TÉCNICO LISBOA

Elementos de Programação
Professor Carlos Caleiro

Satisfatibilidade

Projeto de Computação Evolutiva

103122 **Pedro Antunes**
103734 **Pedro Costa**
103742 **Rui Wang**

Novembro 2021

Índice

Introdução

O problema de satisfatibilidade (SAT) procura determinar se uma dada fórmula normal conjuntiva é satisfazível ou não. Uma fórmula é satisfazível se existir uma atribuição de valores lógicos às variáveis que a torna verdadeira.

O objetivo deste projeto é desenvolver em Python um programa que calcule uma solução para este problema, exata ou aproximada, implementando uma variante do algoritmo adaptativo ASAP que usa o paradigma de computação evolutiva. Foi utilizada programação em camadas: o simulador e as estruturas de dados subjacentes.

O simulador recorre às estruturas de dados para simular a evolução de uma população de indivíduos. Cada indivíduo possui a si associada uma valoração às variáveis da fórmula e evolui através de três eventos distintos - mutação, melhoramento e regeneração - geridos por uma agenda (cadeia de acontecimentos pendentes).

Estruturas auxiliares

1. Bitset

A classe Bitset representa uma sequência de *bits* de tamanho fixo `N`, que se encontram ativos ou inativos. Encontra-se munida das seguintes operações:

- `__init__(size): Inicializa um bitset novo de tamanho size`
- `getSize(): Retorna o tamanho do bitset`
- `st(i): Ativa o bit na posição i`
- `reset(i): Inativa o bit na posição i`
- `flip(i) Altera o estado do bit na posição i`
- `test(i): Retorna o estado do bit na posição i`
- `count(): Retorna o número de bits ativos`

Foram desenvolvidas as seguintes implementações:

- `bitset.py`

  Usa uma lista de valores booleanos para representar os bits nos estados ativo (`True`) ou inativo (`False`)

- `bitset_alt.py`

  Usa uma lista de `1`s e `0`s para representar os bits nos estados ativo (`1`) ou inativo (`0`)

Tipos de dados

1. Fórmula

A classe Formula representa a fórmula normal conjuntiva que se procura satisfazer. Encontra-se munida das seguintes operações:

- `__init__(path)`: Inicializa a fórmula acedendo ao ficheiro de input localizado na diretoria `path`
- `getVarCount()`: Retorna o número de variáveis na fórmula
- `getClauseCount()`: Retorna o número de cláusulas na fórmula
- `evaluate(valoracao)`: Retorna o número de cláusulas da fórmula satisfeitas por `valoracao` (avaliação)

Foram desenvolvidas as seguintes implementações:

- `formula.py`

Nesta implementação a informação de cada cláusula é guardada como uma lista de inteiros tal como representados nos ficheiros cnf. O atributo `self._clauses` é uma lista de todas estas listas.

A avaliação de uma valoração itera todas as listas em `self._clauses` e percorre cada uma até encontrar um literal verdadeiro. Caso encontre, o contador `satisfy` é incrementado por 1. Após terminar a iteração este contador é retornado.

`formula.py`

```
...
    def evaluate(self, valoracao):
        satisfy = 0
        for cl in self._clauses:
            i = 0
            while i < len(cl) and \
                    (valoracao.test(abs(cl[i]) - 1) != (cl[i] > 0)):
                i += 1
            if i < len(cl):
                satisfy += 1
        return satisfy
```

- `formula_alt.py`

Nesta implementação, para cada variável `x` são criados dois números binários b_{x0} e b_{x1} do problema SAT, correspondentes aos literais `-x` e `x`. Seja `N` o número de variáveis e `C` o número de cláusulas do problema SAT. Para cada cláusula `c`, se `c` contém o literal `-x` então o bit de b_{x0} na posição `c` é `1`, caso contrário é `0`, se `c` contém o literal `x` então o bit de b_{x1} na posição `c` é `1`, caso contrário é `0`. Estes números são guardados na matriz `self._columns`

que contém N pares (listas de dois elementos) de inteiros b_{x0}, b_{x1}; é 0-indexada pelo que é necessário subtrair 1 ao índice de cada variável no ficheiro. Note-se também que cada b_{ij} é iniciado com o bit C ativo, que nunca é utilizado, para garantir que todos são do mesmo tamanho e evitar problemas das operações bitwise entre inteiros de tamanhos diferentes, encontrados durante a realização do projeto.

