DÚ 2 - RRT* a stromová struktura

Petr Vaněk

17. listopadu 2013

Úloha je inspirovaná technikou plánování pohybu v prostorech vyšších dimenzí známou pod názvem Rychle náhodně rostoucí stromy (RRT). Konkrétně její variantou známou pod názve RRT* [1, 2, 3], která se od původní verze liší především tím, že se výslednou trajektorii snaží optimalizovat. Technika se využívá v oblasti plánování trajektorií pro roboty a to jak mobilní tak manipulátory. Dále se využívá při výrobě léků a v neposlední řadě pro plánování trajektorií v počítačových hrách, kde má pohyb daných entit vypadat reálně.

1 Algoritmus

Vaším úkolem je do připravené šablony přepsat Algoritmus 1, který reprezentuje metoda `expand` v třídě `RRTStar` a doprogramovat metody potřebné pro jeho funkci. Vstupem do metody je náhodný bod $x_{rand} \in \mathbf{R}^2$, který reprezentuje bod v 2D prostoru a udává směr růstu stromu T. Druhým vstupem je samotný strom T a jeho upravená podoba je zároveň výstupem. V Algoritmu 1 je strom T naznačen jako vstup a výstup pro úplnost, ale v reálném řešení bude součástí třídy `RRTStar` a jediným vstupem do metody bude x_{rand}.

Princip algoritmu je velice jednoduchý. V prvním kroku (řádek 1) je potřeba nalézt ve stromu nejbližšího souseda x_{near} k bodu x_{rand}, což znamená, že strom obsahuje alespoň jeden bod, což je výchozí pozice x_{init}, kterou je strom inicializován. Tyto body se spojí hranou $e_{x_{near},x_{rand}}$ a rozšíří jí strom T (řádek 2). Pokračuje se optimalizační částí algoritmu tak, že se ze stromu

Algorithm 1: RRT* Extend

Input: Random state x_{rand}
Input: Tree T
Output: Tree T

1 $x_{near} \leftarrow \arg\min_{x \in T} d(x, x_{rand})$
2 $T \leftarrow T \cup \{(x_{rand}, e_{x_{near},x_{rand}})\}$
3 $X_{near} \leftarrow \{x | x \in T \backslash \{x_{near}, x_{rand}\} \wedge d(x, x_{rand}) < \epsilon\}$
4 **foreach** $x \in X_{near}$ **do**
5 $c' \leftarrow \texttt{Cost}(x) + d(x, x_{rand})$
6 **if** $c' < \texttt{Cost}(x_{rand})$ **then**
7 `reconnect`
8 **foreach** $x \in X_{near}$ **do**
9 $c' \leftarrow \texttt{Cost}(x_{rand}) + d(x_{rand}, x)$
10 **if** $c' < \texttt{Cost}(x)$ **then**
11 `reconnect`

vybere podmnožina uzlů $X_{near} \subseteq T$ (řádek 3), která v tomto případě představuje kružnici se středem v x_{rand} a zahrnuje všechny body $x \in T$ pro které platí $d(x, x_{rand}) < \epsilon$, kde $d(.,.)$ je funkce vracející Euklidovskou vzdálenost dvou bodů a ϵ je předem stanovená konstanta.

Optimalizace má dvě fáze. V první se hledá postupně přes všechny $x \in X_{near}$ (řádek 4) hrana do x_{rand}, která zmenší aktuální cenu cesty do x_{rand} (řádky 5 a 6). Zde funkce `Cost`(x) vrací aktuální cenu cesty do x. Pokud je cena menší je původní hrana do x_{rand} vyměněna za novou vedoucí z x funkcí `reconnect` (řádek 7). Druhá fáze je obdobná. Postupně se hledá jestli některá hrana z x_{rand} do $x \in X_{near}$ zmenší cenu cesty do x (řádky 8-10). Pokud ano je původní hrana do x nahrazena novou hranou z x_{rand} funkcí `reconnect`.

2 Podpůrné třídy a struktury

Point V úloze se bude pracovat s 2D bodem, který bude reprezentovaný třídou `Point` a bude mít následující strukturu:

```
class Point {
    double x;
    double y;
public:
    Point();
    Point(double x, double y);

    double distance(const Point & p) const;
    friend std::ostream & operator << (std::ostream & os,
        const Point & point);
};
```

Zde je potřeba implementovat:

- implicitní konstruktor
- konstruktor s parametrem `x` a `y`, který nastaví pozici (x, y)
- metodu `distance` která vrací Euklidovskou vzdálenost dvou bodů $\|p - q\|$.
- přetížit `operator <<`, který bude do výstupního proudu vypisovat `x y`.

