**Linguagens Funcionais - Solução de Quebra-Cabeça Sudoku**

**Claudio Massao Uyeno, João Marcus Velasques Faria, Thiago Rodrigo Bucalão**

Departamento de Informática (DIN)  
Universidade Estadual de Maringá (UEM) - Maringá, Pr - Brasil

[claudio\_massao@live.com](mailto:claudio_massao@live.com), [j.marcus.faria@gmail.com](mailto:j.marcus.faria@gmail.com), thiago\_rodrigo90@hotmail.com

**Resumo**

*Este artigo tem como objetivo o aprofundamento sobre as linguagens funcionais. Este se dará através da escolha de um tema relacionado a algum tipo de quebra-cabeças, de tamanho n, com tempo de execução igual ou superior a 2 minutos. É solicitado aos alunos a implementação de um algoritmo, no paradigma Funcional (Racket) e Imperativo (Java), que resolva o quebra-cabeças, e que os tempos de execução sejam descritos, a fim de comparação de resultados, utilizando entradas de complexidades diferentes.*

**Abstract**

*This article aims to deepen about functional languages. This will be through the choice of a topic related to some sort of puzzle of size n, with execution time greater than 2 minutes. It asked students to implement an algorithm, the Functional Paradigm (Racket) and imperative (Java), to solve the puzzles, and the runtimes are described in order to compare results, using inputs of different complexities .*

1. **Introdução**

O problema abordado neste artigo é o jogo Sudoku, ao qual é um jogo de quebra-cabeça de fácil entendimento porém apesar da simplicidade aparente este é um problema matemático computacional, com bastante conhecimento estabelecido mas muito ainda por desvendar.

O artigo está estruturado da seguinte forma. Na seção 2.1, as características do paradigma funcional. Na seção 3.1, linguagens de programação que utilizam o paradigma funcional e suas características. Na seção 4.1, será apresentado sobre o problema e a sua complexidade computacional. As seções 5.1 e 6.1 é abordado a resolução computacional do problema nos diferentes paradigmas: funcional e imperativo/orientado a objetos utilizando as linguagens Racket e Java respectivamente.

Por fim na seção 7.1, será apresentado os resultados comparando as duas abordagens implementadas e na seção 8.1, a conclusão e a seção 9.1, as referências bibliográficas.

* 1. **Paradigma Funcional:**

Segundo o (SEBESTA,2003, p.555), “o objetivo do projeto de uma linguagem de programação funcional é imitar as funções matemáticas no maior grau possível. Está é uma abordagem diferente das demais linguagens que utilizam o método imperativo”.

O Paradigma é caracterizado por aplicações de funções matemáticas e que evita estado ou dados mutáveis ao contrário da programação imperativa que enfatiza mudanças no estado do programa; as linguagens funcionais usam aplicações funcionais, expressões condicionais e recursão para controlar a execução.

Uma função pode ter ou não parâmetros e um simples valor de retorno, um exemplo é a linguagem Racket, esses parâmetros são os valores de entrada da função, e o valor de retorno é o resultado da função.

Sendo assim, temos que as duas características que dominam na programação funcional primeiramente que o significado de uma expressão é o seu valor e o papel do computador é simplesmente obtê-la e segundo de que uma função em uma linguagem funcional pode ser construída, manipulada e resolvida, como qualquer outro tipo de expressão matemática, usando leis algébricas.

* 1. **Linguagens de programação funcional**

Como foi dito na seção anterior, as linguagens de programação ao qual utilizam o paradigma funcional não são muito difundidas no desenvolvimento de software, ficando mais restritas ao meio acadêmico. A figura 2 mostra que as linguagens orientadas a objetos vêm se destacando nos últimos anos. Nota-se que em 2014 a linguagem Python e Java são as que lideram o mercado em seguida temos C++, entretanto as linguagens que suportam o paradigma funcional como a LP Ruby tem se destacado, vale lembrar que Ruby suporta outros tipos de paradigmas também como: orientada a objetos, [imperativa](http://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_imperativa) e [reflexiva](http://pt.wikipedia.org/wiki/Reflex%C3%A3o_(programa%C3%A7%C3%A3o)).

