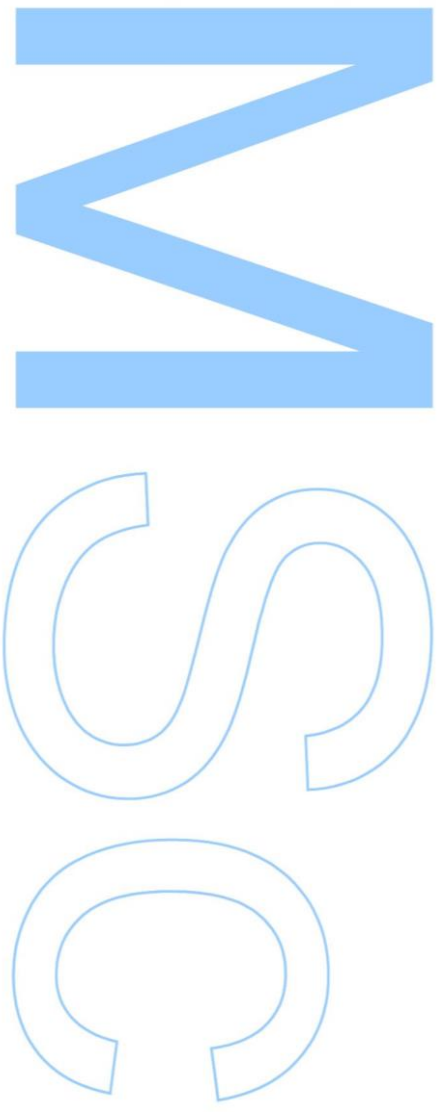
Análise de Desempenho de Redes Bayesianas de Grande Escala em Julia

Mariana Isabel Mourão Lopes
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2019

Orientador
Inês de Castro Dutra, Professor Auxiliar, Faculdade de Ciências da Universidade do Porto



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Resumo

É mostrada uma comparação entre o desempenho dos algoritmos de aprendizagem de Redes Bayesianas implementados na recente linguagem de programação Julia e na linguagem R. São comparadas execuções sequenciais e paralelas em ambas com redes de vários tamanhos. Foi possível verificar pela execução sequencial que os algoritmos implementados pela linguagem de programação Julia tinham tempos de execução muito mais elevados do que os equivalentes em R. A implementação mais rápida em Julia é a da *GraphSearchStrategy* GreedyHillClimbing e a mais demorada é a *GraphSearchStrategy* K2GraphSearch.

Foi adotada uma estratégia de otimização, tentando maximizar o uso do *hardware* disponível, pela abordagem *multi-threading* nativa presente em Julia. Para os conjuntos de dados mais pequenos, ASIA, ALARM e HAILFINDER, não foi possível verificar uma grande diminuição nos tempos de execução com o aumento do número de *threads* utilizadas, havendo até tempos de execução piores. Para conjuntos de dados maiores, como é o caso do ANDES e MUNIN, foi possível verificar uma diminuição dos tempos de execução, sendo o algoritmo que obteve mais melhorias no tempo de execução o da *GraphSearchStrategy* GreedyHillClimbing, chegando a melhorias de mais de 2 horas para o conjunto de dados MUNIN.

Abstract

A comparison is shown between the performance of Bayesian network learning algorithms implemented in the recent Julia programming language and the R language. Sequential and parallel executions are compared in both.

It was possible to verify by sequential execution that the algorithms implemented by the Julia programming language had much higher execution times than the equivalent in R. The fastest implementation in Julia is the GreedyHillClimbing *GraphSearchStrategy*.

An optimization strategy was adopted to try to maximize the use of available hardware by the native multi-threading approach present in Julia. For the smaller data sets, ASIA, ALARM and HAILFINDER, there was an increase in runtimes with increasing number of threads, and even worse runtimes. For larger data sets, such as ANDES and MUNIN, it was possible to verify a decrease in the execution times, and the algorithm that obtained the most improvements in the execution time is the GreedyHillClimbing *GraphSearchStrategy*, reaching over 2 hours improvements to the MUNIN dataset.

Palavras Chave: Redes Bayesianas, Aprendizagem de Estrutura, Aprendizagem de Parâmetros, Computação Paralela

Começo por agradecer à minha orientadora Professora Inês Dutra, por todo o apoio, orientação e paciência. Gostaria de agradecer também à minha familia por todo o apoio que me dão em tudo inclusive na vida académica, sem eles dificilmente teria conseguido alcançar esta etapa da minha vida. Por último mas não menos importante, quero agradecer ao Sérgio Neto por toda a paciência e apoio moral.

Conteúdo

Lista de Tabelas

Lista de Figuras

Acrónimos

AIC *Akaike Information Criterion*

BDe *Bayesian Dirichlet equivalent*

BIC *Bayesian Information Criterion*

CI Causa Indutiva

CC Complexidade Computacional

CP Computação Paralela

DAG *Directed Acyclic Graph*

DAGs *Directed Acyclic Graphs*

DPC Distribuição de Probabilidade Condicional

DPCs Distribuições de Probabilidade Condicionais

ME Maximização de Expectativas

FCI *Fast Causal Inference*

GDA *Gaussian Discriminant Analisys*

GPL *General Public License*

HC *Hill Climbing*

IAMB *Incremental Association Markov Blanket*

IA Inteligência Artificial

LLVM *Low Level Virtual Machine*

MPI *Message-Passing Interface*

MDL *Minimum Description Length*

MIMD *Multiple-Instruction,Multiple-Data*

MISD *Multiple-Instruction, Single-Data*

NB *Naive Bayes*

PVM *Parallel Virtual Machine*

RB Rede Bayesiana

RBs Redes Bayesianas

RL Regressão Logística

SIMD *Single-Instruction, Multiple-Data*

SISD *Single-Instruction, Single-Data*

SVM *Support Vector Machine*

TPC Tabela de Probabilidade Condicional

TPCs Tabelas de Probabilidades Condicionais

TI Teoria de Informação

TIC Testes de Independência Condicional

Capítulo 1

Introdução

Redes Bayesianas (RBs) são modelos gráficos probabilísticos representados por *Directed Acyclic Graphs (DAGs)*, em que cada vértice representa uma variável aleatória de um domínio e as arestas representam relações de dependência direta entre as variáveis, assim, os vértices não conectados representam variáveis que são condicionalmente independentes.

As RBs também são conhecidas como redes causais, redes de crença ou gráficos de dependência probabilística.

As primeiras pesquisas desenvolvidas sobre RBs datam o inicio dos anos 1980, utilizadas na época para ajudar em previsões nos sistemas de Inteligência Artificial (IA).

Têm vindo a ser utilizadas para a solução de vários tipos de problemas, em diversas áreas, como por exemplo em: biologia, negócios e finanças, jogos de computador, visão computacional, hardware e software de computador, medicina, planeamento, psicologia, reconhecimento de fala, controle e diagnóstico de veículos, previsão do tempo [1].

Existem várias linguagens de programação que implementam algoritmos para lidar com RBs, como é o caso do C++, Perl, MATLAB, Python e R, mas alguns desses algoritmos não conseguem ser escaláveis para redes de grande dimensão. Recentemente, foi criada uma nova biblioteca de software que implementa algoritmos de RBs, neste caso, em Julia. Por ser muito recente, Julia ainda não foi exaustivamente testada com conjuntos de dados ou problemas reais e/ou de grande dimensão. Portanto, não está disponível nenhuma análise do seu desempenho e limitações, nem nenhuma comparação com outra linguagem de programação em relação a algoritmos de RBs.

Neste trabalho pretendemos apresentar uma comparação de desempenho, entre os algoritmos de RBs implementados na linguagem de programação Julia e na linguagem R. Assim será possível verificar limitações e/ou melhorias na execução de algoritmos de RBs com Julia.

Nos próximos capítulos apresentamos em Conceitos Fundamentais (Capítulo 2) uma breve explicação dos conceitos e termos considerados essenciais, para compreensão do tema do projeto é dada também uma introdução aos algoritmos de aprendizagem de RBs seguidos das implementações existentes nas linguagens Julia e R; a seguir é mostrado o estado atual

da arte (Capítulo 3) que está direta e indiretamente associado a este estudo. No Capítulo 4 (Desenho e Desenvolvimento) está descrito o software e hardware utilizado assim como todo o processo de escolha, preparação e descrição dos dados; a apresentação de resultados e a sua análise são efetuadas no Capitulo 5. Por último são apresentados as principais conclusões e destaques do projeto assim como trabalhos a realizar (Capítulo 6).

Capítulo 2

Conceitos Fundamentais

Neste capítulo vamos apresentar alguns conceitos importantes para entender a terminologia e expressões de Redes Bayesianas (RBs) que serão utilizadas ao longo da dissertação. Vão ser abordados também os princípios fundamentais das linguagens utilizadas e algumas ideias de Computação Paralela (CP), no contexto de RBs.

2.1 Redes Bayesianas

As RBs são a principal tecnologia para lidar com probabilidades em Inteligência Artificial (IA). Uma Rede Bayesiana (RB) é uma estrutura gráfica que permite representar e raciocinar sobre um domínio com incerteza. Numa RB os vértices representam um conjunto de variáveis aleatórias de um domínio e as arestas representam dependências diretas entre elas. Neste projeto apenas vamos considerar variáveis discretas, nas quais as relações entre elas são quantificadas por Distribuições de Probabilidade Condicionais (DPCs) associadas a cada vértice, representadas por um *Directed Acyclic Graph* (DAG).

Os valores das variáveis devem ser mutuamente exclusivos, o que significa que a variável deve assumir exatamente um dos valores de cada vez. Tipos comuns de vértices discretos são:

- Vértices Booleanos: representam proposições que assumem valores binários *true* (T) e *false* (F).
- Valores ordenados: por exemplo, um vértice chamado Poluição pode representar a exposição de uma pessoa à poluição e assumir um dos valores {baixo, médio ou alto}.
- Valores integrais: por exemplo, um vértice chamado Idade pode representar a idade de uma pessoa e ter valores possíveis de 1 a 120.

2.1.1 Terminologia

A estrutura, ou topologia de uma rede deve capturar relações qualitativas entre variáveis. Em particular, dois vértices devem estar diretamente conectados se um afeta ou causa o outro, com a aresta a indicar a direção do efeito.

Ao falar sobre estrutura de rede é utilizada a seguinte metáfora: um vértice é pai de um filho, se houver uma aresta do primeiro para o segundo. Assim sendo, se houver uma cadeia de vértices, um vértice é antecessor de outro se aparecer mais cedo na cadeia, enquanto que um vértice é descendente de outro se aparecer mais tarde na cadeia.

Outro conceito útil é a cobertura de Markov de um vértice, que consiste nos vértices pais de um vértice, nos seus filhos e nos pais dos seus filhos.

A seguinte terminologia também é frequentemente utilizada e vem da analogia de "árvore" (embora as RBs em geral sejam grafos e não árvores): um vértice sem pais é chamado raiz, enquanto que um vértice sem filhos é chamado folha. Numa RB, significa que os vértices raiz representam as causas originais, enquanto os vértices folha representam os efeitos finais.

Por convenção, para facilitar o exame visual da estrutura da RB, as redes são geralmente dispostas de modo que as arestas apontem de cima para baixo. O que corresponde à "árvore" da RB ser representada com as raízes no topo e folhas na parte inferior.

Depois de especificada a estrutura da RB, a próxima etapa é quantificar as relações entre os vértices conectados, o que é feito pela especificação da Distribuição de Probabilidade Condicional (DPC) para cada vértice. Como estamos a considerar apenas variáveis discretas, assume a forma de uma Tabela de Probabilidade Condicional (TPC).

Primeiro, para cada vértice, precisamos de todas as combinações possíveis de valores para os vértices pais. A cada combinação é chamada de instanciação do conjunto de vértices pais. Para cada instanciação diferente dos valores dos vértices pais, é preciso especificar a probabilidade de o filho receber cada um dos seus valores.

Vértices raiz também têm uma TPC associada, contudo esta apenas contém uma linha que representa as probabilidades à priori. Claramente, se um vértice tem muitos pais ou se os pais têm um grande número de valores, as Tabelas de Probabilidades Condicionais (TPCs) podem ficar muito grandes. O tamanho das TPCs é exponencial no número de pais. Para redes booleanas uma variável com $\mathtt{n}$ pais requer uma TPC com $2^{\mathtt{n}+1}$ probabilidades, pois para cada um dos valores booleanos da variável tem $2^{\mathtt{n}}$ valores, então $2 * 2^{\mathtt{n}} = 2^{\mathtt{n}+1}$.

2.1.2 Funcionamento

As RBs são consideradas representações de distribuições de probabilidade conjunta. Considerando que uma RB contém $\mathtt{v}$ vértices, de $\mathtt{X}_1$ até $\mathtt{X}_\mathtt{v}$, obtidos nessa ordem. Um valor particular na distribuição conjunta é representado por $P(\mathtt{X}_1 = \mathtt{x}_1, \mathtt{X}_2 = \mathtt{x}_2, ..., \mathtt{X}_\mathtt{v} = \mathtt{x}_\mathtt{v})$, ou mais

compactamente, $P(\mathtt{x}_1, \mathtt{x}_2, ..., \mathtt{x}_\mathtt{v})$. A regra da cadeia da teoria de probabilidades permite-nos fatorizar as probabilidades conjuntas para:

$P(\mathtt{x}_1, ..., \mathtt{x}_\mathtt{v}) = \prod_i P(\mathtt{x}_\mathtt{i} | \mathtt{x}_1, ..., \mathtt{x}_{\mathtt{i}-1})$

Sabendo que a estrutura de uma RB implica que o valor de um vértice específico depende apenas dos valores dos seus pais, reduz a:

$P(\mathtt{x}_1, ..., \mathtt{x}_\mathtt{v}) = \prod_i P(\mathtt{x}_\mathtt{i} | \mathtt{Pais}(\mathtt{X}_\mathtt{i}))$

dados os $\mathtt{Pais}(\mathtt{X}_\mathtt{i}) \subseteq \{\mathtt{x}_1, ..., \mathtt{x}_{\mathtt{i}-1}\}$.

2.1.2.1 Algoritmo de construção de rede de Pearl

Dada a condição $\mathtt{Pais}(\mathtt{X}_\mathtt{i}) \subseteq \{\mathtt{x}_1, ..., \mathtt{x}_{\mathtt{i}-1}\}$ permite-nos construir uma rede dada uma determinada ordem dos vértices usando o algoritmo de construção de rede de Pearl [2].

O algoritmo de construção 1 simplesmente processa cada vértice por ordem, adicionando-o à rede existente e adicionando arestas de um conjunto mínimo de pais, de modo que o conjunto de pais torne o vértice atual condicionalmente independente de todos os vértices anteriores.

Algorithm 1 Construção de rede de Pearl

1: Escolhe um conjunto de variáveis relevantes $\{\mathtt{X}_\mathtt{i}\}$ que descrevam o domínio.
2: Escolhe uma ordem para as variáveis, $< \mathtt{X}_1, ..., \mathtt{X}_\mathtt{v} >$.
3: Enquanto existirem variáveis:
   (a) Adiciona a próxima variável $\mathtt{X}_\mathtt{i}$ à rede.
   (b) Adiciona arestas ao vértice $\mathtt{X}_\mathtt{i}$ de um conjunto mínimo de vértices já existentes na rede, $\mathtt{Pais}(\mathtt{X}_\mathtt{i})$, de modo que a seguinte propriedade de independência condicional seja satisfeita:
   $P(\mathtt{X}_\mathtt{i} | \mathtt{X}'_1, ..., \mathtt{X}'_\mathtt{m}) = P(\mathtt{X}_\mathtt{i} | \mathtt{Pais}(\mathtt{X}_\mathtt{i}))$
   onde $\mathtt{X}_\mathtt{i} | \mathtt{X}'_1, ..., \mathtt{X}'_\mathtt{m}$ são todas as variáveis que precedem $\mathtt{X}_\mathtt{i}$ e que não estão em $\mathtt{Pais}(\mathtt{X}_\mathtt{i})$.
   (c) Define a TPC para $\mathtt{X}_\mathtt{i}$.

2.1.2.2 Compacidade e ordenação de vértices

Usando o algoritmo de construção 1 anterior, fica claro que uma ordem de variáveis diferente pode resultar numa estrutura de rede diferente, com ambas a representar a mesma distribuição de probabilidade conjunta.

O objetivo é criar a RB mais compacta possível, por três razões. Primeiro, quanto mais compacto for o modelo, mais tratável, porque vai ter menos valores de probabilidade que exigem especificação; vai ocupar menos memória do computador; as atualizações de probabilidade serão mais eficientes em termos computacionais. Segundo, redes excessivamente densas não conseguem representar independências explicitamente. E terceiro, redes excessivamente densas falham em representar as dependências causais do domínio.

A ordem ideal seria adicionar primeiro as causas raiz, e depois as variáveis que elas influenciam diretamente e continuar até atingir as folhas.

2.1.2.3 Independência condicional

As RBs que satisfazem a propriedade Markov (a qual diz que: não há dependências diretas entre as variáveis que não sejam mostradas explicitamente por meio de arestas), expressam explicitamente independências condicionais das distribuições de probabilidade. A relação entre independência condicional e a estrutura da RB é importante para entender como as RB funcionam.

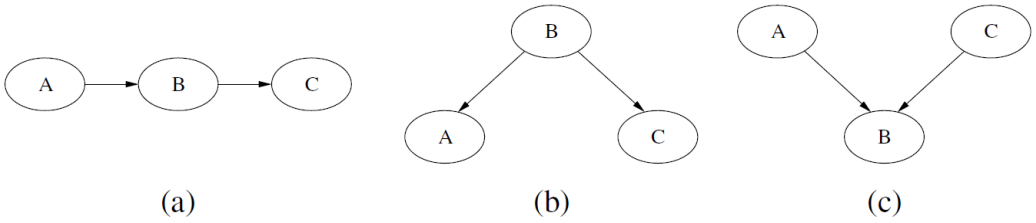


Figura 2.1: (a) cadeia causal; (b) causa comum; (c) efeito comum. (fonte:[3])

- Cadeia Causal: Considerando uma cadeia causal de três nós, onde A causa B que por sua vez causa C, como mostrado na Figura 2.1(a). O que significa que a probabilidade de C, dado B, é exatamente a mesma que a probabilidade de C, dado B e A. Saber que A ocorreu não faz diferença nas nossas crenças sobre C se já sabemos que B ocorreu.
- Causa comum: Duas variáveis A e C com uma causa comum estão representadas na Figura 2.1(b), ou seja, B influencia diretamente as variáveis A e C.
- Efeito comum: Um efeito comum é representado por uma v-estrutura da rede, como na Figura 2.1(c). Representa uma situação em que um vértice (o efeito) tem duas causas. Efeitos comuns (ou seus descendentes) produzem exatamente a estrutura de independência condicional oposta à das cadeias e causas comuns. Ou seja, os pais são marginalmente independentes, mas dependem das informações sobre o efeito comum (ou seja, elas são condicionalmente dependentes). Assim, se observarmos o efeito e, digamos, descobrirmos que uma das causas está ausente, isso aumenta a probabilidade da outra causa.

Portanto, agora podemos ver que construir redes com uma ordem contra a ordem causal pode, e geralmente leva a uma complexidade adicional na forma de arestas extra.

2.1.2.4 D-separação

Dada a propriedade Markov, é possível determinar se um conjunto de vértices X é independente de outro conjunto Y, dado um conjunto de vértices de evidência E. Para fazer isso, introduzimos a noção de d-separação (de separação direção-dependente).

Definição 2.1.1 Caminho (caminho não direcionado). Um caminho entre dois conjuntos de vértices X e Y é qualquer sequência de vértices entre um membro de X e um membro de Y, de modo que cada par de vértices adjacente seja conectado por uma aresta (independentemente da direção) e nenhum vértice aparece na sequência duas vezes.

Definição 2.1.2 Caminho bloqueado. Um caminho é bloqueado, dado um conjunto de vértices E, se houver um vértice Z no caminho para o qual pelo menos uma das três condições é válida:

1. Z está em E e Z tem uma aresta no caminho que leva para dentro e uma aresta para fora (corrente).
2. Z está em E e Z possui duas arestas de caminho que saem (causa comum).
3. Nem Z nem nenhum descendente de Z estão em E, e as duas arestas de caminho levam a Z (efeito comum).

Definição 2.1.3 d-separação. Um conjunto de vértices E, d-separa dois outros conjuntos de vértices X e Y se todo caminho de um vértice em X para um vértice em Y está bloqueado, dado E.

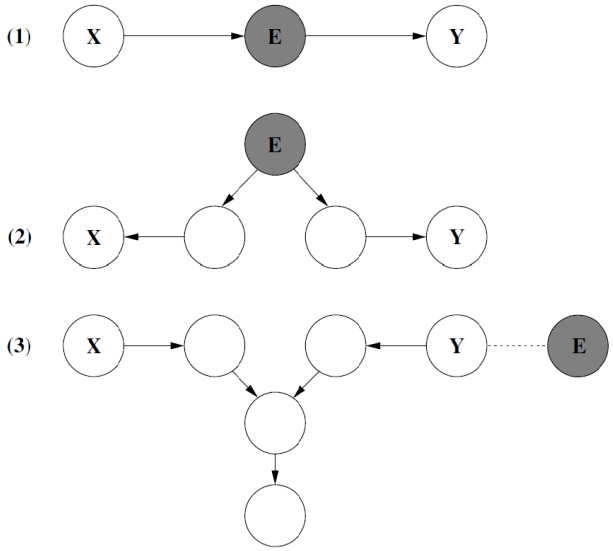


Figura 2.2: Exemplos de 3 tipos de situações nas quais o caminho de X para Y pode ser bloqueado, dado a evidência E. Em cada caso, X e Y são d-separados por E (fonte:[3])

Se X e Y são d-separados por E, então X e Y são condicionalmente independentes, dado E (dada a propriedade Markov). Exemplos dessas três situações de bloqueio são mostrados na Figura 2.2.

Simplificamos o uso de vértices únicos em vez de conjuntos de vértices; e os vértices de evidência E estão sombreados.

Estes conceitos são importantes para a implementação de algoritmos eficientes de inferência e de construção de uma rede.

2.1.3 Aprendizagem

A tarefa de ajustar uma RB é geralmente chamada de aprendizagem e realizada em duas etapas diferentes, que correspondem à seleção de um modelo e à estimação de parâmetros. Esta última pode ser realizada pela estimação de parâmetros das distribuições locais implícitas pela estrutura obtida na etapa anterior. Ambas vão ser abordadas em seguida.

2.1.3.1 Aprendizagem de estrutura

A primeira etapa para ajustar uma RB é chamada aprendizagem de estrutura e consiste em encontrar o DAG que codifica as independências condicionais presentes nos dados. Os algoritmos de aprendizagem de estrutura de RBs podem ser agrupados em duas categorias:

- Algoritmos baseados em restrições: São algoritmos que aprendem a estrutura da rede analisando as relações probabilísticas implícitas pela propriedade de Markov das RBs com Testes de Independência Condicional (TIC) e, em seguida, constroem um grafo que satisfaça a declaração de d-separação correspondente.
- Algoritmos baseados em pontuações: São algoritmos que atribuem uma pontuação a cada grafo candidato a RB e tentam maximizar com algum algoritmo de pesquisa heurística.

Algoritmos baseados em restrições

Os algoritmos baseados em restrições são todos baseados no algoritmo de Causa Indutiva (CI) [4], que fornece um quadro teórico para aprender os modelos da estrutura causal usando TIC e pode ser resumido em três etapas:

1. Primeiro aprende o esqueleto da rede (o grafo não direcionado subjacente à estrutura da rede).
2. Em seguida define todas as direções das arestas que fazem parte da v-estrutura.
3. Por último define as direções das restantes arestas que fazem parte da v-estrutura satisfazendo conforme necessário a restrição de aciclicidade.

Um grande problema do algoritmo de CI é que as duas primeiras etapas não podem ser aplicadas na forma descrita no Algoritmo 2 [3] a qualquer problema do mundo real devido ao número exponencial de possíveis relacionamentos de independência condicional. Isso levou ao desenvolvimento de algoritmos aperfeiçoados, os quais aprendem primeiro a cobertura de Markov

de cada vértice na rede. Esta etapa preliminar simplifica bastante a identificação dos vizinhos de cada vértice, pois a pesquisa pode ser limitada à sua cobertura de Markov. Como resultado, o número de TIC executados pelo algoritmo de aprendizagem e a sua complexidade computacional geral são significativamente reduzidos.

