TÉCNICO LISBOA

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Programação Orientada por Objetos

Projeto

Carlos Henrique Silva - 80872
José Carlos Vieira - 90900
Pedro Carmo - 90989

11 de Maio de 2018

1 Introdução

Num contexto de programação orientada a objetos, este projeto consiste na implementação de uma simulação do comportamento de um conjunto de indivíduos de uma população. Considerando uma matriz, com $n \times m$ pontos, o objetivo é que os indivíduos descubram o melhor caminho entre o ponto inicial e o ponto final. Cada caminho possuí um custo associado ao número de arestas atravessadas. Estas arestas possuem um custo associado, no entanto, quando estas fazem parte de um zona especial, o custo pode ser superior. Há também a possibilidade de introduzir obstáculos, ou seja, pontos na matriz em que nenhum indivíduo se pode posicionar.

A simulação é composta por um conjunto de eventos associado a um indivíduo, que determina o comportamento do mesmo. Como tal, as possibilidades de evento associados a um indivíduo são a morte, um movimento e uma reprodução, sendo que foi também considerado um evento observação que demonstra alguns parâmetros da simulação, com intervalos de tempo de $\frac{\tau}{20}$, sendo τ o tempo total de execução da simulação. Outra questão que define o comportamento da população é a epidemia, que é aplicada quando a população ultrapassa o limite definido, no entanto não é caracterizada como sendo um evento. Por fim, todos os parâmetros de inicialização utilizados para criar a simulação são lidos de um ficheiro em formato XML[1]. Isto requer a utilização do SAX [2]. A execução do programa ocorre com a introdução do comando:

```
java -jar <JAR-ARCHIVE-NAME>.jar <INPUT-XML-FILE>
```

2 Composição

Com base nas especificações pedidas, o problema foi organizado em três pacotes: o *static_comp*, o *dynamic_comp* e o *main*. O primeiro contém classes relativas à estrutura estática da simulação, o segundo composto pelas classes relativas à dinâmica da simulação e a última é apenas composta pela classe *Main*, responsável pela execução do programa bem como o tratamento de erros caso o ficheiro de leitura definido nos argumentos seja inválido.

Dentro do contexto estático, pode-se encontrar a classe *Point*, que tem apenas dois atributos, x e y, que definem a sua posição na *grid*, uma instância da classe *Grid*. Esta última, por sua vez, possuí duas listas constituídas por objetos *Edge*, uma com as arestas (*edges*), associadas aos caminhos possíveis entre os pontos da *grid*, e outra com *specialZones*, com os edges associados às zonas especiais. A classe *Edge* possuí um ponto inicial, um ponto final e o custo associado à sua travessia. Para além disto, a classe *Grid* também possuí uma lista constituída por objetos *Point*, que agrega todos os pontos que são obstáculos. Existe também outra classe *Path*, que é utilizada para agregar os *edges* pelos quais um indivíduo trilha o seu caminho, sendo que este não acumula *edges* de ciclos que, eventualmente, voltem a repetir posições já pertencentes ao *path* desse mesmo indivíduo.

[1] *Extensible Markup Language*
[2] *Simple API for XML*

Relativamente ao domínio dinâmico, existe a classe *Individual*, que define o comportamento do indivíduo ao longo da simulação. A cada indivíduo tem-se associado a variável de conforto, φ, calculada pela fórmula:

$$\varphi(z) = \left(1 - \frac{cost(z) - length(z) + 2}{(cmax - 1) \times length(z) + 3}\right)^k \left(1 - \frac{dist(z, (x_f, y_f))}{n + m + 1}\right)^k \quad (1)$$

De modo a agrupar todos os indivíduos da mesma população criou-se a classe população, *Population* que é constituída por uma lista de indivíduos que permite a adição e remoção dos mesmos.

