Podstawy Sztucznej Inteligencji

Projekt PW.P.2. - Wspinaczka

1 Prowadzący : dr inż. Paweł Wawrzyński

Wykonanie:
Krystian Czapiga
Mariusz Kuc
Konrad Napiórkowski

Treść zadania:

PW.P.2. Wspinaczka

Patrz: zasady ogólne, PW. Zasady i zalecenia.

Program losuje n punktów (uchwytów) na ścianie o wysokości W oraz szerokości 3m. Dodatkowo, na dole ściany umieszcza dwa sztuczne punkty początkowe i na górze ściany dwa punkty końcowe. Dla każdego punktu losuje jego trudność z rozkładu jednostajnego na przedziale [1,2]. Następnie, program korzysta z algorytmu A* do znalezienia najlepszej drogi wszystkich czterech kończyn wspinacza między punktami początkowymi i końcowymi. Ruch wykonany przez wspinacza polega na przemieszczeniu jednej z kończyn między uchwytami. Musi on spełniać następujące warunki:

– punkty, których trzymają się ręce wspinacza są powyżej punktów, których trzymają się nogi (choć dłonie ani nogi niekoniecznie muszą być na różnych punktach),

– największa odległość dzieląca którąś parę kończyn nie może przekraczać 2m.

Koszt ruchu równy jest sumie kosztów uchwytów, na których znajdują się po ruchu kończyny wspinacza.

Interfejs. Prosty interfejs graficzny pokazujący, w formie prostej animacji (ze wspinającym się pająkiem), znajdowane rozwiązanie i przedstawiający podstawowe statystyki dotyczące poszukiwań (głębokość, najlepszy znaleziony koszt dotarcia etc.).

Sprawdzić program dla różnych wartości W oraz n.

Dokumentacja

1 Zestawienie kluczowych decyzji projektowych.

- Projekt wykonany w języku Java.
- Wykorzystanie wzorca projektowego Model-View-Controller.
- Zaimplementowanie dwóch trybów pobierania danych wejściowych. Tryb losowy – wymagany przez założenia projektu, tryb wczytywania z pliku – przydatny przy testowaniu.
- Proste GUI napisane z wykorzystaniem bibliotek AWT i Swing.

2 Instrukcja uruchomienia programu

Tryb losowy

Po uruchomieniu programu należy podać parametry n i W w odpowiednich polach interfejsu.

Wpisz liczbę losowanych uchwytów **n** w pole pod przyciskiem **START**, a wysokość ściany **W** w pole poniżej (po najechaniu wyświetli się informacja, jakie pole należy wpisać w dane miejsce). Liczba uchwytów musi być dodatnią liczbą całkowitą, a wysokość ściany dodatnią liczba rzeczywistą. Po wpisaniu parametrów należy nacisnąć przycisk **START.**

Tryb wczytywania z pliku

Po uruchomieniu programu należy wybrać przycisk **Open File** i wybrać plik z danymi wejściowymi.

Plik wejściowy musi mieć następujący format:

n W

x_1 y_1 $cost_1$ }

… } n razy

x_n y_n $cost_n$ }

gdzie: n – liczba uchwytów, dodatnia całkowita;

W – wysokość ściany, dodatnia rzeczywista;

x_i – współrzędna x i-tego uchwytu, rzeczywista z przedziału [0;3]

y_i – współrzędna y i-tego uchwytu, rzeczywista z przedziały [0;W] gdzie $i<j <=> y_i< y_j$ tzn. uchwyty posegregowane rosnąco względem y;

$cost_i$ – koszt dotarcia do uchwytu, rzeczywisty z przedziału [1;2]

Po wybraniu pliku należy nacisnąć przycisk **START.**

Dalsze działanie

Po naciśnięciu przycisku **START** interfejs zawiesi się na czas działania algorytmu. Jeżeli uda się znaleźć rozwiązanie w środkowym panelu wyświetli się graficzna reprezentacja ściany oraz wspinacza. Przyciskami **NEXT** i **PREV** przechodzimy po kolejnych (poprzednich) stanach wspinacza.

Jeżeli znalezienie rozwiązania nie powiedzie się ściana pozostanie pusta.

3 Struktura programu

Głównym elementem modelu problemu jest ściana zawierająca uchwyty po których będzie wspinał się wspinacz oraz implementacja algorytmu A*.

```
public class Model
{
	private  Wall wall;

	public Model()

	public List<Grip> getFeasibleGrips(final State current, LEG activeLeg)
	public final WallMockup getWallMockup(State currentState)
	public List<Grip> getGripsBetween(double lowerBoud, double upperBound)

	public final Wall getWall()
	public void setWall(Wall wall)
}
```

Klasa Model zawiera tylko jedno pole typu Wall reprezentujące ścianę.

Posiada konstruktor inicjalizujący. Metody `getFeasibleGrips, getWallMockup, getGripsBetween,` wołają analogiczne metody z klasy Wall. Posiada setter i getter pola wall.

```
public class Wall
{
	private final List<Grip> grips; // uchwyty na scianie
	private final int n;    // ilosc uchwytow na scianie
	private final double w; // wysokosc sciany
	private final State start; //stan poczatkowy
	private final State goal; // stan koncowy

	public Wall()
	public Wall(List<Grip> grips, final int n, final double w)

	public List<Grip> getFeasibleGrips(final State current, LEG activeLeg)
	private boolean isStillProbable(State current, LEG activeLeg, int i)
	private int firstProbableGrip(State current, LEG activeLeg)
	public WallMockup getWallMockup(State currentState)
	public List<Grip> getGripsBetween(double lowerBound, double upperBound)

	public final int getN()
	public final double getW()
	public final List<Grip> getGrips()
	public final State getStart()
	public final State getGoal()
}
```

Klasa Wall reprezentuje w modelu główny element problemu czyli ścianę. Zawiera listę uchwytów w polu grips typu List<Grip>, wie ile uchwytów zawiera (pomijając dodatkowe uchwyty startowe i końcowe). Zna swoją wysokość. Ściana zna również stany początkowy i końcowy wspinacza tzn. wie gdzie zaczyna się wspinaczka i gdzie kończy.

