PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

PUCKER

Algoritmo de Apoio à decisão ao jogador de Pôquer No Limit Texas Hold'em

Alexandre Marangoni Costa

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO – CTC
DEPARTAMENTO DE INFORMÁTICA
Curso de Graduação em Engenharia da Computação

Rio de Janeiro, novembro de 2013

Alexandre Marangoni Costa

Pucker

Algoritmo de Apoio à decisão ao jogador de Pôquer No Limit Texas Hold'em

Relatório de Projeto Final, apresentado ao programa de graduação da PUC-Rio como requisito parcial para a obtenção do titulo de Engenheiro de Computação.

Orientador: Marcus Poggi

Rio de Janeiro

Novembro de 2013

“Difícil é tudo aquilo que você ainda não sabe”

(Profª Geiza Hamazaki)

Agradecimentos

Aos meus pais, por terem incentivado e possibilitado a minha inserção no curso de engenharia, em uma cidade distante de onde eu morava, longe deles e de muito do que eu conhecia até então.

Aos meus irmãos, por terem iluminado minha cabeça para as diversas portas que uma universidade pode abrir, e por terem me apoiado em todos os caminhos que eu tomei.

Ao professor Marcus Poggi, pelo tempo, conhecimento e orientação, fundamentais para conclusão desse projeto. Pela amizade e pelos conselhos em assuntos diversos, importantes para participação de um intercâmbio, de um projeto orientado, de uma olimpíada de programação, e inserção no mercado de trabalho.

Aos meus amigos, de Teresina e do Rio de Janeiro, por estarem sempre ao meu lado, mesmo quando eu estou distante.

Resumo

Marangoni Costa, Alexandre. Poggi, Marcus. Pucker: Algoritmo de Apoio à Decisão ao Jogador de Pôquer No Limit Texas Hold'em. Rio de Janeiro, 2013. Número de páginas: 30. Relatório de Projeto Final I – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

O presente trabalho tem como objetivo demonstrar um algoritmo para suportar a decisão do jogador de pôquer, de acordo com o cenário presente no momento da decisão. O método utilizado é conhecido como Rede Bayesiana. Para tanto, foi desenvolvido um arcabouço de funcionalidades, desde artefatos para simulação do jogo, e incorporação de diferentes algoritmos, à classes de contabilização de estatísticas de eficácia dos algoritmos de decisão. Junto ao algoritmo principal, foram implementados outros algoritmos de decisão usados para demonstrar progresso do algoritmo principal.

Palavras-chave
pôquer, rede bayesiana, sistema de apoio à decisão, arvore de decisão

Abstract

Marangoni Costa, Alexandre. Poggi, Marcus. Pucker: Decision Support Algorithm to the Poker No Limit Texas Hold'em Player. Rio de Janeiro, 2013. Número de páginas: 30. Relatório de Projeto Final I – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

The project's goal is to demonstrate an algorithm to support a No Limit Texas Hold'em player to take a decision, given a particular scenario of the game. The method consists of a Bayesian Network. Additionally, it was developed a framework of functionalities such as the game simulation and classes to give statistical calculations of the result. Other decision algorithms were developed, to help evaluate the main algorithms progress.

Keywords
poker, bayesian network, decision-support algorithms, decision tree

Sumário

1 Introdução

1.1 Considerações Iniciais

A todo momento que a palavra “pôquer” for mencionada no presente documento, deve-se compreender que esta referencia a modalidade “No Limit Texas Hold'em”.

Quando a palavra “numeração” de carta de baralho for usada, esta refere-se também aos símbolos Ás, Rei, Dama e Valete.

Apesar da utilização da voz passiva em alguns trechos deste relatório, o projeto foi integralmente desenvolvido pelo aluno.

1.2 Domínio do Problema

Este relatório descreve o framework Pucker: um arcabouço de funcionalidades voltadas à simulação, geração de relatórios, e teste de partidas de pôquer. Adicionalmente, o presente trabalho sugere um método, e algoritmos de apoio à decisão, integrados ao Pucker, capazes de jogar de forma consistente, e obter bons resultados em longo prazo.

1.3 Histórico

Este projeto nasceu como uma proposta de projeto orientado, após o aluno ter cursado a disciplina de análise de algoritmos.

Durante o projeto orientado, o aluno delimitou os domínios do problema, pesquisou possíveis métodos de apoio à decisão que pudessem fazer parte dos resultados, e desenvolveu uma modelagem inicial do projeto, um protótipo em C++.

Após essa fase, foi decidido pela utilização de uma rede bayesiana para responder às incertezas do cenário de pôquer. Durante a etapa de projeto final I, a arquitetura do projeto também sofreu uma alteração. O código escrito em C++ foi portado para Ruby, e a compilação passou a usar uma ferramenta chamada JRuby.

Esta arquitetura permitiu usar a concisa linguagem Ruby para modelar a simulação de um jogo de pôquer, escrever scripts de teste, e reaproveitar o legado de bibliotecas abertas, em Java, que pudessem agilizar o desenvolvimento de partes do projeto.

Durante o projeto final II, foi desenvolvido a simulação completa do jogo de pôquer, em forma de um framework extensível. Isso permitiu a criação de diferentes perfis de jogadores (algoritmos), possibilitando também embates

destes entre si, e a geração de logs e estatísticas de jogadas para cada jogador.

1.4 Pôquer

A modalidade No Limit Texas Hold'em é bem simples. Todos os jogadores recebem 2 cartas. A cada rodada de aposta, cartas são viradas na mesa, e cada jogador usa sua mão mais as cartas da mesa para formar um jogo de 5 cartas.

A primeira rodada de apostas chama-se FLOP e são viradas 3 cartas na mesa. Após analisarem suas próprias mãos, e a mesa, os jogadores podem fazer as apostas. Em seguida, inicia-se a segunda rodada, que chama-se TURN. Nesta, somente uma carta é virada. Os jogadores fazem suas apostas e então mais uma carta é virada, o RIVER. Uma última rodada de apostas acontece, e então os jogadores ativos devem mostrar suas mãos afim de verificar-se quem ganhou.

Em caso de empate, o pote de apostas deve ser distribuído igualmente entre os vencedores.

Ao todo (FLOP+TURN+RIVER), o jogador dispõe de 7 cartas (2 em sua mão, 3 do FLOP, 1 do TURN e 1 do RIVER), mas só pode formar um jogo usando 5. Existem 10 possíveis jogos. A força de um jogo cresce com a numeração.

Os possíveis jogos, em ordem de força, são:

1. Carta alta: vence o jogador com a carta mais alta
2. Par: duas cartas de mesmo número, e outras 3 cartas sem valor
3. Dois pares: dois conjuntos de duas cartas de mesmo número, e outra carta
4. Trinca: três cartas de mesmo número, e outras duas
5. Sequência: cinco cartas de numeração consecutiva.
6. Flush: cinco cartas do mesmo naipe
7. Full House: uma trinca e um par
8. Quadra: quatro cartas de mesma numeração, e um outra
9. Sequência de mesmo naipe
10. Sequência real de mesmo naipe: sequência do 10 ao Ás, do mesmo naipe

Um dado interessante é que 70% das mãos jogadas em “online poker” terminam antes do RIVER, e portanto, o jogador vencedor nem precisa mostrar

sua mão. Ele leva todas as fichas do pote, por ter apostado uma quantia que nenhum outro jogador “paga pra ver”. Isso caracteriza a estratégia agressiva de apostas que o pôquer permite. Quando isso acontece, as fichas do pote são dadas ao vencedor, e uma nova rodada pode recomeçar com a redistribuição de cartas.

Diante disso, pode-se dizer que o pôquer é um jogo de muitas incertezas. O jogador não sabe a próxima carta a ser virada na mesa, o que pode mudar tudo. E, principalmente, ele não sabe as cartas dos outros jogadores. No máximo pode fazer inferências a partir das apostas desses outros jogadores.

Algumas questões que o jogador de pôquer deve considerar num cenário de decisão são:

1. Seu jogo (mão + mesa) é forte?
2. Os outros jogadores estão fortes?
3. Eu posso ganhar essa rodada, baseado na possível força dos outros jogadores, comparado com a força do meu jogo?
4. Eu posso ganhar essa rodada apostando forte para todos os outros jogadores desistirem?
5. Se a mesa está forte, isso me favorece, mas também favorece os outros jogadores.