De forma a realizar a simulação, é necessário organizar os acontecimentos de forma ordeira. Deste modo foi gerada a classe abstrata *Event*, que contém o tempo no qual o evento ocorre e o indivíduo associado, como atributos, e uma função abstrata *action* que é definida consoante o tipo de evento a realizar. Foram criadas as subclasses: *EvDeath*, *EvMove*, *EvReproduction* e a *EvObservation*, delineadas para simular a morte, o movimento e a reprodução do indivíduo e a observação dos parâmetros da simulação, respetivamente. Cada evento, quando gerado, é colocado no PEC[3], que contém uma classe *TreeSet*, que implementa a lista ordenada pelo tempo estipulado através de um objeto *Comparator*.

Finalmente, de modo a interligar a parte dinâmica com a parte estática formulou-se a classe *Simulation*, responsável pela atribuição de todos os elementos estáticos e parâmetros necessários para correr a simulação, bem como pela execução dos respetivos eventos. Estes elementos e parâmetros são fornecidos através de um ficheiro em formato XML, interpretado pela classe *Parser*, que para além de fazer a leitura de todos os elementos, também verifica se estes se encontram de acordo com o ficheiro DTD[4] fornecido, bem como se estes elementos representam dados válidos.

3 Implementação

3.1 Point

A classe *Point* é considerado o elemento mais básico da estrutura estática, utilizado para definir um padrão para descrever a posição de qualquer objeto que necessite ser posicionado na *grid*.

Os seus atributos utilizam visibilidade *private*, dado que não devem ser alterados por outra classe após ser instanciada. Deste modo, são criadas duas funções *getX()* e *getY()*, de forma a possibilitar a leitura dos seus atributos por outra classe. Existe também a redefinição dos métodos *equals* e *hashCode*, de forma a tornar a comparação entre dois pontos baseada nos seus atributos.

[3] *Pending Event Container.*
[4] *Document Type Definition.*

3.2 Edge

De forma a definir a relação entre dois pontos imediatamente próximos, é utilizada a classe *Edge*, constituída por dois objetos que instanciam a classe *Point* e um valor inteiro que corresponde ao custo de travessia de uma extremidade para a outra. É também utilizada na *grid*, para definir os movimentos possíveis dentro da mesma, bem como as zonas de custo especial.

Os seus atributos encontram-se com visibilidade *private*, não permitindo o acesso direto por parte de outras classes, fornecendo *getters* para obter os pontos e o custo, sendo que este último possuí um *setter* com visibilidade *package*. Também aqui é utilizada a redefinição dos métodos *equals* e *hashCode* de forma a permitir que a comparação de cada *Edge* seja baseada no valor dos seus pontos.

3.3 Path

A classe *Path* é utilizada para guardar o conjunto de *edges* (*ArrayList*) pelos quais o indivíduo atravessou, desde o início até ao fim da simulação. É de notar que, caso o indivíduo se mova para um ponto que já está no respetivo *path*, esse ciclo de *edges* é eliminado. Esta restrição é verificada quando um *Edge* é adicionado ao *path*, o que justifica a necessidade de desenvolver uma classe para caracterizar esta estrutura. Mais ainda, esta possuí um atributo de custo, que representa a soma dos custos de todos os *edges*.

Ambos os seus atributos possuem visibilidade *private*, de modo a não serem acedidas diretamente. Isto obriga a que a alteração da *ArrayList* dos *edges* seja feita sempre por intermédio dos métodos *setEdges* ou *addEdge*, que atualizam automaticamente o custo do *path*, assim que o conjunto de *edges* é alterado.

3.4 Grid

Com o intuito de caracterizar o ambiente estático de simulação, existe a classe *Grid*, que é composta por três *ArrayLists*, uma para os obstáculos (classe *Point*), outra para todos os *edges* existentes entre as dimensões definidas (classe *Edge*) e, por fim, uma com todos os *edges* pertencentes às zonas de custo especial. Para além disto, esta possuí ainda os atributos respetivos aos números de colunas e linhas, e dois pontos: o ponto inicial e o ponto final. Todos os atributos mencionados estão com visibilidade *private* e só possuem *getters* para possibilitar o seu acesso, por parte de outras classes. Esta classe implementa uma interface, *IGrid*. Esta interface permite a criação de uma outra classe *Grid*, onde é sabido que irá implementar os métodos nela definidos, e assim a simulação irá correr sem problemas de compatibilidade.