formula_alt.py

```
...
    def __init__(self, path):
        self._varCount = 0
        self._clauseCount = 0
        self._columns = []
        clause = 0
        for line in open(path, "r"):
            if not line.startswith("c") and len(line) > 2:
                args = line.split()
                if args[0] == "p":
                    self._varCount = int(args[2])
                    self._clauseCount = int(args[3])
                    for _ in range(self._varCount):
                        self._columns.append([1 << self._clauseCount]*2)
                else:
                    i = 0
                    while i < len(args) and args[i] != "0":
                        self._columns\
[abs(int(args[i])) - 1][int(int(args[i]) > 0)] |= (1 << clause)
                        i += 1
                    clause += 1
            print(bin(self._columns[i][0]))
            print(bin(self._columns[i][1]))
...
```

A avaliação de uma valoração itera pelas N variáveis e para cada variável x toma b_{x1} caso tenha valor True na valoração ou b_{x0} caso tenho valor False, e vai juntando os bits ativos em algum destes números binários pois estes correspondem a cláusulas satisfeitas. Esta operação corresponde a aplicar bitwise-or de todos os bs.

```
...
    def evaluate(self, valoracao):
        orColumns = 1 << self._clauseCount
        for i in range(self._varCount):
            orColumns |= self._columns[i][valoracao.test(i)]
...
```

Após a iteração, subtrai ao contador `satisfy`, que inicia com o valor `C`, o número de bits inativos (cláusulas não satisfeitas por nenhuma variável) e retorna-o.

```
...
        satisfy = self._clauseCount
        while orColumns & (orColumns + 1):
            satisfy -= 1
            orColumns |= orColumns + 1
        return satisfy
...
```

2. Valoração

A classe Valoracao representa um conjunto de valores lógicos atribuídos às variáveis de uma forma normal conjuntiva, que neste projeto é apelidada de fórmula. Herda as propriedades e métodos da classe Bitset e possui as seguintes operações adicionais:

- `compare(valoracao)`: Compara a valoração com o parâmetro `valoracao` devolvendo `True` se forem equivalentes e `False` caso contrário
- `display()`: Retorna uma representação da valoração em string cujo i-ésimo caractere é `"1"` se o valor lógico atribuído na valoração à i-ésima variável da fórmula é verdadeiro, `"0"` em caso contrário

3. Indivíduo

A classe Individuo representa um indivíduo que possui associados a si vários atributos. Encontra-se munida das seguintes operações:

- `__init__(idt, valoracao)`: Inicializa um indivíduo com identificador `idt`, valoração `valoracao`, Mem vazia, Actv a indicar todos os bits como ativos e PrMut igual a `0.5`
- `getId()`: Retorna o identificador do indivíduo
- `getValoração()`: Retorna a valoração do indivíduo
- `setValoração(valoracao)`: Altera a valoração do indivíduo para `valoracao`
- `getEval()`: Retorna a avaliação da valoração memorizada do indivíduo
- `setEval(eval)`: Altera o avaliação da valoração memorizada do indivíduo para `eval`
- `memorize(valoracao)`: Adiciona `valoracao` à memória do indivíduo
- `getRandomMemVal()`: Retorna uma valoração aleatória da memória do indivíduo
- `forget()`: Apaga a memória do indivíduo
- `valCount()`: Retorna o número de valorações na memória do indivíduo

- `uniqueValCount(): Retorna o número de valorações distintas na memória do indivíduo`
- `isLocked(pos): Retorna True se o bit na posição pos do indivíduo está inativo, False caso contrário`
- `lockBits(): Torna inativos os bits do indivíduo cujos valores lógicos coincidam em todas as valorações memorizadas pelo mesmo`
- `getActvCount(): Retorna o número de bits ativos do indivíduo`
- `getPtMut(): Retorna o valor de PrMut do indivíduo`
- `setPrMut(prMut): Altera o valor de PrMut do indivíduo para prMut`

Foram desenvolvidas as seguintes implementações:

- `individuo.py`

Nesta implementação cada informação associada ao indivíduo é um atributo: `self._id`, `self._val` (valoração), `self._eval`, `self._mem` (lista de valorações), `self._actv` (bitset), `self._PrMut`.

