Projekt inteligentnej śmieciarki

Wojciech Jarmosz, Paweł Stelmach, Hubert Salwin

March 27, 2019

1 Zaprojektowanie środowiska agenta i reprezentacja wiedzy.

Środowisko agenta zostało zaprojektowane przy pomocy biblioteki PyGame. Podstawową strukturą przechowującą siatkę (grida) jest lista list, która zawiera numerację odpowiednich typów komórek (droga, trawa, śmieci). Na początku zerujemy tę strukturę, a następnie losujemy drogi poziome i pionowe w taki sposób, aby żadne dwie drogi nie były do siebie przyległe. Losujemy także śmietniki - domostwa. Elementy wygenerowanej macierzy grida, można opisać następująco:

- **0** - node opisujący trawę
- **1** - node opisujący drogę poziomą
- **2** - node opisujący drogę pionową
- **3** - node opisujący skrzyżowanie dróg
- **4** - node opisujący śmietnik
- **11** i **13** - node opisujący śmieciarkę w poziomie (druga wartość opisuje przypadek gdy śmieciarka znajdzie się na skrzyżowaniu dróg)

- **12** - node opisujący śmieciarkę w pionie

Na tej podstawie za pomoca PyGame generujemy widok całej mapy - odpowiednim numerom komórek przyporządkowywane są odpowiednie sprity. Agent nie ma możliwości przechodzenia przez domy, przez trawę, oraz nie ma możliwości wyjścia poza mapę, może on poruszać się tylko po drogach.

Projekt inteligentnej śmieciarki gr. 4

Wojciech Jarmosz, Paweł Stelmach, Hubert Salwin

May 20, 2019

1 Algorytmy ruchu

1.1 DFS

Aktualna lista ruchów jest przechowywana w zmiennej globalnej - **move_list**. Główna metoda - **create_tree(self, current_move_list, current_grid, recursion_depth)**, **Current_move_list** - aktualna lista ruchów, która po zakończeniu rekurencji jest porównywana z globalną listą ruchów (jeżeli wszystkie śmieci zostały zebrane), **Current_grid** - kopia aktualnej mapy - skopiowana przy użyciu funkcji **copy.deepcopy**, **Recursion_depth** - głębokość rekurencji - by uniknąć zapętlenia.

Funkcja najpierw sprawdza czy wszystkie śmieci zostały zebrane przechodząc przez całą listę i sprawdzając czy w dalszym ciągu istnieją śmieci danego rodzaju.

Następnie sprawdza czy któreś z przylegających pól zawiera śmietnik danego rodzaju jeżeli tak to zbiera pierwszy od lewej do momentu gdy wszystkie śmietniki wokół niego będą puste.

Jeżeli wszystkie pola są puste to wywołuje funkcję w następującej kolejności - lewo, dół, prawo, góra.

Gdy zajdzie w ślepą uliczkę, gdzie nie ma żadnego śmietnika algorytm się kończy

1.2 A*

Algorytm A* bazuje w dużej mierze na metodach algorytmu DFS, z kilkoma różnicami. Po pierwsze, po znalezieniu naszego celu - śmietnika, wyszukiwany jest nowy, najbliższy cel do którego będzie podążała ciężarówka. Jest on wyznaczany za pomocą funkcji heurystyki:

```
def count_heuristic_from_a_to_b(self, current_x, current_y, goal_x, goal_y):
      return abs(goal_x - current_x) + abs(goal_y - current_y)
```

Jest to tzw. dystans manhattanowy. Po drugie, algorytm w odróżnieniu do DFS nie przyjmuje możliwości ruchu w czterech kierunkach, a jedynie w tym który posiada minimalną wartość sumy: **g(x)** + **h(x)**, gdzie **g(x)** jest funkcją wag a **h(x)** funkcją heurystyki Funkcja wagi **g(x)** liczona jest na podstawie ilości możliwych dróg do danego celu - najkorzystniejsza droga posiada możliwie najmniej potrzebnych ścieżek do celu. Algorytm rekurencyjnie sprawdza za każdym razem czy wokół ciężarówki nie ma śmietnika, jeżeli jest, to zbiera go i wyznacza nowy, najbliższy cel względem aktualnej pozycji ciężarówki oraz generuje ruch do listy **move_list**.