Algorithm 2 CI

1: Para cada par de variáveis A e B no conjunto V, procura o subconjunto mínimo $S_{AB} \subset V$(*incluindo* $S = \emptyset$), de modo que A e B sejam independentes, dado S_{AB} e $A, B \notin S_{AB}$. Se não houver esse conjunto, coloca uma aresta não direcionada entre A e B.
2: Para cada par de variáveis não adjacentes A e B com um vizinho comum C, verifica se $C \notin S_{AB}$. Se não for verdade, define a direção das arestas $A - C$ e $C - B$ para $A \rightarrow C$ e $C \leftarrow B$.
3: Define a direção das arestas que ainda não estão direcionadas aplicando recursivamente as duas regras a seguir:
(a) se A é adjacente a B e existe um caminho estritamente direcionado de A para B (um caminho que leva de A a B que não contém arestas não direcionadas), define a direção de $A - B$ para $A \rightarrow B$;
(b) se A e B não são adjacentes, mas $A \rightarrow C$ e $C - B$, altera o último para $C \rightarrow B$.
4: Retorna o DAG resultante (parcialmente concluído).

Algoritmos baseados em pontuações

Os algoritmos baseados em pontuações (também conhecidos como algoritmos de pesquisa e pontuação) representam a aplicação de técnicas gerais de otimização heurística ao problema de aprendizagem de estrutura de uma RB. Cada grafo candidato recebe uma pontuação, que reflete a sua qualidade de ajuste, que o algoritmo tenta maximizar. Alguns exemplos dessa classe de algoritmos são:

- Algoritmos de pesquisa gananciosos, como *Hill Climbing (HC)* com *random restarts* ou pesquisa tabu [5]. São algoritmos que exploram o espaço de pesquisa iniciando uma estrutura de rede (geralmente o grafo vazio) e adicionam, removem ou invertem uma aresta de cada vez, até que a pontuação não possa mais ser melhorada (consulte o Algoritmo 3).
- Algoritmos genéticos, imitam a evolução natural através da seleção iterativa de modelos "mais aptos" e da hibridação das suas características [6]. Neste caso, o espaço de pesquisa é explorado através dos operadores estocásticos de *crossover* (que combina a estrutura de duas redes) e mutação (que introduz alterações aleatórias).
- *Simulated annealing* [5]. É um algoritmo que executa uma pesquisa local estocástica que aceita alterações que aumentam a pontuação da rede e, ao mesmo tempo, permite alterações que a diminuem com uma certa probabilidade.

Uma revisão abrangente destas heurísticas, bem como abordagens relacionadas do campo da IA, é fornecida em Russell e Norvig (2009) [7].

Algorithm 3 HC

1: Escolhe uma estrutura de rede G sobre V, geralmente (mas não necessariamente) vazia.
2: Calcula a pontuação de G, indicada como $Score_G = Score(G)$.
3: Define $maxscore = Score_G$.
4: Repete as etapas a seguir, enquanto a pontuação máxima aumentar:
   (a) para cada adição, remoção ou inversão de aresta possível que não resulta numa rede cíclica:
   i. calcula a pontuação da rede modificada $G^*, Score_{G^*} = Score(G^*)$:
   ii. se $ScoreG^* > Score_G$, define $G = G^*$ e $Score_G = Score^*_G$.
   (b) atualiza o $maxscore$ com o novo valor de $Score_G$.
5: Retorna o DAG G.

2.1.3.2 Aprendizagem de parâmetros

A segunda etapa para ajustar uma RB é chamada aprendizagem de parâmetros. Uma vez que a estrutura do grafo é conhecida da etapa anterior, a tarefa de estimar e atualizar os parâmetros da distribuição global é bastante simplificada se for aplicada a propriedade Markov.

As distribuições locais na prática envolvem um pequeno número de variáveis. Existem duas abordagens principais para a estimativa dos parâmetros na literatura: uma baseada na estimativa de máxima verossimilhança, que se concentra em calcular a probabilidade dados os parâmetros e a outra baseada na estimativa bayesiana, a qual tenta calcular a probabilidade dos parâmetros dado os dados.

O número de parâmetros necessários para identificar exclusivamente a distribuição global, que é a soma do número de parâmetros das distribuições locais, é reduzido porque as relacões de independência condicional codificadas na estrutura da rede ajustam grande parte do espaço de parâmetros.

No entanto, a estimativa de parâmetros ainda é problemática em muitas situações. Por exemplo, em algumas situações podemos ter tamanhos de amostra muito menores que o número de variáveis incluídas no modelo. O que é típico de conjuntos de dados biológicos, como *microarrays*, que têm poucas (dez ou cem) observações e milhares de genes. Nesse cenário, denominado "pequeno n, grande p", as estimativas têm uma alta variabilidade, a menos que seja tomado um cuidado especial na aprendizagem de estrutura e parâmetros [8].

2.1.3.3 Escolher distribuições, TIC e pontuações de rede

Existem muitas opções possíveis para as funções de distribuição global e local, dependendo da natureza dos dados e dos objetivos da análise. No entanto, a literatura concentrou-se principalmente em dois casos:

Variáveis multinomiais: usadas para conjuntos de dados discretos/categóricos e geralmente

denominadas caso discreto. As distribuições global e local são multinomiais, e as últimas são representadas como TPCs. Essa é a suposição mais comum na literatura e são chamadas de RBs discretas.

Variáveis normais multivariadas: essa representação é usada para conjuntos de dados contínuos e, portanto, referida como caso contínuo. A distribuição global é normal multivariada, enquanto as distribuições locais são variáveis aleatórias normais univariadas ligadas por restrições lineares. As distribuições locais são de fato modelos lineares nos quais os pais desempenham o papel de variáveis explicativas. Esses RBs são chamadas redes gaussianas bayesianas.

Nesta dissertação como ja referido estamos apenas a considerar o caso discreto. Neste, a escolha de um conjunto específico de distribuições globais e locais também determina quais os TIC e quais pontuações da rede podem ser usadas para aprender a estrutura de uma RB. Estão disponíveis vários TIC da Teoria de Informação (TI) e estatística clássica para uso em algoritmos de aprendizagem baseados em restrições.

Os TIC e as pontuações da rede para dados discretos são as funções das TPCs implícitas pela estrutura gráfica da rede através das frequências observadas $\{n_{ijk}, i = 1, ..., R, j = 1, ..., C, k = 1, ..., L\}$ para variáveis aleatórias X e Y e todas as configurações das variáveis condicionadas $\mathbf{Z}$. Sendo R o número de observações no conjunto de dados, C denota a cardinalidade do conjunto de pais de X na estrutura da rede, ou seja, o número de valores diferentes nos quais os pais de X podem estar instanciados e L a cardinalidade de X. Dois TIC comuns são os seguintes:

- *Mutual Information* [9], uma medida de distância da teoria de informação definida como

$$MI(X, Y|\mathbf{Z}) = \sum_{i=1}^{R} \sum_{j=1}^{C} \sum_{k=1}^{L} \frac{n_{ijk}}{n} \log \frac{n_{ijk} n_{++k}}{n_{i+k} n_{+jk}}$$

  É proporcional ao *teste de razão log-verossimilhança* G^2 (diferem por um fator $2n$, onde n é o tamanho da amostra) e está relacionado ao desvio dos modelos testados.

- O teste clássico X^2 *de Pearson* para tabelas de contingência,

$$X^2(X, Y|\mathbf{Z}) = \sum_{i=1}^{R} \sum_{j=1}^{C} \sum_{k=1}^{L} \frac{(n_{ijk} - m_{ijk})^2}{m_{ijk}}, m_{ijk} = \frac{n_{i+k} n_{+j_k}}{n_{++k}}$$

Nos dois casos, a hipótese de independência nula pode ser testada usando a distribuição assintótica $\mathcal{X}^2_{(R-1)(C-1)L}$ ou a abordagem de permutação de Monte Carlo [10].

Outras opções possíveis são *Fisher's exact test* e o estimador de retração para *mutual information* [11] [12].

As pontuações da rede frequentemente encontradas na literatura são as seguintes:

- As pontuações *likelihood* e *log-likelihood*, que são equivalentes à medida de entropia usada pelo WEKA [13].

- As pontuações *Akaike Information Criterion* (AIC) e *Bayesian Information Criterion* (BIC), definidas como:

$$AIC = \log L(X_1, ..., X_v) - d$$
$$BIC = \log L(X_1, ..., X_v) - \frac{d}{2} \log n$$

  O último é equivalente ao *minimum description length* descrito por [14] e usado como uma pontuação de RB em [15].

- **Logaritmo do *Bayesian Dirichlet equivalent score*** (bde): uma pontuação equivalente à *posterior density* de *Dirichlet* [16].

- ***Logaritmo da pontuação K2*** (k2): outra *posterior density* de Dirichlet definida como

$$K2 = \prod_{i=1}^{v} K2(X_i)$$
$$K2(X_i) = \prod_{j=1}^{L_i} \frac{(R_i - 1)!}{(\sum_{k=1}^{R_i} n_{ijk} + R_i - 1)!} \prod_{k=1}^{R_i} n_{ijk}!$$

  e originalmente usada em algoritmos de aprendizagem de estrutura do mesmo nome. Ao contrário da pontuação bde, k2 não é equivalente.

Diz-se que essas funções de pontuação são *score equivalent*, pois atribuem a mesma pontuação a redes pertencentes à mesma classe de equivalência. Também são decompostas nos componentes associados a cada vértice, o que é uma vantagem computacional significativa ao aprender a estrutura da rede (as únicas partes da pontuação que precisam ser computadas são as que diferem entre as redes que estão a ser comparadas).

2.2 A linguagem Julia

É uma linguagem de programação de alto nível (uma linguagem com um nível de abstração elevado, longe do código máquina e mais próximo da linguagem humana) projetada para atender requisitos de computação de alto desempenho numérico e científico.

O seu desenvolvimento começou em 2009, por Jeff Bezanson, Stefan Karpinski, Viral B. Shah e Alan Edelman e foi divulgada uma versão de código aberto em Fevereiro de 2012. Desde o seu lançamento foram utilizadas várias versões até à versão estável mais recente, Julia 1.1.1, entregue em 16 de Maio de 2019.

É escrita em C, C++ e Scheme, usando a estrutura do compilador *Low Level Virtual Machine* (LLVM) (uma infraestrutura de compilador escrita em C++, desenvolvida para otimizar tempos de compilação, ligação e execução de programas escritos em várias linguagens de programação), enquanto a maior parte da biblioteca padrão de Julia é implementada na própria linguagem.

Alguns aspetos incomuns do projeto Julia incluem ter um sistema de tipos com polimorfismo paramétrico (uma maneira de tornar uma linguagem mais expressiva, mantendo a segurança total de tipos estática) e a adoção de *multiple dispatch* como o seu principal paradigma de programação. Permite computação concorrente, paralela e distribuída e chamada direta de bibliotecas C e Fortran. O aspeto mais notável da implementação de Julia de acordo com os seus criadores é o seu desempenho, que muitas vezes é o dobro do código C, totalmente otimizado.

Inclui bibliotecas eficientes para cálculos de vírgula flutuante, álgebra linear, geração de números aleatórios e correspondência de expressões regulares. *BayesNets* é uma das muitas bibliotecas disponíveis, e suporta representação, inferência e aprendizagem para RBs, em seguida vamos apresentar algumas das suas funcionalidades.

2.2.1 Biblioteca *BayesNets*

A biblioteca *BayesNets* de Julia implementa vários algoritmos de aprendizagem de estrutura e parâmetros de RBs, os algoritmos para aprendizagem de estrutura apenas implementam abordagens baseadas em pontuações.

2.2.1.1 Aprendizagem

Para a aprendizagem de estrutura, na biblioteca *BayesNets* existem três abordagens disponíveis definidas por *GraphSearchStrategy*.

Uma delas é a K2GraphSearch() que implementa o algoritmo de aprendizagem de estrutura K2, que pode ser executado em tempo polinomial e requer a especificação de uma ordenação topológica dos vértices. Assim, alterar a ordem resulta numa estrutura diferente. As restantes funções são GreedyHillClimbing() e ScanGreedyHillClimbing() que implementam algoritmos de pesquisa heuristica HC (consultar algoritmo 3) e estão implementadas apenas para RBs discretas.

A $ScoringFunction()$ permite extrair uma métrica de pontuação para uma DPC. A pontuação negativa do BIC é implementada em *NegativeBayesianInformationCriterion* e utilizada pelo algoritmo de aprendizagem de estrutura K2.

Para a aprendizagem de parâmetros, a biblioteca *BayesNets* utiliza a função fit(), que consegue aprender cada DPC a partir dos dados.

2.3 A linguagem R

R é um ambiente computacional e uma linguagem de programação que se vem especializando em manipulação, análise e visualização gráfica de dados.

Foi criado originalmente por Ross Ihaka e por Robert Gentleman, e atualmente tem sido

desenvolvido pelo esforço colaborativo de pessoas em vários locais do mundo.

Muitas das funções padrão do R são escritas na própria linguagem . Para tarefas computacionais intensivas, podem ser ligados e chamados códigos C, C++ e Fortran durante a execução.

O código fonte de R está disponível sob a licença GNU *General Public License* (GPL) e as versões binárias pré-compiladas são fornecidas para Windows, Macintosh, e muitos sistemas operativos.

A linguagem R é muito utilizada entre estatísticos e analistas de dados para desenvolver *software* de estatística e análise de dados. R é altamente expansível através do uso de bibliotecas, com dados e funções específicas para diferentes áreas de estudo. O *bnlearn* é uma biblioteca de R que inclui vários algoritmos para aprender a estrutura de RBs com variáveis discretas e contínuas.

2.3.1 Biblioteca *bnlearn*

O *bnlearn* oferece uma ampla variedade de algoritmos de aprendizagem de estrutura, abordagens de aprendizagem de parâmetros e técnicas de inferência, com vários algoritmos de pontuações e TIC disponíveis. É também a única biblioteca que mantém uma clara separação entre a estrutura de uma rede e a distribuição de probabilidades associada, que são implementadas como duas classes diferentes de objetos R.

2.3.1.1 Aprendizagem

Para a aprendizagem de estrutura o *bnlearn* implementa os seguintes algoritmos baseados em restrições (os respetivos nomes de funções estão entre parênteses):

- *Grow-Shrink*(gs): baseado no algoritmo de *Grow-Shrink Markov blanket* [17].
- *Incremental association*(iamb): baseado no algoritmo *Incremental Association Markov Blanket (IAMB)* [18].
- *Fast incremental association*(fast.iamb): uma variante do IAMB[19].
- *Interleaved incremental association*(inter.iamb): outra variante do IAMB[18].

O único algoritmo de aprendizagem baseado em pontuações disponível é a pesquisa gananciosa HC (consultar algoritmo 3) no espaço dos grafos direcionados.

2.3.2 Outras bibliotecas R para redes bayesianas

Existem várias bibliotecas no *CRAN* que lidam com RBs. Podem ser divididas em duas categorias: aquelas que lidam com a aprendizagem de estrutura e as que se concentram apenas na aprendizagem de parâmetros e inferência. As bibliotecas *bnlearn*, *deal*, *pcalg* e catnet enquadram-se na primeira categoria.

O *deal* implementa aprendizagem de estrutura e parâmetros usando uma abordagem bayesiana e lida com dados discretos, contínuos e mistos (assumindo uma distribuição condicional gaussiana). A estrutura da rede é aprendida com pesquisas gananciosas, como descrito no Algoritmo 3, com *posterior density* da rede como função de pontuação e reinicia aleatoriamente para evitar máximos locais.

O *pcalg* fornece uma implementação de código aberto do algoritmo PC [20] e foi projetado especificamente para estimar e medir efeitos causais. Lida com dados discretos e contínuos e pode explicar os efeitos de variáveis latentes na rede. O que é alcançado através de um algoritmo de PC modificado conhecido como *Fast Causal Inference* (FCI) [21].

O *catnet* concentra-se em RBs estáticas e discretas. A aprendizagem da estrutura é realizada em duas etapas. Primeiro, a ordem dos vértices do grafo é aprendida a partir dos dados usando *simulated annealing*; a aprendizagem e previsão de parâmetros também são implementadas. Além disso, uma extensão dessa abordagem de dados mistos (assumindo uma distribuição de mistura gaussiana) foi disponibilizada pelo *CRAN* na biblioteca *mugnet*.

As bibliotecas *gRbase* e *gRain* enquadram-se na segunda categoria. Concentram-se na manipulação de parâmetros da rede, previsão e inferência, assumindo que todas as variáveis são discretas. Nem o *gRbase* nem o *gRain* implementam qualquer estrutura ou algoritmo de aprendizagem de parâmetros; portanto, a RB deve ser completamente especificada pelo utilizador.

Tabela 2.1: Características das bibliotecas R.

	bnlearn	*catnet*	*deal*	*pcalg*	*gRbase*	*gRain*
Dados Discretos	Sim	Sim	Sim	Sim	Sim	Sim
Dados Continuos	Sim	Não	Sim	Sim	Sim	Não
Dados Mistos	Não	Não	Sim	Não	Não	Não
Aprendizagem baseada em restrições	Sim	Não	Não	Sim	Não	Não
Aprendizagem baseada em pontuações	Sim	Sim	Sim	Não	Não	Não
Estimação de parâmetros	Sim	Sim	Sim	Sim	Não	Não

2.4 Computação paralela(CP) para redes bayesianas

A maioria dos problemas na teoria de RBs tem uma complexidade que, na pior das hipóteses, escala exponencialmente com o número de variáveis. Embora os algoritmos mais novos sejam projetados para melhorar a escalabilidade, é inviável analisar dados com mais do que algumas

centenas de variáveis. A CP fornece uma maneira de resolver esse problema, fazendo um melhor uso do *hardware* moderno. Nesta secção, vamos fornecer uma breve visão geral da história e dos conceitos fundamentais da CP.

2.4.1 Fundamentos de computação paralela

Uma maneira simples e eficaz de avaliar o desempenho de um programa que implementa um algoritmo é medir o tempo de execução e os recursos necessários para o executar com êxito. Neste aspeto, o desempenho é influenciado por vários fatores, tanto de *hardware* como de *software*.

O *hardware* em que um programa executa tem influência no seu tempo de execução. O desempenho do *hardware* é medido pelo número de operações que pode executar num determinado período de tempo. Duas medidas deste tipo são as operações de entrada/saída por segundo e operações de vírgula flutuante por segundo.

O desempenho é limitado pelas restrições de produção do *hardware* e cada vez mais por leis físicas fundamentais, como a velocidade da luz e a física de dissipação de calor.

O desempenho do *software* na implementação de um algoritmo depende tanto da sua Complexidade Computacional (CC) como da arquitetura do *software*. A CC classifica os algoritmos de acordo com a sua dificuldade inerente, com especial atenção ao seu comportamento à medida que o tamanho da entrada aumenta (escalabilidade). Portanto, a única maneira de melhorar a escalabilidade é desenvolver um algoritmo melhor para o problema em questão. O que geralmente não é possível, seja porque já estamos a utilizar um algoritmo ideal ou porque o problema em questão não pode ser resolvido de maneira eficiente (ou seja, a sua CC é mais do que polinomial no tamanho da entrada no pior caso). Tais problemas são conhecidos como *NP-hard* e são comuns na teoria de RBs, tanto para aprendizagem de estrutura [22] quanto para inferência [23].

Por outro lado, a arquitetura do *software* pode fazer uma diferença significativa no desempenho geral do programa. Escolher estruturas de dados corretas para o problema pode melhorar ou piorar significativamente a CC de um algoritmo. Adaptar a implementação ao *hardware* específico em que será executado também pode resultar em acelerações.

CP, definida como a execução de vários cálculos simultaneamente, é uma aplicação desta última ideia. Foi originalmente introduzida para superar as limitações de *hardware* em termos de poder computacional; grandes problemas foram divididos em pequenos, que foram resolvidos simultaneamente em multiprocessadores de supercomputadores. As arquiteturas de *hardware* foram então desenvolvidas para aproveitar esse *software*, maximizando o impacto do desempenho do *hardware* para programas implementados adequadamente. Uma classificação dessas arquiteturas de *hardware* [24], de acordo com a natureza dos dados e das operações que eles suportam é:

- *Single-Instruction, Single-Data (SISD)*: uma única unidade de processamento que executa

uma única operação nos mesmos dados.

- *Multiple-Instruction, Single-Data (MISD)*: várias unidades de processamento que executam operações diferentes (de forma independente e assíncrona) nos mesmos dados.
- *Single-Instruction, Multiple-Data (SIMD)*: várias unidades de processamento que executam a mesma operação em vários dados.
- *Multiple-Instruction,Multiple-Data (MIMD)*: várias unidades de processamento que executam operações diferentes em vários dados.

Quase todos os computadores modernos são baseados no modelo MIMD. Todos os processadores modernos têm mais de um *core* e cada *core* suporta várias *threads* de execução simultâneas. Além disso, nos últimos anos, a velocidade dos processadores atingiu um pico, porque algumas restrições físicas mencionadas acima estão a impedir o aumento da escala de frequência. Ao mesmo tempo, o número de *cores* presentes em cada processador ainda está a aumentar e o mesmo para o número de *threads* suportadas por cada *core* [3].

Essa evolução despertou um novo interesse na CP. Os esforços recentes de pesquisa concentraram-se em duas áreas principais: a paralelização de algoritmos existentes e o desenvolvimento de novos algoritmos e bibliotecas de *software* projetadas explicitamente para tirar proveito da CP. No entanto, é importante observar que o grau em que um algoritmo pode alavancar o processamento paralelo depende da natureza do problema que está a tentar resolver. Alguns problemas são embaraçosamente paralelos, isto é, podem ser divididos de tal maneira que cada parte raramente ou nunca precise de comunicar com as outras partes. Outros problemas não podem ser totalmente paralelizados, porque as suas partes precisam de comunicar periodicamente entre si para sincronizar o seu estado. Se são necessárias sincronizações frequentes, falamos de paralelismo de granulação fina e de paralelismo de granulação grossa se as sincronizações forem necessárias apenas algumas vezes durante um longo período de tempo. Finalmente, alguns problemas são inerentemente sequenciais e não podem ser paralelizados.

2.4.2 Computação paralela em Julia

A maioria dos computadores modernos possui mais de uma CPU. Aproveitar o poder de várias CPUs permite que muitos cálculos sejam concluídos mais rapidamente. Existem dois fatores principais que influenciam o desempenho: a velocidade das próprias CPUs e a velocidade do acesso à memória. Um bom ambiente de multiprocessamento deve permitir o controlo sobre a "propriedade" de um pedaço de memória de uma CPU específica.

Julia oferece diferentes níveis de paralelismo, podemos dividi-los em três categorias principais:

1. Julia *Coroutines* (*Green Threading*)
2. *Multi-threading*

3. Processamento *multi-core* ou distribuído

Primeiro, consideraremos Julia *Tasks* (também conhecida como *Coroutines*). Julia também suporta *multi-threading* experimental, que geralmente assume a forma de *loops* paralelos. No final, apresentaremos a abordagem de Julia para a computação *multi-core* ou distribuída. Julia implementa interfaces de forma nativa para distribuir um processo por vários *cores* ou máquinas. Também vamos mencionar bibliotecas externas úteis para programação distribuída.

Coroutines

A plataforma de programação paralela de Julia usa *Tasks* para alternar entre vários cálculos. Para expressar uma ordem de execução entre *threads*, são necessárias primitivas de comunicação. Julia oferece Channel(func :: Function, ctype = Any, csize = 0, taskref = nothing) que cria uma nova tarefa func, vincula-a a um novo canal de tipo ctype e tamanho csize e agenda a tarefa. *Channels* podem servir como uma maneira de comunicar entre tarefas, pois ChannelT(sz :: Int) cria um canal em *buffer* de tipo T e tamanho sz. Sempre que o código executa uma operação de comunicação como *fetch* ou *wait*, a tarefa atual é suspensa e um agendador escolhe outra tarefa para executar. Uma tarefa é reiniciada quando o evento que está à espera é concluído.

Para muitos problemas, não é necessário pensar nas tarefas diretamente. No entanto, podem ser usados para esperar vários eventos ao mesmo tempo, o que fornece *dynamic scheduling*. No *dynamic scheduling*, um programa decide o que computar ou onde computar com base em quando outros trabalhos são concluídos. Isso é necessário para cargas de trabalho imprevisíveis ou desequilibradas, nas quais queremos atribuir mais trabalho aos processos somente quando eles terminam as suas tarefas atuais.

Multi-threading

Além das *Tasks*, Julia suporta nativamente *multi-threading* usando o módulo *Base.Threads* ainda que experiental e no qual as interfaces podem mudar no futuro.

Por padrão, Julia inicia com uma única *thread* de execução. Isso pode ser verificado usando o comando Threads.nthreads(). O número de *threads* que Julia inicia é controlado por uma variável de ambiente chamada JULIA_NUM_THREADS.

Em *multi-threading* a execução é bifurcada e é executada uma função em todas as *threads*. Conhecida como a abordagem *fork-join*, onde encadeamentos paralelos são executados independentemente e, por último, devem ser unidos no encadeamento principal de Julia para permitir a execução em série.