- `individuo_alt.py`

Nesta implementação toda a informação associada ao indivíduo é guardada num dicionário `self._info` com keys: "`id`", "`val`" (valoração), "`eval`", "`mem`" (lista de valorações), "`actv`" (bitset), "`PrMut`".

4. População

A classe População representa um conjunto de indivíduos. Encontra-se munida das seguintes operações:

- `__init__(): Inicializa a população vazia`
- `add(individuo): Adiciona individuo à população`
- `replace(idt, individuo): Substitui o indivíduo de identificador idt por individuo`
- `getIndividuo(idt): Retorna o indivíduo de identificador idt`
- `getAll(): Retorna lista dos indivíduos da população`
- `getRandomOther(idt): Retorna um indivíduo aleatório da população que não o indivíduo de identificador idt`

Foram desenvolvidas as seguintes implementações:

- `populacao.py`

Nesta implementação os indivíduos da população são guardos numa lista `self._elem` ordenados por identificador.

- `populacao_alt.py`

Nesta implementação os indivíduos da população são guardos num dicionário `self._elem` que corresponde a cada id (key) o indivíduo correspondente (value).

5. Evento

A classe Evento representa um acontecimento que atua sobre o sistema alterando o seu estado. Cada evento é caracterizado pela sua categoria, o instante em que é executado e o seu alvo, atuando sobre um único indivíduo (evento local) ou toda a população (evento global). Suporta os métodos:

- `__init__(kind, time, target): Inicializa um novo evento de categoria kind, instante de excução time e alvo target`
- `getKind(): Retorna a categoria do Evento`
- `getTime(): Retorna o instante de execução`
- `getTarget(): Retorna o identificador do indivíduo-alvo ou None`

Foram desenvolvidas as seguintes implementações:

- `evento.py`

  Nesta implementação cada informação associada ao evento é um atributo: `self._kind`, `self._time`, `self._target`.

- `evento_alt.py`

  Nesta implementação toda a informação associada ao evento é guardada num tuplo de 3 elementos `self._info`.

6. CAP

A classe CAP representa uma cadeia de acontecimentos pendentes, ou seja, uma agenda para eventos futuros. Encontra-se munida das seguintes operações:

- `__init__(): Inicializa a agenda vazia`
- `add(evento): Adiciona evento à agenda`
- `next(): Retorna o primeiro evento da agenda`
- `remove(): Remove o primeiro evento da agenda`

Foram desenvolvidas as seguintes implementações:

- `cap.py`

  Nesta implementação os eventos do cap são guardados numa lista ordenada `sefl._eventList` com o evento de menor instante de execução no fim da lista. É realizada uma pesquisa binária para introduzir cada novo evento e os eventos são removidos do fim da lista.

cap.py

```
...
    def add(self, evento):
        left = 0
        right = len(self._eventList)
        while left < right:
            mid = (left + right) // 2
            if evento.getTime() > self._eventList[mid].getTime():
                right = mid
            else:
                left = mid + 1
        self._eventList.insert(left, evento)
...
```

- cap_alt.py

Esta implementação é semelhante a `cap.py` mas mantém uma lista ordenada para cada tipo de evento em `self._eventMap`, que associa cada tipo (key) à lista (value). Ao adicionar um evento novo ao cap é realizada uma pesquisa binária na lista do tipo correspondente e para remover o próximo evento é escolhido o de menor instante de execução entre os últimos elementos de cada lista do `self._eventMap`.

cap.py

```
...
    def add(self, evento):
        if evento.getKind() not in self._eventMap:
            self._eventMap[evento.getKind()] = []
        kindList = self._eventMap[evento.getKind()]
        left = 0
        right = len(kindList)
        while left < right:
            mid = (left + right) // 2
            if evento.getTime() > kindList[mid].getTime():
                right = mid
            else:
                left = mid + 1
        kindList.insert(left, evento)
...
```

Simulador

O simulador importa os módulos:

main.py

```
from math import log
from random import random
import numpy
import copy

from cap import CAP
from evento import Evento
from formula_alt import Formula
from individuo import Individuo
from populacao_alt import Populacao
from valoracao import Valoracao
```

Define as funções `expRandom(m)` que aleatoriza os intervalos de tempo entre eventos do mesmo tipo e alvo e o próprio `simulador(TFim, TMut, TMelh, TReg, In, path)`.