1.3 BFS

Do przechowywania współrzędnych kolejnych możliwych ruchów ciężarówki używana jest kolekcja deque, która pozwala na łatwe i szybkie dodawanie i usuwanie elementów z kolejki. W deque przechowywane są współrzędne punktu wraz z ścieżką, jaka do niego prowadzi. Algorytm działaniem przypomina DFS, z tą różnicą, że za każdym razem, gdy BFS jest rekurencyjnie wywoływany, odwiedzane są wszystkie sąsiadujące, nieodwiedzone pola, a nie tylko jedno.

Implementacja drzewa decyzyjnego

Wojciech Jarmosz, Gr. 4

June 11, 2019

1 Implementacja drzewa decyzyjnego w projekcie inteligentnej śmieciarki

Implementacja metody uczenia maszynowego - drzewa decyzyjnego wymagała stworzenia przeze mnie parsera danych uczących. Odpowiada za to klasa DataParser w pliku **data_parser.py**. Parser danych działa w następujący sposób:

1.1 Generuje nową mapę z której pozyskamy dane uczące.

1.2 Uruchamia na tej mapie algorytm BFS.

1.3 Uzyskujemy listę węzłów - najkrótszą ścieżkę na mapie która odwiedza wszystkie śmietniki.

1.4 Wycinamy z mapy kwadraty o wymiarach 5 x 5, reprezentujące dany stan na podstawie kolejno odwiedzanych węzłów.

1.5 Każdemu polu nadajemy odpowiednią etykietę, w zależności czy jest śmietnikiem, trawą, czy drogą.

1.6 Do sparsowanego stanu dodajemy etykietę ruchu który powinna wykonać śmieciarka.

1.7 Proces powtarza się dla 150 map, co daje nam około 5500 danych uczących.

1.8 Tak przygotowane dane zapisywane są do pliku.

Kiedy dane są już przygotowane, możemy przystąpić do uczenia drzewa decyzyjnego.

```
from sklearn import tree
import json


class DecisionTree:

    X = []
    Y = []
    clf = 0

    def __init__(self):
        self.clf = tree.DecisionTreeClassifier()

    def read_data_from_file(self):
        with open('learning_data.txt', 'rb') as f:
            while True:
                line = f.readline()
                if not line:
                    break
                result = self.process_line(line)

                self.Y.append(result[0][0])
                self.X.append(result[1])
        f.close()

    def process_line(self, line):
        line = line.rstrip().decode('utf-8')
        params = line.split(" ", 1)
        params[0] = json.loads(params[0])
        params[1] = json.loads(params[1])
        return params

    def learn_tree(self):

        self.read_data_from_file()
        self.clf = self.clf.fit(self.X, self.Y)

    def predict_result(self, square):
        return self.clf.predict([square])
```

Klasa DecisionTree implementuje bibliotekę SciPy, odczytuje dane z pliku. Macierz opisująca stan mapy wraz z etykietą ruchu jest przekazywana do drzewa decyzyjnego.

Klasa VisualizeDT zajmuje się wizualizacją uczenia maszynowego. Generuje ona mapę, odczytuje kwadrat stanu względem pozycji cieżarówki a następnie odpytuje drzewo decyzyjne wywołując metodę **predict_result** i otrzymuje ruch który wizualizuje na mapie. W przypadku gdy zwrócony ruch nie jest możliwy do wykonania, losujemy inny, ktory jest możliwy.