Posiada dwa konstruktory. Inicjalizujący pustą ścianę i tworzący wypełnioną ścianę. Metody `getFeasibleGrips, firstProbableGrip, isStillProbable` współpracują z algorytmem A* dostarczając możliwe do złapania uchwyty. Metody `getWallMockup, getGripsBetween` współpracują z modułem View. Posiada stosowne settery i gettery.

```
public class Grip
{
	private int idGrip;
	private final double x;
	private final double y;
	private final double cost;

	public Grip(int idGrip, double x, double y, double cost)

	public boolean isFeasible(State current, LEG activeLeg)
	public boolean isInReach(Grip legGrip)
	public double distance(Grip grip)

	@Override
	public int hashCode()
	@Override
	public boolean equals(Object obj)
	@Override
	public String toString()

	public int getIdGrip()
	public final void setIdGrip(int idGrip)
	public double getX()
	public double getY()
	public double getCost()
}
```

Klasa Grip reprezentuje pojedynczy uchwyt na ścianie. Posiada pola reprezentujące współrzędne x i y oraz kosz dotarcia do tego uchwytu (dowolną kończyną z dowolnego innego uchwytu). Dodatkowo zna swój numer id.

Posiada konstruktor inicjalizujący. Metody `isFeasible, isInReach, distance` współpracują ze stosownymi metodami klasy Wall na rzecz algorytmu A*. Pozostałe to standardowe przeciążone metody.

```
public class AStar implements Algorithm
{
	private final Model model;
	private final Set<State> openSet;
	private final Set<State> closedSet;
	private final Map<State, Double> gScore;
	private final Map<State, Double> fScore;


	public AStar(Model model)

	@Override
	public Path findPath(AbstractState start, AbstractState goal)

	private Map<State, Double> createNeighbourStates(State current)
	private void createNeighbourStatesForOneLeg(Map<State, Double>
neighbourStates, List<Grip> feasibleGrips, State currentState, LEG activeLeg)
	private Double calculateHeuristicCost(State start, State goal)
}
```

Klasa Astar implementuje interfejs Algorithm i jest realizacją algorytmu A*. Posiada pole model dostarczające „problem” do rozwiązania. Pola openSet i closedSet to zbiory typu HashSet zawierające obiekty typu State czyli możliwe stany wspinacza.

Posiada konstruktor inicjalizujący. Metoda `findPath` implementuje algorytm A*. Pozostałe metody współpracują.

```
public class State implements AbstractState
{
    // mapa wskazujaca ktora konczyna jest na ktorym uchwycie, kolejnosc
LEFT_HAND, RIGHT_HAND, LEFT_FOOT, RIGHT_FOOT
    private final Map<LEG, Grip> legState;
    // stan z ktorego przyszlismy do biezacego stanu
    private State previous;
    //stan nastepny do ktorego pojdzie wspinacz
    private State next;
    //koszt dojscia do tego stanu
    private double ownCost;
    //numer stanu w gotowej sciezce
    private int numberOfState;

    public State()
    public State(Map<LEG, Grip> legState, State previous)

    public boolean areHandsOnTheSameGrip()
    public boolean areFeetOnTheSameGrip()

    @Override
    public int hashCode()
    @Override
    public boolean equals(Object obj)
    @Override
    public String toString()
    public State getPrevious()
    public void setPrevious(State previous)
    public Grip getLegGrip(LEG activeLeg)
    public final State getNext()
    public final void setNext(State next)
    public final double getOwnCost()
    public final int getNumberOfState()
    public final void setOwnCost(double ownCost)
    public final void setNumberOfState(int numberOfState)
}
```

Klasa State implementuje interface AbstractState. Reprezentuje bieżące położenie wspinacza na ścianie. Dodatkowo zna następny i poprzedni stan wspinacza oraz koszt jaki poniósł aby dostać się w to miejsce.

Posiada dwa konstruktory. Inicjalizujący pusty obiekt oraz wypełniający podstawowe informacje. Metody `areHandsOnTheSameGrip i areFeetOnTheSameGrip` współpracują z algorytmem A*. Pozostałe metody to standardowe przeciążenia oraz odpowiednie settery i gettery.

4 Wnioski

Program działa i spełnia wszystkie założenia. W związku z charakterystyką działania algorytmu A* i zadanego problemu działanie programu jest dość wolne i silnie zależy od wybranych parametrów. W zadanym problemie wspinacz może przemieścić kończynę na dowolny uchwyt znajdujący się w jego zasięgu (czyli w odległości nie większej niż 2 metry od wszystkich innych kończyn). To znaczy, że w tym zasięgu kolejne Stany wspinacza mogą reprezentować dowolne posunięcie dowolną kończyną w dowolny uchwyt. Ilość rozpatrywanych stanów rośnie geometrycznie. Zbyt gęste rozmieszczenie uchwytów powoduje znaczne wydłużenie pracy algorytmu.

Z naszego doświadczenia wynika, że optymalne wartości parametrów n i W to **n = 5W.** Dzięki temu gęstość uchwytów jest na tyle duża, że zwykle daje się znaleźć rozwiązanie i jednocześnie nie trwa to zbyt długo.