O simulador começa por inicializar a fórmula, variáveis `N` e `C`, flag `foundSolution` e a valoração `solution` que começa não definida.

```
    formula = Formula(path)
    N = formula.getVarCount()
    C = formula.getClauseCount()

    foundSolution = False
    solution = None
```

Inicializa também a população com `In` indivíduos de identificadores de 0 a In-1 inclusive, criando uma valoração aleatória para cada um que irá ser melhorada. Depois, para cada indivíduo, avalia a valoração inicial e guarda o valor no indivíduo; caso encontre alguma solução exata, assinala em `foundSolution` e `solution`.

```
    populacao = Populacao()
    for idt in range(In):
        firstValoracao = Valoracao(N)
        for i in range(N):
            if random() < 0.5:
                firstValoracao.flip(i)
        populacao.add(Individuo(idt, firstValoracao))
```

```
    for individuo in populacao.getAll():
        individuo.setEval(formula.evaluate(individuo.getValoracao()))
        if individuo.getEval() == C:
            foundSolution = True
            solution = individuo.getValoracao()
```

Por fim inicializa a `agenda` com um evento de cada tipo ("`mut`", "`melh`" e "`reg`") e alvo, de instantes de execução aleatórios segundo os parâmetros `TMut` e `TReg` exceto os eventos de melhoramento que têm de ser aplicados às valorações iniciais, sendo atribuídas instante `0`. As variáveis `currentEvent` e `currentTime` são inicializadas com o primeiro evento e entra no loop de simulação que corre até o instante de execução do evento atual exceder o tempo final especificado ou ser encontrada uma solução.

```
    agenda = CAP()
    for individuo in populacao.getAll():
        agenda.add(Evento("mut", expRandom(TMut), individuo.getId()))
        agenda.add(Evento("melh", 0, individuo.getId()))
    agenda.add(Evento("reg", expRandom(TReg), None))

    currentEvent = agenda.next()
    currentTime = currentEvent.getTime()

    while currentTime < TFim and not foundSolution:
```

Em cada passo da simulação podemos executar 1 de 3 tipos de evento:

- **Mutação**

A valoração do indivíduo alvo é copiada e os bits não bloqueados são aleatóriamente alterados de acordo com o PrMut do indivíduo, sendo a nova valoração reavaliada.

```
    if currentEvent.getKind() == "mut":

        individuo = populacao.getIndividuo(currentEvent.getTarget())
        newValoracao = copy.deepcopy(individuo.getValoracao())

        for i in range(N):
            if not individuo.isLocked(i) and \
random() < individuo.getPrMut():
                newValoracao.flip(i)
        newEval = formula.evaluate(newValoracao)
```

Se a nova valoração não for pior que a valoração atual é atribuída ao indivíduo, sendo a antiga memorizada e a avaliação do indivíduo atualizada. Se a nova valoração for perfeita

(e melhor que a atual) então é assinalada nas variáveis próprias. No fim reagendamos este evento de acordo com `TMut`.

```
    if newEval >= individuo.getEval():
        individuo.memorize(individuo.getValoracao())
        individuo.setValoracao(newValoracao)
        individuo.setEval(newEval)
        if newEval == C:
            foundSolution = True
            solution = newValoracao

        agenda.add(Evento("mut", currentTime + expRandom(TMut),
                          individuo.getId()))
```

- **Melhoramento**

Tal como a mutação, é criada uma cópia da valoração do indivíduo alvo. Os bits não bloqueados são percorridos por ordem aleatória, sendo usada uma permutação aleatória de [0, 1, …, N - 1] e saltando os bits bloqueados; para cada bit, alteramo-lo, reavaliamos a valoração (`compEval`) e comparamos com a avaliação antiga, que está sempre guardada em `newEval`. Se não for pior mantemos esta alteração e atualizamos `newEval`, caso contrário revertemos a alteração. Após esta iteração, memorizamos a valoração antiga do indivíduo e guardamos a nova, atualizando também a avaliação do indivíduo. Se a nova valoração for perfeita então é assinalada nas variáveis próprias. No fim reagendamos este evento de acordo com `TMelh`.

```
    elif currentEvent.getKind() == "melh":

        individuo = populacao.getIndividuo(currentEvent.getTarget())
        newValoracao = copy.deepcopy(individuo.getValoracao())
        newEval = individuo.getEval()

        for i in numpy.random.permutation(N):
            if not individuo.isLocked(i):
                newValoracao.flip(i)
                compEval = formula.evaluate(newValoracao)
                if compEval >= newEval:
                    newEval = compEval
                else:
                    newValoracao.flip(i)
        individuo.memorize(individuo.getValoracao())
        individuo.setValoracao(newValoracao)
        individuo.setEval(newEval)
        if newEval == C:
            foundSolution = True
```

```
            solution = newValoracao

        agenda.add(Evento("melh", currentTime + expRandom(TMelh),
                          individuo.getId()))
```