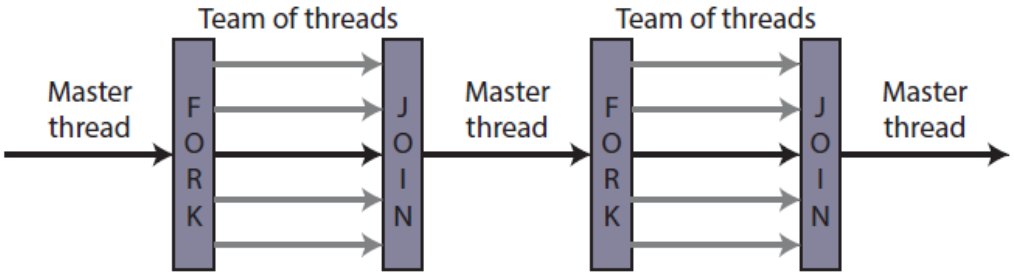


Figura 2.3: Exemplo de um fluxo de execução utilizando a abordagem *fork-join*.

Na abordagem *fork-join* todos os programas iniciam a execução com um processo, a *master thread*. A *master thread* executa sequencialmente até encontrar um construtor paralelo, altura em que cria um *team of threads*. O código delimitado pelo construtor paralelo é executado em paralelo pela *master thread* e pelo *team of threads*. Ao completar a execução paralela, o *team of threads* sincroniza numa barreira implícita com a *master thread*. Quando o *team of threads* termina a execução, a *master thread* continua a execução sequencial até encontrar um novo construtor paralelo tal como na Figura 2.3.

Julia suporta *loops* paralelos usando a macro Threads.@threads. Essa macro é afixada na frente de um *loop for* para indicar a Julia que o *loop* é uma região *multithread*. O espaço de iteração é dividido entre as *threads*.

Observe que, embora o código Julia seja executado em uma única *thread* (por padrão), as bibliotecas usadas por Julia podem iniciar os suas próprias *threads* internas.

Processamento *multi-core* ou distribuído

Uma implementação da computação paralela de memória distribuída é fornecida pelo módulo *Distributed* como parte da biblioteca padrão fornecida com Julia.

Julia fornece um ambiente de multiprocessamento baseado na passagem de mensagens para permitir que os programas sejam executados em vários processos em domínios de memória separados ao mesmo tempo.

A implementação de Julia da passagem de mensagens é diferente de outros ambientes, como o MPI. A comunicação em Julia é geralmente "unilateral", o que significa que o programador precisa gerir explicitamente apenas um processo numa operação de dois processos. Além disso, essas operações normalmente não se parecem com "envio de mensagens" e "recebimento de mensagens", mas sim com operações de nível superior, como chamadas para funções do utilizador.

A programação distribuída em Julia baseia-se em duas primitivas: referências remotas e chamadas remotas . Uma referência remota é um objeto que pode ser usado em qualquer processo para se referir a um objeto armazenado num processo específico. Uma chamada remota é uma solicitação de um processo para chamar uma determinada função em determinados argumentos noutro processo (possivelmente o mesmo).

As referências remotas têm dois tipos: *Future* e *RemoteChannel*.

Uma *RemoteChannel* retorna a *Future* ao resultado. Podemos esperar o término de uma chamada remota chamando *wait* o retorno de *Future* e pode obter o valor total do resultado usando *fetch*.

Cada processo possui um identificador associado. O processo que fornece o *prompt* interativo de Julia sempre tem id igual a 1. Os processos usados por padrão para operações paralelas são chamados de "trabalhadore". Quando há apenas um processo, o processo 1 é considerado um trabalhador. Caso contrário, os trabalhadores são considerados todos os processos, exceto o processo 1.

Começando por julia − p n fornece n processos de trabalho na máquina local. Geralmente, faz sentido igualar n ao número de *threads* da CPU (núcleos lógicos) na máquina. Observe que o argumento −p carrega implicitamente o módulo *Distributed*.

2.4.2.1 Outras bibliotecas para computação paralela

Fora do paralelismo de Julia, há muitas bibliotecas externas que devem ser mencionadas. Por exemplo, *MPI.jl* é um *wrapper* Julia para o protocolo *Message-Passing Interface* (MPI), ou *DistributedArrays.jl*.

2.4.3 Computação paralela em R

O interpretador R só pode executar um comando de cada vez. Para tirar proveito de vários processadores, o R deve ser compilado com uma implementação *multi-threaded*. O que levou ao desenvolvimento de várias bibliotecas que lidam com CP. O *bnlearn* foi desenvolvido para trabalhar com:

- A biblioteca *snow*, [25] que fornece suporte para CP simples usando o modelo *master-slave*. *snow* gera um número configurável de processos R em segundo plano (os processos escravos). O utilizador pode copiar os dados e enviar comandos a partir do terminal do R em que está a trabalhar (o processo mestre). A comunicação entre esses processos é gerida usando *sockets* TCP padrão ou os mecanismos fornecidos pelas bibliotecas *Rmpi* e *rpvm*.
- A biblioteca *Rmpi* [26] , é uma interface R para as bibliotecas C que implementam de fato *MPI*, um protocolo de comunicação independente da linguagem, projetado para programar computadores paralelos.
- A biblioteca *rpvm* [27] , é uma interface R para o *software Parallel Virtual Machine (PVM)*. O *PVM* é projetado para permitir que uma rede de máquinas Unix e Windows heterogéneas seja usada como um único processador paralelo distribuído.

- A biblioteca *rsprng*, [28] fornece geradores de números aleatórios independentes para os escravos gerados pela *snow*.

2.4.3.1 Aprendizagem

Sabemos pela literatura que o problema de aprender a estrutura de RBs é muito difícil de resolver. A sua CC, medida com o número necessário de TIC ou pontuações de rede, é superexponencial no número de vértices no pior caso e polinomial na maioria das situações do mundo real. Além disso, a CC dos TIC e das próprias pontuações da rede deve ser levada em consideração; na maioria dos casos, é linear no tamanho da amostra. Em R, calcular probabilidades condicionais requer uma passagem sobre os dados.

Na prática, a maioria dos algoritmos de aprendizagem de estrutura pode ser aplicada a conjuntos de dados com algumas centenas de variáveis, no máximo. Implementações paralelas de algoritmos de aprendizagem, fornecem um aumento significativo no desempenho, melhorando assim a capacidade de lidar com grandes redes. No entanto, é importante observar que o tempo de execução reduz no máximo linearmente com o número de processos escravos. Por esse motivo, a paralelização não pode ser considerada uma solução universal, embora possa ser útil em muitas situações.

Algoritmos de aprendizagem de estrutura baseados em restrições

Os algoritmos baseados em restrições exibem um paralelismo de granulação grossa, porque só precisam sincronizar as suas partes algumas vezes. Se examinarmos novamente o algoritmo de CI 2, podemos ver que:

1. A primeira etapa é embaraçosamente paralela, porque cada conjunto de d-separação pode ser aprendido independentemente dos outros. Outra solução é dividir esta etapa numa parte para cada vértice, que aprenderá todos os conjuntos de d-separação que envolvem esse vértice específico. A primeira abordagem pode tirar vantagem de um número maior de processadores, enquanto a segunda possui menos sobrecarga devido ao menor número de partes em execução paralela.

2. O mesmo acontece para a segunda etapa. Uma vez que todos os conjuntos de d-separação são conhecidos, ela é embaraçosamente paralela e pode ser dividido da mesma maneira que na primeira.

3. A terceira etapa é sequencial, porque cada iteração requer o estado da anterior.

Portanto, as informações disponíveis para os processos escravos devem ser sincronizadas e apenas coletadas entre a primeira e a segunda etapa e entre a segunda e a terceira etapa.

A maioria dos algoritmos modernos baseados em restrições, que aprendem as coberturas de Markov dos vértices como uma etapa intermediária, exigem uma sincronização adicional.

Algoritmos de aprendizagem de estrutura baseados em pontuações

Os algoritmos de aprendizagem baseados em pontuações beneficiam de várias décadas de esforços de pesquisa com o objetivo de tirar proveito da CP nas heurísticas de otimização.

A maioria dos algoritmos baseados em pontuações é inerentemente sequencial por natureza. Considerando o algoritmo HC 3. A cada iteração, o estado da iteração anterior é usado como ponto de partida para a pesquisa de uma nova e melhor estrutura de rede. O que torna a sua implementação paralela um problema desafiador.

Uma solução possível é fornecer uma implementação paralela dos cálculos executados numa única iteração e permitir que o processo mestre execute as iterações de maneira sequencial, sincronizando o estado dos escravos de cada vez. Isso reduziria um problema sequencial a um problema paralelo de granularidade fina; conhecido como *move acceleration model* se cada escravo computar parte da pontuação de cada rede candidata ou *parallel moves model* se cada escravo gerir algumas das redes candidatas. No entanto, é provável que o ganho de desempenho resultante seja compensado pela sobrecarga das comunicações entre os processos mestre e escravo.

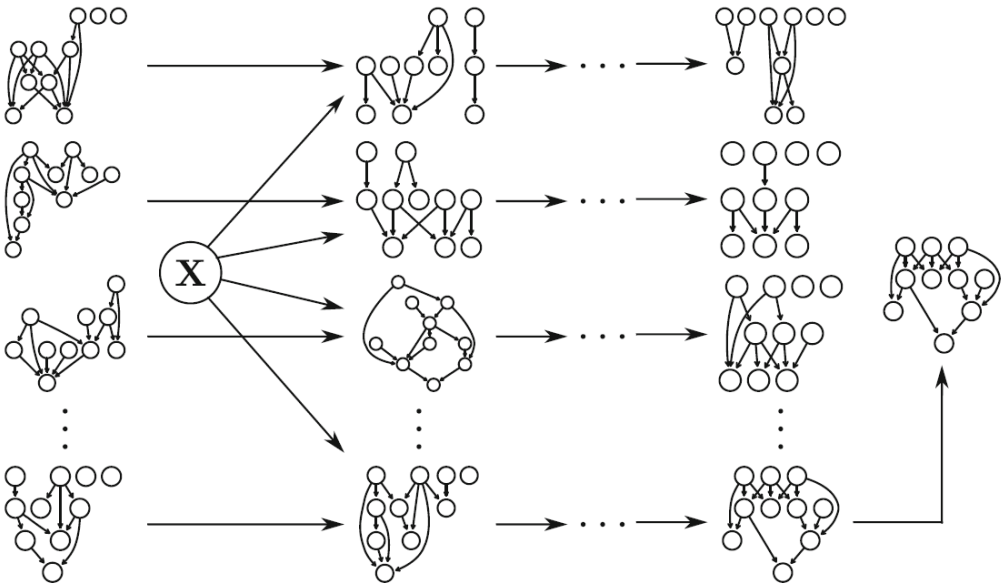


Figura 2.4: Etapas de uma implementação paralela de um algoritmo de aprendizagem baseado em pontuações.(fonte:[3])

Outra solução, chamada de *parallel multistart model*, é ilustrada na Figura 2.4 consiste em inicializar várias instâncias de um algoritmo baseado em pontuação com diferentes redes iniciais. O uso de pontos de partida diferentes para a pesquisa melhora a capacidade do algoritmo de cobrir o espaço de pesquisa e resulta em soluções melhores e mais robustas. Por exemplo, mesmo que uma das instâncias fique presa num máximo local, outra poderá encontrar o máximo global. Nesse caso, uma solução abaixo do ideal é simplesmente descartada.

É importante observar que o tempo de execução para cada modelo não é reduzido pelo *parallel*

multistart, porque cada uma das instâncias executadas pelos processos escravos leva em média tanto tempo quanto o algoritmo original baseado em pontuação.

Aprendizagem de parâmetros

A aprendizagem de parâmetros é outro problema embaraçosamente paralelo. Uma vez conhecida a estrutura da rede, a decomposição da distribuição global nas distribuições locais fornece uma maneira natural de dividir a estimativa dos parâmetros entre os escravos. A distribuição de cada vértice depende apenas dos valores dos seus pais e possui um número limitado de parâmetros; portanto, a quantidade de dados copiados para e dos processos escravos é muito pequena. Além disso, atribuir uma variável de cada vez a um processo escravo permite um uso eficiente de um grande número de processadores.

É importante observar que a estimativa de parâmetros é eficiente em termos de CC em comparação com a maioria dos outros problemas relacionados a RBs, tanto na aprendizagem de estruturas quanto na inferência. É provável que a redução no tempo de execução resultante de uma implementação paralela seja insignificante em toda a análise.

Capítulo 3

Algoritmos e Implementações de Redes Bayesianas

Neste capítulo pretendemos discutir e analisar trabalhos relacionados aos temas abordados nesta dissertação, que correspondem a implementações de algoritmos para aprender Redes Bayesianas (RBs) (tanto para estruturas como para parâmetros) e qual o seu desempenho. Vamos abordar também trabalhos que envolvam formas possíveis de otimizar esse mesmo desempenho.

Como já referido anteriormente pretendemos obter uma comparação de desempenho dos algoritmos de RBs entre a linguagem Julia e a linguagem R. Assim sendo, e como iremos utilizar a biblioteca *bnlearn* do R é pertinente falarmos nos dois seguintes artigos *Learning Bayesian Networks with the bnlearn R Package* [29] e *Bayesian Network Constraint-Based Structure Learning Algorithms: Parallel and Optimized Implementations in the bnlearn R Package* [30] do M. Scutari, que abordam o uso de algoritmos de aprendizagem de RBs na linguagem R.

O primeiro, descreve os algoritmos implementados na biblioteca *bnlearn*, para aprendizagem de estrutura de uma Rede Bayesiana (RB) com variáveis discretas e contínuas. Aborda também de que modo os algoritmos baseados em restrições e pontuações implementados podem melhorar o seu desempenho utilizando funcionalidades fornecidas pela biblioteca *snow* (também do R), por computação paralela.

O segundo, apresenta formas otimizadas e paralelas de aprender a estrutura de uma RB para algoritmos baseados em restrições. Pois os algoritmos baseados em pontuação já beneficiam das pesquisas de teoria da otimização, utilizando a maximização da função de pontuação, o que não acontece na abordagem baseada em restrições. Assim, este artigo mostra como é implementado *backtracking* e como ele degrada a estabilidade da aprendizagem da estrutura de uma RB com pouco ganho em velocidade. Alternativamente descreve uma arquitetura e uma estrutura de software que podem ser utilizadas para paralelizar os algoritmos de aprendizagem de estrutura baseados em restrições (implementados no *bnlearn*).

Apesar de serem várias as novas bibliotecas de *software* que foram introduzidas para o estudo das RBs, até agora existem poucas publicações sobre comparações entre elas. Não existindo um consenso sobre quais delas são melhores.

No artigo *"Comparison of software packages for Bayesian network learning in gene regulatory relationship mining"* [31] é feita uma comparação entre nove tipos de bibliotecas que são amplamente utilizadas para a aprendizagem de RBs. Sendo elas, o *bnlearn*, *deal* e *BNArray* da linguagem R, *CGBayesNets*, *BN Toolkit* do MATLAB, *SeqSpider* do Perl, *BNFinder2* do Python, *CyNetworkBMA* existente no Java e R e *Bayesian Network Webserver* do PHP e HTML5. São utilizadas para testes e experiências as bibliotecas *bnlearn*, *deal*, *CGBayesNets*. São usados os métodos gs, hc, iamb, mmpc, rsmax, tabu, fastiamb, interiamb, and mmhc da biblioteca *bnlearn* do R e os três métodos PC, K2 e GS da biblioteca *CGBayesNet* do MATLAB. O conjunto de dados utilizado foi o *benchamark* ALARM, que em que cada ficheiro zip inclui 10 conjuntos de dados que são usados para aprendizagem em diferentes tamanhos de amostras (500,1000,5000). Os resultados mostram que a *accuracy* de aprendizagem de diferentes métodos é diferente para o mesmo conjunto de dados, sendo a *accuracy* dos métodos da biblioteca *CGBayesNet* menor do que os do *bnlearn* e do *deal*. Para a eficiência, o *deal* é o que apresenta maior tempo de aprendizagem. Os melhores tempos de aprendizagem são os do *bnlearn*, usualmente perto dos 0.001 segundos. Para os três métodos do *CGBayesNet*, o método GS é o que usa maior tempo, enquanto que o método K2 usa o menor. Logo, a eficiência de aprendizagem é aproximadamente a mesma para amostras de tamanho diferente. Neste artigo foram mostradas várias bibliotecas de *software* relevantes, assim como duas métricas de avaliação do seu desempenho (*accuracy* e eficiência). Mostrando que o *bnlearn*, tem uma boa eficiência.

Visto o foco deste trabalho ser a comparação em RBs de larga escala, são bastante interessantes para o estudo publicações que mostrem o que tem sido efetuado em termos de otimizações e paralelismo em algoritmos de RBs.

O artigo *A parallel algorithm for Bayesian network structure learning from large data sets* [32] considera um algoritmo paralelo para aprendizagem de estrutura de uma RB a partir de grandes conjuntos de dados. O algoritmo PC é um algoritmo baseado em restrições que consiste em cinco etapas onde a primeira etapa realiza um conjunto de Testes de Independência Condicional (TIC) e as quatro restantes são relativas à identificação da estrutura da RB usando os resultados dos TIC. Descreve uma nova abordagem para a paralelização dos TIC, que são a etapa que consome mais tempo. O algoritmo PC paralelo proposto é avaliado em conjuntos de dados gerados aleatoriamente a partir de cinco RBs diferentes. Compara também, o algoritmo proposto, numa abordagem baseada em processos, onde cada processo gere um subconjunto de dados sobre todas as variáveis na RB. Mostrando resultados com melhorias significativas no desempenho de tempo usando ambas as abordagens.

No artigo *Map-Reduce for Machine Learning on Multicore* [33] desenvolvem um método de programação paralela, que é facilmente aplicado a muitos algoritmos de aprendizagem diferentes. Este trabalho faz uma abordagem contrastante com a maior parte dos restantes, que tentam projetar maneiras de acelerar apenas um algoritmo. Neste é mostrado que alguns algoritmos,

podem ser escritos numa "forma de soma", que permite que sejam mais facilmente paralelizados em computadores multicore. Adapta o paradigma *map-reduce* do Google para mostrar esta técnica de aceleração paralela num conjunto de algoritmos de aprendizagem sendo eles: regressão linear ponderada localmente, *k-means*, Regressão Logística (RL), *Naive Bayes* (NB), *Support Vector Machine* (SVM), ICA, PCA, *Gaussian Discriminant Analisys* (GDA), Maximização de Expectativas (ME) e retropropagação Os resultados experimentais mostram uma aceleração linear com um número crescente de processadores.

Em *Accelerating Bayesian Network Parameter Learning Using Hadoop and MapReduce* [34] formulam a aprendizagem de paramêtros (para dados completos) e o clássico algoritmo de ME (para dados incompletos) na estrutura do *map-reduce*. Analisa analiticamente e experimentalmente a velocidade que pode ser obtida pelo uso de *map-reduce*. Apresenta uma implementação *Hadoop* e relata as acelerações em contraste com o caso sequencial, e compara várias configurações *Hadoop* para RBs de diferentes tamanhos e estruturas. Conclui que o map-reduce pode aumentar a velocidade em comparação com o algoritmo sequencial de ME para aprender com 20 ou menos casos. Investigam o beneficio do map-reduce para a aprendizagem de RBs com conjuntos de dados com até 1.000.000 de observações.

As referências anteriores são importantes para o projeto pois consideram diferentes visões e abordagens possíveis para a aprendizagem de RBs. Nos primeiros artigos mencionados foi possível obter uma ampla visão de todos os métodos implementados pela linguagem R na biblioteca *bnlearn* o que é bastante útil para uma possível utilização destes ao longo do projeto. No terceiro a comparação entre várias bibliotecas de diferentes linguagens deu uma boa indicação que biblioteca *bnlearn* da linguagem R seria uma das mais eficientes para aprendizagem de RBs em grande escala. Todas as restantes referências são também importantes para o projeto, porque abordam diferentes maneiras de otimizar os tempos de aprendizagem dos algoritmos de RB, sendo pela otimização de etapas mais demoradas para um único algoritmo e também pela sua abordagem baseada em processos ou utilizando métodos de programação paralela que podem abranger vários algoritmos com apenas uma abordagem, assim como exploram técnicas e implementações de *map-reduce* e *Hadoop* para o aumento da velocidade de aprendizagem das RBs.

Capítulo 4

Desenho e Desenvolvimento

Neste capítulo começamos por mostrar a configuração do ambiente utilizado no desenvolvimento experimental, devido à influência que o tipo de *hardware* e *software* assumem na execução de experiências que envolvem tempo de execução de algoritmos. Os conjuntos de dados também afetam o comportamento dos algoritmos, características como o tamanho e a relação entre o número de colunas e linhas são fatores importantes, por este motivo seguidamente descrevemos cada um dos conjuntos de dados e a preparação necessária para a sua utilização. Por último são descritos os parâmetros utilizados nas experiências.

4.1 Configuração do ambiente

Para a realização de experiências foram utilizadas duas máquinas, pois como já foi dito o *hardware* é um fator que influencia os resultados obtidos, sendo assim, são descritas na Tabela 4.1 as suas características e daqui em diante vamos referir-nos a cada uma das máquinas apenas pelo nome registado na mesma tabela (Maq1 e Sibila).

Tabela 4.1: Características do *hardware* utilizado.

	Maq1	Sibila
Memória RAM	15GiB[1]	251GiB
Processador	Intel(R) Core(TM) i7-4510U CPU 2.00GHz	AMD Opteron(tm) Processor 6376
CPU(s)	4	64
Thread(s) per core	2	2
Core(s) per socket	2	8
Socket(s)	1	4
CPU MHz	2793.483	1400.000
CPU max MHz	3100	2300
CPU min MHz	800	1400

[1] 1 GiB = 1.073741824 GB

Durante todo o desenvolvimento do projeto, acedemos à máquina Sibila remotamente, estabelecendo uma ligação SSH.

Ao longo do desenvolvimento desta dissertação foram utilizadas duas linguagens de programação, R e Julia. A razão pela qual foi escolhido o R para comparação com Julia, foi por já fornecer implementações para aprendizagem em Redes Bayesianas (RBs) e assim existirem os recursos necessários à possível comparação final. Como *IDEs* foram utilizados o *RStudio*, que é o mais indicado para o desenvolvimento de projetos em R e o *ATOM* para desenvolvimento Julia. As especificações do *software* utilizado nas experiências é mostrado a seguir na Tabela 4.2.

Tabela 4.2: Características do *software* utilizado.

	Maq1	Sibila
Sistema Operativo	Ubuntu 18.04.3 LTS (Bionic Beaver)	Fedora 20 (Heisenbug)
IDEs	RStudio Version 1.1.46 ATOM 1.39.0x64	-
Linguagens de Programação	R version 3.5.1 (2018-07-02) Julia Version 1.1.1 (2019-05-16)	R version 3.5.1 (2018-07-02) Julia Version 1.1.1 (2019-05-16)
Bibliotecas R	*bnlearn* *Rgraphviz* *snow*	
Bibliotecas Julia	*BayesNets* *CSV* *DataFrames* *Discretizers* *TikzPictures* *TikzGraphs* *BenchmarkTools* *Distributed*	

4.2 Descrição dos dados

Na literatura são utilizadas diversas RBs de referência como *benchmarks* e estão disponíveis em diferentes formatos. Neste trabalho foram utilizados vários conjuntos de dados, com diferentes tamanhos e complexidades, obtidos do *Bayesian Network Repository* e da biblioteca *bnlearn* que inclui alguns conjuntos de dados.

Dividimos as RBs em cinco categorias, pelo seu número de vértices (que correspondem ao número de colunas dos conjuntos de dados), primeiro as redes pequenas com menos de 20 vértices, redes médias com entre 20 e 50 vértices, redes grandes entre 50 a 100 vértices, redes muito grandes de 100 a 1000 vértices e por fim redes massivas com mais de 1000 vértices e escolhemos um conjunto de dados de cada categoria apresentados na Tabela 4.3 e descritos em seguida.

Tabela 4.3: Características dos conjuntos de dados utilizados nas experiências.

Categoria	Conjunto de Dados	Colunas	Observações
Pequena (< 20)	ASIA	8	5000
Média (20 - 50)	ALARM	37	20000
Grande (50 - 100)	HAILFINDER	56	20000
Muito Grande (100 - 1000)	ANDES	223	1000, 20000
Massiva (> 1000)	MUNIN	1041	1000, 20000

- ASIA, pequeno conjunto de dados sintéticos [35] sobre doenças pulmonares (tuberculose, cancro de pulmão ou bronquite) e visitas à Ásia.
- ALARM, [36] uma Rede Bayesiana (RB) projetada por médicos especialistas para fornecer um sistema de mensagens de alarme para pacientes em unidades de terapia intensiva com base na saída de um número de sinais vitais em dispositivos de monitoramento.
- HAILFINDER é um conjunto de dados gerado a partir da rede de referência com o mesmo nome. É uma RB projetada para prever granizo severo no verão no nordeste do Colorado.
- ANDES, [37] um sistema de tutoria inteligente baseado num modelo de estudante para o campo da física Newtoniana clássica, desenvolvido na Universidade de Pittsburg e na Academia Naval dos Estados Unidos. Lida com a avaliação do conhecimento a longo prazo, o reconhecimento de planos e a previsão de ações dos alunos durante os exercícios de solução de problemas.
- MUNIN, [38] uma RB projetada por especialistas para interpretar resultados de exames eletromiográficos e a diagnosticar um grande conjunto de doenças comuns a partir de condições físicas, como a atrofia e fibras nervosas ativas.