3.5 Event

A classe *Event* é uma classe abstrata com dois atributos, um tempo e um indivíduo com visibilidade *private* e de tipo *final*, de modo aos seus valores serem imutáveis após a construção de um objeto da subclasse de *Event*. Esta classe implementa a interface

IEvent que contém a função *action* descrita posteriormente.

Deste modo, possui um método abstrato *action*, que representa a ação que o evento vai realizar e também possui duas funções, *getTime* e *getIndividual* que permite o acesso ao tempo e o indivíduo do evento. Foi escolhido com intuito de poder ser estendido para vários diferentes tipos de eventos.

É preciso definir o método abstrato nas suas subclasses de modo a construir um objeto. Como tal, para o bom funcionamento do programa criou-se 4 subclasses: *EvDeath*, *EvMove*, *EvReproduction* e *EvObservation*. Em seguida, estão descritas a definição do método *action* para cada subclasse.

3.5.1 EvDeath

Retira todos os eventos associados a esse indivíduo do *Pec* e elimina-o da lista de indivíduos da população. Para percorrer os eventos do *Pec* utilizou-se um *Iterator* do tipo *Event* para tornar o funcionamento do método mais rápido.

3.5.2 EvMove

Em primeiro lugar, é gerado um novo evento de movimento para o indivíduo em questão e de seguida é efetuado o deslocamento do mesmo. De seguida é testado se o indivíduo possui melhores caraterísticas que o melhor indivíduo até esse instante. Esta verificação depende se o ponto final já foi atingido ou não. Se ainda não foi atingido é atualizado o *best_inidivual* sempre que o conforto é maior que o atual, caso contrário é atualizado sempre que o *move* é para a posição final e o custo é menor que o atual.

3.5.3 EvObservation

Imprime o resultado das observações da população durante a simulação do programa em intervalos de tempo definidos. O formato da impressão deve ser o seguinte:

```
Observation number:
                    Present instant:                instant
                    Number of realized events:      events
                    Population size:                size
                    Final point has been hit:       yes/no
                    Path of the best fit individual: {(x_1, y_1),...,(x_j, y_j)}
                    Cost/Comfort:                   cost/comfort
```

Figura 1: Formato do *print* das observações

3.5.4 EvReproduction

Cria um novo evento de reprodução para o evento em questão e cria um filho, ou seja, uma nova referência de *Individual* com base no caminho que o pai já percorreu e no

conforto do mesmo. O comprimento do prefixo (*length*) é calculado do seguinte modo:

$$length = 0.9 * length_{father} + 0.1 * comfort_{father} \tag{2}$$

De seguida, cria os eventos para o novo indivíduo e adiciona-o à lista de indivíduos da população e atualiza *best_inidivual* caso não tenha chegado à posição final e o conforto seja melhor que o atual.

3.6 PEC

A classe *PEC* constituí um conjunto de eventos ordenados pelo tempo. Para tal utilizou-se o *TreeSet*, uma classe implementada com a interface *SortedSet* que para ser definida, necessita de um objeto *Comparator* do tipo *Event*, que através do método *compare* organiza-os consoante o tempo do evento e o indivíduo associado. Esta função está estabelecida para não permitir eventos do mesmo tipo (mesma subclasse), mesmo tempo e mesmo indivíduo. Exceto esta situação, todas as outras são possíveis. Este atributo é privado para não poder ser alterado fora da classe, sendo permitido o acesso através de um método *get*.

3.7 Individual

A classe *Individual* possui duas variáveis estáticas que serão iguais para todos os indivíduos: a grelha, objeto do tipo *Grid* e o parâmetro de conforto. Os atributos específicos a cada instância são o conforto, a sua posição(*Point*), o caminho percorrido(*Path*), a distância à posição final e a distância já percorrida. Estes parâmetros são todos privados de modo a não poderem ser mudados fora da classe. Como tal, foram criados métodos (*getters*) para cada atributo, que permitem ter acesso a cada variável.