- **Regeneração**

  No caso da regeneração iteramos por todos os indivíduos da população.

```
    elif currentEvent.getKind() == "reg":

        for individuo in populacao.getAll():
```

Caso o indivíduo tenha pelo menos 10 valorações memorizadas, se não existem pelo menos 3 distintas colonizamo-lo com uma valoração `newValoracao` aleatória da memória de um outro indivíduo aleatório da população `colonizer`; existe o caso degenerado de apenas existir 1 indivíduo na população, no qual foi decidido usar uma valoração aleatória.

```
            if individuo.valCount() >= 10:

                if individuo.uniqueValCount() < 3:
                    if In == 1:
                        newValoracao = Valoracao(N)
                        for i in range(N):
                            if random() < 0.5:
                                newValoracao.flip(i)
                    else:
                        colonizer = \
populacao.getRandomOther(individuo.getId())
                        if colonizer.valCount() == 0:
                            newValoracao = \
copy.deepcopy(colonizer.getValoracao())
                        else:
                            newValoracao = \
copy.deepcopy(colonizer.getRandomMemVal())
```

O processo de colonização a partir de uma valoração consiste no melhoramento da valoração, usando o mesmo código do evento de melhoramento sem a verificação por bits bloqueados, e a criação de um novo indivíduo de mesmo identificador com a nova valoração que substitui o indivíduo anterior na população. Se a nova valoração for perfeita então é assinalada nas variáveis próprias.

```
                    newEval = formula.evaluate(newValoracao)
```

```
                    for i in numpy.random.permutation(N):
                        newValoracao.flip(i)
                        compEval = formula.evaluate(newValoracao)
                        if compEval >= newEval:
                            newEval = compEval
                        else:
                            newValoracao.flip(i)
                    idt = individuo.getId()
                    newIndividuo = Individuo(idt, newValoracao)
                    newIndividuo.setEval(newEval)
                    populacao.replace(idt, newIndividuo)
                    if newEval == C:
                        foundSolution = True
                        solution = newValoracao
```

Se existem pelo menos 3 valorações distintas memorizadas bloqueamos os bits comuns a todas elas no indivíduo chamando `individuo.lockBits()`, removemos todas as valorações da memória usando `individuo.forget()`, e alteramos a probablidade de mutação PrMut para o quociente entre o número de bits bloqueados `individuo.getActvCount()` e o dobro do número de variáveis `N`.

```
                else:
                    individuo.lockBits()
                    individuo.forget()
                    individuo.setPrMut( \
individuo.getActvCount() / (2 * N))
```

Caso o indivíduo não tenha 10 valorações memorizadas permanece inalterado. No final reagendamos este evento de acordo com `TReg`.

```
        agenda.add(Evento("reg", currentTime + expRandom(TReg), None))
```

Após executar o evento, este é removido da agenda e atualizamos `currentEvent` e `currentTime` para o próximo evento.

```
    agenda.remove()
    currentEvent = agenda.next()
    currentTime = currentEvent.getTime()
```

Ultimamente, após a simulação terminar, se uma solução foi encontrada é retornado um coeficiente `1` e a representação em string da valoração encontrada, senão percorremos todos os indivíduos da população guardando sempre a melhor avaliação `maxEval` e a

valoração correspondentes `bestVal`, sendo `maxEval` inicializada com `-1` para garantir que é menor que qualquer avaliação da população; é retornado o coeficiente de adaptação calculado pelo quociente entre a avaliação `maxEval` por o número de cláusulas `C` e a representação em string da valoração correspondente.

```
if foundSolution:
    maxEval = C
    bestVal = solution
else:
    maxEval = -1
    bestVal = None
    for individuo in populacao.getAll():
        if individuo.getEval() > maxEval:
            maxEval = individuo.getEval()
            bestVal = individuo.getValoracao()

return (maxEval / C, bestVal.display())
```