4.3 Preparação dos dados

Por facilidade os conjuntos de dados foram todos carregados apartir do *RStudio*, já que a biblioteca *bnlearn* contém instâncias dos conjuntos de dados ASIA, ALARM e HAILFINDER só foi necessária a sua leitura direta, aplicando as funções dados, () e str(). Para os restantes conjuntos de dados, (ANDES e MUNIN) foram descarregados e utilizados os ficheiros .RDA[2] presentes no *Bayesian Network Repository* através da função load() e seguidamente foram geradas observações aleatórias, com a função rbn(), com 1000 e 20000 observações cada um. Todos os conjuntos de dados foram escritos em ficheiros .csv no *RStudio*, para a sua posterior utilização pela linguagem de programação Julia.

Procedeu-se à importação dos ficheiros .csv criados no *RStudio* para serem usados por Julia, através da fução CSV.read() da biblioteca *CSV* de Julia. Efetuou-se uma renomeação das

[2].RDA (ou .rda) é a abreviação de .RData (ou .rdata)

colunas dos conjuntos de dados (correspondentes aos vértices nas RBs), porque as funções para aprendizagem de estrutura de RBs da biblioteca *BayesNets* de Julia, não aceitavam alguns caracteres utilizados nos nomes das colunas iniciais.

Como duas das funções do *BayesNets*, utilizadas para a aprendizagem de estrutura das RBs apenas estão implementadas para dados discretos, foi necessário realizar-se uma discretização dos dados importados, utilizando a função encode() da biblioteca *Discretizers*, realizou-se então uma discretização categórica ao longo das colunas dos conjuntos de dados.

Para garantir que os dados avaliados eram iguais para ambas as linguagens de programação e a sua comparação o mais justa possível, foram escritos em ficheiros .csv os dados já discretizados com a função CSV.write(), de maneira a poderem ser importados pelo R para a aprendizagem das RBs.

4.4 Desenvolvimento experimental

Depois de obtidos os conjuntos de dados e preparados para a sua utilização ser possível em ambas as linguagens de programação, foram submetidos aos algoritmos de aprendizagem presentes na biblioteca *BayesNets* de Julia e os correspondentes na biblioteca *bnlearn* da linguagem R.

4.4.1 Algoritmos de aprendizagem utilizados em Julia

Em Julia, como já referido, estão implementados três algoritmos para a aprendizagem de estrutura de RBs definidos como *GraphSearchStrategy*.

Uma das implementações segue o algoritmo K2, chamado em Julia pela *GraphSearchStrategy* K2GraphSearch que tem como argumentos de entrada:

- uma ordenação de variáveis;
- os tipos de Distribuição de Probabilidade Condicional (DPC), na mesma ordem dada anteriormente;
- o número máximo de pais por DPC;
- e uma métrica (a qual tenta maximizar).

No desenvolvimento experimental, e como não existe um conhecimento específico sobre todas as variáves de todos os conjuntos de dados, não efetuámos uma relação causa efeito entre elas. Optámos por seguir a ordem em que as variáveis ocorriam no ficheiro .csv, para a ordenação de variáveis requerida. Para o tipo das Distribuições de Probabilidade Condicionais (DPCs) utilizamos *DiscreteCPD = CategoricalCPD {Categorical}*. Apenas realizámos testes para números máximos de pais iguais a 2 e 3, devido ao tempo acrescido de execução dos algoritmos com maior

número máximo de pais. A única métrica implementada para este algoritmo é uma função de pontuação para o *Bayesian Information Criterion* (BIC) negativo, definido como:

$$BIC = -2 * L + k * \log n$$

onde

$L = \mathsf{logpdf}(\mathsf{cpd}, \mathsf{dados})$, é a probabilidade de log dos dados sobre a DPC

$k = \mathsf{nparams}(\mathsf{cpd})$, é o número de parâmetros livres a serem estimados

$n = \mathsf{nrow}(\mathsf{dados})$, é o tamanho da amostra

As duas restantes implementações são chamadas pelas *GraphSearchStrategy* GreedyHillClimbing e ScanGreedyHillClimbing e só são implementadas para RBs discretas, que são redes nas quais todas as variáveis são inteiros e toda a distribuição é categórica.Ambas as implementações seguem, como o nome indica, uma estratégia *Hill Climbing* (HC), na qual procuram melhorar a sua pontuação através de adições, remoções ou inversões de uma aresta de cada vez.

A *GraphSearchStrategy* GreedyHillClimbing tem como argumentos de entrada:

- uma ScoreComponentCache() do conjunto de dados, a qual armazena pontuações numa fila de prioridades, de modo que os algoritmos de pesquisa de grafos saibam quando uma construção específica de rede já foi feita;
- o número máximo de pais;
- e um prior.

Na *GraphSearchStrategy* ScanGreedyHillClimbing os argumentos de entrada são novamente:

- uma ScoreComponentCache();
- o número máximo de pais;
- uma profundidade máxima :
- e um prior.

Como na *GraphSearchStrategy K2GraphSearch* no desenvolvimento experimentall, utilizámos em ambas as funções como valores para o número máximo de pais 2 e 3. No prior foram utilizados os valores *default*, que coresponde a *DirichletPrior = UniformPrior (1.0)*, assim como para a profundidade máxima na *GraphSearchStrategy* ScanGreedyHillClimbing foi usado o valor *default* que é igual a 1.

A aprendizagem de parâmetros de uma RB está implementada pela função fit() da biblioteca *Distributions*, a qual aprende cada DPC a partir dos dados. Recebe como argumentos de entrada o tipo de DPC, o conjunto de dados, e a *GraphSearchStrategy* que define a abordagem para a aprendizagem de estrutura.

4.4.2 Algoritmos de aprendizagem utilizados em R

Como em Julia só estão implementados algoritmos de aprendizagem de estrutura baseados em pontuações, e a única abordagem baseada em pontuações implementada em R pela biblioteca *bnlearn* é o algoritmo HC (descrito acima pelo algoritmo 3), para a comparação ser o mais justa possível utilizámos a função hc() da biblioteca *bnlearn* de R para a aprendizagem da estrutura.

Para comparação com a implementação utilizada pela *GraphSearchStrategy* K2GraphSearch de Julia utilizámos a função hc() de R com score = "bic" e números máximos de pais 2 e 3.

Para as implementações de HC utilizadas pelas *GraphSearchStrategy* GreedyHillClimbing e ScanGreedyHillClimbing escolhemos a mesma função R mas com uma implementação de pontuação diferente, score = "bds".

Para a aprendizagem de parâmetros a linguagem R disponibiliza a função bn.fit() que tem como argumentos de entrada: o modelo aprendido pelo algoritmo de aprendizagem de estrutura escohido, neste caso o hc() com as diferentes funções de pontuação e o conjunto de dados.

Para uma compreensão mais fácil são mostradas na Tabela 4.4 as comparações que foram efetuadas nas experiências.

Tabela 4.4: Funções em comparação entre as duas linguagens de programação no desenvolvimento experimental, onde dados representa um *dataframe* do conjunto de dados.

	Julia	R
Aprendizagem de Estrutura	K2Graphsearch(order = names(dados), cpdtype = DiscreteCPD, max_n_parents=2 metric = NegativeBayesianInformationCriterion())	hc(dados, score = "bic", maxp = 2)
	GreedyHillClimbing(ScoreComponentCache(dados), max_n_parents=2, prior=UniformPrior(1.0))	hc(dados, score = "bds", maxp = 2)
	ScanGreedyHillClimbing(ScoreComponentCache(dados), max_n_parents=2, max_deph=1, prior=UniformPrior(1.0))	hc(dados, score = "bds", maxp = 2)
Aprendizagem de Parâmetros	Distributions.fit(Type{CPD}, dados, GraphSearchStrategy)	bn.fit(modelo aprendido, dados)

4.4.3 Abordagem Paralela

Como já referido os algoritmos de aprendizagem de RBs têm complexidades que escalam exponencialmente com o número de váriáveis. Como a única maneira de melhorar a escalabilidade é desenvolver um algoritmo melhor ou adaptar a implementação ao *hardware* em que o algoritmo será executado, optámos pela segunda opção para a avaliação da escalabilidade dos algoritmos utilizados para comparação.

Como a linguagem de programação Julia tem três tipos de paralelismo, escolhemos uma abordagem *multi-threading* para avaliação da escalabilidade dos algoritmos implementados na biblioteca *BayesNets*.

Para a Maq1, como só tem 2 *cores* e cada um deles com 2 *threads*, limitámos a nossa avaliação a 1, 2 e 4 *threads*. A Sibila tem disponíveis 8 *cores* cada um com 2 *threads*, então foi possível avaliar o desempenho dos algoritmos com 1, 2, 4, e 8 *threads*, não foi efetuada uma avaliação com 16 *threads* pois como a máquina era partilhada com mais utilizadores o uso das 16 *threads* poderia gerar conflitos e resultados menos viáveis pela competitividade entre os processos.

Para isto, utilizámos a variável de ambiente JULIA_NUM_THREADS, para os respetivos números de *threads* que pretendíamos utilizar na experiência e adicionando a macro Threads.@threads, nos blocos de código dos diferentes algoritmos implementados possíveis de ser executados por várias *threads*, para indicar a Julia que eram regiões de *multi-threading*.

Para a linguagem R, como referido anteriormente o *bnlearn* foi desenvolvido para trabalhar com a biblioteca *snow*, que gera processos R em segundo plano. Utilizámos a função makecluster() definida com argumentos iguais a 2 e 4 na Maq1 e com 2, 4 e 8 na Sibila.

4.4.4 Métricas de desempenho

Para avaliarmos o desempenho quantitativo dos algoritmos implementados em comparação neste projeto medimos os seus tempos de execução e os recursos necessários para executarem.

Na linguagem de programação Julia, realizámos a avaliação do tempo de execução da aprendizagem de cada RB, através da macro @time da biblioteca *BenchamarkTools*.

Em R foi utilizada a função system.time() para medir o tempo de execução da aprendizagem de cada RB.

Para avaliar o desempenho qualitativo foi calculado o *log-likelihood* das redes aprendidas.

Em Julia foi calculado através da função logpdf() que retorna o logaritmo da densidade da probabilidade em x.

Na linguagem R foi utilizada a função score() que calcula a pontuação da RB, com argumento type = "loglik" a pontuação *multinomial log-likelihood*, que é equivalente à medida de entropia usada no Weka [13].

Capítulo 5

Resultados e Análise

Neste capítulo vamos apresentar e discutir os resultados obtidos pela execução do desenvolvimento experimental descrito anteriormente, em duas abordagens uma sequencial e outra paralela. Em cada um dos testes apresentados, o tempo de execução em segundos foi obtido da segunda execução de cada aprendizagem realizada e é apresentado nos gráficos em escala logarítmica. A opção ideal era um número maior de execuções e efetuar a média dos tempos de execução, mas por questões de limitação de tempo não foi possível, pois alguns algoritmos demoram mais de 10 dias a executar, que foi o tempo considerado limite.

5.1 Abordagem sequencial

Nesta secção analisamos os resultados obtidos através da execução sequencial de todos os algoritmos em estudo.

5.1.1 Diferente número de pais

Os tempos de execução (em segundos, em escala logarítmica) obtidos pela execução de todos os algoritmos implementados na biblioteca *BayesNets* de Julia com um conjunto de dados de cada categoria e para números máximos de pais com valores 2 e 3 estão mostrados nas Figuras 5.1, 5.2a e 5.2b. Nas Figuras 5.3a e 5.3b apresentamos os tempos obtidos pela execução sequencial dos algoritmos da biblioteca *bnlearn* de R considerados para comparação.

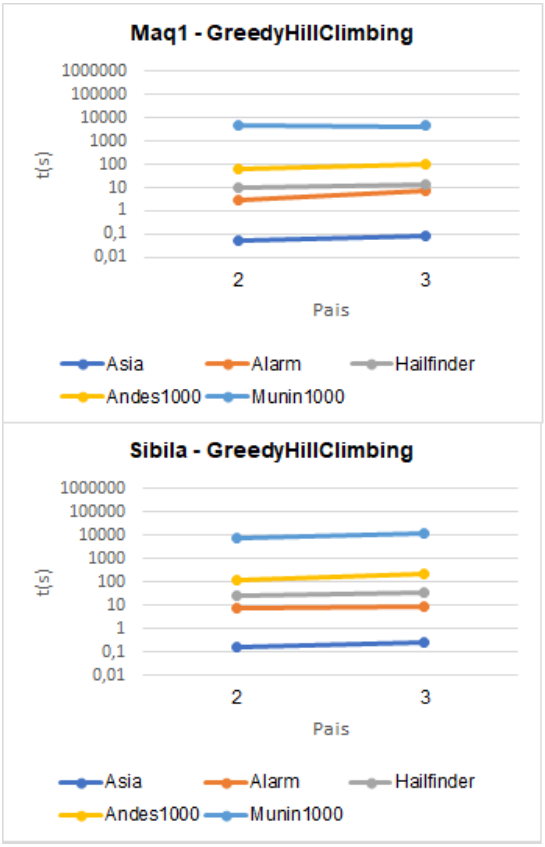
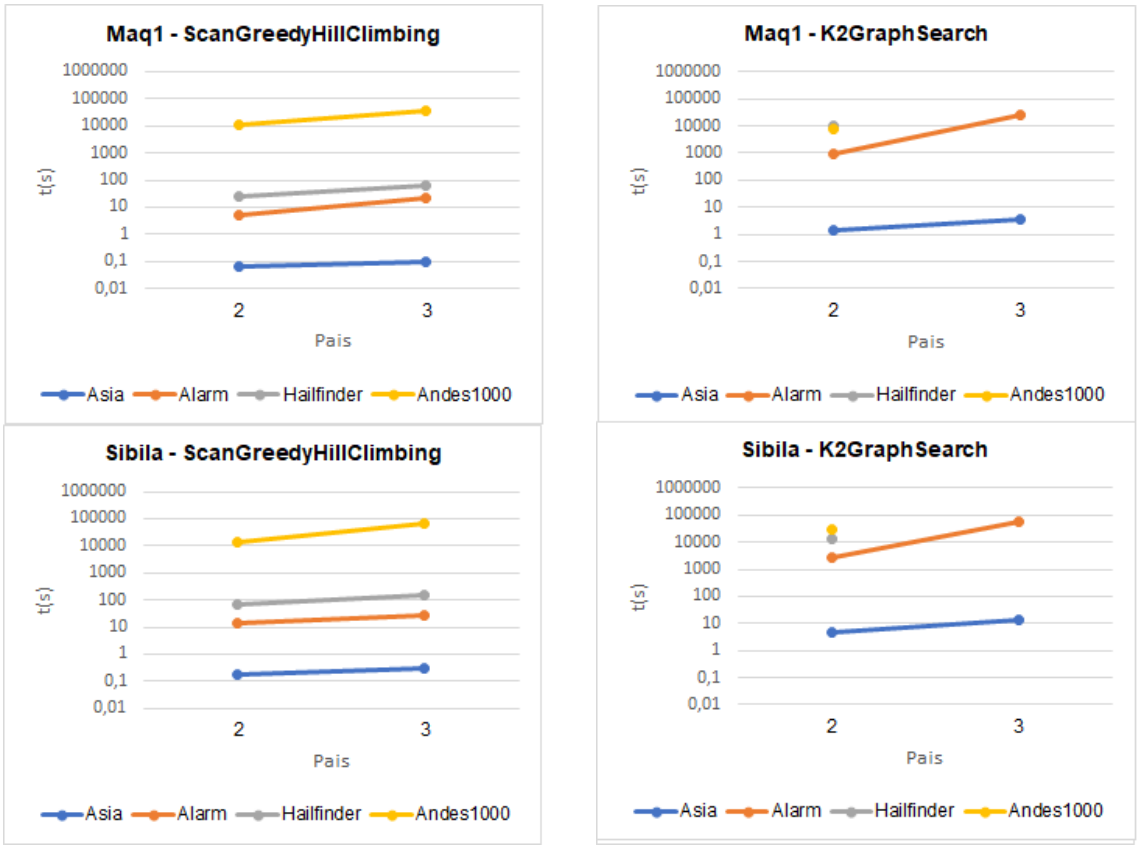


Figura 5.1: Tempos de execução com *GraphSearchStrategy* GreedyHillClimbing para os conjuntos de dados ASIA, ALARM, HAILFINDER, ANDES e MUNIN com 1000 observações, com número máximo de pais 2 e 3, nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

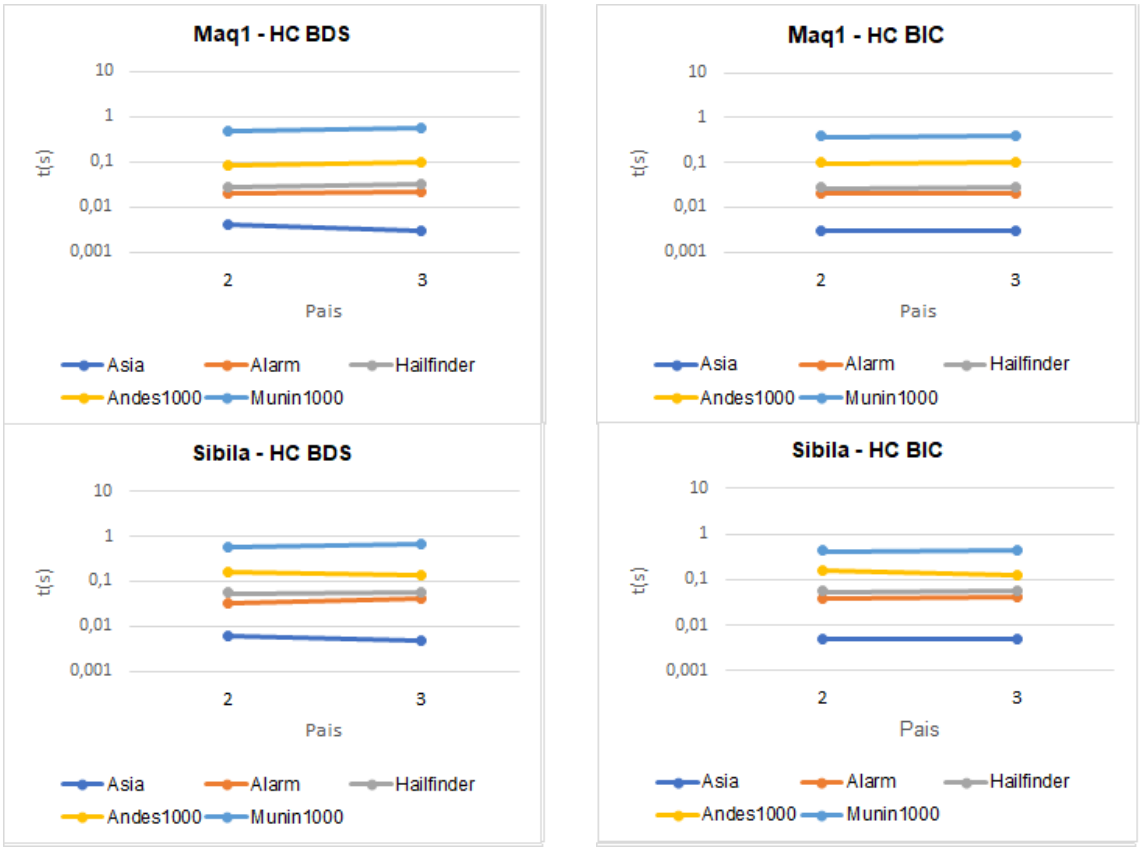
Não foi possível obter resultados para o conjunto de dados MUNIN com 1000 observações, para os algoritmos com *GraphSearchStrategy* ScanGreedyHillClimbing e K2GraphSearch de Julia porque o seu tempo de execução excedeu o tempo definido como limite de execução (10 dias). Assim nas Figuras 5.2a, 5.2b só estão representados os restantes conjuntos de dados.

Para o algoritmo com *GraphSearchStrategy* K2GraphSearch também não conseguimos obter resultados em tempo útil para os conjuntos de dados HAILFINDER e ANDES com 1000 observações com número máximo de pais igual a 3. Por esse motivo só estão representados os resultados para número máximo de pais igual a 2, nesses dois conjuntos de dados para a *GraphSearchStrategy* K2GraphSearch.



(a) ScanGreedyHillClinmbing (b) K2GraphSearch

Figura 5.2: Tempos de execução das respetivas *GraphSearchStrategy* para os conjuntos de dados ASIA, ALARM, HAILFINDER, ANDES com 1000 observações, com número máximo de pais 2 e 3, nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila)



(a) hc(score = "bds") (b) hc(score = "bic")

Figura 5.3: Tempos de execução das respetivas funções para os conjuntos de dados ASIA, ALARM, HAILFINDER, ANDES e MUNIN com 1000 observações, com número máximo de pais 2 e 3, nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

5.1.1.1 Mesmo número de observações

Visto o foco deste trabalho ser a comparação em Redes Bayesianas (RBs) de larga escala, para uma melhor compreensão da escalabilidade dos algoritmos de aprendizagem considerados, agrupamos os diferentes conjuntos de dados com 20000 observações (ALARM, HAILFINDER, ANDES e MUNIN) e em seguida apresentamos os tempos de execução (em segundos, em escala logarítmica) obtidos em todos os algoritmos de ambas as linguagens e para números máximos de pais com valores 2 e 3 nas Figuras 5.4, 5.5a, 5.5b, 5.6a e 5.6b.

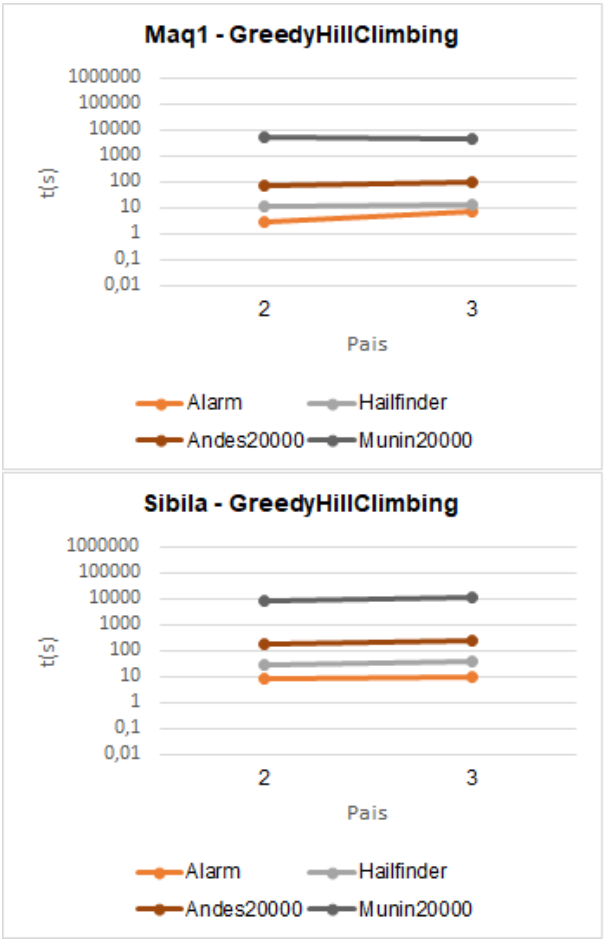
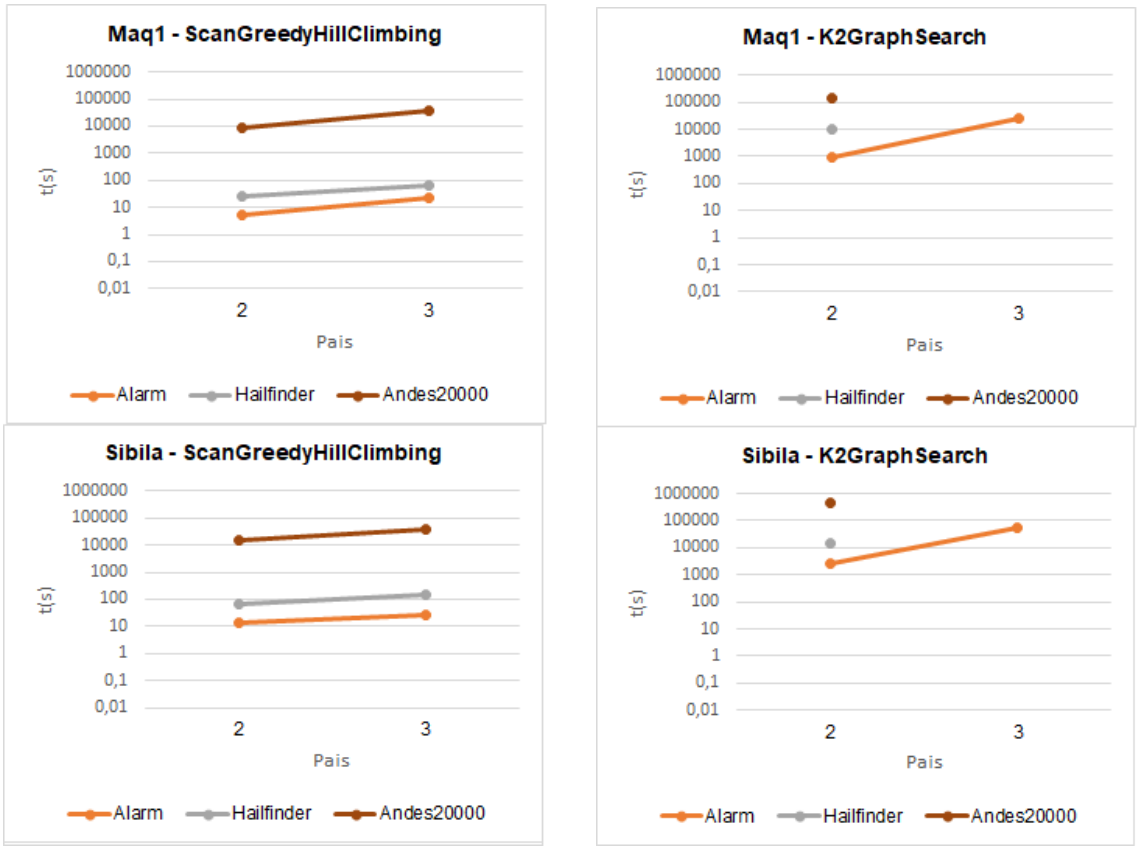


Figura 5.4: Tempos de execução com *GraphSearchStrategy* GreedyHillClimbing para os conjuntos de dados com 20000 observações, com número máximo de pais 2 e 3, nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

Igualmente como no conjunto de dados MUNIN com 1000 observações, também não foi possível obter resultados para o mesmo com 20000 observações, para os algoritmos com *GraphSearchStrategy* implementados com ScanGreedyHillClimbing e K2GraphSearch de Julia, porque o seu tempo de execução excedeu o tempo definido como limite de execução (10 dias). Assim nas Figuras 5.5a, 5.5b só estão representados os restantes conjuntos de dados.