Esta classe tem dois construtores possíveis. O primeiro é utilizado quando se inicializa a grelha com os indivíduos iniciais e o segundo é utilizado quando um nasce por ação de reprodução de outro indivíduo visto que precisa de ter em conta os parâmetros do indivíduo-pai (caminho e conforto). Ambos têm visibilidade *package* por apenas ser possível construir indivíduos através da população ou no evento de reprodução.

O método *calculateDist* é responsável por atualizar o valor da distância até à posição final. Este número é calculado através da equação, sendo x e y as coordenadas da posição atual do indivíduo:

$$dist = |x - x_{final}| + |y - y_{final}| \tag{3}$$

A função *calculateComfort* está encarregada de atualizar o valor do conforto de cada indivíduo após a realização de um movimento ou após a construção de um objeto *Individual*. Esta variável será calculada com base na equação 1, correspondendo as seguintes incógnitas a:

1. *cost*, o custo do caminho
2. *length*, o caminho já percorrido

3. *dist*, a distância ao ponto final
4. c_{max}, o custo máximo de uma aresta na grelha
5. n, m que representam as dimensões da grelha

Estes últimos dois métodos são privados visto que o cálculo destas variáveis é apenas efetuado dentro da classe, após ações específicas, como por exemplo, um deslocamento.

Finalmente existe o método *move*, incumbido de efetuar o movimento do indivíduo quando é realizado um evento de mobilidade. Este método tem visibilidade *package* por apenas haver um deslocamento quando ocorre um evento de movimento, não podendo ser realizado noutra ocasião.

3.8 Population

Uma população é representada por uma *LinkedList* do tipo indivíduo. Esta lista tem um número inicial e máximo de indivíduos (dado pelas variáveis *initial_pop* e *max_pop*). Deste modo, para construir um objeto do tipo *Population* são necessários estes dois atributos. Todos os parâmetros são privados e os últimos dois são do tipo *final* de modo a serem constantes após a sua definição no construtor. De forma a ter acesso à lista de indivíduos foi disponibilizado um método *get*.

No método *startPopulating* cria-se a população inicial de indivíduos. Este método chama a função *addIndividual* o número de vezes igual ao atributo *initial_pop*.

De seguida, a função *addIndividual* verifica se o número máximo de indivíduos foi atingido. Em caso afirmativo acontece uma epidemia que elimina um número aleatório de indivíduos e no mínimo deixa 5 sobreviventes, valor invariável definido na variável de classe *NR_SURVIVORS*, que representam os indivíduos com melhor conforto. O método *epidemics* é privado por apenas poder ser chamado na própria classe. A função auxiliar *getIndMaxComfort* também é privada pela mesma razão.

3.9 Simulation

Nesta classe é feita a interligação entre os dois pacotes: estático e dinâmico. Como tal vai ser composto por uma população (*Population*) que vai permitir interagir com os indivíduos, uma referência de *Event* para o evento atual, um objeto de *Individual* para o indivíduo com o melhor caminho em cada instante da simulação, um *boolean* que indica se o ponto final já foi atingido, o instante atual e um contador de eventos para possibilitar a impressão de quantos eventos já foram realizados na *EvObservation*.

Os restantes atributos correspondem ao instante final, a um conjunto de eventos, *PEC*, a uma grelha (*Grid*) que representa o mapa e a parte estática e aos parâmetros de movimento, morte e reprodução, necessários para calcular o tempo de cada subclasse de evento(*EvMove*, *EvDeath* e *EvReproduction* respetivamente). Estes parâmetros são estipulados como finais devido à sua constância ao longo da simulação.

