entre linhas de processamento, portanto a paralelização do algoritmo `PFS` pode trazer ganhos em tempo de execução. Desta maneira, paralelizamos este algoritmo, e para isso utilizamos a biblioteca *OpenMP*.

Entretanto, apesar da etapa de percorrimento das árvores ser quase independente entre linhas de processamento paralelas, a etapa de atualização da floresta dual a que foi percorrida é dependente. Além de possuir seções críticas, em que apenas uma linha de processamento pode avançar de cada vez, a atualização da floresta dual pode ter condições de corrida, isto é, o resultado da atualização pode depender da ordem em que as linhas de processamento avançam no código. Condições de corrida não tratadas podem inclusive levar a inconsistências na representação do espaço de busca.

Este algoritmo paralelo foi chamado de `PPFS`. Testamos o desempenho deste algoritmo comparado ao `PFS` e os resultados são apresentados na tabela 4. Podemos notar que o tempo de execução do algoritmo paralelo é pior do que a versão original, portanto o trabalho adicional de controle de linhas de processamento não é recompensado pelo ganho de tempo de processamento paralelo. Isto aconteceu porque a etapa de ramificação, que pode ser feita em paralelo sem grandes dificuldades, é curta em relação a etapa de atualização de floresta dual, que possui diversas seções críticas e também um código adicional para controle de condições de corrida.

Tabela 4: Comparação entre os algoritmos `PFS` e `PPFS`. O algoritmo `PPFS` apresenta um número similar de média de chamadas da função custo ao `PFS`, mas possui tempo de execução médio maior.

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$\|S\|$	$2^{\|S\|}$	PFS	PPFS	PFS	PPFS
10	1024	0.011 ± 0.001	0.146 ± 0.027	643.6 ± 133.0	626.5 ± 151.1
11	2048	0.017 ± 0.004	0.227 ± 0.062	1151.0 ± 359.7	1135.6 ± 381.1
12	4096	0.029 ± 0.007	0.385 ± 0.113	2173.1 ± 652.5	2139.2 ± 710.3
13	8192	0.049 ± 0.015	0.640 ± 0.237	3839.8 ± 1376.6	3743.5 ± 1532.9
14	16384	0.104 ± 0.037	1.337 ± 0.513	8175.9 ± 3037.4	8026.4 ± 3303.7
15	32768	0.163 ± 0.078	2.010 ± 1.096	12459.6 ± 6164.5	12062.2 ± 6897.2
16	65536	0.360 ± 0.163	4.483 ± 2.030	27027.3 ± 12397.0	26835.7 ± 12446.9
17	131072	0.664 ± 0.362	8.072 ± 4.370	48001.9 ± 26149.2	48093.4 ± 26233.2
18	262144	1.250 ± 0.690	14.341 ± 8.062	85880.9 ± 47950.8	86050.8 ± 49186.8
19	524288	2.936 ± 1.629	34.639 ± 19.074	198503.0 ± 108116.1	197832.5 ± 110659.6
20	1048576	5.024 ± 3.097	61.038 ± 38.250	321495.8 ± 198004.3	318507.0 ± 199354.3

2.5 Elaboração do UBB-PFS

2.6 Elaboração de um algoritmo de aproximação

2.7 Testes com instâncias reais do problema de seleção de características

3 Avaliação e disseminação de resultados

4 Conclusão

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., Gustavo Estrela, Carlos E. Ferreira e Junior Barrera. "featsel: A framework for benchmarking of feature selection algorithms and cost functions". SoftwareX 6 (2017), pp. 193-197.

[3] Bryant, Randal E. "Graph-based algorithms for boolean function manipulation."IEEE Transactions on Computers, 100.8 (1986): 677-691.

[4] Gustavo E. Matos e Marcelo S. Reis. "Estudos de estruturas de dados eficientes para abordar o problema de otimização U-curve". Relatório científico final FAPESP, Instituto Butantan, Brasil (2015).