Do mesmo modo para o algoritmo com *GraphSearchStrategy* K2GraphSearch também não conseguimos obter resultados em tempo útil para os conjuntos de dados HAILFINDER e ANDES com 20000 observações com número máximo de pais igual a 3. Por esse motivo só estão representados os resultados para número máximo de pais igual a 2, nesses dois conjuntos de dados para a *GraphSearchStrategy* K2GraphSearch.



(a) ScanGreedyHillClimbing (b) K2GraphSearch

Figura 5.5: Tempos de execução das respetivas *GraphSearchStrategy* para os conjuntos de dados com 20000 observações, com número máximo de pais 2 e 3, nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

(a) hc(score = "bds") (b) hc(score = "bic")

Figura 5.6: Tempos de execução das respetivas funções para os conjuntos de dados com 20000 observações, com número máximo de pais 2 e 3, nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

Como podemos verificar o aumento do número de vértices numa Rede Bayesiana (RB) implica um maior tempo de execução para todos os algoritmos em ambas as linguagens e nas duas máquinas utilizadas, como já era esperado.

Como visto para redes massivas (>1000 vértices) não foi possível obter resultados em tempo útil para os algoritmos implementados pelas *GraphSearchStrategy* ScanGreedyHillClimbing e K2GraphSearch. Também não foi possível obter resultados para os conjuntos de dados HAILFINDER e ANDES com 1000 e 20000 observações para a *GraphSearchStrategy* K2GraphSearch com número máximo de pais igual a 3 em ambas as máquinas utilizadas. Verificamos assim que o aumento do número máximo de pais implica maiores tempos de execução, sendo mais significativo nas *GraphSearchStrategy* ScanGreedyHillClimbing e K2GraphSearch de Julia e quase insignificante nos algoritmos implementados na linguagem R.

Podemos verificar também que nos conjuntos de dados com diferentes números de observações (ANDES e MUNIN) os tempos de execução não se alteram significativamente, o que nos permite afirmar que o aumento do número de observações não influência diretamente os tempos de

aprendizagem dos algoritmos em estudo.

5.1.2 Comparação de algoritmos

Em seguida são apresentados os tempos de execução de todos os algoritmos considerados no estudo, para os conjuntos de dados ASIA, ALARM, HAILFINDER, e ANDES e MUNIN com 1000 observações, nas Figuras 5.7 e 5.9, assim como para os conjuntos de dados com 20000 observações, nas Figuras 5.8 e 5.10, com número máximo de pais igual a 2 do lado esquerdo e com número máximo de pais igual 3 do lado direito para ambas as máquinas utilizadas (Maq1 em cima e Sibila em baixo).

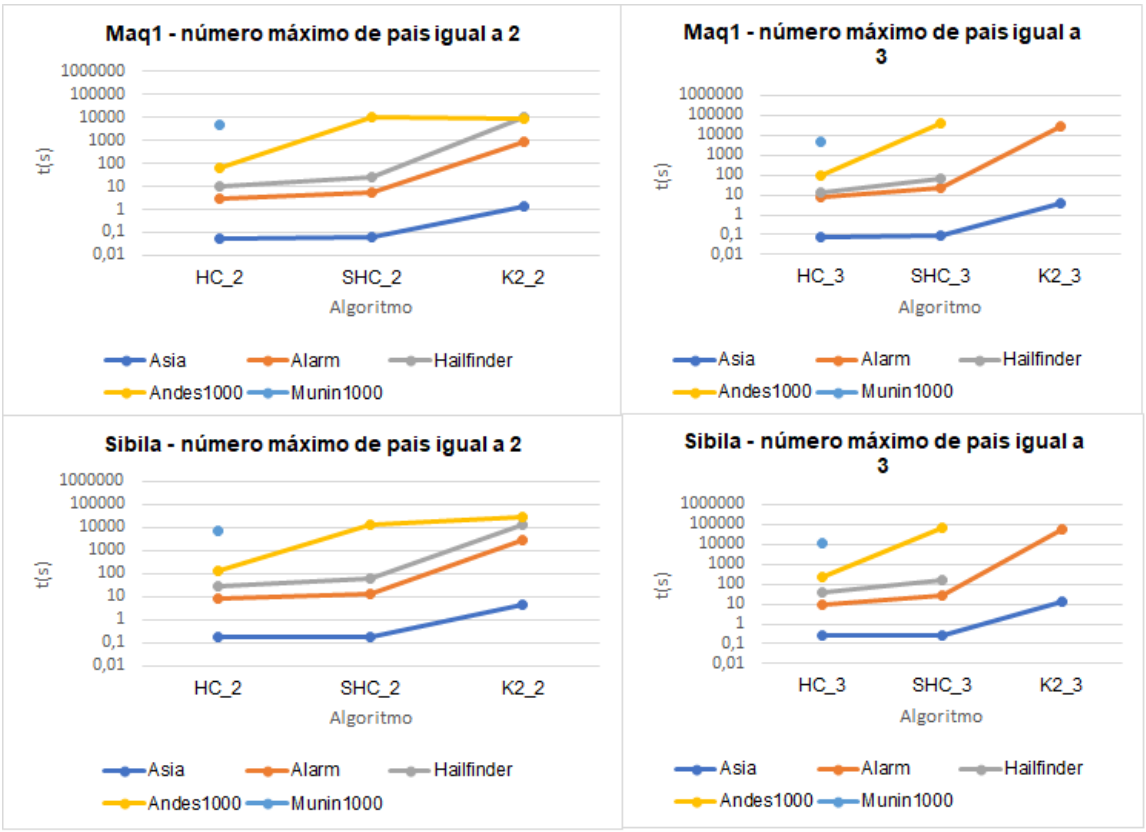


Figura 5.7: Tempos de execução das 3 *GraphSearchStrategy* de Julia, para os conjuntos de dados ASIA, ALARM, HAILFINDER, e ANDES e MUNIN com 1000 observações, com número máximo de pais igual a 2 (do lado esquerdo) e 3 (do lado direito), nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

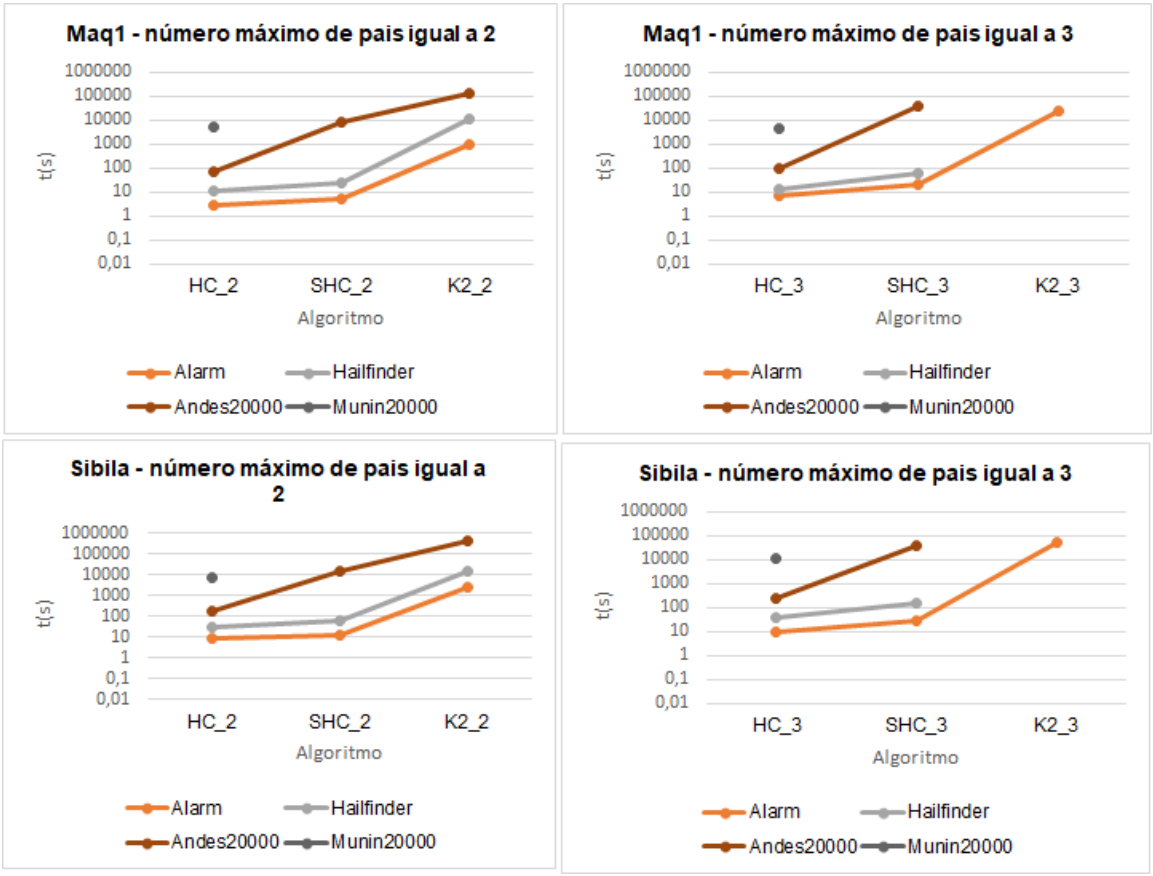


Figura 5.8: Tempos de execução das 3 *GraphSearchStrategy* de Julia, para os conjuntos de dados ASIA, ALARM, HAILFINDER, e ANDES e MUNIN com 20000 observações, com número máximo de pais igual a 2 (do lado esquerdo) e 3 (do lado direito), nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

Verificamos que o algoritmo de aprendizagem em Julia que leva menos tempo para concluir a sua execução é o da *GraphSearchStrategy* GreedyHillClimbing (HC_* nas Figuras) e o que demora mais tempo é o da *GraphSearchStrategy* K2GraphSearch (K2_* nas Figuras), como já era esperado. Pois pelo estudo das implementações de ambos os algoritmos é visível que a *GraphSearchStrategy* K2GraphSearch tem uma complexidade cúbica como pode ser visto no Apêndice A.3 enquanto que a *GraphSearchStrategy* GreedyHillClimbing aprende a RB iterativamente como pode ser visto no Apêndice A.1. No caso da *GraphSearchStrategy* ScanGreedyHillClimbing verificamos que a sua implementação é uma implementação recursiva do GreedyHillClimbing como podemos observar no Apêndice A.2.

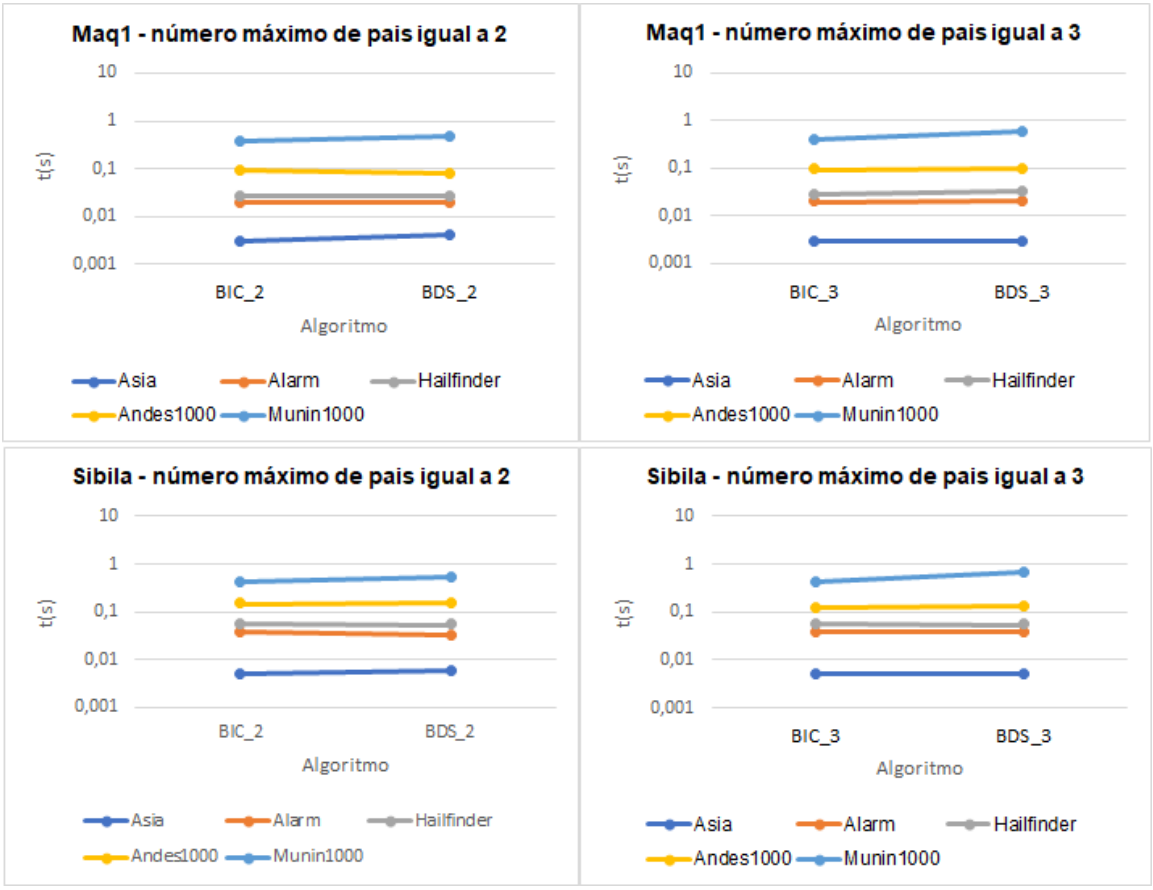


Figura 5.9: Tempos de execução das 2 funções de aprendizagem de estrutura implementadas em R, para os conjuntos de dados ASIA, ALARM, HAILFINDER, e ANDES e MUNIN com 1000 observações, com número máximo de pais igual a 2 (do lado esquerdo) e 3 (do lado direito), nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

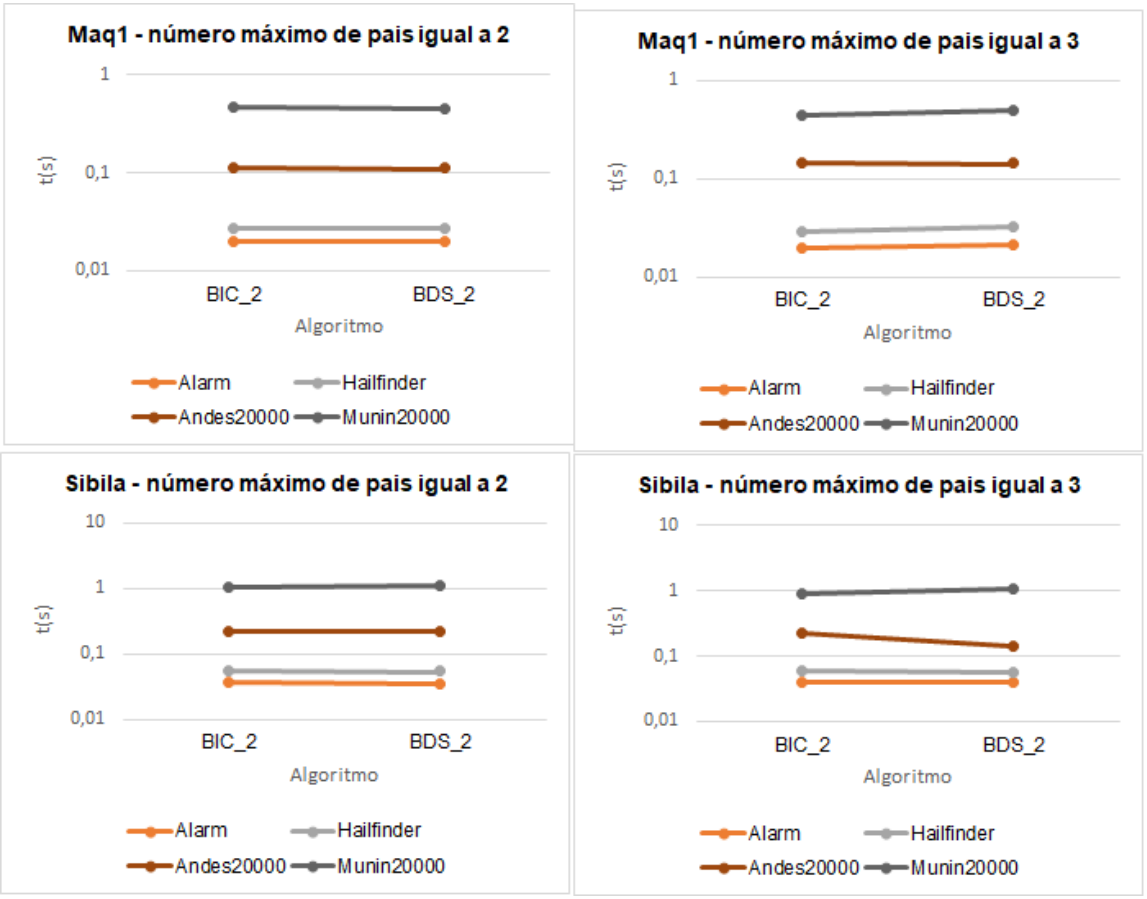


Figura 5.10: Tempos de execução das 2 funções de aprendizagem de estrutura implementadas em R, para os conjuntos de dados ASIA, ALARM, HAILFINDER, e ANDES e MUNIN com 20000 observações, com número máximo de pais igual a 2 (do lado esquerdo) e 3 (do lado direito), nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

Verificamos que os tempos de execução dos algoritmos de R são muito semelhantes (BIC_* e BDS_* nas Figuras).

Para comparação entre linguagens é facilmente visível que os algoritmos implementados em R são muito mais rápidos a aprender uma RB do que os implementados em Julia. Embora os autores mencionem que Julia pode ser mais rápida do que C isto não se verifica na biblioteca *Bayesnets*. É possível verificar que para todos os algoritmos considerados, de ambas as linguagens de programação e com diferentes números máximos de pais testados, os tempos de execução da Maq1 são inferiores aos da Sibila, o que pode ser explicado pelo fato da velocidade de CPU máxima da Maq1 ser superior (3100 MHz) à da Sibila (2300 MHz).

5.1.3 Características das redes bayesianas aprendidas

Pela verificação de igual número de arestas aprendido e LogPdf é possível afirmar que ambas as máquinas (Maq1 e Sibila) aprendem redes de classe equivalente para todos os conjuntos de

dados, em todos os algoritmos implementados em *BayesNets* e com igual número máximo de pais, essas características estão expressas nas tabelas 5.1, 5.3, 5.5, 5.7 e 5.9.

Também pela observação de mesmo número de arestas aprendido e iguais Score:log-lik afirmamos que as duas máquinas aprendem redes de classe equivalente para todos os conjuntos de dados, nos algoritmos testados no *bnlearn*, características essas apresentadas nas Tabelas 5.2, 5.4, 5.6 , 5.8 e 5.10.

Tabela 5.1: Características das Redes Bayesianas aprendidas com o conjunto de dados ASIA nos vários algoritmos da linguagem Julia.

Algoritmo	MaxPais	Arestas	LogPdf
GHC	2	9	-11035.0284887824
ScanGHC		9	-11033.8771853801
K2GS		7	-11034.8971672668
GHC	3	11	-11026.9289739635
ScanGHC		10	-11028.6867934127
K2GS		7	-11034.8971672668

Tabela 5.2: Características das Redes Bayesianas aprendidas com o conjunto de dados ASIA nos vários algoritmos da linguagem R.

Algoritmo	MaxPais	Arestas	Score: log-lik
HC BIC	2	7	-11034.9
HC BDS		7	-11034.9
HC BIC	3	7	-11034.9
HC BDS		7	-11034.9

Pela observação das Tabelas 5.1 e 5.2, a RB com melhor qualidade é a rede aprendida pelo algoritmo GHC com número máximo de pais igual a 3, porque é a que tem maior pontuação (identificada na tabela a verde). Para número máximo de pais igual a 2 podemos concluir que a rede com melhor qualidade é a aprendida pelo algoritmo ScanGC (identificada na tabela a azul).

Em seguida apresentamos as estruturas das RBs aprendidas para o conjunto de dados ASIA, com números máximos de pais iguais a 2 e 3, nas Figuras 5.11 e 5.12 respetivamente. Do lado esquerdo encontra-se representada a RB conhecida como *benchmark* (Real nas Figuras) que foi criada por especialistas [35], e que será considerada a rede "verdadeira". Seguidamente encontram-se as redes aprendidas por todos os algoritmos em estudo, nas quais as arestas aprendidas corretamente em função da primeira, estão preenchidas a preto, arestas ausentes ou com direções diferentes da primeira estão a vermelho e arestas não aprendidas a azul e desenhadas utilizando uma linha tracejada.

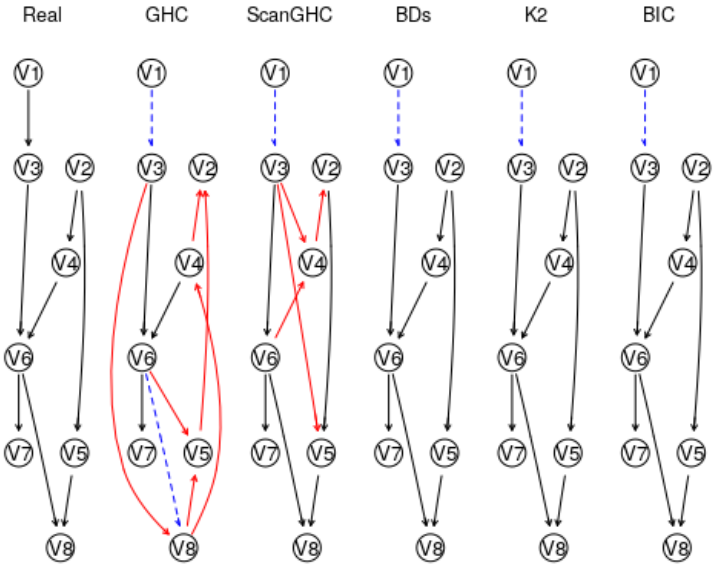


Figura 5.11: Estruturas de rede aprendidas para o conjunto de dados ASIA por todos os algoritmos com número máximo de pais igual a 2.