1.5 Redes Bayesianas

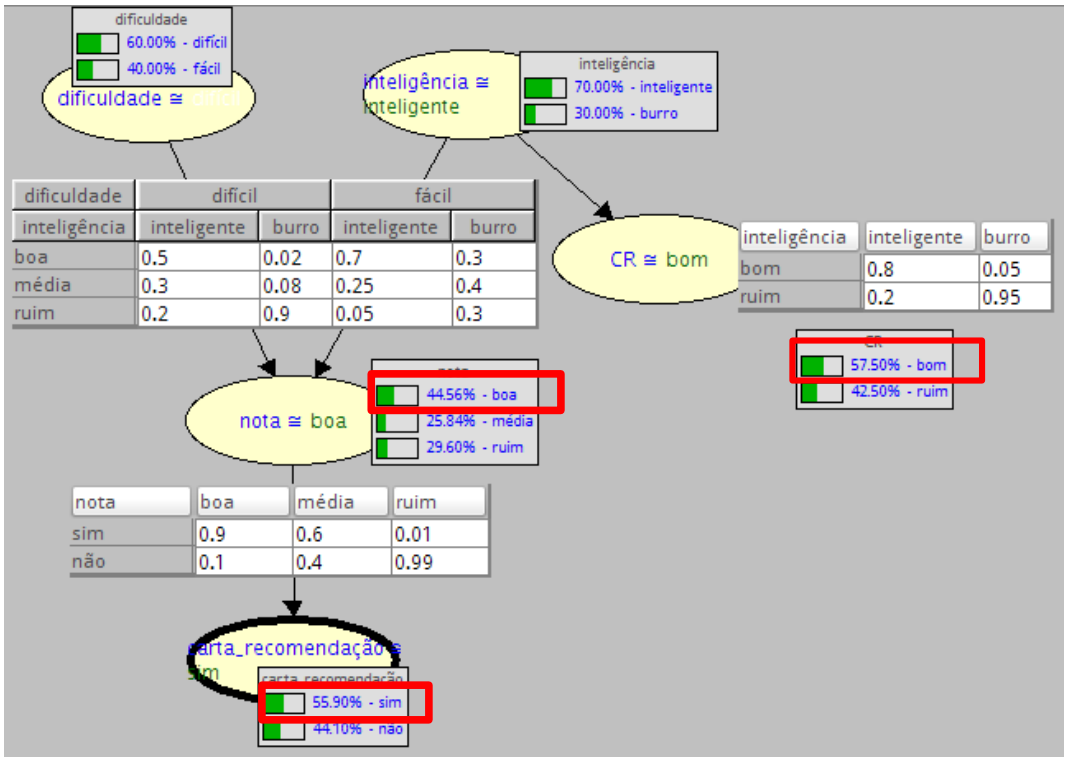
O objetivo deste trabalho é a construção de um algoritmo capaz de jogar pôquer, e obter bons resultados no longo prazo. O pôquer é um jogo de muitas incertezas. Em nenhum momento o jogador tem conhecimento de 100% do cenário. Isso torna a rede bayesiana um método conveniente para modelar o conhecimento de um cenário de pôquer.

Uma rede bayesiana pode ser pensada como um grafo de variáveis aleatórias, cada uma com sua distribuição de probabilidades. Um nó pode influenciar a distribuição de outro nó, criando uma distribuição condicional de probabilidade para o nó influenciado. Sua probabilidade varia de acordo com a variável que a influencia.

Para ilustrar, a rede bayesiana abaixo foi modelada para ajudar um professor a decidir se dá ou não uma carta de recomendação para um aluno. Essa decisão leva em conta cinco variáveis aleatórias: dificuldade da disciplina que ele ministrou ao aluno, inteligência do aluno, nota do aluno na sua disciplina, coeficiente de rendimento do aluno (CR), e a decisão de dar ou não

a carta.

É trivial pensar que a dificuldade do curso, e a inteligência do aluno influenciam diretamente na nota que este aluno recebe. A inteligência do aluno também influencia no seu CR. Porém o professor não quer sujeitar sua decisão ao CR do aluno, dado que conhece bons alunos com CR baixo. A nota do aluno em sua disciplina parece fazer mais sentido nessa decisão.



dificuldade	difícil		fácil	
inteligência	inteligente	burro	inteligente	burro
boa	0.5	0.02	0.7	0.3
média	0.3	0.08	0.25	0.4
ruim	0.2	0.9	0.05	0.3

inteligência	inteligente	burro
bom	0.8	0.05
ruim	0.2	0.95

nota	boa	média	ruim
sim	0.9	0.6	0.01
não	0.1	0.4	0.99

Afim de aprofundar o conhecimento sobre o cálculo feito por uma rede bayesiana, vamos analisar três inferências feitas por essa rede.

1. Porque esta rede indica que um aluno tem 57.5% de chance de ter um bom CR?

Um aluno tem 0.7 chance de ser inteligente: $I^1 = 0.7$

Um aluno tem 0.8 chance de ter bom CR, se for inteligente: $CR(I^1) = 0.8$

$$0.7 * 0.8 = 0.56$$

Um aluno tem 0.3 chance de ser burro: $I^2 = 0.3$

Um aluno tem 0.05 chance de ter bom CR, se for burro: $CR(I^2) = 0.05$

$$0.3 * 0.05 = 0.015$$

E para concluir:

$$0.56 + 0.015 = 0.575$$

2. Porque esta rede indica que um aluno tem 44.56% de chance de tirar uma boa nota?

   Pelo mesmo raciocínio:

$$D^1 * I^1 * N^1(D^1, I^1) +$$
$$D^1 * I^2 * N^1(D^1, I^2) +$$
$$D^2 * I^1 * N^1(D^2, I^1) +$$
$$D^2 * I^2 * N^1(D^2, I^2) =$$

$$0.6 * 0.7 * 0.5 +$$
$$0.6 * 0.3 * 0.02 +$$
$$0.4 * 0.7 * 0.7 +$$
$$0.4 * 0.3 * 0.3 =$$
$$0.4456$$

3. Porque esta rede indica que um aluno tem 55.9% de chance de receber uma carta de recomendação desse professor?

   Pelo mesmo raciocínio:

$$0.4456 * 0.9 + 0.2584 * 0.6 + 0.296 * 0.01 = 0.55904$$

1.6 Motivações

Pesquisadores do ramo da inteligência artificial sempre tiveram interesse no desenvolvimento de algoritmos que pudessem competir com o homem. Por exemplo, no xadrez, já existem algoritmos capazes de vencer experientes jogadores.

O pôquer apresenta um desafio particular devido à falta de informações referentes ao cenário do jogo em todos os momentos de decisão. Por exemplo, as 5 cartas viradas na mesa (flop, turn e river) podem mudar completamente o valor das cartas do jogador. Além disso, as desconhecidas cartas portadas

pelos outros jogadores influenciam no resultado final da jogada.

A relevância do problema abordado parte do pressuposto de que não existe resposta correta para a saída do algoritmo de decisão. Diante disso, a avaliação dos algoritmos criados é feita a partir da comparação, em longo prazo, do desempenho destes sobre um algoritmo que responde aleatoriamente. Além disso, vale ressaltar que o algoritmo final foi aperfeiçoado a partir de várias versões mais simples, em um processo gradual, a medida que o projeto foi progredindo.

Estas versões mais simples, assim como o algoritmo aleatório, serviram para avaliar o desempenho do algoritmo final, buscando sempre um algoritmo que maximizasse seu número de vitórias em rodadas de pôquer, em relação aos algoritmos mais simples.

Esse projeto é interessante como projeto final pela sua originalidade e complexidade. Além disso, ele engloba conhecimentos de diversos conteúdos ministrados durante o curso de Engenharia. Entre esses, vale a pena destacar as matérias de Análise de Algoritmos, por todo o conhecimento referente a complexidade de algoritmos, e raciocínio lógico para implementação de estruturas de dado complexas, e para a utilização do método de rede bayesiana, Programação Orientada a Objetos e Programação Modular, para compreensão da importância de testes e modularização em sistemas complexos, e para garantia da qualidade do código implementado.

1.7 Ambiente Computacional

O sistema operacional usado para desenvolvimento é a distribuição linux Elementary OS 3.2, um fork do Ubuntu 12.04, entretanto isso não restringe o sistema operacional cliente, já que a linguagem de programação usada é multiplataforma.