Resultados

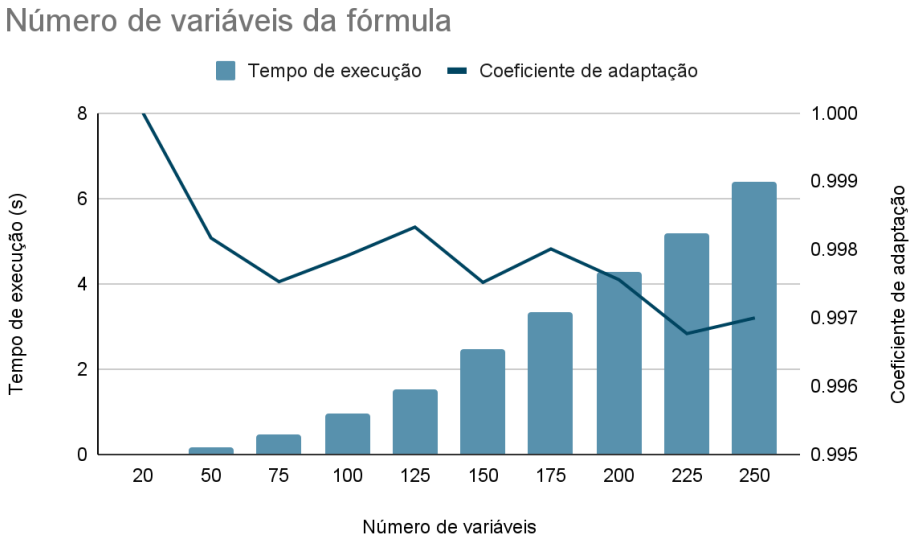
Realizaram-se testes para avaliar a *performance* do programa em diversos cenários, variando cada um dos parâmetros:

- Número de variáveis da fórmula normal conjuntiva
- Tempo de simulação
- Tempo médio do processo de Mutação
- Tempo médio do processo de Melhoramento
- Tempo médio do processo de Regeneração
- Número de Indivíduos da população

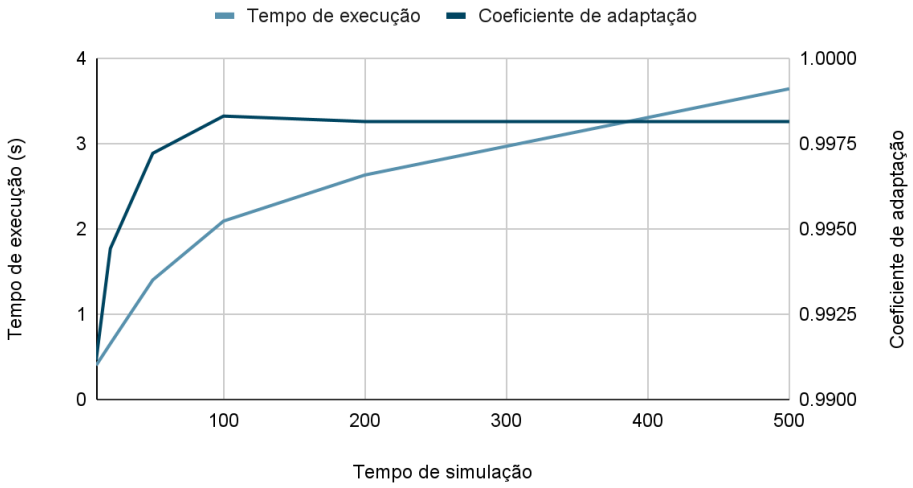
A fim de testar o programa, escolheram-se os seguintes valores *default* para cada um dos parâmetros. Observe-se que o ficheiro escolhido tem 150 variáveis.

```
(TFim=100, TMut=5, TMelh=5, TReg=5, In=10, path="ProblemSet/uf150-01.cnf")
```

