Relatório da Atividade 1: Logical Clock

Isabelle Ferreira de Oliveira
CES-27 - Engenharia da Computação 2020
Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos, Brasil
isabelle.ferreira3000@gmail.com

***Resumo*—Esse relatório documenta a implementação da simulação de processos rodando e trocando seus relógios lógicos entre si (Logical Clock definido por Lamport). Esses relógios foram tanto escalares quanto vetoriais.**

***Index Terms*—Relógio lógico, Relógio lógico escalar, Relógio lógico vetorial, algoritmo de Lamport**

I. Implementação

A. Tarefa 1: Relógio Lógico Escalar

A primeira etapa se tratou da implementação da simulação para o Relógio Lógico Escalar de Lamport, segundo o algoritmo descrito nos slides da aula e conforme o solicitado no roteiro da atividade. Essa implementação foi feita de forma bastante análoga à maneira das dicas fornecidas no roteiro.

Pode-se começar analisando-se a função main(), cujo código foi apresentado abaixo.

```
func main() {
  initConnections()

  // close connections when it's over
  defer ServConn.Close()
  for i := 0; i < nPorts; i++ {
    defer AllConn[i].Close()
  }

  // read "processID" from user input
  go readInput(ch)

  for {
    // Server
    go doServerJob()

    select {
    case processID, valid := <-ch:
      if valid {
        // update clock
        logicalClock++

        //Client
        if processID == myID {
          fmt.Println("logicalClock
              atualizado:", logicalClock)
        } else {
          fmt.Println("logicalClock
              enviado:", logicalClock)
          go doClientJob(processID,
              logicalClock)
        }
      } else {
        fmt.Println("Channel closed!")
      }
    default:
      time.Sleep(time.Second * 1)
    }
  }
```

Na main(), primeiramente são iniciadas as conexões de servidores e clientes, a partir da chamada de initConnections(). Nessa função, também é iniciado o relógio lógico desse processo em questão, além de serem setados seu ID, e as portas de todos os processos.

Após isso, é iniciada uma thread para ler as entradas do usuário a partir da função readInput().

Em seguida, é iniciado o loop de fazer o trabalho de servidor (ou seja, atualiza-se o relógio lógico caso chegue uma mensagem de outro processo, por meio da thread doServerJob()) e fica-se esperando uma mensagem do usuário no canal criado *ch*.

Ao se receber esse input do usuário, atualiza-se o relógio lógico e, a partir daí, duas ações podem ser tomadas a depender do conteúdo da entrada:

- faz-se o trabalho de cliente (enviando uma mensagem a um outro processo por meio da thread doClientJob()) caso a entrada seja o ID de outro processo;
- ou apenas imprime-se o valor atual do relógio lógico caso a entrada seja o próprio ID desse processo.

Abaixo, segue-se o código dessa função initConnections().

```
func initConnections() {
  nPorts = len(os.Args) - 2

  // my process
  logicalClock = 0
  auxMyID, err := strconv.Atoi(os.Args[1])
  myID = auxMyID
  myPort = os.Args[myID+1]

  // Server
  ServerAddr, err :=
      net.ResolveUDPAddr("udp", myPort)
  aux, err := net.ListenUDP("udp", ServerAddr)
  ServConn = aux

  // Clients
```

```
  for i := 0; i < nPorts; i++ {
    aPort := os.Args[i+2]

    ServerAddr, err :=
        net.ResolveUDPAddr("udp","127.0.0.1"
        + aPort)
    LocalAddr, err :=
        net.ResolveUDPAddr("udp",
        "127.0.0.1:0")
    auxConn, err := net.DialUDP("udp",
        LocalAddr, ServerAddr)
    AllConn = append(AllConn, auxConn)
  }
}
```

Vale ressaltar também que, para esse código e todos os outros dessa atividade, sempre após a setagem da variável *err*, referente a um possível erro advindo de algumas funções, também era chamada a função CheckError(err), que imprime o erro e interrompe o processo caso houvesse algum erro.