O ambiente de desenvolvimento é bem diverso. Para a base do sistema, como classes básicas de baralho (table.Deck e table.Card), mão (table.Hand) e de avaliação dos vencedores (table.HandEvaluator), a linguagem Java foi a escolhida, devido à quantidade de código passível de ser reutilizado, além da portabilidade característica dessa linguagem. Para scripts de testes, de geração de relatórios e de controle da execução do jogo é usada a linguagem JRuby, que nada mais é do que a implementação da linguagem Ruby para a máquina virtual Java. Assim todos esses scripts tem fácil acesso ao código Java.

Além destes, todo o desenvolvimento do código foi feito usando a metodologia Test-Driven Development, para garantir a qualidade e documentação do código escrito. Isso quer dizer que antes de escrever uma dada funcionalidade, como um jogador, primeiro escreviam-se testes que verificavam se o comportamento do jogador era o esperado. Estes testes falhavam, já que o jogador ainda não havia sido escrito. Em seguida, escrevia-se um mínimo de código responsável por passar o teste escrito. Com o teste passando, o código do jogador desejado era refatorado.

Isso garantiu a qualidade do software desenvolvido, mesmo antes dele estar terminado. Antes do sistema ser capaz de simular jogadas completas de pôquer, vários fragmentos do código-fonte já estavam testados. Assim, foi possível criar e testar jogadores, mesmo antes destes poderem jogar entre si. Isso possibilitou o aluno focar no algoritmo de decisão, e no estudo de uma rede bayesiana eficaz para o problema, ao invés de focar somente na simulação do jogo.

Como consequência do ambiente e metodologias usados, o sistema é capaz de gerar uma documentação automática do código-fonte, toda vez que os testes são rodados. Isso é possível pois cada teste descreve a funcionalidade que está sendo testada, em linguagem humana. Essa descrição é usada para gerar a documentação do sistema, presente no anexo, e nos arquivos documentation.html, documentation.markdown e documentation.txt.

2 Estado da Arte

O pôquer tem sido largamente estudado no meio acadêmico-científico desde 1950. Dentre suas variantes, até o presente momento, somente uma foi resolvida de forma ótima [1], e o No Limit Texas Hold'em é um dos mais complexos, devido à configuração de suas variáveis aleatórias.

Em 2003, Billings et al. apresentou uma solução próxima de ser ótima para a variação Heads-Up Limit Texas Hold'em, usando Teoria dos Jogos.

Os trabalhos mais avançados referentes a Texas Hold'em são aproximações de equilíbrio de Nash em jogos na forma extensiva [2][3].

Em 2007 e 2008, o programa de computador POLARIS, desenvolvido na Universidade de Alberta nos Estados Unidos, competiu contra humanos profissionais de pôquer. Na primeira vez, o programa perdeu por pouco para dois jogadores humanos, enquanto na segunda vez derrotou um grupo de experientes jogadores. A Universidade de Alberta atualmente possui o mais

bem sucedido grupo de estudos de pôquer, que contam com publicações na área desde 1995.

Uma grande parte dos algoritmos escritos para este domínio baseia-se em algoritmos não determinísticos de inteligência artificial, como os algoritmos genéticos. Por essa razão, o presente trabalho abordará o problema a partir de um algoritmo considerado determinístico. Este será baseado em uma rede bayesiana, um modelo que se mostrou aceitável para a representação do conhecimento incerto e incompleto presente numa jogada de pôquer, por meio da Teoria da Probabilidade Bayesiana, publicada pelo matemático Thomas Bayes, em 1763.

A rede bayesiana é um grafo direcionado acíclico, uma árvore, no qual os nós representam as variáveis do problema, no caso uma jogada de pôquer, e as arestas representam a dependência probabilística entre essas variáveis. No caso do pôquer isso é muito conveniente, pois essas variáveis são capazes de representar a incerteza, a partir da representação do custo das arestas como probabilidades presentes numa jogada de pôquer.[4]

Diante disso, é fácil perceber que o ajuste no peso das arestas implica diretamente na alteração do comportamento do algoritmo. Isso resulta na flexibilidade direta que esse método oferece ao sistema, pois durante projeto final II, verificou-se a possibilidade de testar abordagens dinâmicas ou estáticas para implementação do conhecimento probabilístico do algoritmo de decisão.

3 Objetivos

O trabalho consiste principalmente de algoritmos de decisão. Estes recebem informações de um cenário de pôquer, e, baseado nessas entradas, produz em sua saída uma decisão (fold, bet, all-in).

De acordo com o planejamento feito durante o projeto final I, o primeiro objetivo a ser alcançado era um algoritmo que no longo prazo vencesse um algoritmo aleatório. Isso garantiu que, assumindo que pôquer não é um jogo de azar (como a roleta), a rede bayesiana permite uma modelagem satisfatória da solução. Esse objetivo foi conquistado com sucesso, ainda no início do projeto.

O segundo objetivo a ser alcançado era um algoritmo capaz de ganhar de um algoritmo determinístico existente, de qualidade previamente testada. Infelizmente não foi achado nenhuma fonte que pudesse fornecer um algoritmo determinístico satisfatório. Além disso, foi constatado que o desenvolvimento

de um algoritmo desse porte consumiria bastante recursos do projeto, que já estava extenso. Portanto, esse objetivo foi substituído pelo desenvolvimento de múltiplas redes bayesianas, e o teste exaustivo dessas redes entre si. Esses testes produziram conhecimento suficiente para o desenvolvimento de mais redes, e foi dessa forma que o projeto ganhou progresso.

O sistema é inteiramente novo, entretanto, partes do código foram reutilizadas a partir de bibliotecas livres de licenças proprietárias. Vale ressaltar que este código reutilizado não faz parte do objetivo principal do projeto (apoio à decisão). Como exemplo, pode-se destacar o código que avalia as cartas e determina o(s) vencedor(es). Este é um algoritmo relativamente complicado, cujo domínio de estudo não faz parte do escopo deste projeto.

O sistema pode ser usado por jogadores reais de pôquer, tanto para estudo quanto para treino. Além disso, esse algoritmo pode ser de grande ajuda na montagem de estratégias para situações reais de jogo, como um campeonato, uma mesa repleta de jogadores conservadores, ou o inverso, uma mesa repleta de jogadores agressivos.

O sistema é capaz de simular, de forma independente, todo o desenrolar de jogadas de pôquer. Dessa forma, o sistema é flexível o suficiente para incentivar a continuação deste trabalho por desenvolvedores interessados. Estes encontrarão facilidade em criar outros algoritmos, e submete-los a rodadas de pôquer contra os algoritmos já desenvolvidos no presente trabalho. Basta criar uma subclasse, e reescrever um método.

Além disso, desenvolvedores interessados também encontrarão facilidade em modificar o código desenvolvido, pois este é acompanhado de um sólido conjunto de testes unitários, testes funcionais e testes de integração, que garantem a qualidade do sistema, e fornecem segurança ao desenvolvedor que precisa fazer grandes alterações. Qualquer bug inserido no sistema terá grande chances de ser detectado pelos testes, que cobrem 90% da arquitetura.

4 Atividades Realizadas

Esse projeto foi desenvolvido ao longo de três períodos. O primeiro durante uma disciplina de projeto orientado, e os dois períodos que compõem o projeto final.

Abaixo uma lista completa das atividades realizadas:

- Projeto Orientado
  - Modelagem inicial da simulação de um jogo de pôquer em C++
  - Estudos iniciais sobre redes bayesianas
- Projeto Final I
  - Pesquisa sobre estudo da arte acerca de pôquer e acerca de software aberto Java, para agilizar o desenvolvimento de pontos complexos da simulação
  - Integração de bibliotecas open-source em Java, à scripts de teste em Ruby via ferramenta conhecida como JRuby
  - Inicio da implementação do sistema, via scripts de teste

O projeto final, realizado no segundo período de 2013, e objeto deste relatório, foi o que concentrou o maior número de atividades realizadas. No anexo I, pode-se conferir um log completo das atividades realizadas diretamente no código-fonte.

De forma mais geral, pode-se listar:

- Um framework, chamado Pucker, composto de classes de nome autoexplicativo: Card, Deck, Hand, HandEvaluator, Dealer, PlayerGroup, Game, Player
- Diferentes algoritmos para demonstrar o progresso do algoritmo criado: DummyPlayer, SimpleBnPlayer, BnPlayer
- Um algoritmo final com a melhor estratégia encontrada: SmartBnPlayer
- Um sólido conjunto de testes unitários, funcionais e de integração (coverage: 90.76%)
- Documentação do código-fonte (anexo II), gerada automaticamente pelos testes
- Classes para contabilidade de estatísticas em tempo real, e log

4.1 Estudos Preliminares

Antes do projeto ter início, o aluno já havia cursado disciplinas de inteligência artificial, pesquisa operacional, análise de algoritmos e programação linear, tanto na universidade PUC-Rio quanto durante intercâmbio na França, onde estudou na SUPMECA. O aluno estava confortável em programar diferentes metodologias de apoio à decisão.

