RRT ALGORITHM – IMPLEMENTARE

Explicare algoritm: Algoritmul **RRT** funcționează prin construirea progresivă a unui arbore în spațiul de configurare al robotului sau al obiectului pentru care este planificat. Spațiul de configurare reprezintă toate stările sau configurațiile posibile ale robotului. Arborele începe cu o configurație inițială și apoi explorează spațiul de configurare prin creșterea în mod repetat a arborelui către noduri aleatorii.

Etape ale algoritmului::

- Inițializare: Începeți cu un arbore T gol, constând dintr-un singur nod reprezentând configurația inițială a robotului.

- Generare noduri: generați o configurație aleatorie în spațiul de configurare. Această configurație servește ca o țintă sau o configurație de obiectiv.

- Extindere: Găsiți cel mai apropiat nod din arborele T de configurația generată aleatoriu. Acest nod devine „cel mai apropiat vecin”.

- Steer: Determinați o nouă configurație prin direcția de la cel mai apropiat vecin către configurația generată aleatoriu, ținând cont de constrângerile mișcării robotului.

- Verificarea coliziunilor: Verificați dacă noua configurație nu se ciocnește, adică nu se intersectează cu niciun obstacol din mediu. Dacă nu este fără coliziuni, reveniți la pasul 2 și generați o nouă configurație aleatorie.
- Adăugarea noului nod: Dacă noua configurație este fără coliziuni, adăugați-o ca nod nou în arborele T, conectat la cel mai apropiat vecin.

- Verificare obiectiv: Verificați dacă nodul adăugat nou este aproape de configurația obiectivului. Dacă este suficient de aproape, a fost găsită o cale fezabilă și algoritmul se termină. În caz contrar, reveniți la pasul 2 și continuați extinderea arborelui.
- Afiseaza calea: Odată ce algoritmul se termină, extrage calea de la configurația inițială la configurația finala, traversând arborele de la nodul obiectiv (goal) înapoi la nodul inițial.

Explicare implementare:

Definim clasa RRT, care are ca atribute nodul start, nodul gol, biasul si step_size si urmatoarele metode:

Metoda **generate_random_point** este folosita pt pasul 2, adica pentru a popula imaginea cu noduri random. Se foloseste biasul pt a determina daca nodul este nod goal sau un nod random. Am folosit bias pentru a imbunatati pathurile gasite, astfel, un nod aleator este generat prin compararea unei valori aleatoare distribuite uniform între 0 și 1 cu valoarea biasului. Dacă valoarea aleatorie este mai mică decât biasul, nodul este nodul goal. În caz contrar, este generat un punct aleatoriu în limitele hărții.

Prin introducerea biasului față de punctul țintă, algoritmul devine mai orientat spre nodul goal și tinde să exploreze spațiul din vecinatatea lui. Poate fi util pt găsirea rapidă a unei căi către obiectiv, mai ales când punctele de început și obiective sunt îndepărtate sau când mediul are o structură complexă cu pasaje înguste.

Metoda **get_nearest_node** gaseste cel mai apropiat nod. Se foloseste distanta euclidiana. Pt. o performanta mai buna / gasirea mai buna a drumurilor se poate folosi distanta Manhattan (norma L1), sau distanta Chebyshev.

Metoda **steer** direcționează de la un nod la altul prin calculul unui vector de direcție și scalarea acestuia la dimensiunea pasului. Dacă distanța dintre cele două noduri este mai mare decât step_size, vectorul de direcție este redus, scalat la distanta si step_size. Nodul rezultat este returnat.

Metoda **is_collision_free** verifică dacă există o coliziune între două noduri utilizând algoritmul **Bresenham**. Acesta generează un set de puncte de-a lungul liniei dintre cele două noduri și verifică dacă vreunul dintre acele puncte cade pe un obstacol de pe hartă.

Metoda **bresenham_line** implementează algoritmul Bresenham line pentru a genera un set de puncte de-a lungul unei linii între două puncte date. Este o metoda ajutatoare pentru metoda care verifica coliziunea.

Metoda **find_path** este algoritmul RRT principal pentru a găsi o cale de la punctul de plecare până la nodul goal. Cu ajutorul metodelor definite anterior:

Acesta generează în mod iterativ puncte aleatorii, găsește cel mai apropiat nod în arborele RRT, se îndreaptă către punctul aleator și verifică căile fără coliziune. Dacă se găsește o cale fără coliziune către un nou nod, aceasta este adăugată în arbore. Algoritmul se termină fie când este atins nodul goal, fie după un număr maxim de iterații (prestabilit la 10.000).