Essas chamadas de funções foram suprimidas do relatório a fim de simplificar a apresentação dos códigos, e por entender-se que não se trata da ideia principal dos códigos desenvolvidos.

A função readInput() segue bem semelhante àquela apresenta na Dica 3 do roteiro, com a diferença de aceitar um canal de inteiro ao invés de um canal de string. Assim, a função é capaz de ler o ID que o usuário digitar.

Já a função doServerJob(), apresentada abaixo, também segue bem semelhante à apresentada na função main() do código do servidor fornecido na Dica 1. A diferença está na retirada do loop *for* e do fechamento da conexão, uma vez que essas etapas se equivalem aos apresentados na main() da atividade (função já apresentada acima). Outra diferença também é a impressão do relógio lógico recebido por mensagem e, em seguida, a impressão do valor atualizado. Segue abaixo o código descrito.

```
func doServerJob() {
  buf := make([]byte, 1024)

  n, _, err := ServConn.ReadFromUDP(buf)

  aux := string(buf[0:n])
  otherLogicalClock, err := strconv.Atoi(aux)
  fmt.Println("Received", otherLogicalClock)

  // updating logical clock
  logicalClock = max(otherLogicalClock,
      logicalClock) + 1
  fmt.Println("logicalClock atualizado:",
      logicalClock)
}
```

Por fim, a função doClientJob() também seguiu de forma semelhante ao código apresentado na função main() do código do cliente fornecido na Dica 1. As alterações também foram semelhantes: retirouse o loop *for* e o fechamento da conexão. Além diso, o conteúdo da mensagem a ser enviada foi alterado para o relógio lógico atual do processo em questão. Segue abaixo o código descrito.

```
func doClientJob(otherProcessID int,
    logicalClock int) {
  otherProcess := otherProcessID - 1

  msg := strconv.Itoa(logicalClock)
  buf := []byte(msg)

  _,err := AllConn[otherProcess].Write(buf)

  time.Sleep(time.Second * 1)
}
```

B. Tarefa 2: Relógio Lógico Vetorial

Já essa etapa se tratou da implementação do método act() também da classe DQNAgent de dqn_agent.py. Nesse método, era escolhido e retornado uma ação de acordo com a política ϵ-greedy.

Essa implementação foi feita de forma bastante análoga à maneira do laboratório 12 [3]. Assim, gerou-se um número aleatório entre 0 e 1 e, caso esse valor aleatório seja menor que epsilon, então uma ação aleatória é escolhida; caso contrário, é escolhida a ação gulosa, através do retorno do índice do máximo elemento do array *model.predict(state)[0]*.

C. Reward Engineering

Nesse momento, foi implementado o método reward_engineering_mountain_car() de utils.py, script também fornecido no código base do laboratório. Nesse método, eram calculadas e retornadas as recompensas intermediárias "artificias", chamadas reward engineering, a fim de tornar o treino mais rápido no ambiente do Mountain Car.

Essa implementação foi feita conforme as equações apresentadas na seção 4.3 do roteiro do laboratório [1], ou seja, assim como apresentado no pseudo-código em Python a seguir.

```
package main

import "fmt"

func main() {
   fmt.Println("Hello World!")
}
```

Os valores de position, start, velocity e next_position também eram fornecidos no roteiro [1], e bastava substituí-los no pseudo-código acima.

D. Treinamento usando DQN

Bastava treinar o modelo implementado, executando o script train_dqn.py, também do código base, e observar os resultados e os gráficos obtidos.

E. Avaliação da Política

Bastava aplicar o modelo implementado no ambiente do Mountain Car, executando o script evaluate_dqn.py, também do código base, e observar a simulação, os resultados e os gráficos obtidos.

II. Resultados e Conclusões

O summary do modelo implementado em make_model() foi apresentado na Figura 1, e condiz com os requisitos pedidos na Tabela 3 do roteiro do laboratório [1].