VowpalWabbit

Przygotowanie danych**: Hubert Salwin**

moim zadaniem było napisanie parsera, który przetwarza stan mapy na format nadający się do metody uczenia jaką jest vowpal wabbit. Na początku tworzę mapę o określonej przez użytkownika rozdzielczości i umieszczam na niej ciężarówkę. Potem za pomocą algorytmu BFS znajduję optymalną drogę ciężarówki pomiędzy wszystkimi śmietnikami na mapie i przekazuję do parsera listę ruchów. Następnie po każdym ruchu wycinam to co znajduje się w polu widzenia ciężarówki oraz ruch, który według BFS powinna wykonać ciężarówka. Pole widzenia ciężarówki to kwadrat nxn której centralnym punktem jest ciężarówka i może być w każdej chwili zmieniony na dowolną liczbę nieparzystą >= 3. Na początku odpowiednio dodaję do mapy "obwódkę" z trawy o grubości zależnej od pola widzenia, tak, żeby nie zostało wycięte pole widzenia spoza indeksu mapy, gdy ciężarówka znajduje się blisko jej krawędzi. Informacje są gromadzone i zapisywane do pliku z dowolnej liczby losowo generowanych map

Uczenie vowpala i wyświetlanie rezultatu: **Pawel Stelmach**

Dane postaci:
2 | f0:1 f1:1 f2:1 f3:1 f4:1 f5:2 f6:2 f7:1 f8:0 f9:1 f10:0 f11:0 f12:1 f13:3 f14:1 f15:0 f16:3 f17:1 f18:3 f19:1 f20:1 f21:1 f22:1 f23:1 f24:1
Gdzie pierwsza liczba jest ruchem (np: 2 zbiera śmieci)
Po zebraniu danych z 25000 map gdzie jedna mapa posiada około 100 ruchów i wybraniu najlepszej kombinacji oznaczania ruchów, wycinania kwadratów z mapy oraz trenowania powstał plik trained_5, ważący(95 mb), w którym zapisywane są ruch od 0 do 4 gdzie 0,1,2,3,4 to kolejna ruch w górę, dół,zbieranie,prawo,lewo. Z mapy wycinane są kwadraty 5x5
Niestety z założeniami jakie mamy Nie udało się dobrze wytrenować vowpala ponieważ często nie odróżnia różnicy pomiędzy ruchem ruchami mimo że pula danych była bardzo duża.
By śmieciarka się nie blokowała dodałem losowanie ruchu w przypadku gdy waga poprzedniego ruchu jest taka sama jak waga aktualnego ruch jednak śmieciarka może w dalszym ciągu się zaklinować

1. Przygotowanie danych:
```
$ vw vowpal_data.txt -l 1 -c --passes 100 --holdout_off -f trained_5
```
2. Wczytywanie wytrenowanego vowpala do pythona
```
vw = pyvw.vw("--quiet -i trained_5")
```
3. Przedstawienie aktualnej pozycji w sposób zrozumiały dla vowpala
4. Obliczania wagi na podstawie przygotowanego wcześniej pliku
```
def my_predict(vw, s):
    ex = vw.example(s)
    pp = 0.
    for f,v in ex.iter_features():
        pp += vw.get_weight(f) * v
    return pp
```

5. Wykonanie ruchu na podstawie obliczonej wagi

```
def calculate_move(wage):
    if wage >= 1.5 and wage <= 2.4:
        return 'collect'
    elif wage >=2.4 and wage <=3.3:
        return Move.MOVE_LEFT
    elif wage >=3.3:
        return Move.MOVE_RIGHT
    elif wage <=0.5:
        return Move.MOVE_TOP
    elif wage >=0.5 and wage <=1.5:
        return Move.MOVE_DOWN
```

Podsumowanie projektu

Początkowe założenia projektu były następujące:
śmieciarka ma wyjechać i zebrać wszystkie śmieci wybranego koloru, następnie ma wrócić do stacji puste śmietniki zostaną uzupełnione, a śmieciarka ponownie wyjeżdża zbierać tym razem inny rodzaj śmieci
niestety nie udało się nam tego osiągnąć mimo, że zrezygnowaliśmy z wielu funkcji nauczona śmieciarka dalej nie funkcjonuje w pełni poprawnie często nie może odróżnić ruchów od siebie nawet przy bardzo dużej puli danych. gdybym jeszcze raz miał tworzyć ten projekt postarałbym się zamodelować dane w taki sposób by każdy ruch był dosyć odróżniały od siebie co pozwoliłoby algorytmom uczącym na lepsze rozpoznanie poszczególnych ruchów.