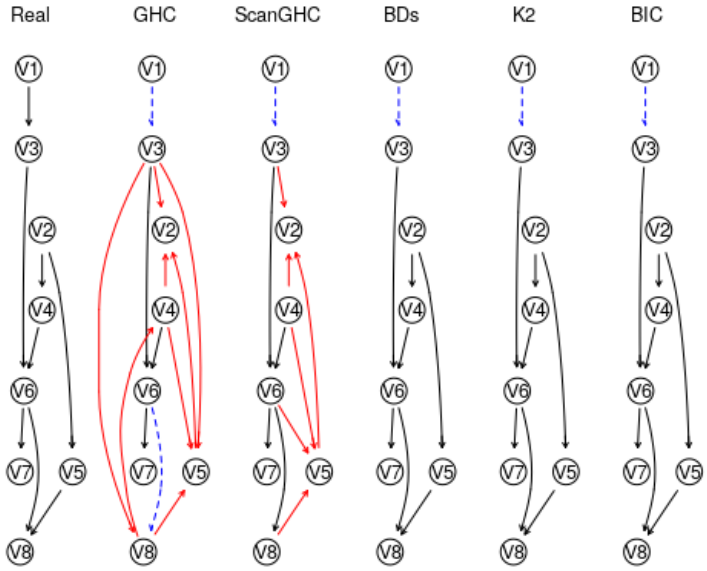


Figura 5.12: Estruturas de rede aprendidas para o conjunto de dados ASIA por todos os algoritmos com número máximo de pais igual a 3.

Ao observar as figuras anteriores intuitivamente diríamos que as redes que melhor representariam o *benchmark* seriam as dos algoritmos da biblioteca *bnlearn* com ambos os *scores* e o implementado pelo algoritmo K2 de Julia. O que contradiz as pontuações obtidas pela execução dos algoritmos.

É possível chegar a esta contradição em diferentes conjuntos de dados, pois tendo em

consideração que alguns *benchmarks*, como é o caso da rede ASIA, são redes criadas baseando-se no conhecimento de especialistas e não seguindo um algoritmo especificamente, assim as arestas criadas são as consideradas como causas e efeitos relevantes para esses especialistas, que podem não ser as que melhoram uma pontuação como acontece pela execução de um algoritmo. Assim a rede considerada como "verdadeira" pode não ser a que traduz uma melhor qualidade, mas sim a semanticamente melhor para os especialistas.

Com o aumento do número de variáveis num conjunto de dados torna-se impossível fazer estas comparações manualmente, assim como, extremamente difícil criar relações viáveis apenas com base no conhecimento de especialistas, por isso é que é importante automatizar os processos de aprendizagem para se conseguirem obter resultados mais rápidos e fidedignos. Embora já existam ferramentas que permitem comparar grafos, estas exigem preparações nos grafos a serem lidos o que se encontra fora do escopo do nosso trabalho. Sendo assim decidimos que nos conjuntos de dados que se seguem apenas analisaremos as pontuações obtidas pela execução dos algoritmos e o número de arestas aprendido e não as comparamos estruturalmente com os respetivos *benchmarks*.

Tabela 5.3: Características das Redes Bayesianas aprendidas com o conjunto de dados ALARM nos vários algoritmos da linguagem Julia.

Algoritmo	MaxPais	Arestas	LogPdf
GHC	2	53	-218184.517435974
ScanGHC		51	-220271.101878574
K2GS		61	-239498.744670881
GHC	3	54	-216713.937342424
ScanGHC		53	-216714.267939992
K2GS		80	-226396.857137122

Tabela 5.4: Características das Redes Bayesianas aprendidas com o conjunto de dados ALARM nos vários algoritmos da linguagem R.

Algoritmo	MaxPais	Arestas	Score: log- lik
HC BIC	2	50	-221276.2
HC BDS		52	-220859.6
HC BIC	3	53	-217488.6
HC BDS		55	-216288.6

Pelas tabelas 5.3 e 5.4, verificamos que a RB com melhor pontuação é a rede aprendida pelo algoritmo HC BDS de R com número máximo de pais igual a 3, (identificada na tabela a verde). Para a linguagem Julia o algoritmo que aprendeu a melhor pontuação foi o GHC também com número máximo de pais igual a 3 (identificada na tabela a violeta). Para número máximo de pais igual a 2 podemos concluir que a rede com melhor qualidade é a aprendida pelo GHC (identificada na tabela a azul).

Tabela 5.5: Características das Redes Bayesianas aprendidas com o conjunto de dados HAILFINDER nos vários algoritmos da linguagem Julia.

Algoritmo	MaxPais	Arestas	LogPdf
GHC	2	82	-985814.692455501
ScanGHC		82	-986008.204533399
K2GS		58	-989542.227590665
GHC	3	87	-981737.370029849
ScanGHC		87	-981737.370029849

Tabela 5.6: Características das Redes Bayesianas aprendidas com o conjunto de dados HAILFINDER nos vários algoritmos da linguagem R.

Algoritmo	MaxPais	Arestas	Score: log- lik
HC BIC	2	62	-985461.1
HC BDS		60	-986803.8
HC BIC	3	64	-981720.1
HC BDS		64	-982319

Pelas tabelas 5.5 e 5.6, verificamos que a RB com melhor pontuação é a rede aprendida pelo algoritmo HC BIC de R com número máximo de pais igual a 3, (identificada na tabela a verde). Para a linguagem Julia o algoritmo que aprendeu a melhor pontuação foi o ScanGHC também com número máximo de pais igual a 3 (identificada na tabela a violeta). Para número máximo de pais igual a 2 podemos concluir que a rede com melhor qualidade é a aprendida pelo HC BIC (identificada na tabela a azul).

Tabela 5.7: Características das Redes Bayesianas aprendidas com o conjunto de dados ANDES nos vários algoritmos da linguagem Julia.

Observações	Algoritmo	MaxPais	Arestas	LogPdf
1000	GHC	2	410	-94119.3531358949
	ScanGHC		402	-94368.7165882503
	K2GS		228	-94557.923598877
20000	GHC		388	-1882917.21238852
	ScanGHC		392	-1899554.40880454
	K2GS		256	-1897955.10327357
1000	GHC	3	512	-92469.3831537619
	ScanGHC		539	-92527.4732067445
20000	GHC		434	-1863025.15705913
	ScanGHC		448	-1867368.46015871

Tabela 5.8: Características das Redes Bayesianas aprendidas com o conjunto de dados ANDES nos vários algoritmos da linguagem R.

Observações	Algoritmo	MaxPais	Arestas	Score: log- lik
1000	HC BIC	2	327	-93846.73
	HC BDS		346	-93784.31
20000	HC BIC		349	-1887729
	HC BDs		350	-1887734
1000	HC BIC	3	342	-92463.81
	HC BDS		385	-92303.98
20000	HC BIC		353	-1861448
	HC BDS		357	-1861446

Pelas tabelas 5.7 e 5.8, verificamos que a RB com melhor pontuação é a rede aprendida pelo algoritmo HC BDS de R com número máximo de pais igual a 3, para ambos os números de observações (identificadas na tabela a verde). Para a linguagem Julia o algoritmo que aprendeu a melhor pontuação foi o GHC também com número máximo de pais igual a 3 (identificadas nas tabelas a violeta). Para número máximo de pais igual a 2 podemos concluir que a rede com melhor qualidade é a aprendida pelo HC BDS para 1000 observações, enquanto que para 20000 observações é o algoritmo GHC (identificadas na tabela a azul).

Podemos observar que o aumento do número de observações implica uma pior pontuação da rede. Contudo para ambos os números de observações aprenderem as melhores redes com os mesmos algoritmos.

Como para o conjunto de dados MUNIN não foi possível obter resultados no tempo definido como limite de execução (10 dias) para os algoritmos ScanGHC e K2 de Julia, nas Tabelas 5.9 e 5.10 só estão representadas as pontuações para os restantes algoritmos em estudo.

Tabela 5.9: Características das Redes Bayesianas aprendidas com o conjunto de dados MUNIN nos vários algoritmos da linguagem Julia.

Observações	Algoritmo	MaxPais	Arestas	LogPdf
1000	GHC	2	1128	-184990.437253634
20000			1069	-3565105.12833238
1000	GHC	3	1335	-183376.087333998
20000			1191	-3531256.57929736

Tabela 5.10: Características das Redes Bayesianas aprendidas com o conjunto de dados MUNIN nos vários algoritmos da linguagem R.

Observações	Algoritmo	MaxPais	Arestas	Score: log- lik
1000	HC BIC	2	682	-195682,1
	HC BDS		1130	-171680,6
20000	HC BIC		919	-3548003
	HC BDS		1024	-3520702
1000	HC BIC	3	681	-195645,5
	HC BDS		1313	-169828,5
20000	HC BIC		928	-3539251
	HC BDS		1110	-3488849

Pelas tabelas 5.9 e 5.10, verificamos que a RB com melhor pontuação é a rede aprendida pelo algoritmo HC BDS de R com número máximo de pais igual a 3, para ambos os números de observações (identificadas na tabela a verde). Para a linguagem Julia o algoritmo que aprendeu a melhor pontuação foi o GHC também com número máximo de pais igual a 3, para ambos os números de observações (identificadas na tabela a violeta). Para número máximo de pais igual a 2 podemos concluir que a rede com melhor qualidade é a aprendida pelo HC BDS para ambos os números de observações (identificadas nas tabelas a azul).

Como já observado no conjunto de dados ANDES, o aumento do número de observações implica uma pior pontuação da rede e também para ambos os números de observações aprendem as melhores redes com os mesmos algoritmos.

Depois de observar todas as tabelas das características aprendidas para todos os conjuntos de dados podemos afirmar que a linguagem R ainda aprende melhores redes qualitativamente do que a linguagem Julia. Também é notável que muitas vezes o algoritmo que dá as segundas melhores aprendizagens é o GHC com número máximo de pais igual a 3. Ao contrário do que acontece com o aumento do número de observações, quando aumentamos o número máximo de pais as redes aprendidas tendem a ser melhores (mas têm um custo temporal maior).

5.2 Abordagem paralela

Nesta secção analisamos os resultados obtidos da execução paralela de todos os algoritmos em estudo através de uma abordagem *multi-threading*. Para a Maq1, a nossa avaliação foi efetuada com 1, 2 e 4 *threads* e na Sibila foi possível avaliar o desempenho dos algoritmos com 1, 2, 4, e 8 *threads*.

5.2.1 Diferente número de pais

Os tempos de execução (em segundos, em escala logarítmica) obtidos pela execução paralela de todos os algoritmos implementados na biblioteca *BayesNets* de Julia com um conjunto de dados de cada categoria, para números máximos de pais com valores 2 e 3 estão apresentados nas Figuras 5.13, 5.14 e 5.15. As Figuras 5.16 e 5.17 mostram os tempos obtidos na execução paralela dos algoritmos da biblioteca *bnlearn* considerados para comparação.

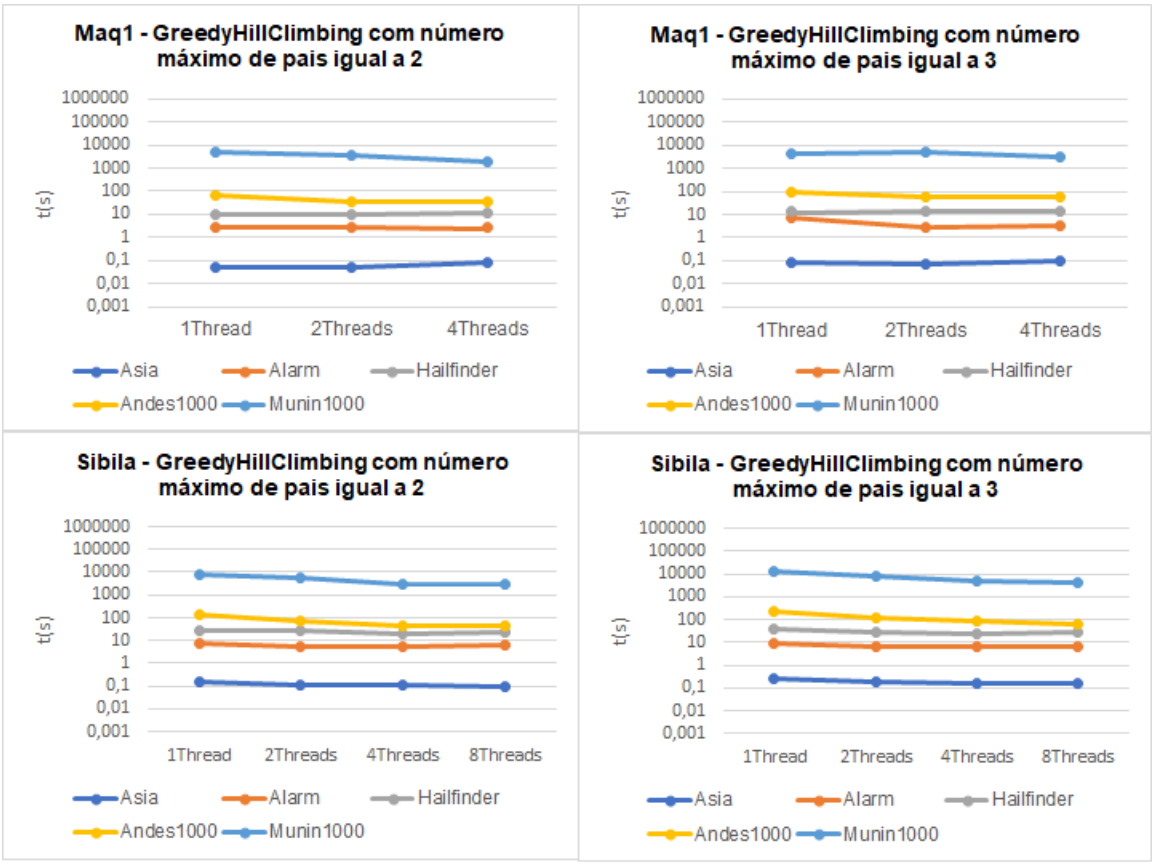


Figura 5.13: Tempos de execução com *GraphSearchStrategy* GreedyHillClimbing para os conjuntos de dados ASIA, ALARM, HAILFINDER, e ANDES e MUNIN com 1000 observações, com diferentes números de *threads* e número máximo de pais iguais a 2 (do lado esquerdo) e 3 (do lado direito), nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

Para os algoritmos com *GraphSearchStrategy* ScanGreedyHillClimbing e K2GraphSearch de Julia não foi possível obter resultados para o conjunto de dados MUNIN com 1000 observações, porque o seu tempo de execução excedeu o tempo definido como limite (10 dias). Assim nas Figuras 5.14, 5.15 só estão representados os restantes conjuntos de dados.

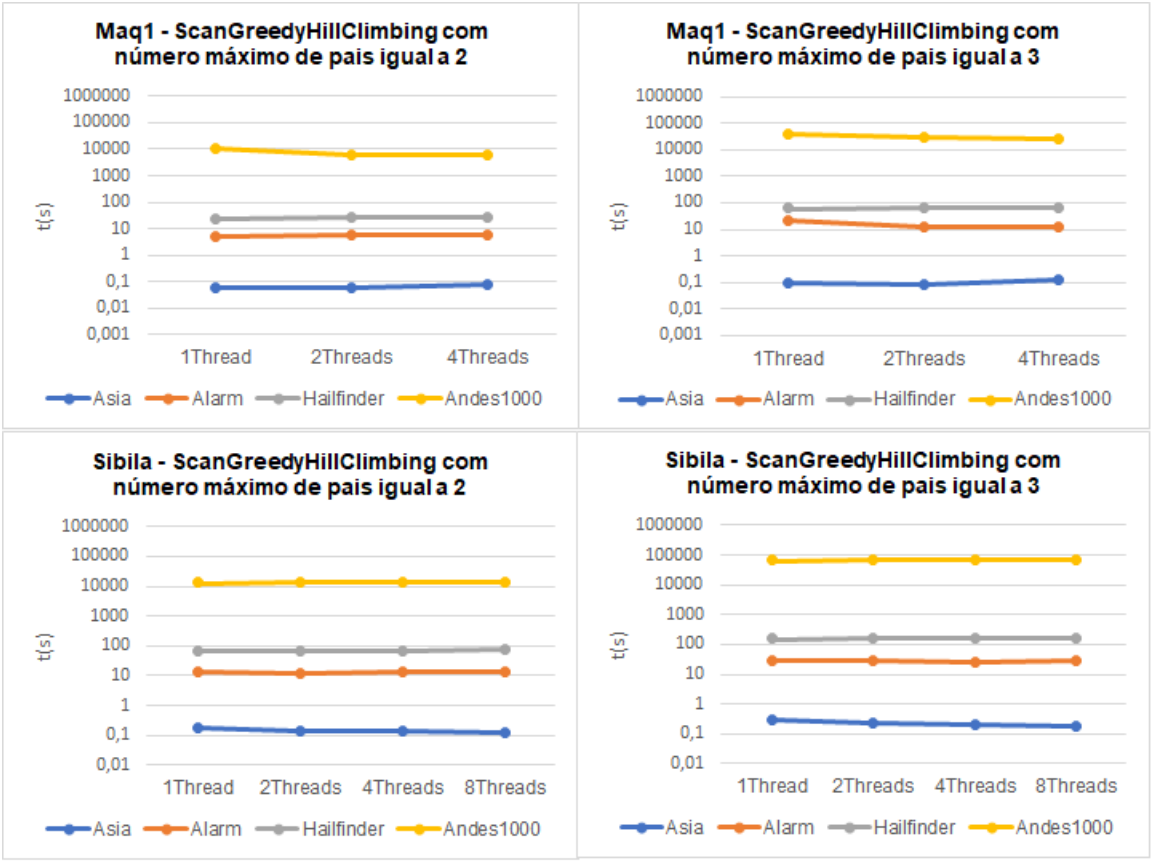


Figura 5.14: Tempos de execução com *GraphSearchStrategy* ScanGreedyHillClimbing para os conjuntos de dados ASIA, ALARM, HAILFINDER e ANDES com 1000 observações, com diferentes números de *threads* e número máximo de pais iguais a 2 (do lado esquerdo) e 3 (do lado direito), nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

Para o algoritmo com *GraphSearchStrategy* K2GraphSearch não foi possível obter resultados para o conjunto de dados HAILFINDER e ANDES com 1000 observações com número máximo de pais igual a 3, porque o seu tempo de execução excedeu o tempo definido como limite (10 dias). Assim na Figura 5.15 para número máximo de pais igual a 3 só estão representados os conjuntos de dados ASIA e ALARM.

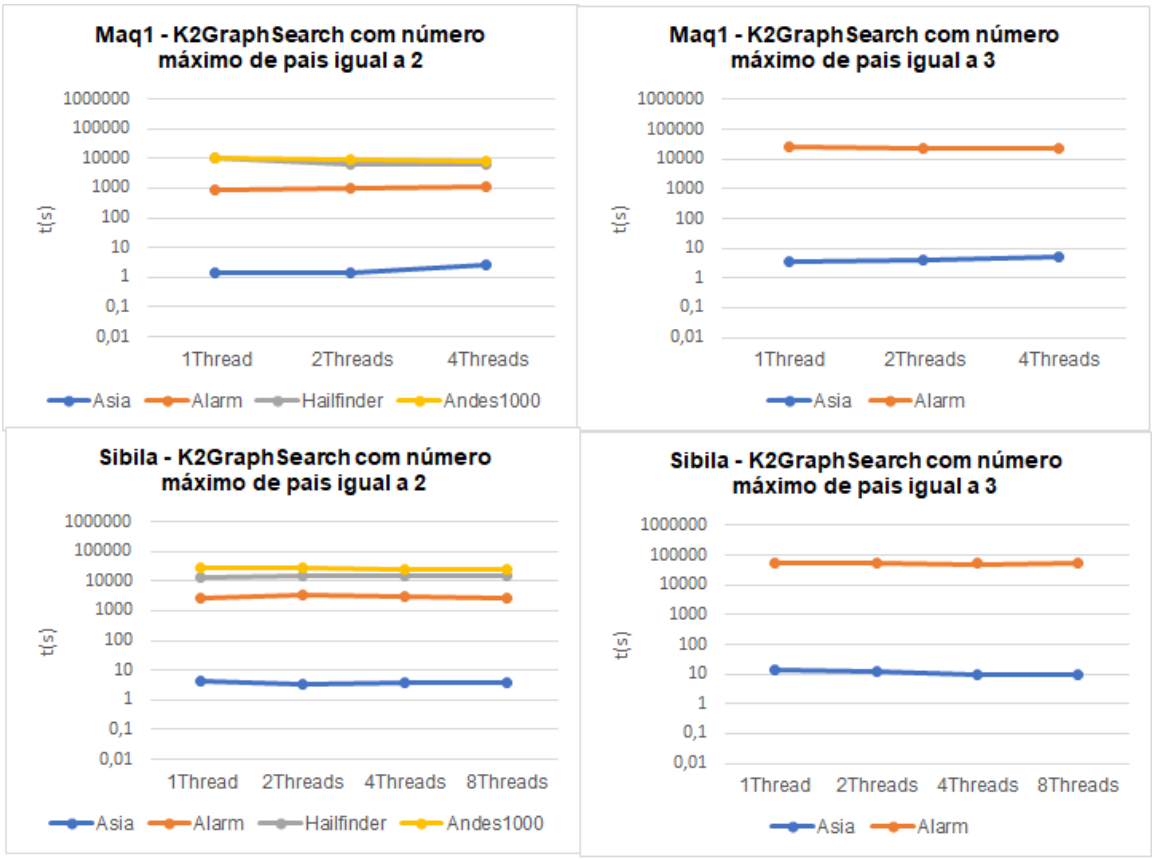


Figura 5.15: Tempos de execução da função K2Graphsearch para os conjuntos de dados ASIA e ALARM, HAILFINDER, e ANDES com 1000 observações, com diferentes números de *threads* e número máximo de pais iguais a 2 (do lado esquerdo) e 3 (do lado direito), nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

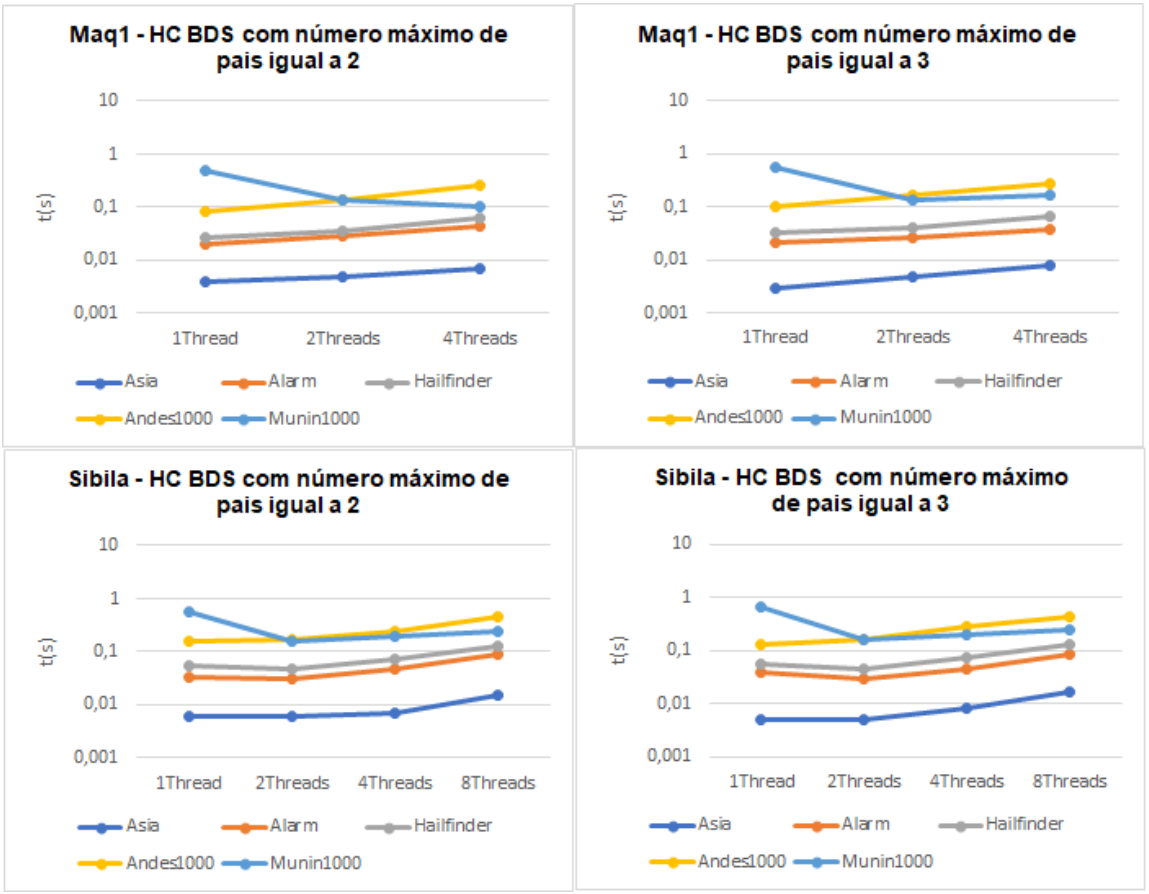


Figura 5.16: Tempos de execução da função hc(score = "bds") para os conjuntos de dados ASIA e ALARM, HAILFINDER, ANDES e MUNIN com 1000 observações, com diferentes números de *threads* e número máximo de pais iguais a 2 (do lado esquerdo) e 3 (do lado direito), nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