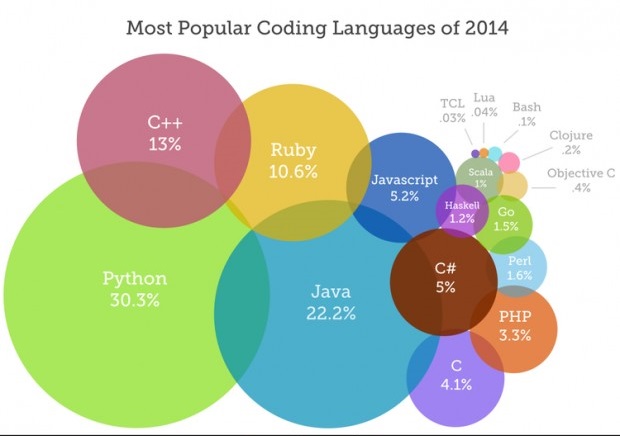


Figura 1: Popularidade das LP no ano de 2014.

Entretanto notamos que Haskell, uma LP puramente funcional é uma das menos utilizadas. Vamos descrever um pouco sobre algumas linguagens que abordam a o paradigma funcional, escolhemos as seguintes linguagens: Haskell, LISP, Racket, Miranda.

* 1. **Haskell**

Como já mencionado, é uma linguagem puramente funcional que leva esse nome em homenagem ao lógico Haskell Curry.

A linguagem Haskell tem suporte a funções recursivas e tipos de dados, casamento de padrões, *lis comprehensions*, *guar statements* e avaliação preguiçosa. A combinação destas características pode fazer com que a construção de funções que seriam complexas em uma linguagem procedural torna-se uma tarefa quase trivial em Haskell. Outra característica é que Haskell é estaticamente tipada, ou seja, o compilador saberá quais partes do código é um número, o que é uma string e assim por diante.

* 1. **LISP**

Lisp é uma família de linguagens de programação concebida por [John McCarthy](http://pt.wikipedia.org/wiki/John_McCarthy) em [1958](http://pt.wikipedia.org/wiki/1958). Num célebre artigo, ele mostra que é possível usar exclusivamente funções matemáticas como estruturas de dados elementares. A linguagem Lisp foi projetada primariamente para o processamento de dados simbólicos.[] Ela é uma linguagem formal matemática. Durante os anos de [1970](http://pt.wikipedia.org/wiki/1970) e [1980](http://pt.wikipedia.org/wiki/1980), Lisp se tornou a principal linguagem da comunidade de inteligência artificial.

A linguagem LISP é interpretada, onde o usuário digita expressões em uma linguagem formal definida e recebe de volta a avaliação de sua expressão. Deste ponto de vista podemos pensar no LISP como uma calculadora, que ao invés de avaliar expressões aritméticas avalia expressões simbólicas, chamadas de expressões.[] Cada programa em LISP, é, portanto, uma expressão. As expressões são de tamanho indefinido e tem uma estrutura de árvore binária. A estrutura de utilização da memória disponível é na forma de listas, pois livra o programador da necessidade de alocar espaços diferentes para o programa e para os dados, fazendo com que os dados e os programas sejam homogêneos, característica única da linguagem LISP.

* 1. **Racket**

É uma é uma [linguagem de programação](http://pt.wikipedia.org/wiki/Linguagem_de_programa%C3%A7%C3%A3o) [multiparadigma](http://pt.wikipedia.org/wiki/Multiparadigma) que suporta [programação funcional](http://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_funcional) e [procedural](http://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_procedural). A linguagem é utilizada numa variedade de contextos, tais como [scripting](http://pt.wikipedia.org/wiki/Linguagem_de_script), programação de uso geral, a educação de ciência da computação e pesquisa.

* 1. **ML**

É uma [linguagem de programação](http://pt.wikipedia.org/wiki/Linguagem_de_programa%C3%A7%C3%A3o) [funcional](http://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_funcional) de proposta geral desenvolvida por [Robin Milner](http://pt.wikipedia.org/wiki/Robin_Milner) e outros no final dos [anos 1970](http://pt.wikipedia.org/wiki/Anos_1970) na [Universidade de Edimburgo](http://pt.wikipedia.org/wiki/Universidade_de_Edimburgo), cuja sintaxe foi inspirada pelo [ISWIM](http://pt.wikipedia.org/w/index.php?title=ISWIM&action=edit&redlink=1). É considerada uma linguagem funcional impura, por permitir a programação imperativa, ao contrário de outras linguagens funcionais como [Haskell](http://pt.wikipedia.org/wiki/Haskell_(linguagem_de_programa%C3%A7%C3%A3o)), por esta razão também é considerada uma linguagem de programação multi-paradigma.