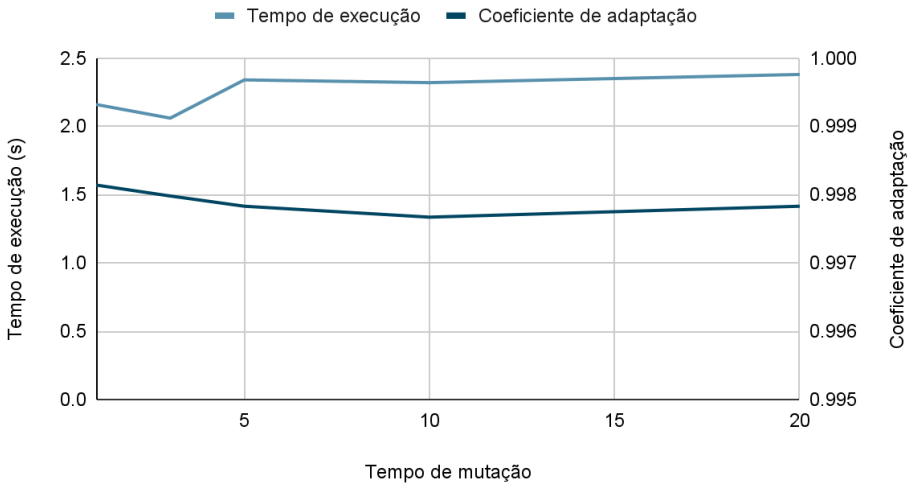
Apresentam-se de seguida 6 gráficos dos tempos de execução (em segundos) e coeficiente de adaptação em função da variação de um único parâmetro, mantendo os restantes inalterados. Cada teste foi realizado 10 vezes para minimizar o impacto da natureza aleatória do programa nos resultados obtidos. As simulações foram todas executadas no mesmo computador.