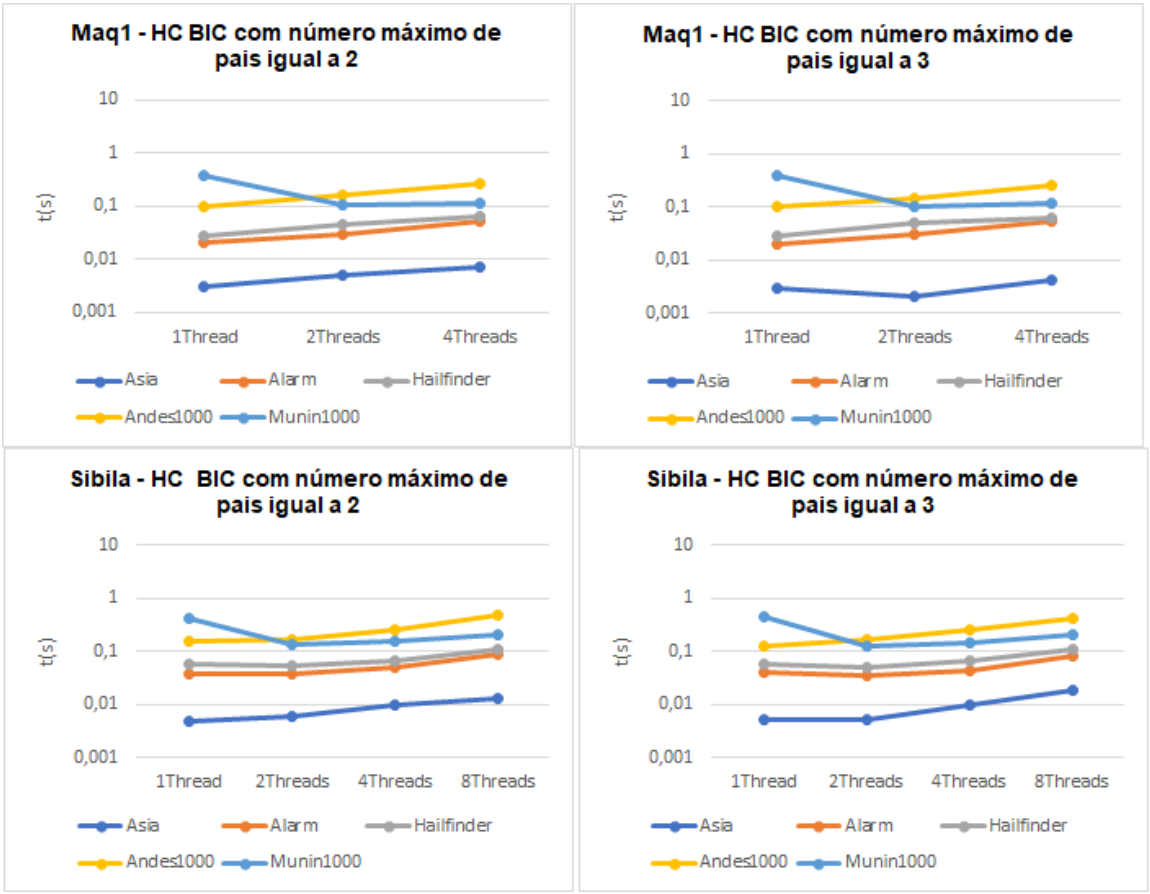


Figura 5.17: Tempos de execução da função hc(score = "bic") para os conjuntos de dados ASIA e ALARM, HAILFINDER, ANDES e MUNIN com 1000 observações, com diferentes números de *threads* e número máximo de pais iguais a 2 (do lado esquerdo) e 3 (do lado direito), nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

5.2.1.1 Mesmo número de observações

Do mesmo modo como comparado na abordagem sequencial, também nesta agrupamos os diferentes conjuntos de dados com 20000 observações (ALARM, HAILFINDER, ANDES e MUNIN) e em seguida apresentamos os tempos de execução (em segundos, em escala logarítmica) obtidos em todos os algoritmos de ambas as linguagens e para números máximos de pais com valores 2 e 3 nas Figuras 5.18, 5.19, 5.20, 5.21 e 5.22.

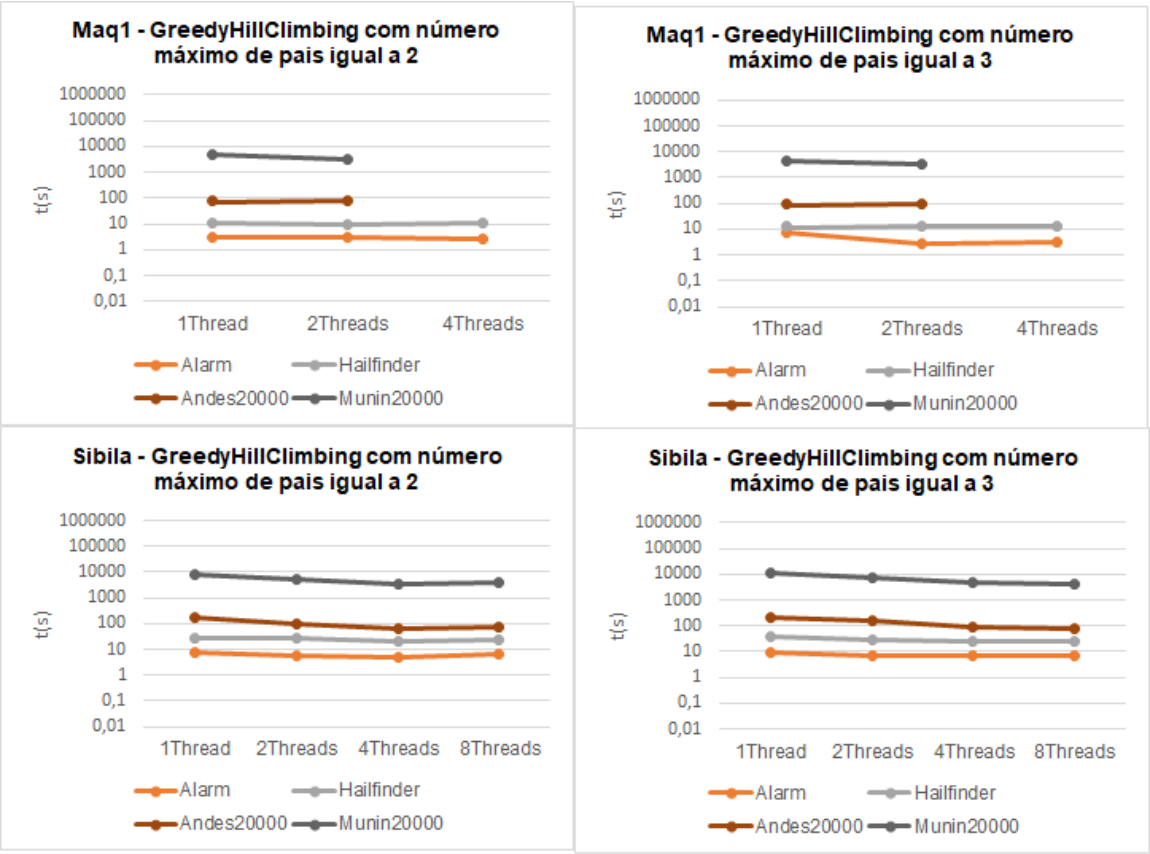


Figura 5.18: Tempos de execução com *GraphSearchStrategy* GreedyHillClimbing para os conjuntos de dados com 20000 observações, com número máximo de pais 2 e 3, nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

Como é possível verificar na Maq1 não foi possível obter resultados com 4 *threads* para os dois maiores conjuntos de dados (ANDES e MUNIN) pois a memória necessária para a execução do algoritmo excedeu o limite de memória existente na máquina (15GiB de RAM).

Como no conjunto de dados MUNIN com 1000 observações, também não foi possível obter resultados para o mesmo com 20000 observações, para os *GraphSearchStrategy* ScanGreedyHillClimbing e K2GraphSearch de Julia, porque o seu tempo de execução excedeu o tempo definido como limite de execução (10 dias). Assim nas Figuras 5.19, 5.20 só estão representados os restantes conjuntos de dados.

Igualmente para o algoritmo com *GraphSearchStrategy* K2GraphSearch também não conseguimos obter resultados em tempo útil para os conjuntos de dados HAILFINDER e ANDES com 20000 observações com número máximo de pais igual a 3. Por esse motivo só estão representados os resultados para número máximo de pais igual a 2, nesses dois conjuntos de dados para a *GraphSearchStrategy* K2GraphSearch.

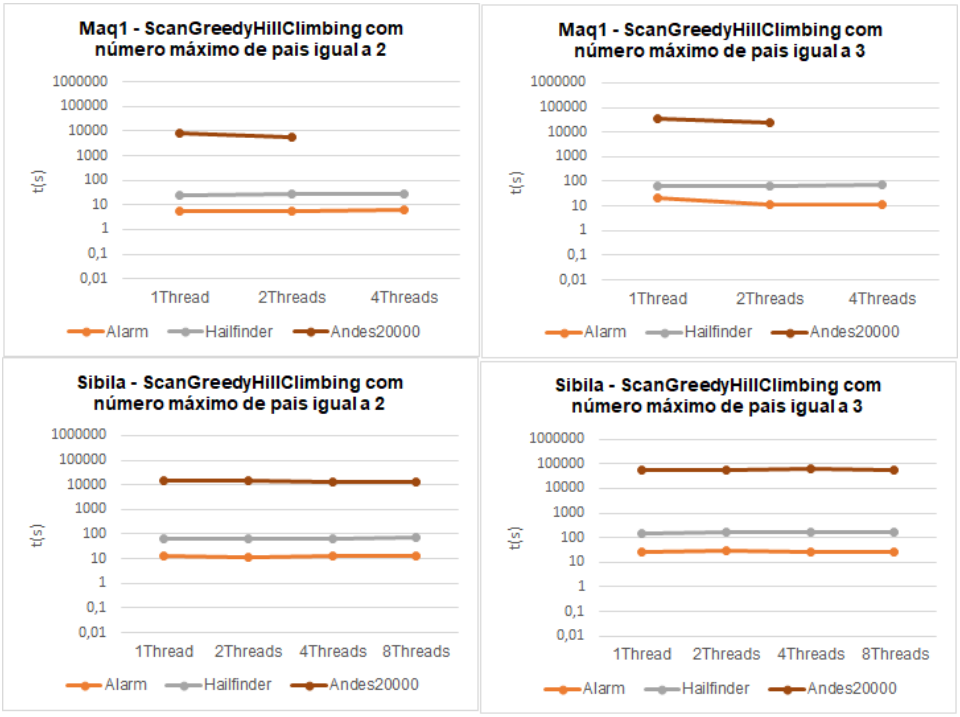


Figura 5.19: Tempos de execução com *GraphSearchStrategy* ScanGreedyHillClimbing para os conjuntos de dados com 20000 observações, com número máximo de pais 2 e 3, nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

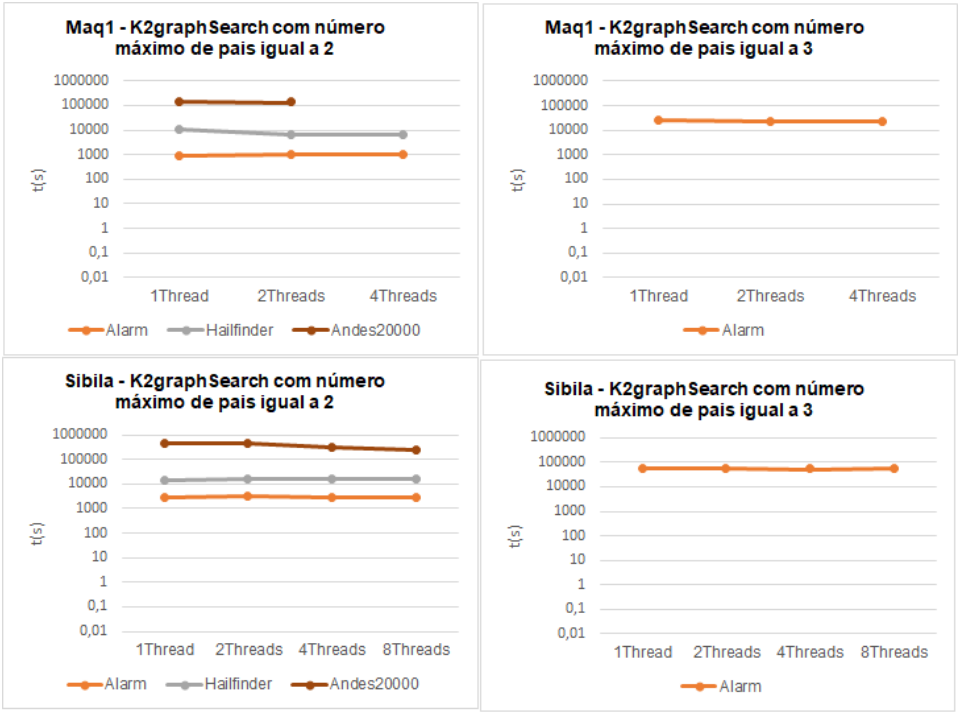


Figura 5.20: Tempos de execução com *GraphSearchStrategy* K2GraphSearch para os conjuntos de dados com 20000 observações, com número máximo de pais 2 e 3, nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

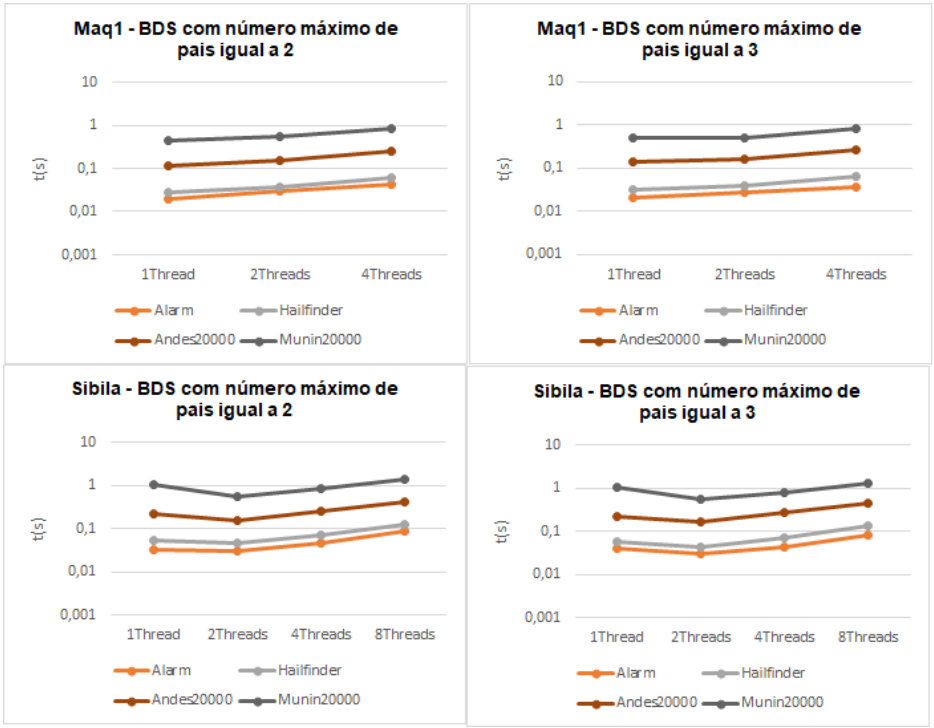


Figura 5.21: Tempos de execução da função hc(score = "bds") para os conjuntos de dados com 20000 observações, com número máximo de pais 2 e 3, nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

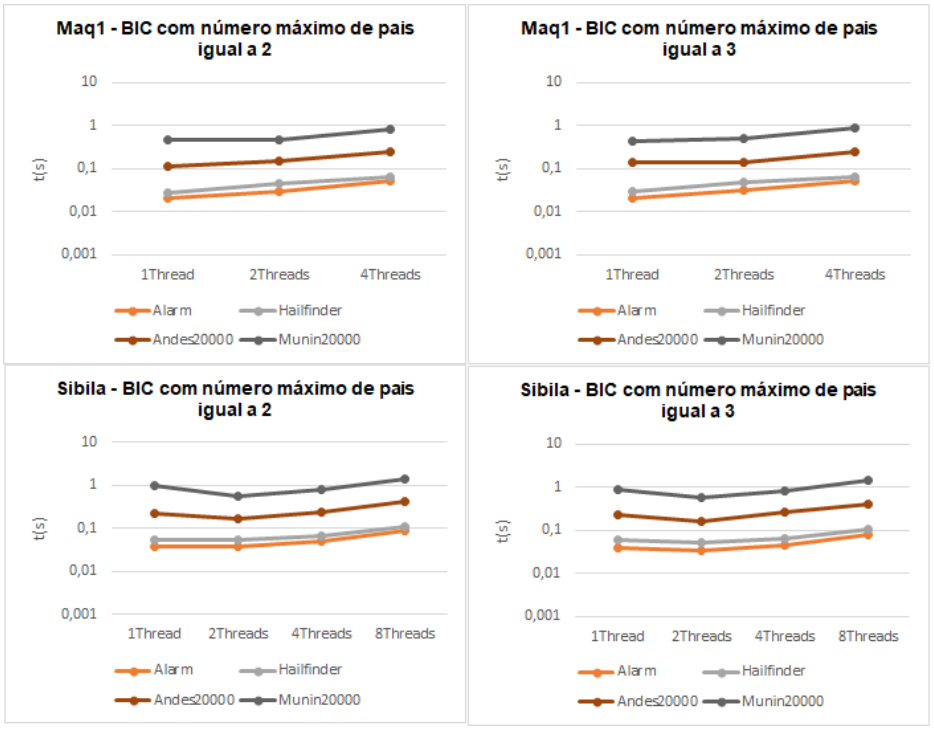


Figura 5.22: Tempos de execução da função hc(score = "bic") para os conjuntos de dados com 20000 observações, com número máximo de pais 2 e 3, nas duas máquinas utilizadas (em cima a Maq1 e em baixo a Sibila).

Assim como já observado na abordagem sequencial, confirma-se que o aumento do número de vértices numa RB implica um maior tempo de execução para todos os algoritmos em ambas as linguagens nas duas máquinas utilizadas, como já era esperado.

Igualmente para o conjunto de dados MUNIN não foi possível obter resultados em tempo útil para os algoritmos implementados pelas *GraphSearchStrategy* ScanGreedyHillClimbing e K2GraphSearch, assim como para os conjuntos de dados HAILFINDER e ANDES com 1000 e 20000 observações para a *GraphSearchStrategy* K2GraphSearch com número máximo de pais igual a 3 em ambas as máquinas utilizadas. Como verificado na execução sequencial, confirma-se que o aumento do número máximo de pais implica um maior tempo de execução, sendo mais significativo nas *GraphSearchStrategy* ScanGreedyHillClimbing e K2GraphSearch e quase insignificante nos algoritmos implementados na linguagem R.

Podemos confirmar também, como já visto na abordagem sequencial que nos conjuntos de dados com diferentes números de observações (ANDES e MUNIN) os tempos de execução não se alteram significativamente, e assim, que o aumento do número de observações não influência diretamente os tempos de aprendizagem dos algoritmos em estudo.

Ao observar os gráficos da linguagem Julia com diferentes números de *threads* (Figuras 5.13, 5.14, 5.15 5.18, 5.19 e 5.20) é percetível que os tempos de execução dos diferentes algoritmos em estudo no geral diminuiu.

Para a linguagem Julia, os conjuntos de dados mais pequenos, ASIA e ALARM, não beneficiam significativamente com o a utilização de um maior número de *threads*. Verifica-se até um aumento dos tempos de execução no conjunto de dados ASIA para 4 *threads* na Maq1 para todas as *GraphSearchStrategy* implementadas, visto a dimensão não ser elevada o suficiente e a velocidade máxima de CPU da Maq1 não justificar o uso de mais *threads*. Pelo contrário a Sibila beneficia mesmo que pouco da utilização de mais *threads*, pois a sua velocidade máxima de CPU é menor do que a da Maq1.

Com o aumento de dimensão dos conjuntos de dados verifica-se uma maior diminuição dos tempos de execução ao utilizar um maior número de *threads*, sendo essa diminuição mais significativa na Sibila do que na Maq1. Visto que a segunda tem uma velocidade de CPU mais elevada faz com que a paralelização seja menos benéfica nesta do que na Sibila.

Para a linguagem R (Figuras 5.16, 5.17, 5.21 e 5.22), observa-se um aumento dos tempos de execução com o aumento do número de *threads* utilizadas no conjunto de dados ASIA para os dois algoritmos, nas duas máquinas utilizadas. Para o conjunto de dados ANDES com 1000 observações, os melhores tempos na Maq1 foram obtidos pela utilização de 2 *threads*, enquanto que na Sibila se obteram os melhores tempos com apenas uma *thread*. Para o conjunto de dados ALARM, ANDES e MUNIN com 20000 observações, os melhores tempos obtidos na Maq1 verificam-se com a utilização de apenas uma *thread*, enquanto que na Sibila se verificam com 2 *threads* e tem piores tempos de execução para os restantes números de *threads*. Para os conjuntos de dados HAILFINDER e MUNIN com 1000 observações, os melhores tempos são obtidos em ambas as máquinas com a utilização de 2 *threads*. Visto a aprendizagem dos algoritmos

implementados em R na execução sequencial ser bastante rápida a adição de *threads* na execução paralela não contribui para a diminuição dos tempos de execução, pois para os conjuntos de dados utilizados a quantidade de computação não justifica um aumento de processadores.

Na abordagem paralela, confirmamos que o algoritmo de aprendizagem em Julia com menores tempos de execução é o implementado com *GraphSearchStrategy* GreedyHillClimbing e que devido à sua implementação iterativa também é o que mais beneficia com a paralelização do seu código, como confirmado pela observação da dimimuição dos tempos de execução. Enquanto que para a *GraphSearchStrategy* SanGreedyHillClimbing verificámos que a rede é aprendida de forma recursiva e que a sua paralelização não vai diminuir significativamente os tempos de execução. Para o algoritmo com maiores tempos de execução, o da *GraphSearchStrategy* K2GraphSearch pelo estudo da sua implementação verificámos que tinha uma complexidade cúbica e que só poderíamos aplicar *multi-threading* no primeiro ciclo for, assim a utilização de maior número de *threads* não iria diminuir significativamente os tempos de execução, como podemos verificar.

Verificamos que os tempos de execução dos algoritmos de R são muito semelhantes em ambas as abordagens.

Confirmamos mais uma vez que os algoritmos implementados em R são muito mais rápidos a aprender uma RB do que os implementados em Julia, e que não beneficiam significativamente com uma abordagem paralela, piorando até alguns tempos de execução.

Confirmamos também que para todos os algoritmos considerados de ambas as linguagens de programação e diferentes números máximos de pais os tempos de execução da Maq1 são inferiores aos da Sibila.

5.2.2 Características das redes bayesianas aprendidas

Pela verificação de igual número de arestas aprendido e LogPdf é possível afirmar que ambas as máquinas (Maq1 e Sibila) aprendem a mesma rede ou redes de classe equivalente para os conjuntos de dados ASIA e ALARM em todos os algoritmos implementados em *BayesNets* com igual número máximo de pais e para os diferentes números de *threads* utilizadas, assim as características das redes aprendidas são iguais às da Abordagem Sequencial (Tabelas 5.1 e 5.3).

Do mesmo modo a observação do mesmo número de arestas aprendido e iguais Score:log-lik permite-nos afirmar que as duas máquinas aprendem redes de classe equivalente para os conjuntos de dados ASIA, ALARM, HAILFINDER e MUNIN com os diferentes números de observações e diferentes números de *threads*, nos algoritmos testados no *bnlearn* com igual número máximo de pais e para os diferentes números de *threads* utilizadas, assim as características das redes aprendidas são iguais às da Abordagem Sequencial (Tabelas 5.2, 5.4, 5.6 e 5.10).

Vamos agora analisar os conjuntos de dados que obtiveram redes diferentes ou de diferente classe das aprendidas pela abordagem sequencial. Em primeiro para o conjunto de dados HAILFINDER apenas o algoritmo GHC com número de pais igual a 3 aprende uma rede diferente

da abordagem sequencial sendo mostrado na Tabela 5.11, onde os valores diferentes da Abordagem Sequencial estão mostrados a negrito. Todos os restantes aprendem redes de equivalentes ou de classe equivalente à abordagem sequencial (Tabela 5.5).