Além disso o aluno usa cotidianamente a linguagem Ruby, e aplica a

metodologia Test-Driven Development em projetos pessoais e profissionais, usando a linguagem Ruby.

4.2 Estudos Conceituais e de Tecnologia

O contato com as “Bayesian Networks” se iniciou durante o projeto orientado, à partir de publicações escritas por alguns dos criadores dessa ferramenta: Judea Pearl [5] e Max Henrion [6]. E continuou a partir de aulas ministradas online pela professora Daphne Koller [7].

Em pouco tempo o aluno já estava familiarizado com o software SAMIAM[8] usado para projetar as redes bayesianas usadas no projeto (e para desenhar as redes bayesianas nas imagens deste documento). Essas redes podiam ser testadas em Ruby usando a biblioteca open-source SBN[9].

O aluno já trabalhava com Ruby, porém nunca havia usado o JRuby para colocar código Ruby em contato com bibliotecas Java. Esse passo foi crucial para o estabelecimento da arquitetura do sistema Pucker, pois permitiu integrar redes bayesianas implementadas usando a biblioteca SBN[9], com o código-fonte do projeto em Ruby, e com as classes reutilizadas em Java.

O primeiro teste criado em Ruby usava um código desenvolvido por uma equipe na Universidade de Alberta[10], para comparar duas mãos de pôquer, respondendo qual a mais forte. Esse código foi usado até o fim do projeto para avaliar e premiar os jogadores ao fim de cada jogada.

Essa integração Java-Ruby foi feita a partir de scripts de teste usando a ferramenta de teste Rspec, e esse é o coração do projeto, pois foi por onde ele se iniciou, como pode-se conferir no primeiro commit do Anexo I, há 1 ano atrás. Após os scripts validarem a possibilidade dessa arquitetura, o projeto progrediu a partir da criação de mais scripts, que definiram o modelo conceitual do projeto, e precederam a criação das funcionalidades do jogo.

4.3 Testes e Protótipos para Aprendizado e Demonstração

O protótipo completo resultado deste trabalho chama-se Pucker, e encontra-se disponível no repositório[11]:

https://github.com/alexandremcosta/pucker/

Dentro do diretório spec/ encontram-se 63 casos de teste que navegam por aproximadamente 90% de todo o protótipo.

Este protótipo é capaz de executar e gerar estatistísticas de jogadas de pôquer. Ele facilita a criação de novos perfis de jogadores, caso essa pesquisa

tenha continuidade, e fornece ferramentas para analisar a taxa de sucesso dos algoritmos criados.

Junto ao protótipo responsável pela simulação e teste, foram criados 5 tipos diferentes de algoritmos de decisão, presentes no diretório lib/, sendo estes:

- Um algoritmo determinístico que sempre aposta o mínimo possível (Player)
- Um algoritmo aleatório, que aposta randomicamente (DummyPlayer)
- Um algoritmo bayesiano simples, que leva em conta somente a força de sua mão, e é a superclasse que hospeda o método criado
- Um algoritmo bayesiano mediano que leva em conta tanto a sua mão como a possibilidade dos outros jogadores também estarem fortes
- Um algoritmo bayesiano avançado, que no longo prazo vence todos os outros, pois leva em conta a posição do jogador para determinar a validade das incertezas relacionadas ao cenário

4.4 Método

O método para desenvolvimento desse projeto foi baseado na metodologia Test-Driven Development, que, de acordo com a Wikipedia, pode ser descrito como:

> *"Desenvolvimento orientado a testes é uma técnica de desenvolvimento de software que baseia em um ciclo curto de repetições: Primeiramente o desenvolvedor escreve um caso de teste automatizado que define uma melhoria desejada ou uma nova funcionalidade. Então, é produzido código que possa ser validado pelo teste para posteriormente o código ser refatorado para um código sob padrões aceitáveis."*

Numa tentativa de seguir esse padrão para a criação das diferentes redes bayesianas ao longo do projeto, e garantir a estratégia desejada a partir de testes automatizados, o desenvolvimento foi orientado pelo seguinte método:

1. Projetar uma rede bayesiana usando o software SAMIAM
2. Criar uma rede bayesiana em Ruby, seguindo o modelo em SAMIAM, no arquivo: lib/pucker/bayesian_networks.rb
3. Criar um conjunto de testes no arquivo: spec/pucker/bn_players_spec.rb
4. Criar um jogador que use a rede bayesiana, e valide os casos de teste

implementados: lib/pucker/bn_players.rb

5. Refatorar código gerado, e melhorar casos de teste

4.5 Cronograma

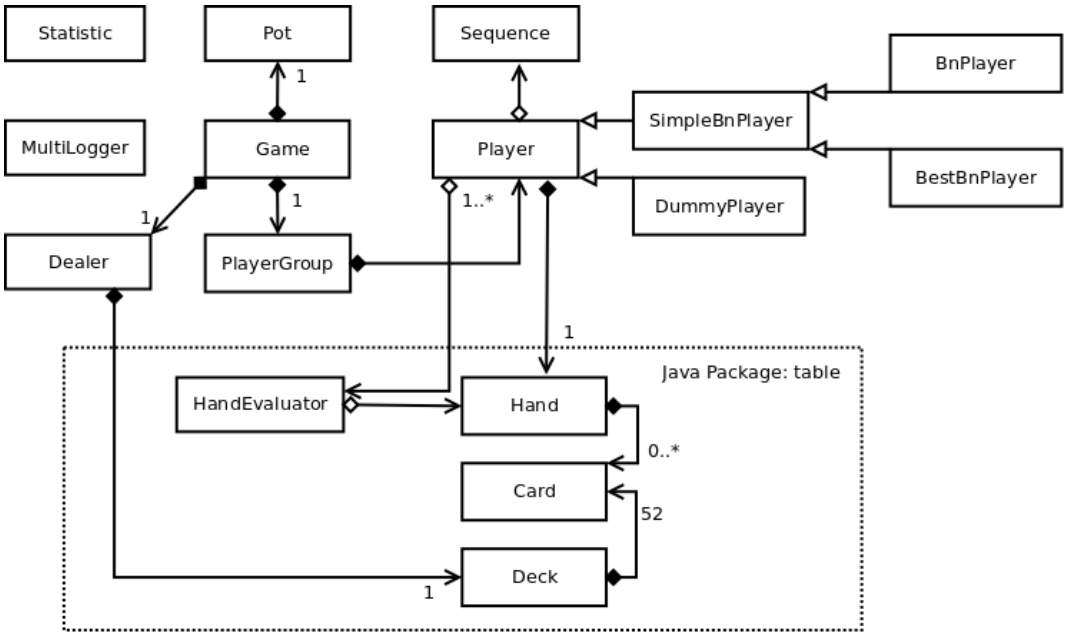
O cronograma planejado durante o projeto final I foi fielmente seguido. Abaixo, demonstro uma versão mais detalhada do que a que foi apresentada no relatório de projeto final I.

- Projeto Final I
  - SET/OUT/NOV 2012 – Estudo e definição das plataformas e técnicas a serem utilizadas
  - OUT 2012 – Início da produção do relatório
  - NOV/DEZ 2012 – Modelagem do projeto (UML), e início do desenvolvimento
  - NOV/DEZ 2012 – Atualização do relatório

- Projeto Final II
  - AGO/SET/OUT 2013 – Desenvolvimento da simulação do jogo
  - OUT/NOV 2013 – Implementação e desenvolvimento das redes bayesianas
  - NOV 2013 – Finalização e acabamento do relatório

5 Projeto e Especificação do Sistema

O projeto foi modelado de acordo com o seguinte diagrama:



Esse diagrama simplifica o entendimento da arquitetura do sistema. Os principais detalhes dessa arquitetura são:

- A classe Game é o ponto central da arquitetura. Ela faz a integração de todos os módulos independentes do sistema, sendo o ponto de partida de uma simulação de um jogo
- A classe Game não sabe nada em relação aos jogadores. Ela se comunica com eles via PlayerGroup, que é um container de Players. Isso permite implementar jogadores independentes da implementação do jogo. A principal consequência disso é que os jogadores criados podem ser facilmente adaptados para qualquer modalidade de pôquer
- A classe Player encapsula o funcionamento básico do jogador, permitindo fácil criação de diferentes perfis de jogador via herança de comportamento, ou seja, criando uma subclasse
- A comunicação entre o projeto desenvolvido Pucker, e o código reutilizado em Java é feito por intermédio das classes Dealer e Player

5.1 Simplificações

Com o intuito de agilizar o processo de desenvolvimento, e focar nas partes críticas do projeto, foi adotada uma simplificação ao jogo de pôquer: antes da fase FLOP, nenhum jogador realiza qualquer tipo de apostas. Ou seja, as tomadas de decisão só acontecem a partir do FLOP, quando já existem três cartas na mesa.

