Project baseado em ProjectLeic2016Tecnico

Abstract

Neste projecto vamos desenvolver vários algoritmos para resolver um jogo. O projecto terá 3 fases incluindo: i) criação de ferramentas básicas; ii) desenvolvimento de métodos de procura não guiada para problemas pequenos; iii) desenvolvimento de métodos de procura guiada e optimização do método de resolução de forma a conseguir resolver problemas de maior dimensão. O jogo é o VectorRacer

1 Descrição do jogo : VectorRacer

Este jogo VectorRacer (também chamado RaceTrack ou formula 1 de papel) é uma corrida de carros com uma dinâmica muito simplificada e que pode ser feito com caneta e papel quadriculado. Para uma descrição e história ver `https://en.wikipedia.org/wiki/Racetrack_(game)` aqui tambem https://youtu.be/sWBTLtVdBPE

1.1 Movimento e Regras

O movimento do carro é muito simples e respeita uma dinâmica de primeira ordem em que as acções são equivalentes a um conjunto de acelerações. Em cada instante é possível acelerar -1, 0 ou 1 unidades em cada direção. Ou seja o conjunto de acelerações em cada direção é $A = \{-1, 0, +1\}$ e portanto $a = (a_l, a_c)$ com $a_l, a_c \in A$.

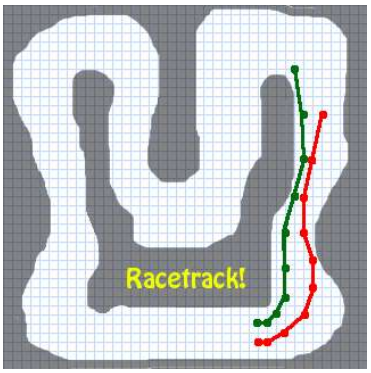

Figure 1: Exemplo de corrida com as primeiras jogadas de 2 jogadores.

Se o carro na jogada t esta na posição $p^t = (p_l, p_c)$ com a velocidade $v^t = (v_l, v_c)$, na jogada a seguir ele vai estar na posição:

$$p^{t+1} = p^t + v^t + a$$

Detalhadamente para cada direcção:

$$\begin{aligned} p_l^{t+1} &= p_l^t + v_l^t + a_l \\ p_c^{t+1} &= p_c^t + v_c^t + a_c \end{aligned}$$

Considera-se a notação `(linha,coluna)` para os vectores.
Considera-se a notação `(linha,coluna)` para os vectores.

Sendo $a = (a_l, a_c)$ a aceleração, a nova velocidade será então:

$$\begin{aligned} v_l^{t+1} &= v_l^t + a_l \\ v_c^{t+1} &= v_c^t + a_c \end{aligned}$$

Se o carro sair da pista ele volta a posição anterior com velocidade zero.

$$\begin{aligned} &\text{if not ontrack(p)} \\ &\qquad p^{t+1} = p^t \\ &\qquad v^{t+1} = (0,0) \end{aligned}$$

Quando o carro sai da pista há uma penalização de -20 pontos. A cada acção corresponde uma penalização de -1 ponto. Chegar a um dos estados da meta dá uma bonificação de 100 pontos.

A corrida termina quando o carro toca numa das posição marcada com meta, independentemente da velocidade.

O objectivo é chegar à meta obtendo a maior pontuação possível. Como as estratégias de procura estão orientadas para obter a solução de menor custo vamos considerar os valores simétricos dos apresentados acima para os custos das acções:

movimento 1

saída de pista 20

chegar à meta -100

1.3 Estruturas de dados

O ficheiro `datastructures.lisp` inclui a estrutura que define uma pista, a definição do estado do carro e a definição do problema que deverão ser usadas no projecto.

1.3.1 Definição de uma pista

```
;;; Definition of track
;;;  * size - size of track
;;;  * env - track where nil are obstacles, everything else is the track
;;;  * startpos - initial position
;;;  * endpositions - valid final positions
(defstruct track
  size
  env
  startpos
  endpositions)
```

O tamanho da pista é representado em Common Lisp por uma lista com dois elementos em que o primeiro elemento representa o número de linhas e o segundo o número de colunas.

A pista é representada por uma lista de listas. Cada elemento da lista representa uma linha completa em que cada elemento dessa lista representa o que está numa dada coluna. Os obstáculos são representados por NIL e as posições que podem ser usadas pelos carros por T.

A posição inicial é representada como uma lista de dois elementos em que o primeiro representa a linha e o segundo a coluna.

A linha da meta é representada por uma lista de posições.