As características do ML são incluir chamada-a-valor avaliação e estratégia, em primeira classe funções de [gerenciamento automático de memória](http://pt.wikipedia.org/wiki/Gerenciamento_de_mem%C3%B3ria) através de [coleta seletiva de lixo](http://pt.wikipedia.org/wiki/Coletor_de_lixo), [polimorfismo](http://pt.wikipedia.org/wiki/Polimorfismo) paramétrico, tipagem estática, tipo inferência, tipos de dados algébricos, correspondência padrão, exceção e manuseio.

* 1. **Sudoku: Um problema NP-Completo**

O jogo é formado por um quadrado bidimensional, no qual sua versão clássica tem um total de 9 x 9 = 81 células, e essas por sua vez, é composto por nove quadrados menores a figura 2 apresenta o formato do jogo em seu estágio inicial (à esquerda) e estado final (à direita).

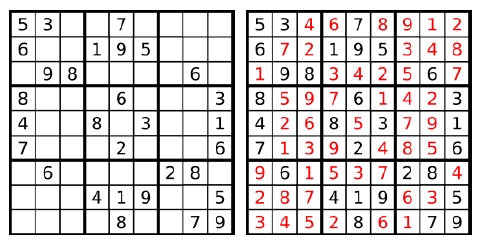


Figura 2: Exemplo do jogo Sudoku, estado inicial (esquerda) e estado final(direita).

Portanto o objetivo do Sudoku é preencher os espaços em branco do quadrado maior, que está dividido em nove grids, com os números de 1 a 9. Os algarismos não podem se repetir na mesma coluna, linha ou grid.

A palavra “Sudoku” significa “número sozinho” em japonês, o que mostra exatamente o objetivo do jogo. O jogo foi criado na verdade no ocidente, na década de setenta pelo arquiteto Howard Garns, mas começou a ganhar popularidade no final de 2004 quando começou a ser publicado diariamente na sessão de *puzzles* do jornal *The Times*.

Como já foi mencionado aparentemente é um problema simples de resolver, porém segundo (ALBUQUERQUE, 2013) o jogo Sudoku é um problema computacional NP-Completo Garey (1979), o qual a prova foi realizada por Takayuki Yato e Takahiro Seta, da Universidade de Tóquio Yato (2003). O problema inicial era propor um algoritmo que resolvesse o Sudoku de todos os tamanhos e níveis de dificuldade, entretanto não foi obtido êxito pois o tempo necessário para encontrar a solução aumenta drasticamente à medida que *n* cresce, ou seja, solução tende ao infinito.

“Determinar o número exato de quadrados possíveis se revelou uma tarefa extremamente difícil e que é o obtido de forma aproximada, estimando-se o número de quadrados do Sudoku válidos como sendo 6.670.903.752.021.072.936.960. Isso se refere as soluções derivadas de qualquer quadrado. Se forem contados apenas uma vez os quadrados que podem ser reduzidos a uma configuração equivalente, o número final cai para 5.472.730.538. Apesar desta redução, mesmo que fosse solucionado um quadrado por minuto e se vivesse cem anos, só seria possível resolver cerca de 1% do jogo.”

Segundo (FREITAS, 2003) para a prova da NP-Completude, provar que o jogo de Sudoku pertence a NP, basta se propor um algoritmo que reconheça se um dado jogo totalmente preenchido é realmente uma solução do Jogo. Isto pode ser feito em tempo polinomial, varrendo-se as linhas, colunas e blocos fazendo a verificação se cada númeor do intervalo de 1 a 9 aparece exatamente uma vez, devolvendo a resposta se “sim” é solução ou “não”, não é solução.

A segunda parte da prova da NP-Completude é provar que o jogo é NP-Difícil realizando uma redução polinomial de um problema já provado NP-Difícil. E. Yato e S. Yato (2003) provaram fazendo uma redução polinomial do quadrado latino para o Sudoku, ou seja, mostraram como tornar um quadrado latino n x n incompleto e criar, através dele um Sudoku n2 x n2 que, se resolvido, fornece também a resposta para o quadrado latino. Desta forma, demostraram que resolver o Sudoku é tão difícil quanto completar um quadrado latino.