5.2 Preparando o ambiente

Para usar o Pucker deve-se inicialmente preparar o ambiente, conforme as instruções abaixo:

- Instalar o ambiente JRuby
  - O Java Development Kit deve estar instalado
  - Recomenda-se usar a versão 1.7.4, usada em desenvolvimento [12]
  - Recomenda-se usar o software RVM para instalação do JRuby [13]

```
$ \curl -L https://get.rvm.io | bash -s stable –ruby=jruby-1.7.4
```

- Instalar o Bundler [14]

```
$ gem install bundler
```

- Instalar os pacotes usados no projeto

```
$ bundle install
```

5.3 Rodando os testes

Os testes foram escritos usando uma ferramenta de testes automatizados chamada Rspec. Esta ferramente permite a geração de uma documentação do sistema. Um exemplo de documentação gerada pode ser conferido no anexo II.

1. Para rodar, execute o seguinte comando na pasta raiz do projeto:

```
$ rspec
```

2. Para gerar documentação:
   - Diretamente no terminal, ou standard output, STDOUT

```
$ rspec --format doc
```

   - Em TXT

```
$ rspec --format doc > documentation.txt
```

   - Em HTML

```
$ rspec --format html > documentation.html
```

   - Em Markdown

```
$ rspec --require spec/markdown_formatter.rb --format MarkdownFormatter > documentation.markdown
```

5.4 Simulando jogadas

Para simular, por exemplo, 100 jogadas, usando um jogador do tipo BestBnPlayer contra outros quatro jogadores do tipo SimpleBnPlayer, deve-se navegar até a pasta *lib/* do projeto, e executar o "interactive ruby shell" afim de usar as classes do projeto:

```
pucker/      $ cd lib/
pucker/lib/ $ irb
> require 'pucker'
> game = Pucker::Game.new
> 100.times { game.play }
```

Um detalhado log, contendo informações em tempo real sobre o desenrolar das jogadas, será impresso no terminal, bem como registrado no arquivo pucker.log na pasta raiz do projeto.

5.5 Classes para simulação de rodadas de pôquer

5.5.1 Game

É o coração do Pucker. O método mais importante que os objetos da classe Game executam é o #play. Esse método executa uma jogada completa de pôquer, chamando os seguintes métodos auxiliares:

- #setup_game: configura a posição dos jogadores, fichas e baralho
- #collect_blinds: coleta apostas pré-flop
- #deal_flop, #deal_turn, #deal_river: nomes auto-explicativos
- #reward: distribui as apostas aos vencedores

5.5.2 PlayerGroup

Esta classe encapsula um container de jogadores. Ela é a interface entre o jogo e os jogadores, e permite que se mude a modalidade de pôquer jogada, com um mínimo de adaptação aos jogadores.

Ela faz o uso de um design pattern conhecido como Delegator [15], de forma a permitir o uso de uma sintaxe conveniente para se acessar e iterar entre os jogadores de um dado Game.

Como por exemplo, para se iterar imprimindo o ID e as fichas de todos os jogadores:

```
> players = Pucker::PlayerGroup.new
> players.each { |p| print "#{p.id} #{p.stack}" }
```

Se o desejado for fazer isso somente entre os jogadores ativos (que não desistiram):

```
> players.eligible.each { |p| print “#{p.id} #{p.stack}” }
```

Outros exemplos adicionais:

```
> players.first
> players.last
> players[2] # acessa o terceiro jogador
> players.size # retorna o número total de jogadores
> players.rotate # realoca os jogadores, mudando o SMALL_BLIND
```

5.5.1 Player

Agrupa todas as mensagens que um Player deve ser capaz de responder. Fazendo isso, essa classe simplifica a criação de novos jogadores com diferentes estratégias.

Por exemplo, um jogador deve ser capaz de:

```
> player = Pucker::Player.new
> player.set_hand(Card.new(1), Card.new(2))
> player.bet(min_bet: 0)
> player.reward(100)
> player.hand_rank(table_cards_array)
```

Na última linha acima, vemos que um jogador deve ser capaz de calcular a força da sua mão, com base num Array de cartas presentes na mesa. Para isso ele faz uso de uma classe Java chamada HandEvaluator, escrita e distribuída livremente pela Universidade de Alberta.

5.6 Código Java reutilizado (Universidade de Alberta)

Para simplificação do processo de desenvolvimento, o projeto reutilizou classes implementadas em Java, por uma equipe da Universidade de Alberta que há anos contribuem para o estado da arte relacionada a métodos de apóio a decisão em pôquer.

A licença do código reusado está presente junto ao mesmo, no diretório lib/pucker/table.

Por motivos de organização do sistema, todas as classes Java foram agrupadas em um package chamado “table”. Isso permite a comunicação entre código Ruby e Java em duas linhas, via JRuby:

```
require 'java'
java_import 'table.*'
```

A seguir, uma lista de todas as classes Java do projeto:

- Card: representa uma carta, com simbolo/número, e naipe
- Deck: baralho convencional de 52 cartas, permite shuffle e reset
- Hand: conjunto de cartas representando uma mão
- HandEvaluator: é capaz de calcular o ranking de uma mão. Quanto maior esse ranking, mais forte é a mão, de acordo com as regras do pôquer

5.7 Classes para recolhimento de estatísticas e log de jogadas

5.7.1 MultiLogger

Ao longo de uma simulação de jogada, diversos dados são produzidos. Esses dados são mostrados na tela graças a uma classe chamada MultiLogger.

Essa classe encapsula o funcionamento do Logger, uma classe nativa, presente na biblioteca standard do Ruby, adicionando a capacidade de impressão dos logs em múltiplos outputs.

Dessa forma, durante a execução dos testes automatizados, o sistema não imprime o log na tela, mas sim num arquivo separado, na raíz do projeto, chamado test.log.

Em contrapartida, quando a simulação é executada no “interactive ruby shell”, o log é registrado tanto na tela (STDOUT), quanto no arquivo pucker.log, na raíz do projeto.

5.7.2 Statistic

Essa classe é responsável pelas estatísticas geradas em tempo real, sobre o desempenho dos algoritmos criados.

5.8 Redes Bayesianas Criadas

As redes bayesianas criadas durante o desenvolvimento do projeto encontram-se no arquivo: lib/pucker/bayesian_networks.rb.

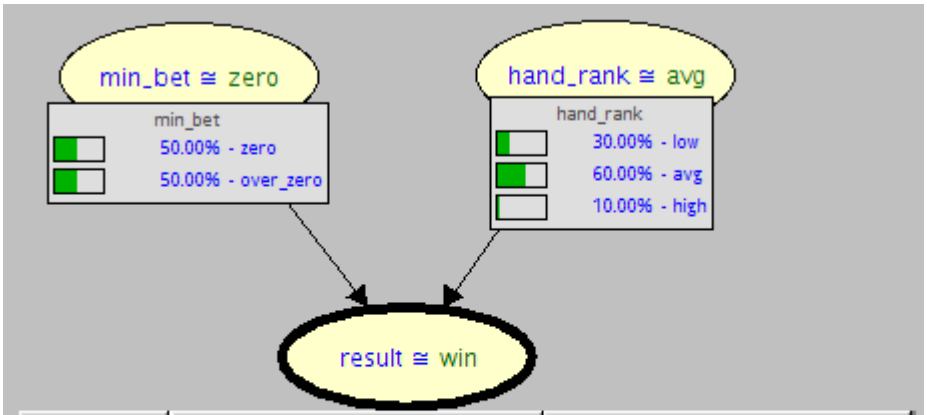
A implementação das redes foi feita usando a biblioteca open-source SBN[9], e esta implementação será demonstrada abaixo.

Ao todo, foram criadas mais de 10 redes bayesianas diferentes. Porém somente três foram importantes suficientemente para serem incluídas na versão final do Pucker: SimpleBn, GoodBn, e BestBn.