O ficheiro `track0.lisp` inclui um exemplo do código lisp necessário para criar uma pista na representação descrita acima.

```
(setf *env* '((nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil)
              (nil nil nil nil nil t   t   t   t   t   t   t   t   t   nil nil nil nil)
              (nil nil nil nil nil t   t   t   t   t   t   t   t   t   t   nil nil nil)
              (nil t   t   t   t   t   t   nil nil nil nil nil t   t   t   t   t   nil)
              (nil t   t   t   t   t   nil nil nil nil nil nil nil t   t   t   t   nil)
              (nil t   t   t   t   nil nil nil nil nil nil nil nil nil t   t   t   nil)
              (nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil)))
```

A função `orderlistofcoordinates` poder-se-a revelar muito útil para se poderem comparar estructuras do tipo track.

Uma função que seja capaz de ler ficheiros de pista $*.txt$ e que transforme em *structtrack* será muito útil para poder testar pistas diferentes.

Devera existir uma função que leia ficheiros de texto com a representação externa de uma pista. É forncecido um exemplo de uma pista no ficheiro `track0.txt`

```
XXXXXXXXXXXXXXXXXX
XXXXX0000000000XXXX
XXXXX0000000000XXX
X000000XXXXX000EEX
XS0000XXXXXXX00EEX
X0000XXXXXXXXX0EEX
XXXXXXXXXXXXXXXXXX
```

Aqui 0 representa a pista, X são os obstáculos, S é a posição inicial e E são as posições finais.

1.3.2 Estado do carro

```
;;; State of the car
;;;  * pos - position
;;;  * vel - velocity
;;;  * action - action that was used to generate the state
;;;  * cost - cost of the action
;;;  * track - VectorRace track
;;;  * other - additional information
(defstruct state
  pos
  vel
  action
  cost
  track
  other)
```

A posição em que o carro se encontra é representada em Common Lisp como uma lista de dois elementos em que o primeiro representa a linha e o segundo a coluna, por exemplo (0 3) representa a posição na linha 0 coluna 3. A origem das coordenadas é a posição (0 0) e representa o canto superior esquerdo de uma pista.

A velocidade a que o carro se movimenta é representada em Common Lisp como uma lista de dois elementos em que o primeiro representa a velocidade na direcção horizontal e o segundo a a velocidade na direcção vertical. Velocidades positivas deslocam o carro em direcções nas quais o valor das coordenadas é crescente, enquanto que velocidades negativas deslocam o carro em direcções nas quais o valor das coordenadas é negativo.

A acção que deu origem ao estado é representada por uma lista de dois elementos em que o primeiro representa a acelaração na direcção horizontal e o segundo a acelaração na direcção vertical, tal como para a velocidade.

O custo representa o custo da acção que originou a transição de estado. Atenção aqui é o custo da acção e não o custo acumulado.

A pista é representada tal como descrito no ponto anterior.

O campo `other` pode servir para guardar outra informação que considere relevante.

1.3.3 Estrutura de um problema

```
;;; Definition of a problem
;;;  * initial-state
;;;  * fn-nextstate - function that computes the successors of a state
;;;  * fn-isGoal - funtion that identifies a goal state
;;;  * fn-h - heuristic function
(defstruct problem
  initial-state
  fn-nextStates
  fn-isGoal
  fn-h)
```

Um problema tem que ter definido qual o estado inicial.

Uma função que gera os sucessores de um estado, ou seja uma função que aplicada a um estado gera uma lista com todos os estados que podem ser atingidos a partir desse.

Uma função que permite identificar um estado objectivo em que o jogo terminou.

Para as procuras guiadas é necessário também uma função heurística que aplicada a um estado estima a distancia desse estado ao estado objectivo mais próximo.

2 1a Fase

2.1 Verificar se há um obstáculo numa dada posição da pista

Criar uma função `isObstaclep` que dada uma posição e uma pista devolve o valor lógico T se existir um obstáculo na posição e NIL no caso contrário.

```
"Exercise 1.1 - isObstaclep "
> (isObstaclep <pos> <track>)
NIL
```

2.2 Verificar se um estado é objectivo

Criar uma função `isGoalp` que dado um estado devolve o valor lógico T se o estado for objectivo e NIL caso não seja.

```
"Exercise 1.2 - isGoalp"
> (isGoalp <state>)
T
```

2.3 Calcular o estado seguinte dada uma acção

Criar uma função `nextState` que dado um estado e uma acção devolve o estado que resulta de aplicar a acção ao estado fornecido.

```
"Exercise 1.3 - nextState"
> (nextState <state> <action>)
<next state>
```