Todos os atributos desta classe são privados de modo a proteger a memória das variáveis respetivas. São disponibilizados métodos *get* para ter acesso ao valor das variáveis e

também há dois métodos *set* que possibilitam alterar o valor do melhor indivíduo e da variável que indica se o fim foi alcançado. Estas funções apenas são utilizadas nas classes da parte dinâmica, sendo de visibilidade *package*.

O construtor efetua o *Parser* do ficheiro *XML* e inicializa toda as variáveis com os parâmetros necessários. Pelo método *createPopulation* ser chamado dentro do construtor, esta função deve ser apenas usada dentro da classe, sendo definida como privada.

O método *startSimulation* corre enquanto que o instante atual da simulação não alcançar o valor final. Note que há utilização de polimorfismo quando se chama o método *action* no evento atual, dado que embora o *Pec* tenha um *TreeSet* do tipo *Event*, as referências ao objeto são de subclasses de *Event* e como tal apenas são definidas durante o funcionamento do programa.

O método auxiliar chamado *expRandom*, utilizado para calcular o tempo dos eventos é um método de classe, por ser igual independentemente do objeto que a chama.

3.10 Parser

Uma das funcionalidades da aplicação é fornecer ao utilizador a possibilidade de introduzir um ficheiro em formato XML, com os elementos e respetivos valores necessários para executar a simulação. Esta funcionalidade é habilitada pela implementação da classe *Parser*, que estende a classe *DefaultHandler*, contido na biblioteca SAX. Esta API fornece os métodos *startElement*, chamada quando é detetado um elemento, e *characters*, chamada quando é detetado texto no interior de uma secção de dados de um elemento. Outros métodos herdados são o *fatalError*, o *error* e o *warning*, que são chamados quando existe um erro fatal, um erro ou um aviso na validação do ficheiro XML contra o ficheiro DTD. Estas últimas lançam uma *SAXException* para a classe *Simulation* e acabam por ser tratadas no método *main*, uma vez que esta se encontra dentro de um bloco *try/catch*. A mensagem da exceção lançada é construída na função, de acordo com o problema.

Relativamente à interpretação dos elementos e respetivos valores, esta classe possuí um atributo do tipo *HashMap<String,Integer[]>*, onde são guardados os elementos em formato de vector de inteiros, identificados por uma *String* única. Por esta razão, quando a *tag* associada ao nome do elemento, como é o caso da situação em que há vários elementos *zone*, é necessário garantir que a *String* que identifica o elemento seja único. Sendo assim, existe um contador que incrementa o seu valor sempre que a *tag* é repetida, e concatena-o ao nome da *tag*. Deste modo será necessário identificar todos os elementos com a *String*: $tag + i$.

As funções *getPoint*, *getEdge* e *getInteger* fornecem uma forma de obter o valor desejado através de uma *tag*, sendo que se esta não existir, ou o valor não for válido, estas geram uma exceção do tipo *SAXException* que será captada da mesma forma que as anteriores. Na figura 2 pode-se observar um exemplo de mensagem de erro para o caso em que segundo o elemento *zone* tem o *xinitial* mal escrito no ficheiro.

```
Could not start simulation due to:
        Error at 9      Attribute "xinitial" is required and must be specified for element type "zone".
```

Figura 2: Mensagem de erro em que segundo o elemento *zone* tem o *xinitial* mal escrito no ficheiro.

4 Testes

Com o intuito de verificar a existência de erros durante a execução da aplicação, foram estabelecidos vários ficheiros de entrada, variando alguns parâmetros.