Tabela 5.11: Características das Redes Bayesianas aprendidas com o conjunto de dados HAILFINDER nos vários algoritmos da linguagem Julia com diferente número de *threads*.

Threads	Algoritmo	MaxPais	Arestas	LogPdf
1	GHC	3	87	-981737.370029849
8	GHC		**86**	**-981739.315191239**

Observando a Tabela 5.11 para o algoritmo GHC com número máximo de pais igual a 3 a abordagem paralela obtém uma rede de pior qualidade, mas muito próxima da abordagem sequencial, pois os seus LogPdf's são muito próximos e o número de arestas apenas divergem numa aresta.

Para o conjunto de dados ANDES com 1000 observações, todos os algoritmos implementados por *BayesNets* aprendem redes equivalentes à Abordagem Sequencial para os diferentes números de *threads* na Maq1 e na Sibila. Para o conjunto de dados ANDES com 20000 observações apenas o algoritmo GHC com número máximo de pais igual a 2 com 4 e 8 *threads* aprende redes diferentes da Abordagem Sequencial sendo mostrados na Tabela 5.12, na qual os valores diferentes da Abordagem Sequencial estão mostrados a negrito.

Tabela 5.12: Características das Redes Bayesianas aprendidas com o conjunto de dados ANDES nos vários algoritmos da linguagem Julia com diferente número de *threads*.

Observações	Threads	Algoritmo	MaxPais	Arestas	LogPdf
20000	1	GHC	2	388	-1 882 917.21238852
	4			**394**	**-945004,039056778**
	8			**392**	**-1882995,09991679**

Observando a tabela 5.12 a abordagem paralela com 4 *threads* obtém uma rede de melhor qualidade do que a aprendida pela abordagem sequencial, mas para 8 *threads* a rede aprendida tem pior qualidade, considerando os valores dos LogPdf's.

Na linguagem R, para os conjunto de dados ANDES com 20000 observações, os algoritmos testados pelo *bnlearn* aprendem redes equivalentes à Abordagem Sequencial para os diferentes números de *threads* na Maq1 e na Sibila (Tabela 5.8). Para o conjunto de dados Andes com 1000 observações, as redes aprendidas para diferentes números de *threads* são diferentes da Abordagem Sequencial e são mostradas as características aprendidas na Tabela 5.13, na qual os valores diferentes da Abordagem Sequencial estão mostrados a negrito.

Tabela 5.13: Características das Redes Bayesianas aprendidas com o conjunto de dados ANDES nos vários algoritmos da linguagem R com diferente número de *threads*.

Observações	Threads	Algoritmo	MaxPais	Arestas	Score:log-lik
1000	1	HC BIC	2	327	-93846,73
	2			**349**	**-1887729**
	4			**349**	**-1887729**
	8			**349**	**-1887729**
	1	HC BDS		346	-93784.31
	2			**350**	**-1887734**
	4			**350**	**-1887734**
	8			**350**	**-1887734**
1000	1	HC BIC	3	342	-92463,81
	2			**353**	**-1861448**
	4			**353**	**-1861448**
	8			**353**	**-1861448**
	1	HC BDS		385	-92303.98
	2			**357**	**-1861446**
	4			**357**	**-1861446**
	8			**357**	**-1861446**

Observando a tabela 5.13 a abordagem paralela para todos os diferentes números de *threads* obtêm redes de pior qualidade do que a aprendida pela abordagem sequencial, considerando os valores dos LogPdf's.

Para o conjunto de dados MUNIN ambas as máquinas aprendem a mesma rede no único algoritmo em que foi possível obter resultados para a linguagem Julia, mas aprendem redes diferentes com números de *threads* iguais a 2 e 8 no conjunto de dados com 1000 observações, e com números de *threads* iguais a 4 e 8 no conjunto de dados com 20000 observações como mostramos na Tabela 5.14.

Tabela 5.14: Características das Redes Bayesianas aprendidas com o conjunto de dados MUNIN nos vários algoritmos da linguagem Julia com diferentes números de *threads*.

Observações	Threads	Algoritmo	MaxPais	Arestas	LogPdf
1000	1	GHC	2	1028	-184990,437253634
	2			**1027**	**-184993,131752802**
20000	1			1069	-3565105,12833238
	4			1069	**-3565078,30570966**
	8			**1064**	**-3563653,98567271**
1000	1	GHC	3	1335	-183376,087333998
	8			**1333**	**-183379,146476279**
20000	1			1191	-3531256,57929736
	8			**1190**	**-3531919,86716583**

Observando a tabela 5.14 a abordagem paralela para o conjunto de dados com 1000 observações

obtêm redes de pior qualidade do que a aprendida pela abordagem sequencial, considerando os valores dos LogPdf's, para os diferentes números máximos de pais. Para o conjunto de dados com 20000 observações a abordagem paralela com 4 e 8 *threads* para número máximo de pais igual a 2 obtêm redes com melhor qualidade do que a aprendida pela abordagem sequencial, enquanto que para número máximo de pais igual a 3 a abordagem paralela com 8 *threads* aprende uma rede de pior qualidade.

Capítulo 6

Conclusões e Trabalho Futuro

Esta dissertação mostra uma comparação entre o desempenho dos algoritmos de aprendizagem de Redes Bayesianas (RBs) implementados na recente linguagem de programação Julia e na linguagem R. São comparadas execuções sequenciais e paralelas em ambas.

Foi possível verificar pela execução sequencial que os algoritmos implementados pela linguagem de programação Julia tinham tempos de execução muito mais elevados do que os equivalentes em R. Chegando a um limite de espera para a sua conclusão de 10 dias e ainda assim alguns não terminaram as suas execuções.

Verificou-se que a implementação mais rápida em Julia é a da *GraphSearchStrategy* GreedyHillClimbing, a qual leva entre 0,05 e 10,54 segundos a executar o menor conjunto de dados selecionado, ASIA, e entre 1,30 e 3,97 horas a executar o maior conjunto de dados selecionado, MUNIN com 20000 observações.

Para o algoritmo mais demorado, implementado pela *GraphSearchStrategy* K2GraphSearch demora entre 1,35 e 14,91 segundos para o menor conjunto de dados e entre 2,26 e 7,92 horas para o conjunto de dados ANDES, o maior para o qual foi possível obter resultados no tempo limite de 10 dias.

Confirma-se também o já esperado, o aumento do número de vértices e número máximo de pais aumentam os tempos de execução, para todo os algoritmos implementados.

Como os algoritmos de aprendizagem de RBs escalam exponencialmente com o número de váriáveis. Foi adotada uma estratégia de otimização, tentando maximizar o uso do *hardware* disponível, pela abordagem *multi-threading* nativa presente em Julia. Porque nos algoritmos implementados em Julia os seus tempos de execução eram muito elevados, mas os mesmos não tiravam proveito de todo o hardware disponível, assim dividindo os cálculos por várias *threads* conseguimos melhorar os seus desempenhos ao nível dos tempos de execução.

Para os conjuntos de dados mais pequenos, ASIA, ALARM e HAILFINDER, não foi possível verificar uma grande diminuição nos tempos de execução com o aumento do número de *threads*, havendo até tempos de execução piores, o que também era esperado pela pequena dimensão dos

dados.

Para conjuntos de dados maiores, como é o caso do ANDES e MUNIN, foi possível verificar uma diminuição dos tempos de execução, sendo o algoritmo que obteve mais melhorias no tempo de execução o da *GraphSearchStrategy* GreedyHillClimbing, chegando a melhorias de mais de 2 horas para o conjunto de dados MUNIN.

Para a linguagem R a biblioteca utilizada *bnlearn* já era compatível com outras bibliotecas com implementações paralelas, nós utilizamos a biblioteca *snow* para as experiências paralelas em R.

Com o desenvolvimento desta dissertação foi possível identificar uma grande limitação nas implementações de aprendizagem de RBs existentes na linguagem Julia, mostrando que esta ainda não está preparada para abordar RBs de grande escala, enquanto que a linguagem R tem um bom desempenho e nos conjuntos de dados utilizados não necessitava da utilização de paralelização.

Como trabalhos futuros, sugerimos uma abordagem de paralelização diferente com uma analise das complexidades dos algoritmos e possíveis melhorias nas implementações disponíveis, pois só fazer uso do *hardware* disponível não foi suficiente para obter melhorias significativas nos tempos de execução nos algoritmos implementados pela linguagem de programação Julia. Também seria interessante abordar mais detalhes sobre a forma como as linguagens utilizadas, as características dos conjuntos de dados e uma diferente estratégia de paralelização influenciam o desempenho dos algoritmos.

Referências

[1] Neapolitan RE, et al. Learning bayesian networks. vol. 38. Pearson Prentice Hall Upper Saddle River, NJ; 2004.

[2] Pearl J. Probabilistic reasoning in intelligent systems: networks of plausible inference. Elsevier; 2014.

[3] Nagarajan R, Scutari M, Lèbre S. Bayesian networks in r. Springer. 2013;122:125–127.

[4] Verma T, Pearl J. Equivalence and synthesis of causal models. UCLA, Computer Science Department; 1991.

[5] Bouckaert RR. Bayesian belief networks: from construction to inference; 1995.

[6] Larrañaga P, Sierra B, Gallego MJ, Michelena MJ, Picaza JM. Learning Bayesian networks by genetic algorithms: a case study in the prediction of survival in malignant skin melanoma. In: Conference on Artificial Intelligence in Medicine in Europe. Springer; 1997. p. 261–272.

[7] Russell Stuart J, Norvig P. Artificial intelligence: A modern approach. Prentice Hall; 2009.

[8] Castelo R, Roverato A. A robust procedure for Gaussian graphical model search from microarray data with p larger than n. Journal of Machine Learning Research. 2006;7(Dec):2621–2650.

[9] Cover T, Thomas J. Elements of Information Theory 2nd edn Wiley. Hoboken; 2006.

[10] Edwards D. Introduction to graphical modelling. Springer Science & Business Media; 2012.

[11] Hausser J, Strimmer K. Entropy inference and the James-Stein estimator, with application to nonlinear gene association networks. Journal of Machine Learning Research. 2009;10(Jul):1469–1484.

[12] Scutari M, Brogini A. Bayesian network structure learning with permutation tests. Communications in Statistics-Theory and Methods. 2012;41(16-17):3233–3243.

[13] Witten IH, Frank E, Hall MA, Pal CJ. Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann; 2016.

[14] Rissanen J. Modeling by shortest data description. Automatica. 1978;14(5):465–471.

[15] Lam W, Bacchus F. Learning Bayesian belief networks: An approach based on the MDL principle. Computational intelligence. 1994;10(3):269–293.

[16] Heckerman D, Geiger D, Chickering DM. Learning Bayesian networks: The combination of knowledge and statistical data. Machine learning. 1995;20(3):197–243.

[17] Margaritis D. Learning Bayesian network model structure from data. Carnegie-Mellon Univ Pittsburgh Pa School of Computer Science; 2003.

[18] Tsamardinos I, Aliferis CF, Statnikov AR, Statnikov E. Algorithms for Large Scale Markov Blanket Discovery. In: FLAIRS conference. vol. 2; 2003. p. 376–380.

[19] Yaramakala S, Margaritis D. Speculative Markov blanket discovery for optimal feature selection. In: Fifth IEEE International Conference on Data Mining (ICDM'05). IEEE; 2005. p. 4–pp.

[20] Spirtes P, Glymour C. An algorithm for fast recovery of sparse causal graphs. Social science computer review. 1991;9(1):62–72.

[21] Spirtes P, Glymour CN, Scheines R, Heckerman D, Meek C, Cooper G, et al. Causation, prediction, and search. MIT press; 2000.

[22] Chickering DM. Learning Bayesian networks is NP-complete. In: Learning from data. Springer; 1996. p. 121–130.

[23] Cooper GF. The computational complexity of probabilistic inference using Bayesian belief networks. Artificial intelligence. 1990;42(2-3):393–405.

[24] Flynn MJ. Some computer organizations and their effectiveness. IEEE transactions on computers. 1972;100(9):948–960.

[25] Tierney L, Rossini A, Li N, Sevcikova H. snow: Simple network of workstations. R package version 03-3, URL http://CRAN R-project org/package= snow. 2008;.

[26] Yu H. Rmpi: Interface (Wrapper) to MPI (Message-Passing Interface). R package version 0.5-8; 2010.

[27] Nagarajan R, Scutari M, Lèbre S. Parallel Computing for Bayesian Networks. In: Bayesian Networks in R. Springer; 2013. p. 103–123.

[28] Li N. rsprng: R Interface to SPRNG (Scalable Parallel Random Number Generators). R package version. 2010;1.

[29] Scutari M. Learning Bayesian networks with the bnlearn R package. arXiv preprint arXiv:09083817. 2009;.

[30] Scutari M. Bayesian network constraint-based structure learning algorithms: Parallel and optimised implementations in the bnlearn r package. arXiv preprint arXiv:14067648. 2014;.

[31] Kang Y, Yang X, Sun M, Hu J, Zhong Z, Liu J. Comparison of software packages for Bayesian network learning in gene regulatory relationship mining. In: 2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD). IEEE; 2017. p. 2010–2015.

[32] Madsen AL, Jensen F, Salmerón A, Langseth H, Nielsen TD. A parallel algorithm for Bayesian network structure learning from large data sets. Knowledge-Based Systems. 2017;117:46–55.

[33] Chu CT, Kim SK, Lin YA, Yu Y, Bradski G, Olukotun K, et al. Map-reduce for machine learning on multicore. In: Advances in neural information processing systems; 2007. p. 281–288.

[34] Basak A, Brinster I, Ma X, Mengshoel OJ. Accelerating Bayesian network parameter learning using Hadoop and MapReduce. In: Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications. ACM; 2012. p. 101–108.

[35] Lauritzen SL, Spiegelhalter DJ. Local computations with probabilities on graphical structures and their application to expert systems. Journal of the Royal Statistical Society: Series B (Methodological). 1988;50(2):157–194.

[36] Beinlich IA, Suermondt HJ, Chavez RM, Cooper GF. The ALARM monitoring system: A case study with two probabilistic inference techniques for belief networks. In: AIME 89. Springer; 1989. p. 247–256.

[37] Conati C, Gertner AS, VanLehn K, Druzdzel MJ. On-line student modeling for coached problem solving using Bayesian networks. In: User Modeling. Springer; 1997. p. 231–242.

[38] Andreassen S, Jensen FV, Andersen SK, Falck B, Kjærulff U, Woldbye M, et al. MUNIN: an expert EMG Assistant. In: Computer-aided electromyography and expert systems. Pergamon Press; 1989. p. 255–277.

Apêndice A

Algoritmos implementados em Julia

Apresentação das implementações dos algoritmos de aprendizagem presentes na biblioteca *Bayesnests* de Julia para mais referências, consulte https://github.com/sisl/BayesNets.jl.

A.1 *GraphSearchStrategy* GreedyHillClimbing

```
function Distributions.fit(::Type{DiscreteBayesNet}, data::DataFrame, params::GreedyHillClimbing;
    ncategories::Vector{Int} = map!(i->infer_number_of_instantiations(data[!,i]), Array(Int)(undef, ncol(data)), 1:ncol(data)),
    )

    n = ncol(data)
    parent_list = map!(i->Int[], Array{Vector{Int}}(undef, n), 1:n)
    datamat = convert(Matrix{Int}, data)'
    score_components = bayesian_score_components(parent_list, ncategories, datamat, params.prior, params.cache)

    while true
        best_diff = 0.0
        best_parent_list = parent_list
        for i in 1:n

            # 1) add an edge (j->i)
            if length(parent_list[i]) < params.max_n_parents
                for j in deleteat!(collect(1:n), parent_list[i])
                    if adding_edge_preserves_acyclicity(parent_list, j, i)
                        new_parents = sort!(push!(copy(parent_list[i]), j))
                        new_component_score = bayesian_score_component(i, new_parents, ncategories, datamat, params.prior, params.cache)
                        if new_component_score - score_components[i] > best_diff
                            best_diff = new_component_score - score_components[i]
                            best_parent_list = deepcopy(parent_list)
                            best_parent_list[i] = new_parents
                        end
                    end
                end
            end

            # 2) remove an edge
            for (idx, j) in enumerate(parent_list[i])

                new_parents = deleteat!(copy(parent_list[i]), idx)
                new_component_score = bayesian_score_component(i, new_parents, ncategories, datamat, params.prior, params.cache)
                if new_component_score - score_components[i] > best_diff
                    best_diff = new_component_score - score_components[i]
                    best_parent_list = deepcopy(parent_list)
                    best_parent_list[i] = new_parents
                end

                # 3) flip an edge
                new_parent_list = deepcopy(parent_list) # TODO: make this more efficient
                deleteat!(new_parent_list[i], idx)

                if adding_edge_preserves_acyclicity(new_parent_list, i, j)
                    sort!(push!(new_parent_list[j], i))
                    new_diff = bayesian_score_component(i, new_parent_list[i], ncategories, datamat, params.prior, params.cache) - score_components[i]
                    new_diff += bayesian_score_component(j, new_parent_list[j], ncategories, datamat, params.prior, params.cache) - score_components[j]
                    if new_diff > best_diff
```

Figura A.1: Código implementado pela biblioteca *BayesNets* para aprendizagem de parâmetros com a *GraphSearchStrategy*: GreedyHillClimbing -parte1 (fonte[GHC]).

```
                        best_diff = new_diff
                        best_parent_list = new_parent_list
                    end
                end
            end
        end

        if best_diff > 0.0
            parent_list = best_parent_list
            score_components = bayesian_score_components(parent_list, ncategories, datamat, params.prior, params.cache)
        else
            break
        end
    end

    # construct the BayesNet
    cpds = Array(DiscreteCPD)(undef, n)
    varnames = names(data)
    for i in 1:n
        name = varnames[i]
        parents = varnames[parent_list[i]]
        cpds[i] = fit(DiscreteCPD, data, name, parents, params.prior, parental_ncategories=ncategories[parent_list[i]], target_ncategories=ncategories[i])
    end
    BayesNet(cpds)
end
```

Figura A.2: Código implementado pela biblioteca *BayesNets* para aprendizagem de parâmetros com a *GraphSearchStrategy*: GreedyHillClimbing -parte2 (fonte[GHC]).

A.2 *GraphSearchStrategy* ScanGreedyHillClimbing

```
function Distributions.fit(::Type{DiscreteBayesNet}, data::DataFrame, params::ScanGreedyHillClimbing;
    ncategories::Vector{Int} = map!(i->infer_number_of_instantiations(data[!,i]), Array{Int}(undef, ncol(data)), 1:ncol(data)),
    )

    n = ncol(data)
    parent_list = map!(i->Int[], Array{Vector{Int}}(undef, n), 1:n)
    datamat = convert(Matrix{Int}, data)'
    score_components = bayesian_score_components(parent_list, ncategories, datamat, params.prior, params.cache)

    # 0 depth
    depth = 0
    best, out_score = greedy_score(score_components, n, parent_list, datamat, params, ncategories)

    # > 1 depth
    greedy_parents = out_score
    while depth < params.max_depth
        depth += 1

        # Scan parameters
        best_parent_list = parent_list
        complete = true
        for i in 1:n

            # 1) add an edge (j->i)
            for j in deleteat!(collect(1:n), parent_list[i])
                if adding_edge_preserves_acyclicity(parent_list, j, i)
                    if length(parent_list[i]) < params.max_n_parents
                        new_parent_list = copy(parent_list)
                        new_parents = sort!(push!(new_parent_list[i], j))
                        new_score, out_score = greedy_score(score_components, n, new_parent_list, datamat, params, ncategories)
                        if new_score > best
                            best = new_score
                            complete = false
                            best_parent_list = new_parent_list
                            greedy_parents = out_score
                        end
                    end
                end
            end

            # 2) did this improve our greedy score?
            # Add the best edge and continue
            parent_list = best_parent_list
        end
        if complete
            break
        end
    end

```

Figura A.3: Código implementado pela biblioteca *BayesNets* para aprendizagem de parâmetros com a *GraphSearchStrategy*: ScanGreedyHillClimbing -parte1 (fonte[SGHC]).

```
    # compute the greedy solution
    parent_list = greedy_parents

    # construct the BayesNet
    cpds = Array{DiscreteCPD}(undef, n)
    varnames = names(data)
    for i in 1:n
        name = varnames[i]
        parents = varnames[parent_list[i]]
        cpds[i] = fit(DiscreteCPD, data, name, parents, params.prior, parental_ncategories=ncategories[parent_list[i]], target_ncategories=ncategories[i])
    end
    BayesNet(cpds)
end
```

Figura A.4: Código implementado pela biblioteca *BayesNets* para aprendizagem de parâmetros com a *GraphSearchStrategy*: ScanGreedyHillClimbing -parte2 (fonte[SGHC]).

```
function greedy_score(score_components, n, prior_parent_list, datamat, params::ScanGreedyHillClimbing, ncategories::Vector{Int})
    parent_list = prior_parent_list
    while true

        # Compute score
        best_diff = 0.0
        best_parent_list = parent_list
        for i in 1:n

            # 1) add an edge (j->i)
            if length(parent_list[i]) < params.max_n_parents
                for j in deleteat!(collect(1:n), parent_list[i])
                    if adding_edge_preserves_acyclicity(parent_list, j, i)
                        new_parents = sort!(push!(copy(parent_list[i]), j))
                        #println("A")
                        new_component_score = bayesian_score_component(i, new_parents, ncategories, datamat, params.prior, params.cache)
                        #println("B")
                        if new_component_score - score_components[i] > best_diff
                            best_diff = new_component_score - score_components[i]
                            best_parent_list = deepcopy(parent_list)
                            best_parent_list[i] = new_parents
                        end
                    end
                end
            end

            # 2) remove an edge
            for (idx, j) in enumerate(parent_list[i])

                new_parents = deleteat!(copy(parent_list[i]), idx)
                new_component_score = bayesian_score_component(i, new_parents, ncategories, datamat, params.prior, params.cache)
                if new_component_score - score_components[i] > best_diff
                    best_diff = new_component_score - score_components[i]
                    best_parent_list = deepcopy(parent_list)
                    best_parent_list[i] = new_parents
                end


                # 3) flip an edge
                if length(parent_list[j]) < params.max_n_parents
                    deleteat!(parent_list[i], idx)
                    if adding_edge_preserves_acyclicity(parent_list, i, j)
                        sort!(push!(parent_list[j], i))
                        new_diff = bayesian_score_component(i, copy(parent_list[i]), ncategories, datamat, params.prior, params.cache) - score_components[i]
                        new_diff += bayesian_score_component(j, copy(parent_list[j]), ncategories, datamat, params.prior, params.cache) - score_components[j]
                        if new_diff > best_diff
                            best_diff = new_diff
                            best_parent_list = deepcopy(parent_list)
                        end
```

Figura A.5: Código implementado pela biblioteca *BayesNets* para aprendizagem de parâmetros com a *GraphSearchStrategy*: ScanGreedyHillClimbing, e função recursiva greedy$_s$core-parte1 (fonte[SGHC]).

```
                    deleteat!(parent_list[j], findall((in)(i), parent_list[j]))
                end
                sort!(push!(parent_list[i], j))
            end
        end
    end

    if best_diff > 0.0
        parent_list = best_parent_list
        score_components = bayesian_score_components(parent_list, ncategories, datamat, params.prior, params.cache)
    else
        break
    end
end
sum(score_components), parent_list
end
```

Figura A.6: Código implementado pela biblioteca *BayesNets* para aprendizagem de parâmetros com a *GraphSearchStrategy*: ScanGreedyHillClimbing, e função recursiva greedy$_s$core-parte2 (fonte[SGHC]).

A.3 *GraphSearchStrategy* K2GraphSearch

```
function Distributions.fit(::Type{BayesNet{C}}, data::DataFrame, params::K2GraphSearch) where {C<:CPD}

    N = length(params.order)
    cpds = Array{C}(undef, N)
    for i in 1 : N

        cpd_type = params.cpd_types[i]
        target = params.order[i]

        best_score = -Inf

        # find the best parent to add
        potential_parents = params.order[1:i-1]
        for nparents in 0:min(params.max_n_parents,i-1)
            for parents in subsets(potential_parents, nparents)
                cpd = fit(cpd_type, data, target, parents)
                score = score_component(params.metric, cpd, data)

                if score > best_score
                    best_score = score
                    cpds[i] = cpd
                end
            end
        end
    end

    BayesNet(cpds)
end
Distributions.fit(::Type{BayesNet}, data::DataFrame, params::K2GraphSearch) = fit(BayesNet{CPD}, data, params)
```

Figura A.7: Código implementado pela biblioteca *BayesNets* para aprendizagem de parâmetros com a *GraphSearchStrategy*: K2GraphSearch (fonte[K2]).