5.8.1 SimpleBn

Esta rede é usada pelo jogador SimpleBnPlayer, e leva em conta a mínima aposta que o jogador pode fazer, bem como a força da mão do jogador.

Abaixo tempos uma representação dessa rede bayesiana, feita com auxilio do software SAMIAM:



min_bet	zero			over_zero		
hand_rank	low	avg	high	low	avg	high
win	0.6	0.8	0.9	0.1	0.6	0.9
lose	0.4	0.2	0.1	0.9	0.4	0.1

Essa rede é tão simples que sua implementação cabe em 6 linhas:

```
net = Sbn::Net.new("min_bet_hand_rank")
min_bet = Sbn::Variable.new(net, :min_bet, [0.5, 0.5], [:zero, :over_zero])
hand_rank = Sbn::Variable.new(net, :hand_rank, [0.3, 0.6, 0.1], [:low, :avg, :high])
result = Sbn::Variable.new(net, :result, [0.6,0.4,0.8,0.2,0.9,0.1,0.1,0.9,0.6,0.4,0.9,0.1])
min_bet.add_child(result)
hand_rank.add_child(result)
```

Na documentação do jogador SimpleBnPlayer encontra-se uma descrição perfeita do comportamento deste algoritmo:

```
Pucker::SimpleBnPlayer
  #bet
    when has low hand
      when min_bet equals zero
        should check
      when min_bet greater than zero
        should fold
    when has avg hand
      when min_bet equals zero
        should raise
      when min_bet greater than zero
```

```
should check
when has high hand
  when min_bet equals zero
    should raise
  when min_bet greater than zero
    should raise
```

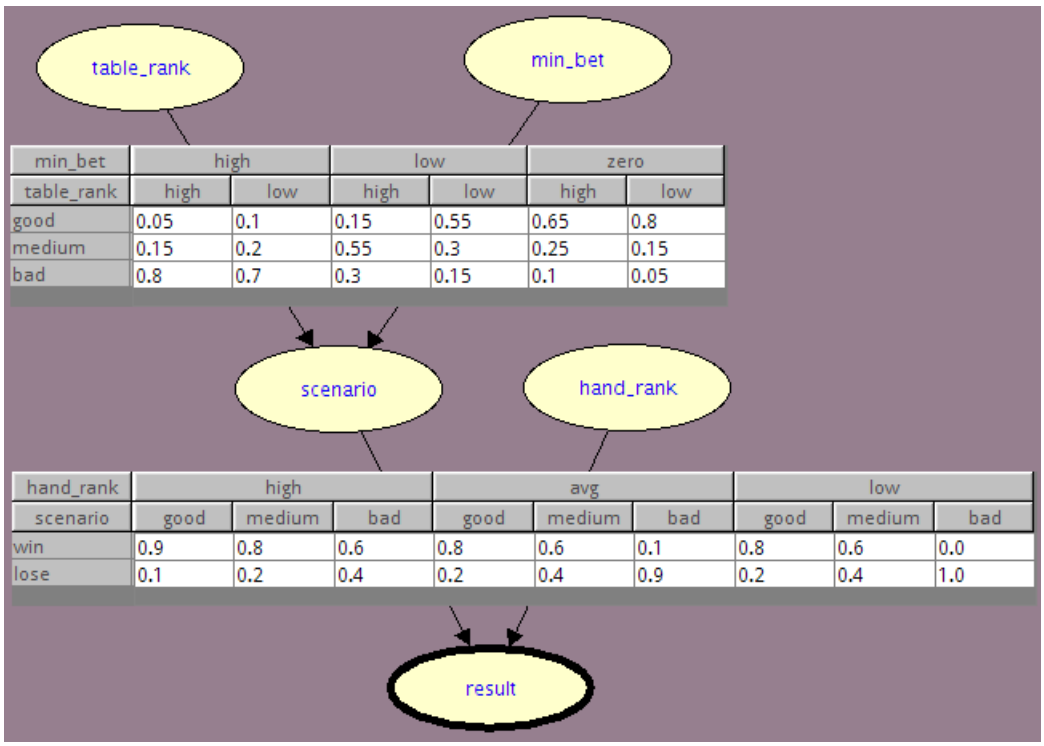
5.8.2 GoodBn

Esta rede é usada pelo jogador BnPlayer, e leva em conta a mínima aposta que o jogador pode fazer, bem como a força da mão do jogador e a força da mesa.

Se a mão do jogador estiver forte unicamente devido às cartas da mesa estarem forte, este jogador é mais cauteloso, pois as cartas da mesa também favorecem os outros jogadores.

A grande contribuição dessa rede ao sistema, é que esta rede permite uma estratégia de blefe ao raciocínio dos jogadores. Assim sendo, mesmo que um jogador não tenha cartas boas, se o cenário for favorável, ele faz uma aposta alta para forçar a desistência dos outros jogadores, e ganhar a quantia colocada pelos blinds (apostas mínimas que precedem o FLOP).

Abaixo tempos uma representação dessa rede bayesiana, feita com auxilio do software SAMIAM:



min_bet	high		low		zero	
table_rank	high	low	high	low	high	low
good	0.05	0.1	0.15	0.55	0.65	0.8
medium	0.15	0.2	0.55	0.3	0.25	0.15
bad	0.8	0.7	0.3	0.15	0.1	0.05

hand_rank	high			avg			low		
scenario	good	medium	bad	good	medium	bad	good	medium	bad
win	0.9	0.8	0.6	0.8	0.6	0.1	0.8	0.6	0.0
lose	0.1	0.2	0.4	0.2	0.4	0.9	0.2	0.4	1.0

Analisando a imagem acima, podemos ver que mesmo que um jogador

tenha uma mão “low”, se o cenário for “good” ele será encorajado a apostar (win: 80%, lose: 20%).

Esta rede tem uma implementação mais complexa, mas vamos demonstra-la afim de exemplificar e justificar o uso da biblioteca SBN para rapidez do processo de desenvolvimento das redes.

```
net = Sbn::Net.new("scenario_hand_rank")
min_bet = Sbn::Variable.new(net, :min_bet, [0.3, 0.4, 0.3], [:high, :low, :zero])
table_rank = Sbn::Variable.new(net, :table_rank, [0.2, 0.8], [:high, :low])
scenario = Sbn::Variable.new(net, :scenario,
    [0.05, 0.15, 0.8, 0.1, 0.2, 0.7, 0.05, 0.8, 0.15, 0.08, 0.9, 0.02, 0.85, 0.13, 0.02, 0.9, 0.1, 0.0],
    [:good, :medium, :bad])
min_bet.add_child(scenario)
table_rank.add_child(scenario)
hand_rank = Sbn::Variable.new(net, :hand_rank, [0.1, 0.6, 0.3], [:high, :avg, :low])
result = Sbn::Variable.new(net, :result,
    [1.0, 0.0, 0.8, 0.2, 0.6, 0.4, 0.999, 0.001, 0.4, 0.6, 0.1, 0.9, 0.99, 0.01, 0.3, 0.7, 0.0, 1.0])
hand_rank.add_child(result)
scenario.add_child(result)
```

5.8.3 BestBn

Essa foi a rede com o melhor desempenho entre todas criadas no decorrer do projeto. Ela é usada pelo jogador BestBnPlayer.

Sua principal característica é que ela leva em conta a posição do jogador para determinar se as informações que possui a cerca do cenário são confiáveis.

Isso é interessante pois quando o jogador encontra-se numa posição desfavorável, não há muito valor em saber que a aposta mínima é baixa. Isso acontece pois vários outros jogadores vão apostar em seguida, o que pode aumentar o valor da aposta mínima.