1. **Sudoku: Implementação em Racket**

A implementação do Sudoku utilizando a LP funcional Racket foi dada da seguinte maneira: A ideia do algoritmo é, inicialmente, definir a estrutura do quadro, ou tabuleiro, do jogo, na forma de listas. Estas listas são definidas com funções genéricas (posicao, coluna, linha), e em seguida preenchidas com os os numerais, previamente definidos (1 - 9).

Em seguida, cada coordenada é definida e organizada em forma de caixas, ou seja, os quadrantes, uma vez que estes também são determinantes para o preenchimento dos numeros.

Após isso, todas as posições são vetorizadas, e as comparações são feitas para todos os elementos que possuem o valor zero. Neste caso, a ideia é verificar todas as possibilidades que podem ser colocadas no lugar do zero e, a medida que um numero não se torna possível de ocupar determinado lugar, ele é eliminado. Este processo é repetido em backtracking até que exista apenas um numero, e este será a solução para um determinado "quadrado" do tabuleiro.

A execução da função **solucionar** se dá em conjunto com a função **solucionar-de.** A primeira verifica o primeiro elemento que está com o número zero (através da função **primeiro-livre**). A segunda verifica, através da lista de números disponíveis (função **num-possiveis**), qual o numeral mais adequado para ocupar a "casa".

No final da execução, o vetor é convertido de volta a uma lista de listas através das funções **converte** (que transforma o vetor em uma lista) e **separar** (que divide o vetor de 81 números em 9 vetores de 9 elementos), e em seguida a solução, através da função **print-quadro**, juntamente com o tempo de execução(através do comando *time* fornecido pelo racket), é apresentada. A figura 3 mostra um exemplo de saída obtida na execução do algorítmo:

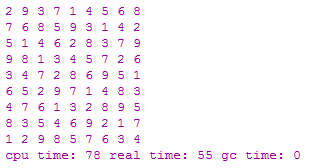


Figura 3: Saída obtida na execução na linguagem Racket

1. **Sudoku: Implementação em Java**

Na implementação do Sudoku utilizando LP imperativa, optamos pela linguagem Java, onde o programa foi dividido nas seguintes classes: Tabuleiro, Celula, Algoritmos e SudokuSolver2.

A classe Tabuleiro é composta pelos seguintes atributos:

* **Pai:** é do tipo Tabuleiro e utilizado para armazenar o endereço de memória do tabuleiro pai do tabuleiro atual;
* **Tabuleiro:** é formado por uma matriz 9x9 do tipo Celula;
* **nZeros:** é do tipo inteiro e representa a quantidade de zeros no tabuleiro.

Além disso, a classe apresenta as seguintes funções:

* **importTabuleiro:** é responsável por fazer a leitura do arquivo do tipo “.txt” e armazenar os dados montando o tabuleiro para ser solucionado. Ao mesmo tempo que ele faz a leitura, ele prepara o tabuleiro para iniciar a busca pela solução colocando em todas as celulas vazias (com valor 0) uma lista do tipo Celula contendo todos os valores possíveis para aquela celula (valores de 1 a 9);
* **mostraTabuleiro:** responsável por mostrar um determinado tabuleiro que seja visualmente agradável ao usuário;
* **mostraPossiveis:** função utilizada para auxiliar no desenvolvimento e teste do algoritmo essa função não tem o objetivo de encontrar a solução final.

A classe Celula é composta dos seguintes atributos:

* **listaPossivies:** é a lista de elementos inteiros. Ela armazena os possíveis valores que cabem em uma determinada celula (posição do tabuleiro);
* **troca:** é do tipo booleano. Utilizado para fixar os elementos e impedir que haja alteração no valor presente na celula, o que poderia alterar completamente o tabuleiro original;
* **valorFixo:** tipo inteiro. Contém o melhor valor considerado para aquela celula.