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 24)                72
_________________________________________________________________
dense_2 (Dense)              (None, 24)                600
_________________________________________________________________
dense_3 (Dense)              (None, 3)                 75
=================================================================
Total params: 747
Trainable params: 747
Non-trainable params: 0
```

Figura 1. Sumário do modelo implementado em Keras.

.

Já a Figura 2 representa as recompensas acumulativas advindas do treinamento do modelo em 300 episódios. Esse resultado dependem diretamente da correta implementação e funcionamento dos métodos make_model() e act().

Pode-se dizer que esse gráfico condiz com o esperado, uma vez que é possível notar inicialmente recompensas pequenas para os primeiros episódios e, mais ou menos a partir do episódio 80, tornou-se frequente recompensas com valores elevados, chegando a valores próximos de 40, indicando um aprendizado significantemente correto.

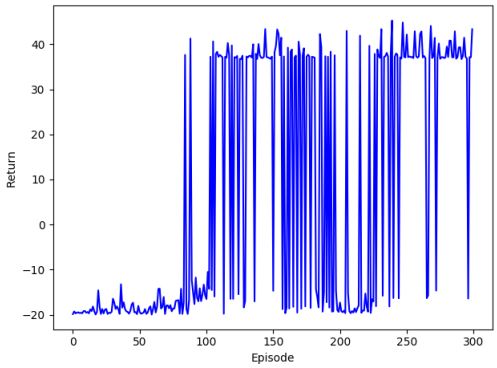

Figura 2. Recompensa acumulativa com o passar dos episódios, no treinamento do modelo para 300 episódios.

.

Já a aplicação do modelo implementado no ambiente do Mountain Car gerou as Figuras de 3 a 5.

A partir da Figura 3, pode-se concluir que a implementação e treino chegaram em resultados satisfatórios, uma vez que grande parte das recompensas acumuladas foi alta, próximas de 40, chegando no final de 30 episódios a uma média de 27.8, conforme apresentado na Figura 4.

Por fim, acerca da Figura 5, pode-se observar que:

- Para velocidades para direita, quase unanimamente a decisão do carro é continuar para direita. Exclui-se disso as situações de posição muito à esquerda e velocidades altas, na qual é decidido fazer nada, e de velocidades para direita muito baixas, na qual pouquíssimas vezes o carro decide ir para esquerda, talvez já se enquadrando nas intenções descritas no próximo item.
- Para velocidades para esquerda, as decisões do carro diferem bastante da posição na qual ele se encontra. Para posições mais a esquerda, o carro decide continuar indo para esquerda, talvez para pegar impulso da subida e, quando por fim chegar em posições mais a esquerda (consequentemente mais altas) possíveis, decidir ir com velocidade para direita. Já para posições relativamente próximas da posição objetivo, aparecem também decisões de não fazer nada, indicando que o carro irá mais para esquerda e cairá na situação anteriormente descrita, na qual ele decidirá continuar indo para esquerda e pegará o impulso da elevação.

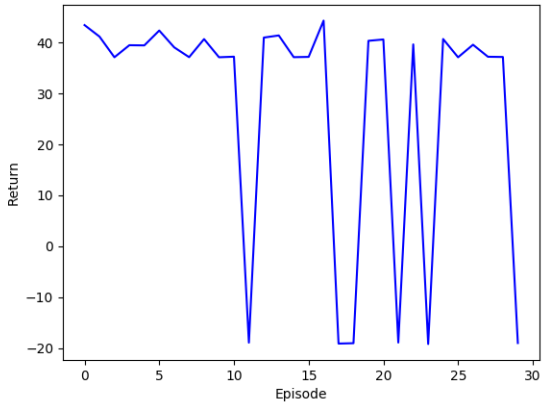

Figura 3. Representação em cores da tabela de action-value calculada, para algoritmo de Sarsa.

.