Metoda **plot_tree** ilustreaza arborele RRT și calea rezultată pe hartă folosind matplotlib. Acesta trasează harta ca o imagine în tonuri de gri și suprapune marginile copacului. Nodurile start si goal sunt afișate ca puncte de dispersie verde și, respectiv, roșu. Dacă este găsită o cale (adică, nodul goal este în arbore), aceasta urmărește calea de la goal înapoi până la început folosind relațiile părinte-copil memorate in arbore.

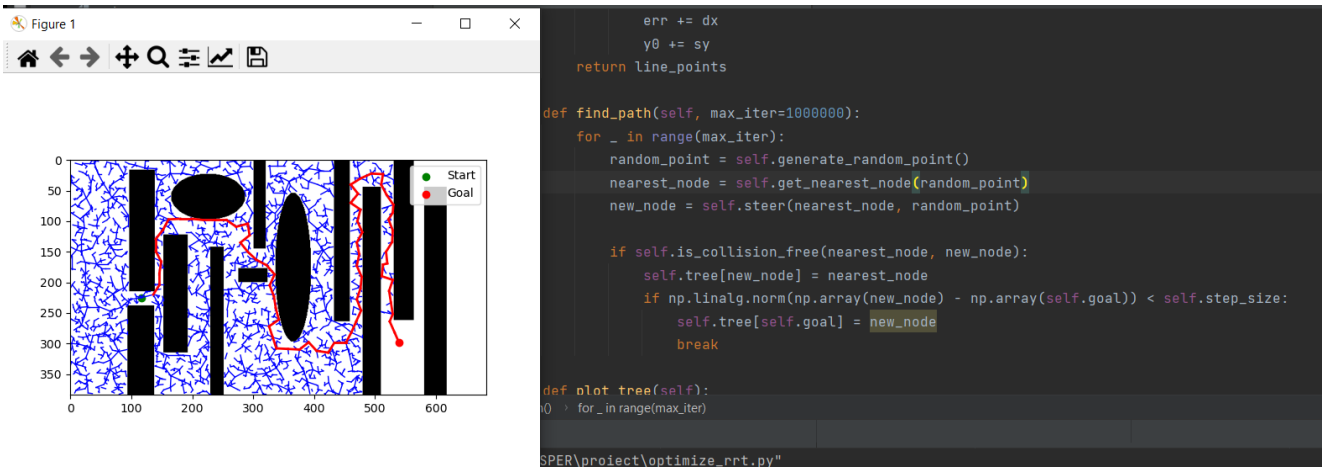
Metoda **main()** este punctul de “intrare” al programului. Citeste imaginea, o converteste in grayscale, initializeaza un RRT apeland functia **choose_start_goal**, care permite inputul prin interactionarea cu harta (dublu click o data pt a seta pozitia de start, apare pozitia de start, dublu click iar pentru a seta pozitia goal, dupa care va aparea si aceasta. Inchide fereastra pt a continua)

Pe scurt, acest cod definește o clasă RRT care implementează algoritmul RRT pentru planificarea traseului. Acesta permite utilizatorului să selecteze nodurile start si goal pe o imagine de hartă și vizualizează arborele RRT și calea rezultată. Algoritmul construiește progresiv un arbore conectând nodurile de la start la goal, evitând coliziunile cu obstacolele din hartă.

Folosire: pentru a folosi codul, pur si simplu rulati codul, apeland functia main(“fisier.png”), unde fisier.png este numele hartii.

To-do:

Optimizari: Codul nu functioneaza destul de bine in cazul unei harti de complexitate mare, cu multe obstacole si spatii foarte inguste daca nodurile start si goal sunt foarte departe, sau le gaseste, dar necesita timp si resurse, si nu e eficient (vezi exemplul de mai jos). Idei de optimizare: Se poate calcula cu o alta distanta, nu cea euclidiana; Se poate aplica path-smoothening pt a elimina nodurile ne-necesare, si facand astfel drumul mai direct.



Observam ca pentru o harta destul de complicata, cu multe obstacole si puncte foarte departate, a fost nevoie de un max_iter = 1 milion, step_size folosit fiind de 20.