  Pro bod $x = 1.1$ a $y = 2.2$ vypíše

  ```
  1.1 2.2
  ```

Node Samotný strom je reprezentován uzly, které definuje struktura `Node` a obsahuje polohu uzlu v 2D prostoru pomocí prvku `point`, ukazatel na předka `parent`, ukazatel na potomka `child` a ukazatel na sourozence `sibling`.

```
struct Node {
    Point point;
    Node * parent;
    Node * child;
    Node * sibling;
    Node * next;
    double cost;
    unsigned int id;
};
```

Uzel bude použit ve stromové struktuře, a proto bude mít uzel vždy jenom jednoho předka. Jako sourozenec se nazývá uzel, který má stejného předka. Pokud uzel nemá žádného sourozence ukazuje ukazatel `sibling` na sebe samého. Jak přibývají další sourozenci je mezi nimi ukazatelem `sibling` vytvořen cyklický spojový seznam. Pokud uzel nemá žádného potomka je ukazatel `child` nastaven na `NULL` jinak ukazatel ukazuje na prvního potomka, kterého uzel získal. Jediný uzel, který nemá žádného předka je kořen stromu a ten má nastaven ukazatel `parent` na `NULL`. Ukazatel `next` slouží jako jednosměrný spojový seznam do kterého se postupně přidává každý nově vytvořený uzel. Hodnota `cost` obsahuje celkovou cenu cesty do tohoto uzlu. Hodnota `id` obsahuje identifikační číslo uzlu, které pro kořen je 0 a s každým novým uzlem je o 1 větší.

Zde je potřeba implementovat:

- implicitní konstruktor
- přetížit `operator <<`, který vypíše `id(cost): point`

  Pro uzel s id = 2, cost = 2.13 a point = {1.1, 2.2} vypíše

  ```
  2(2.13): 1.1 2.2
  ```

Strom Samotný strom obsahuje ukazatel na počáteční uzel (kořen) `first` a na poslední přidaný uzel `last`

```
class Tree {
    static Node * const first;
    static Node * last;
public:
    Tree(Node * const root);
    ~Tree(void);

    Node * add(const Point & point, Node * parent);
    Node * nearest(const Point & point) const;
    std::vector<Node *> rNearest(const Point & point, double radius) const;
};
```

Je potřeba implementovat následující metody umožňující:

- vytvořit nový uzel pro bod `point` s předkem `parent` pomocí metody `add`
- nalézt nejbližšího souseda k bodu `point` metodou `nearest`
- nalézt uzly ve vzdálenosti $> \epsilon$ od bodu `point` metodou `rNearest`, kde ϵ reprezentuje vstupní parametr `radius`
- přetížit `operator <<`, který vypíše postupně všechny uzly ve stromu ve formát pro konkrétní uzel `uzel p uzel->parent`

  Pokud strom obsahuje 3 prvky, kde kořen je v bodu {0, 0} a má dva potomky, první s id = 1 na souřadnicích {1.1, 2.2} a druhý s id = 2 na souřadnicích {1, 3.3} vypíše se následující výstup

  ```
  0(0): 0 0
  1(2.45967): 1.1 2.2 p 0(0): 0 0
  2(3.44819): 1 3.3 p 0(0): 0 0
  ```

RRT* Poslední část je třída `RRTStar`, která implementuje modifikovaný algoritmus. Obsahuje ukazatel na strom `tree`, ukazatel na kořen stromu `root` a konstantní `RADIUS`, který v Algoritmu 1 koresponduje s ϵ.

```
class RRTStar {
    Tree * tree;
    Node * root;
    const double RADIUS;
public:
    RRTStar(const Point & point, double radius);
    ~RRTStar(void);
    void iterate(const vector<Point> & points);
    void extend(const Point & rand);
};
```

Zde je potřeba implementovat:

- konstruktor, který parametrem `point` vytvoří kořen a tím se inicializuje strom `tree` a parametrem `radius` se nastaví `RADIUS`.
- metodu `extend` která implementuje Algoritmus 1.
- přetížený `operator <<`, který do výstupního proudu vypíše strom vytvořený algoritmem.

V šabloně je implementovaná metoda `RRTStar::iterate` a funkce `main`, které by při odevzdávaní měli zůstat ve stejné podobě.

Vstupní soubor data se zadávají jako text na standardní vstup ve formátu:

```
radisu
p0x p0y
p1x p1y
p2x p2y
...
```

kde první je `radius`, který bude algoritmus používat pro hledání nejbližších sousedů. Následují souřadnice kořene stromu (`p0x, p0y`), a pak pokračují souřadnice náhodných bodů (`p1x, p1y, p2x, p2y, ...`) jejichž počet samozřejmě není omezen.