A classe Algoritmo contém todas as funções responsáveis por encontrar a solução final sendo elas:

* **ehSolucao:** faz a chamada das funções verificaLinhaColuna e verificaQuadrante;
* **verificaLinhaColuna:** responsável por pecorrer todas as linhas e colunas e verificar se não existem 2 ou mais células com o mesmo número;
* **verificaQuadrante:** responsável por verificar se no mesmo quadrante (matrix 3x3) não possui o mesmo valor 2 ou mais vezes;
* **removeImproprios:** percorre todo o tabuleiro buscando as celulas vazias e removendo da sua “listaPossiveis” todos os valores que não servem para aquela posição considerando as regras do jogo;
* **fixaTodos:** percorre o tabuleiro procurando por posições que contenham apenas um único elemento em sua “listaPossiveis”, fixando esse valor;
* **copiaTabuleiro:** função para criar as cópias do tabuleiro para que seus “filhos” possam ser gerados para evitar o “shallow copy” presente na linguagem java. Não utilizamos a função “clone” já existente, pois precisaríamos copiar outros atributos para o novo tabuleiro além do tabuleiro em si.
* **buscaMelhor:** procura a melhor posição para tentar fixar os valores e gerar os próximos filhos;
* **fazFilhos:** função que gera os possíveis filhos que podem ser gerados a partir de uma posição no tabuleiro, selecionado pela função “buscaMelhor”;
* **atualizaPossiveis:** utiliza a mesma ideia da função “fixaTodos”;
* **soluciona:** função que chama as funções anteriores para gerar a solução.
  1. **Resultados dos testes: Racket vs. Java**

Nesta seção iremos verificar as comparações de tempo de execução utilizando as duas implementações iremos fazer quatro comparações, primeiramente com uma entrada no nível fácil de solução, em seguida uma médio e uma difícil e por fim será comparado todas as entradas simultaneamente.

O primeiro teste de entrada que iremos utilizar é uma entrada fácil a figura 4 apresenta a comparação do tempo de execução nas duas linguagens.

Figura 4: Teste dos algoritmos no nível fácil.

O segundo teste de entrada que iremos utilizar é uma entrada de nível médio e é apresentado na figura 5.

Figura 5: Teste dos algoritmos no nível médio.

O terceiro teste de entrada que iremos utilizar é uma entrada de nível difícil e sua comparação de tempo entre as duas linguagens é dada pela figura 6.

Figura 6: Teste dos algoritmos no nível médio.

Houveram alguns casos de nível difíceis em que o algoritmo em Java não terminou a execução.

* 1. **Conclusão**

A princípio, sentimos um pouco de dificuldade ao iniciar a implementação, pois conforme a figura 1 mostrou, poucos são os que utilizam a linguagem Racket. Mas nos permitiu colocar em prática e aprender um pouco mais sobre esse tipo de linguagem.

Ao desenvolvermos este trabalho fomos capazes de aprender um pouco mais sobre o histórico da linguagem de programação funcional e do seu funcionamento, em especial a linguagem Racket, que foi a linguagem definida para a implementação da versão funcional do algoritmo.

Analisando os gráficos, podemos perceber que na maioria dos casos a implementação na linguagem funcional apresentou melhor desempenho sobre a linguagem imperativa. Porém, nos deparamos com um único caso em que ocorreu o inverso.

Por fim, podemos concluir que a linguagem funcional tende a apresentar um desempenho melhor sobre a linguagem imperativa, apesar de ser um pouco mais complexa, provavelmente devido ao fato de que pouco sabíamos sobre ela. E que nossos algoritmos estão longe de serem ideais, pois houve discrepância e variações na solução de alguns problemas.

**9.1. Referências bibliográficas**

ALBUQUERQUE, L; FREITAS, R; O jogo Sudoku: pré-coloração estendida em hipergrafos e algoritmos de enumeração implícita. Simpósio Brasileiro de Pesquisa Operacional, Natal/RN, Setembro 2013.

Robert Sebesta, Concepts Of Programming Languages, 4ªedição. Capítulo 14.

<<http://homepages.dcc.ufmg.br/~loureiro/md/md_0Introducao.pdf>> <Acessado 11/11/2014>

<<http://pt.wikipedia.org/wiki/Haskell_(linguagem_de_programa%C3%A7%C3%A3o)>> <Acessado 11/11/2014>

<<http://pt.wikipedia.org/wiki/Racket>> <Acessado 11/11/2014>