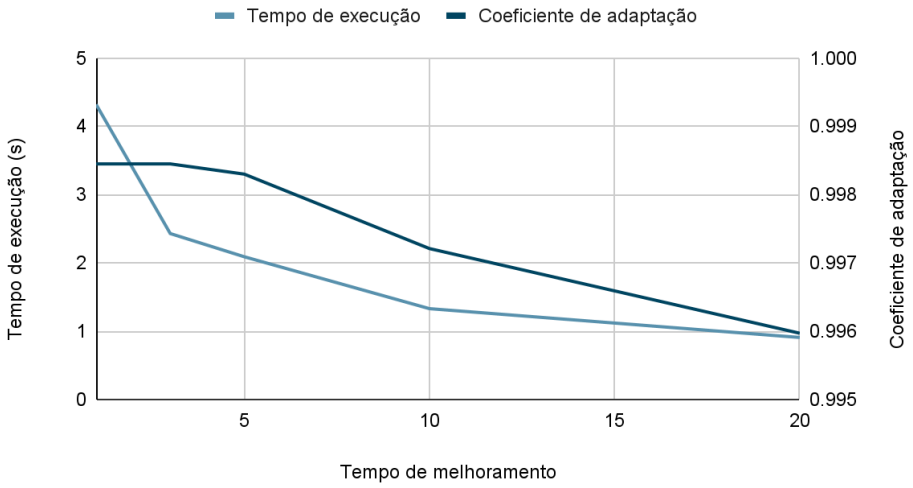
Tempo de simulação



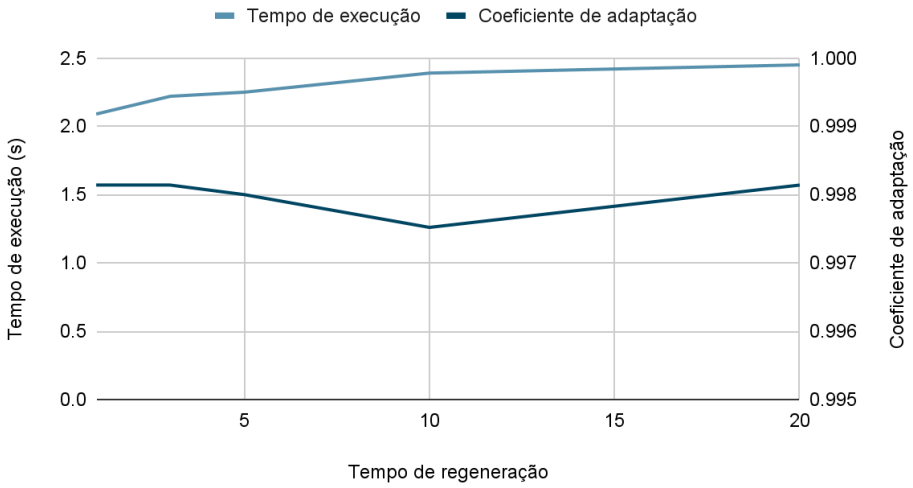
Tempo de mutação



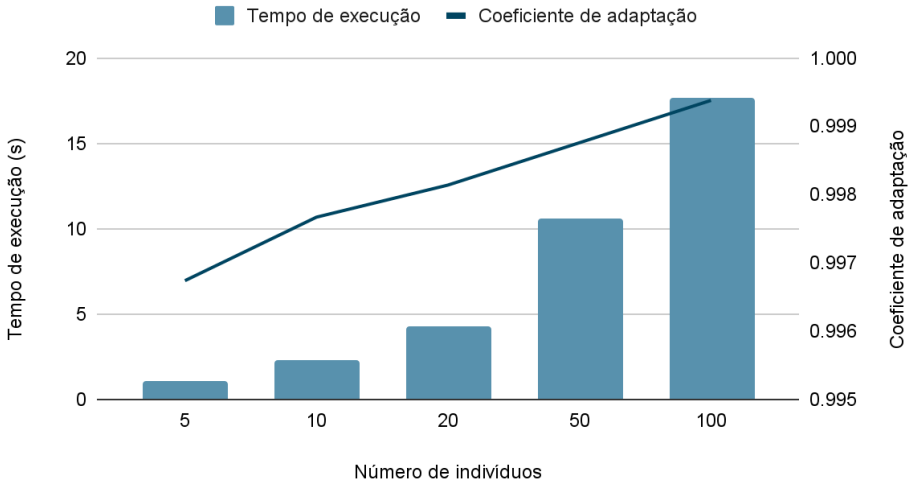
Tempo de melhoramento



Tempo de regeneração



Número de indivíduos



Discussão

Tempo de execução

Uma análise cuidada dos resultados evidencia que o tempo de execução do programa aumenta de forma aproximadamente linear com o número de variáveis da fórmula. De facto, durante a realização deste projeto, pudemos constatar que uma porção significativa do tempo de execução do programa é gasta no cálculo do coeficiente de adaptação de uma valoração, que por sua vez aumenta linearmente com o número de cláusulas e de variáveis!

O tempo de simulação, que à primeira vista poderia aparentar ter um simples efeito linear no tempo de execução do programa, comporta-se de forma ligeiramente diferente. Isto pode dever-se à seguinte possibilidade: com maior tempo de execução, um maior número de casos teste será resolvido antecipadamente, atenuando o aumento do tempo de execução.

Os gráficos “Tempo de mutação” e “Tempo de regeneração” sugerem que estes eventos não têm grande impacto no *runtime* do programa. Por outro lado, o aumento do tempo de melhoramento, que resulta na diminuição da frequência desse mesmo evento, reduz significativamente o tempo de execução, o que vai de encontro com a previsão de que o cálculo do coeficiente de adaptação de uma valoração é dispendioso, pois o melhoramento é o evento que mais o usa.

Por fim, o gráfico “Número de indivíduos” mostra o óbvio, um maior número de indivíduos exige um maior tempo de execução.

Coeficiente de adaptação

Em geral, todos os coeficientes obtidos são muito próximos de 1, o que confere mérito ao algoritmo. Todavia, é possível tecer alguns comentários pertinentes.

O aumento do número de variáveis da fórmula piora ligeiramente a *performance* do programa.

Os gráficos “Tempo de simulação” e “Número de indivíduos” apresentam uma clara tendência crescente, que converge para 1. Com tempo de execução ou população grandes o suficiente, qualquer fórmula satisfazível deve ser resolvida pelo algoritmo.

Os gráficos “Tempo de mutação” e “Tempo de regeneração” revelam que estes eventos não afetam muito o coeficiente de adaptação. Por sua vez, o aumento do tempo de melhoramento (diminuição da sua frequência) provoca a diminuição do coeficiente. Este parece ser o evento mais determinante em toda a simulação.