4.1 Teste 1

O primeiro teste é corrido com o ficheiro fornecido pelo enunciado. Segundo a figura 3 é possível não só observar o melhor caminho e os parâmetros obtidos para o *best fit individual*, mas também uma representação gráfica do melhor *path*, que se pode verificar correto. É de notar que em todos os testes executados, o ponto final foi sempre atingido, embora possam ser obtidos com custos diferentes, isto é, com caminhos diferentes. Note que [B] representa cada posição do melhor caminho, [I] representa o ponto inicial, [O] representa um obstáculo e [F] representa o ponto final.

```
        Path of the best fit individual:        {(1,1),(1,2),(2,2),(3,2),(3,3),(4,3),(5,3),(5,4)}

[B][I](1,1)      [O](2,1)         (3,1)         (4,1)         (5,1)

[B]   (1,2)   [B]   (2,2)   [B]   (3,2)      [O](4,2)         (5,2)

      (1,3)      [O](2,3)   [B]   (3,3)   [B]   (4,3)   [B]   (5,3)

      (1,4)      [O](2,4)         (3,4)         (4,4)   [B][F](5,4)
```

Figura 3: Visualização gráfica do melhor *path* obtido com o ficheiro *test_1.xml*.

4.2 Teste 2

Neste segundo teste, tendo o primeiro teste como base, foram alterados todos os parâmetros da simulação. O resultado do melhor *path* encontra-se representado na figura 4, onde se pode ver que um dos indivíduos conseguiu encontrar o mesmo caminho que no teste anterior. Não foi destacado nenhum erro com a alteração destes parâmetros.

4.3 Teste 3

No terceiro teste, em relação ao primeiro, foram alterados os parâmetros da morte, de reprodução e do movimento. Uma vez que neste caso o parâmetro de morte está com o valor 2 e os restantes possuem o valor 5, a probabilidade de gerar um evento mais cedo *EvDeath* é maior que a dos restantes. Este teste tem também como objetivo verificar se a aplicação estaria ou não preparada para a eventualidade de não haver nenhum

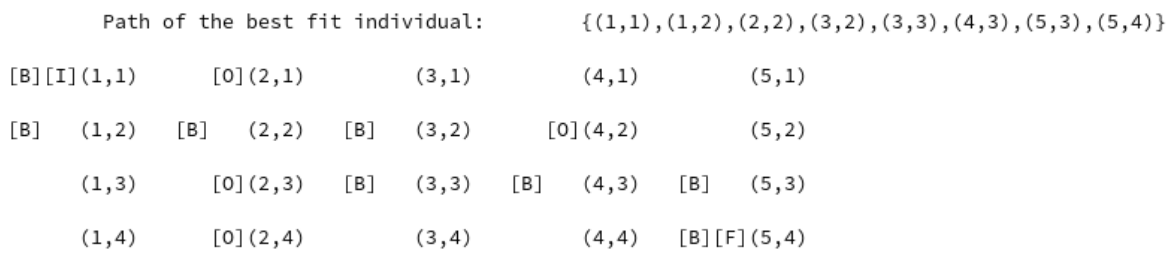

Figura 4: Visualização gráfica do melhor *path* obtido com o ficheiro *test_2.xml*.

best fit individual, bem como a validação de que os parâmetros estão a influenciar a simulação de acordo com o esperado, como se pode observar na figura 5.

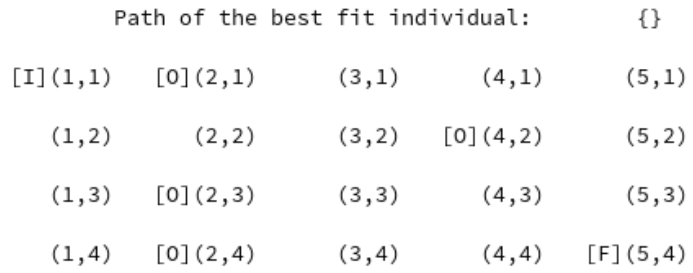

Figura 5: Visualização gráfica do melhor *path* obtido com o ficheiro *test_3.xml*.

4.4 Teste 4

No penúltimo teste, relativamente ao primeiro teste, foi alterado o tamanho da *grid* e o número de obstáculos. Também para esta situação, o *best fit individual* consegue atingir o ponto final, como se confirma com a figura 6.