3 Vzorový příklad

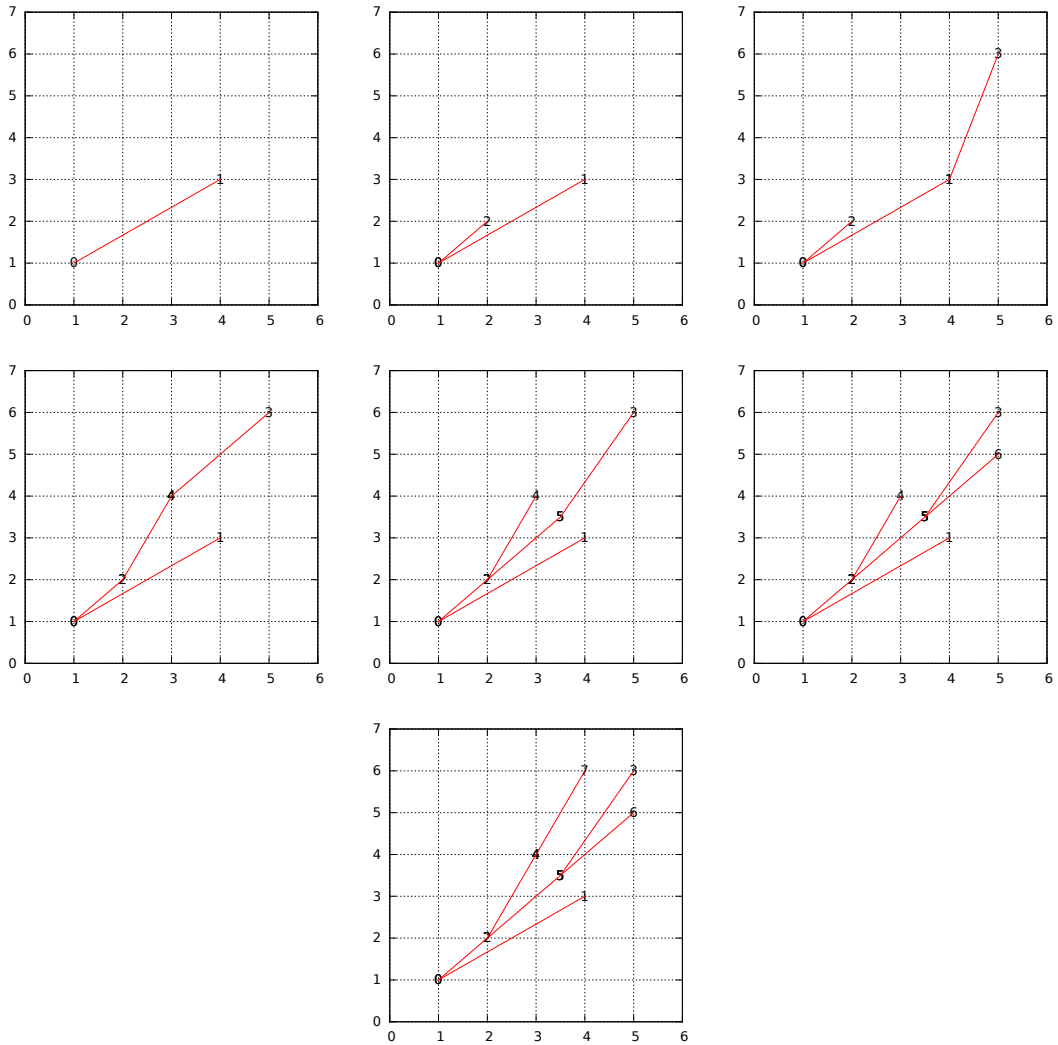
Program by pro následující vstupní data

```
3
1 1
4 3
2 2
5 6
3 4
3.5 3.5
5 5
4 6
```

měl vygenerovat tento výstup

```
0(0): 1 1
1(3.60555): 4 3 p 0(0): 1 1
2(1.41421): 2 2 p 0(0): 1 1
3(6.45101): 5 6 p 5(3.53553): 3.5 3.5
4(3.65028): 3 4 p 2(1.41421): 2 2
5(3.53553): 3.5 3.5 p 2(1.41421): 2 2
6(5.65685): 5 5 p 5(3.53553): 3.5 3.5
7(5.88635): 4 6 p 4(3.65028): 3 4
```

Graficky znázorněný postup algoritmu pro vzorová data je na Obrázku 1.



Obrázek 1: Postup algoritmu pro ukázková vstupní data

Reference

[1] S. Karaman and E. Frazzoli. Sampling-based motion planning with deterministic μ-calculus specifications. In *IEEE Conference on Decision and Control (CDC)*, Shanghai, China, December 2009.

[2] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning. In *Robotics: Science and Systems (RSS)*, Zaragoza, Spain, June 2010.

[3] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *Int. J. Rob. Res.*, 30(7):846–894, 2011.