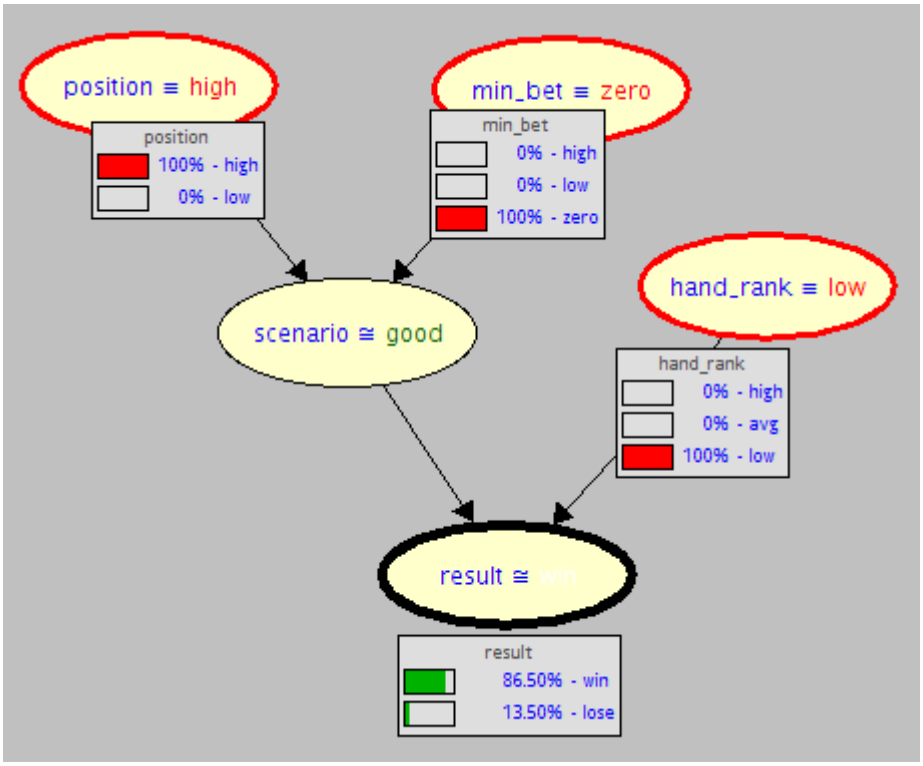
Além disso, como os jogadores seguintes ainda não tomaram qualquer decisão, portanto o cenário não abrange nenhuma informação acerca de suas situações.

Em todas as simulações feitas durante o desenvolvimento do projeto, após 1000 jogadas, esta rede sempre apresentava uma eficiência de 5 ou mais vezes maior do que o melhor de seus adversários.

Dificilmente um jogador com esta rede participa de jogadas ruim. Além disso, o jogador BestBnPlayer, que a utiliza, aposta um maior número de fichas se a chance de vitória calculada pela rede for maior que 90%. Se essa chance for somente moderada, a aposta é menor.

Além disso, essa rede faz o uso de blefes inteligentes quando sabe que os outros jogadores estão em posições desfavoráveis, e com mãos fracas.

Um exemplo dessa situação é demonstrado na imagem abaixo:



Nesta imagem, podemos ver que, apesar do jogador ter uma mão fraca (low), ele irá apostar uma alta quantia, pois sua posição é boa (high), e nenhum jogador fez qualquer tipo de aposta antes (min_bet == zero).

A implementação dessa rede é parecida com a implementação da rede GoodBn, e será omitida para fins de simplificação deste relatório.

Entretanto, é importante considerar o comportamento desta rede em diferentes cenários de aposta, conforme a documentação gerada. Em negrito está destacada a situação exemplificada acima:

```
Pucker::BestBnPlayer
  #bet
    when has low hand
      when scenario is bad
        should fold
      when scenario is medium
        should fold
      when scenario is good
        should raise
    when has avg hand
      when scenario is bad
        should fold
      when scenario is medium
        should fold
      when scenario is good
        should raise
    when has high hand
      when scenario is bad
        should check
      when scenario is medium
        should raise
```

```
when scenario is good
  should raise
```

5.9 Criando um novo jogador

Para criar uma nova estratégia de jogo, recomenda-se o desenvolvimento de uma modelagem inicial usando o software SAMIAM, para simplificar a consideração de diferentes cenários de jogo.

Com essa modelagem em mãos, a implementação é trivial, e pode ser incluida no arquivo lib/pucker/bayesian_networks.rb. O desenvolvedor pode basear-se pelas redes pré-existentes, para compreender como a biblioteca SBN, para criação de redes bayesianas, funciona.

Em seguida, deve-se criar uma subclasse de SimpleBnPlayer, reimplementando o método #build_bayesian_network, de acordo com a rede bayesiana criada no passo anterior. Exemplos de como isso é feito podem ser encontrados no arquivo lib/pucker/bn_players.rb, que contem todos os jogadores inteligentes criados.

O desenvolvedor também deve sobrescrever o método #build_evidence, que recebe informações acerca do cenário de decisão, processa, e entrega a rede bayesiana criada, num formato padrão para ser consumido pela biblioteca SBN.

Recomenda-se o desenvolvimento de testes de integração, que validam se o comportamento do jogador está de acordo com o esperado. Esta prática foi usada em todas as partes desse projeto, e deve ser continuada por desenvolvedores que tem interesse em contribuir com este trabalho.

6 Implementação e Avaliação

6.1 Planejamento e execução de testes

Ao todo, o projeto conta com 63 casos de teste, que passam por 90% de todo o código Ruby presente no Pucker. Entre eles, estão presentes testes unitários, testes funcionais, e testes de integração. Isso foi possível pois este projeto final foi integralmente desenvolvido à luz da metodologia TDD – Test-Driven Development, conforme explicado do tópico 4.4.

A ferramenta escolhida para automatização dos testes chama-se Rspec [16]. Esta ferramenta é a mais popular entre os desenvolvedores da linguagem Ruby. Atualmente ela é amplamente utilizada para teste e especificação de sistemas web desenvolvidos usando o framework Ruby on Rails.

Entretanto, dado a compatibilidade quase que 100% entre o JRuby e o Ruby, ela foi incluída neste projeto sem grandes contratempos.

O Rspec encoraja a especificação do comportamento do código testado, ao invés de sua implementação. Essa prática é conhecida como Behaviour-Driven Development.

Juntamente com cada caso de teste, é fornecida uma string que descreve o escopo do teste. Estas strings são usadas pelo Rspec para criar a documentação do sistema, que acompanha o crescimento dos testes, fornecendo aos desenvolvedores essa documentação com um mínimo de esforço. Na seção 5.3 foi exemplificado como o Rspec pode ser usado para executar todos os testes presentes no Pucker, e gerar sua documentação de forma automatica.

Abaixo é exemplificado como criar um caso de teste, e como esse caso será usado na produção da documentação.

Para a classe Dealer, uma das primeiras criadas neste projeto, existe o seguinte teste, no arquivo spec/pucker/dealer_spec.rb:

```
describe Dealer do
  describe "#deal" do
   it "deals cards" do
     ...
   end
    it "deals 52 different cards" do
     ...
   end
   it "doesn't deal 53 cards" do
     ...
   end
  end
end
```

Este caso de teste produz a seguinte documentação:

```
Pucker::Dealer
  #deal
    deals cards
    deals 52 different cards
    doesn't deal 53 cards
```

6.2 Comentários sobre a implementação

6.2.1 Problemas Encontrados

Durante a implementação, vários contratempos foram observados. O primeiro é que o aluno nunca teve experiência prévia na aplicação de redes bayesianas em qualquer problema real. Isso dificultou o desenvolvimento das

primeiras redes bayesianas, pois não era claro o que, nem como, deveria ser modelado.

Um outro ponto fraco do projeto, observado durante o desenvolvimento, foi a complexidade da simulação de um jogo de pôquer. Este é um jogo bastante complexo, e dependendo do desenrolar das apostas, vários cenários diferentes podem tomar conta do jogo, principalmente no que diz respeito à coleta de apostas, e aos potes de fichas formados.

Essa coleta segue um loop complexo, que depende das fichas de cada jogador, das apostas que cada jogador fez, e do último jogador que fez a maior aposta. Essa complexidade foi responsável pelo atraso ocorrido no projeto durante o mês de outubro, quando boa parte do tempo foi empregada apenas nesta parte do jogo, e na resolução de bugs que apareciam em cada caso extremo dessa coleta de apostas.

Isso resultou num código complexo, difícil de ser testado, e de difícil manutenção, presente no método #collect_bets da classe Game. Este método é sem dúvidas o método mais polêmico do projeto, por ser a principal fonte de execução do sistema, e não conter testes, como o resto do sistema.

Uma decisão inteligente, seria ter começado o projeto por analisar simulações de partidas de pôquer já presentes no universo open-source. Concentrando, dessa forma, todo o tempo do projeto nas redes bayesianas.

7 Considerações Finais

O presente trabalho tem uma característica ímpar se comparado com outros tipos de projeto final: foi integralmente pensado pelo aluno. Desde a sugestão do tema, da forma como este tema seria abordado, das características da solução final, ao desenvolvimento de códigos para auxiliar na simulação dos resultados, e na visualização destes. A única decisão que não tomada pelo aluno foi com relação ao método de apoio à decisão usado. Nesse caso, o aluno pôde contar com a experiência de seu orientador na sugestão do uso de redes bayesianas.