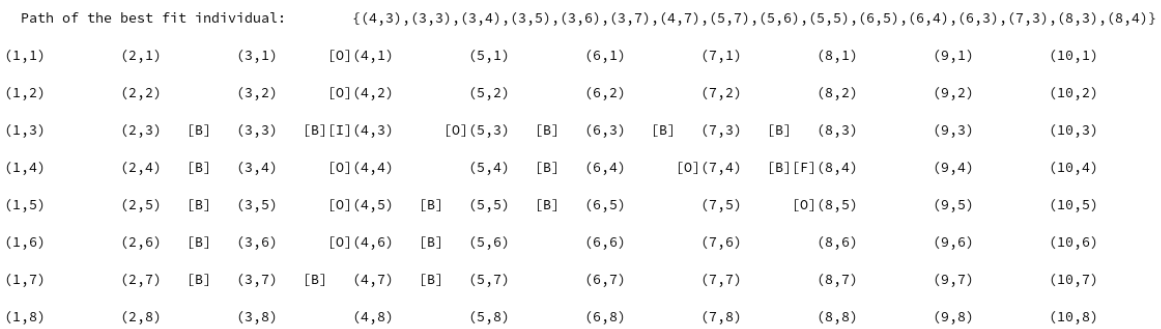

Figura 6: Visualização gráfica do melhor *path* obtido com o ficheiro *test_4.xml*.

4.5 Teste 5

Para o último teste foi tudo alterado. Como o tamanho da grelha é relativamente grande em relação aos anteriores testes e dado que os pontos final e inicial se encontram

nas extremidades da grelha, o *best fit individual* não atinge o ponto final. Também se verificou, com este teste, uma menor frequência na semelhança dos *paths* trilhados pelo indivíduo. Um dos resultados obtidos ao correr este teste pode ser observado na figura 7, e pode-se concluir que é possível alterar qualquer valor do documento, sem causar exceções ou outras situações problemáticas durante a execução da simulação.

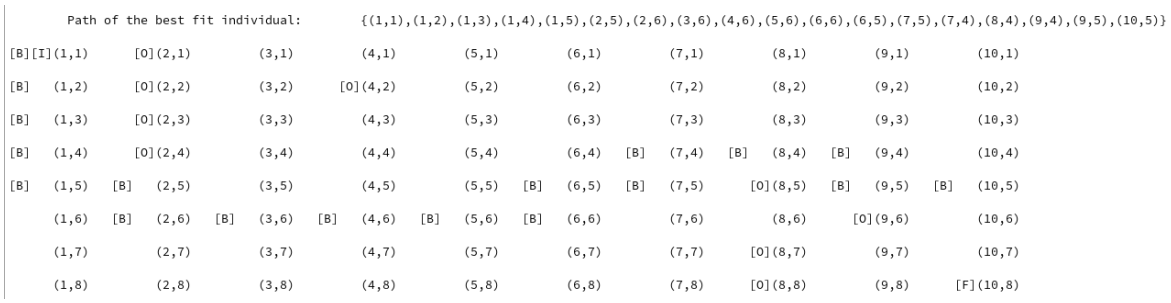

Figura 7: Visualização gráfica do melhor *path* obtido com o ficheiro *test_5.xml*.

5 Conclusão

Concluí-se que após a implementação do design previamente descrito, o problema inicial foi resolvido. O programa é capaz de encontrar o caminho com o menor custo entre o ponto final e inicial. O objetivo foi alcançado através do uso de caraterísticas de programação orientada por objetos tais como polimorfismo e herança, utilizado nas subclasses de *Event* e classes que implementaram interfaces desenvolvidas na solução, possibilitando a implantação de diferentes métodos com a mesma assinatura sem qualquer problema adicional. O código desenvolvido possibilita extensibilidade através das interfaces criadas e através da classe abstrata *Event*. As restantes classes podem ser herdadas mas algumas funcionalidades podem não ser utilizadas visto serem intrínsecas à solução desenvolvida, como o caso de *Individual* que é utilizada para apoio à *Population* ou a *Simulation* que é o núcleo do código, ou por serem classes primitivas que facilmente são reproduzidas.