```
episode: 1/30, time: 107, score: 43.4619, epsilon: 0.0
episode: 2/30, time: 94, score: 41.2016, epsilon: 0.0
episode: 3/30, time: 159, score: 37.1377, epsilon: 0.0
episode: 4/30, time: 85, score: 39.5076, epsilon: 0.0
episode: 5/30, time: 84, score: 39.4702, epsilon: 0.0
episode: 6/30, time: 101, score: 42.3907, epsilon: 0.0
episode: 7/30, time: 83, score: 39.0879, epsilon: 0.0
episode: 8/30, time: 160, score: 37.1565, epsilon: 0.0
episode: 9/30, time: 91, score: 40.7103, epsilon: 0.0
episode: 10/30, time: 159, score: 37.138, epsilon: 0.0
episode: 11/30, time: 167, score: 37.2522, epsilon: 0.0
episode: 12/30, time: 200, score: -18.9488, epsilon: 0.0
episode: 13/30, time: 92, score: 41.0069, epsilon: 0.0
episode: 14/30, time: 95, score: 41.4258, epsilon: 0.0
episode: 15/30, time: 160, score: 37.1562, epsilon: 0.0
episode: 16/30, time: 163, score: 37.2112, epsilon: 0.0
episode: 17/30, time: 113, score: 44.3414, epsilon: 0.0
episode: 18/30, time: 200, score: -19.131, epsilon: 0.0
episode: 19/30, time: 200, score: -19.0591, epsilon: 0.0
episode: 20/30, time: 89, score: 40.3713, epsilon: 0.0
episode: 21/30, time: 90, score: 40.6353, epsilon: 0.0
episode: 22/30, time: 200, score: -18.9305, epsilon: 0.0
episode: 23/30, time: 86, score: 39.6682, epsilon: 0.0
episode: 24/30, time: 200, score: -19.2182, epsilon: 0.0
episode: 25/30, time: 91, score: 40.7233, epsilon: 0.0
episode: 26/30, time: 160, score: 37.1333, epsilon: 0.0
episode: 27/30, time: 85, score: 39.6013, epsilon: 0.0
episode: 28/30, time: 165, score: 37.2369, epsilon: 0.0
episode: 29/30, time: 161, score: 37.1958, epsilon: 0.0
episode: 30/30, time: 200, score: -19.0234, epsilon: 0.0
Mean return:  27.79701181215042
```

Figura 4. Recompensa acumulada em função das iterações, para algoritmo de Sarsa.

.

Como as decisões aprendidas e tomadas pelo carro fizeram sentido e puderam ser interpretadas satisfatoriamente, pode-se dizer que a proposta do laboratório foi corretamente im-

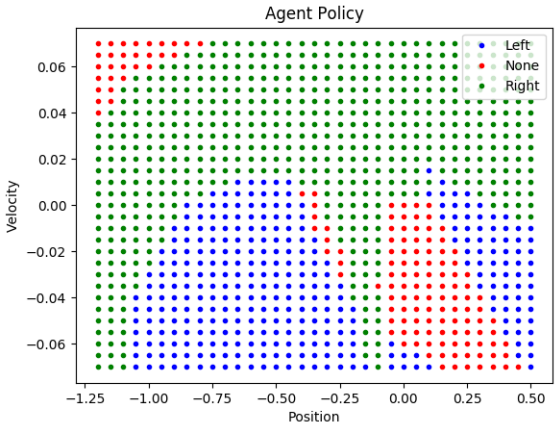

Figura 5. Representação em cores da tabela de greedy-policy calculada, para algoritmo de Sarsa.

.

plementada e se mostrou satisfatória em resolver o problema proposto.

REFERÊNCIAS

[1] M. Maximo, “Roteiro: Laboratório 12 - Deep Q-Learning”. Instituto Tecnológico de Aeronáutica, Departamento de Computação. CT-213, 2019.

[2] M. Maximo, “Roteiro: Laboratório 8 - Imitation Learning com Keras”. Instituto Tecnológico de Aeronáutica, Departamento de Computação. CT-213, 2019.

[3] M. Maximo, “Roteiro: Laboratório 12 - Aprendizado por Reforço Livre de Modelo”. Instituto Tecnológico de Aeronáutica, Departamento de Computação. CT-213, 2019.