Este trabalho firma um ponto inicial, entre a universidade e o tema abordado, pois foi o primeiro trabalho nesta área dentro da PUC-Rio. Além disso, o projeto Pucker encoraja sua própria continuação, oferecendo um sólido conjunto de testes automatizados aos desenvolvedores. Espera-se que isso gere confiança e estímulo a qualquer um que vier a estender esse sistema.

Sugestões futuras

Como a produção de redes bayesianas, baseadas no bom senso do programador para definição de seus parâmetros, já foi amplamente abordado, recomenda-se que, se houver continuação dessa pesquisa, a mesma se dê pela busca de métodos mais automáticos, e eficientes para definição desses parâmetros. Uma boa alternativa seriam métodos não determinísticos, de inteligência artificial, que pudessem fazer uma otimização dos parâmetros da rede, como por exemplo Algoritmos Genéticos [17].

8 Referências Bibliográficas

[1] (Gilpin and Sandholm (2005)) A. Gilpin and T. Sandholm. Optimal Rhode Island hold'em poker. In AAAI national conference, pp. 1684-1685, 2005.

[2] http://en.wikipedia.org/wiki/Extensive-form_game

[3] (Gilpin et al. (2007)) A. Gilpin, S. Hoda, J. Pena, and T. Sandholm. Gradient-based algorithms for finding Nash equilibria in extensive form games. In 3rd International Workshop on Internet and Network Economics (WINE), 2007.

[4] http://en.wikipedia.org/wiki/Bayesian_network

[5] J. Pearl, Fusion, propagation, and structuring in belief networks, A rtif Intell. 29 (1986)241-288.

[6] Max Henrion: Propagating uncertainty in Bayesiannetworks by probabilistic logic sampling.UAI, 1986, pp.149-164

[7] http://coursera.org/course/pgm

[8] http://reasoning.cs.ucla.edu/samiam/

[9] https://github.com/cayblood/sbn

[10] http://webdocs.cs.ualberta.ca/~games/poker/

[11] https://github.com/alexandremcosta/pucker/

[12] http://jruby.org/download

[13] https://rvm.io/

[14] http://bundler.io/

[15] http://en.wikipedia.org/wiki/Delegation_pattern

[16] http://rspec.info/

[17] http://en.wikipedia.org/wiki/Genetic_algorithm

Anexo I – Um git log em ordem cronológica reversa

BestBnPlayer improvement on specs (2 days ago)
BestBnPlayer raises more if its sure to win (2 days ago)
best player (2 days ago)
BnPlayer (2 days ago)
generate statistics (4 days ago)
improve simple_bn_player (4 days ago)
create BnPlayer (8 days ago)
solved bug! (9 days ago)
debug bets - missing bug: players that already betted, should complete bet (10daysago)
calibrate DummyPlayers (11 days ago)
Game#collect_bets only if player is active (13 days ago)
MultiLogger logs only if not in test environment (13 days ago)
PlayerGroup#reset adds stack instead of creating another player (13 days ago)
logger (13 days ago)
Game uses DummyPlayer, and Player has ID (13 days ago)
README: irb instructions (3 weeks ago)
README (3 weeks ago)
Game#reward reward players (3 weeks ago)
Game#elegigle_players_by_rank sort players (3 weeks ago)
Player#hand_rank (3 weeks ago)
specs: Pot#get_from_all and Pot#empty? (3 weeks ago)
pending tests (3 weeks ago)
handle sidepots (3 weeks ago)
create class Pot to help calculating sidepots (4 weeks ago)
game#collect_bets: business model finished. Missing: handle sidepots (4 weeks ago)
DummyPlayer tests (4 weeks ago)
DummyPlayer#fold refactor (4 weeks ago)
game supports bet raises (4 weeks ago)
.ruby-gemset (4 weeks ago)
create player subclasses (4 weeks ago)
missing new_round var setting (7 weeks ago)
exchange .rvmrc to .ruby-version (8 weeks ago)
evaluate and reward game winners (9 weeks ago)
change magic number to constants (3 months ago)
reverting back readme (3 months ago)
readme: tests output (3 months ago)
PlayerGroup: delegate rotate to container (3 months ago)
Player#active (3 months ago)
Create README.md (3 months ago)
Game: play and its helper methods (3 months ago)
Player: stack and get_from_stack (3 months ago)
PlayerGroup: initializer receives stack amount, and rotate positions (3 months ago)
game: collect bets and deal table cards (3 months ago)
refactoring PlayerGroup: composition over inheritance (3 months ago)
game setup and players set hands (3 months ago)
players group class creation (3 months ago)
requiring relative on pucker.rb (3 months ago)
game and player class creation (3 months ago)
remove cucumber (3 months ago)
add wip filter to RSpec (3 months ago)
dealer refactor (3 months ago)
updating jruby (3 months ago)
Using RSpec instead of Cucumber for dealer (1 year, 1 month ago)
Dealer deals cards 01 (1 year, 1 month ago)
Initial Commit: jruby, java and rspec structure (1 year, 1 month ago)

Anexo II – Documentação do código-fonte

Esta documentação pode ser gerada a partir da execução dos testes deste projeto, usando o seguinte comando, na raiz do projeto:

rspec --format doc

```
Pucker::Dealer
  #deal
    deals cards
    deals 52 different cards
    doesn't deal 53 cards
  #reset
    allows dealer to continue dealing
    should deal cards in different order

Pucker::Game
  #initialize
    should have 5 players by default
    should allow definition of number of players
  #collect_bets
    collect bets (PENDING: Not yet implemented)
  private methods
    #prepare_players
      should rebuy players with insuficient stack
      should set all players active and remove allin states
      should rotate players
    integration tests for rewards
      #eligible_players_by_rank
        sort players
      reward eligible_players_by_rank
        rewards players

Pucker::PlayerGroup
  #[]
    should delegate to container
  #set_hands
    should give 2 cards to each player
  #reset
    shouldnt change players size
    when players have ZERO stack
      should increase back its stack

Pucker::Player
  #initialize
    should have a uniq id
  #bet_if_active
    should return what #bet returns
    when player is NOT active
      should be false
  #bet
    should check every time
  #get_from_stack
    when player has more
      should get amount
    when player has less
      should zero stack
      should desactivate player
  #set_hand
    should put 2 cards on player's hand
  #hand_rank
    should rank players hand according to table cards
    shouldnt call HandEvaluator again if table cards hasnt changed
  protected methods
    #full_hand
      should misc players and table cards

Pucker::DummyPlayer
  #bet
    when he folds
      should be false
```

```
    when he checks
      should == 10
    when he raises
      should return a value between min_value and stack

Pucker::BestBnPlayer
  #bet
    when has low hand
      when scenario is bad
        should fold
      when scenario is medium
        should fold
      when scenario is good
        should raise
    when has avg hand
      when scenario is bad
        should fold
      when scenario is medium
        should fold
      when scenario is good
        should raise
    when has high hand
      when scenario is bad
        should check
      when scenario is medium
        should raise
      when scenario is good
        should raise

Pucker::BnPlayer
  #bet
    when has low hand
      when scenario is bad
        should fold
      when scenario is medium
        should fold
      when scenario is good
        should raise
    when has avg hand
      when scenario is bad
        should fold
      when scenario is medium
        should fold
      when scenario is good
        should raise
    when has high hand
      when scenario is bad
        should check
      when scenario is medium
        should raise
      when scenario is good
        should raise

Pucker::SimpleBnPlayer
  #bet
    when has low hand
      when min_bet equals zero
        should check
      when min_bet greater than zero
        should fold
    when has avg hand
      when min_bet equals zero
        should raise
      when min_bet greater than zero
        should check
    when has high hand
      when min_bet equals zero
        should raise
      when min_bet greater than zero
        should raise

Pucker::Pot
  #all_bets
    should return a hash
```

```
  #add_bet
    when player hasnt betted yet
      should increase the number of contributors
  #sum
    when directed called
      should misc too pots
    when called via +=
      should misc too pots in place
  #total_contributed_by
    when player hasnt betted
      should return ZERO
  #empty
    should be empty when initialized
    should be empty when there arent bets
  #get_from_all
    should get an amount from every players bet

Pending:
  Pucker::Game#collect_bets collect bets
    # Not yet implemented
    # ./spec/pucker/game_spec.rb:19

Finished in 10.5 seconds
63 examples, 0 failures, 1 pending
Coverage report generated for RSpec to /home/alexandre/Dev/pucker/coverage. 619 / 682 LOC (90.76%) covered.
```