Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Robotos pakolási műveletek hatékony tervezése

Szakdolgozat

Készítette
Bodor Máté

Konzulens
dr. Kovács András
dr. Hullám Gábor

2019. december 2.

Tartalomjegyzék

Kivonat **i**

Abstract **ii**

1. Bevezetés HIV **1**

2. Irodalmi áttekintés HIV **3**
2.1. Robotok kinematikai modellezése . . . 3
2.1.1. Feladat tér . . . 3
2.1.2. Konfigurációs tér . . . 3
2.1.3. Denavit–Hartenberg paraméterek . . . 4
2.1.4. Direkt kinematika . . . 4
2.1.5. Inverz kinematika . . . 4
2.1.5.1. Zárt alakú megoldás . . . 5
2.1.5.2. Numerikus megoldás . . . 5
2.2. Sorrendtervezés . . . 5
2.2.1. Sorrendtervezési modellek . . . 5
2.2.1.1. TSP . . . 6
2.2.1.2. ATSP . . . 6
2.2.1.3. GTSP . . . 6
2.2.1.4. SSP . . . 6
2.2.2. Megoldó eljárások . . . 6
2.2.2.1. Lokális keresés . . . 7
2.2.2.2. VLNS . . . 7
2.2.2.3. Hegymászó keresés . . . 7
2.2.2.4. Szimulált lehűtés HIV . . . 7
2.2.2.5. Tabu keresés HIV . . . 8
2.3. Pályatervezés . . . 8
2.3.1. Kombinatorikus tervezés . . . 9
2.3.2. Mintavétel alapú tervezés . . . 9
2.3.2.1. PRM HIV . . . 9
2.3.2.2. RRT HIV . . . 9

3. Feladat definíció nov 15 **11**
3.1. Bemeneti paraméterek . . . 11
3.1.1. Munkadarabok száma . . . 12
3.1.2. Munkadarabok kiinduló- és célhelyzete . . . 12
3.1.3. Robotkar kiinduló- és végállapota . . . 12

3.1.4. Robotkar és a cella modellje . . . 12
3.1.5. Robotkar általános paraméterei . . . 13
3.1.5.1. A kar csuklóinak száma . . . 13
3.1.5.2. Denavit-Hartenberg paraméterek . . . 13
3.1.5.3. Sebesség- és gyorsulásparaméterek . . . 13
3.1.5.4. Megfogó paraméterei . . . 14
3.2. Megoldási terv . . . 14
3.2.1. Inverz kinematika . . . 14
3.2.2. Ütköző robotkonfigurációk kiszűrése . . . 14
3.2.3. Sorrendtervezés és robotkonfigárációs lista választás . . . 15
3.2.4. Pályatervezés . . . 15

4. Megoldás nov 26 **16**
4.1. Munkafolyamat . . . 16
4.2. Inverz kinematika . . . 16
4.3. Sorrendtervezés és konfiguráció választás . . . 16
4.4. Pályatervezés . . . 16

5. Implementáció HIV **17**
5.1. CollisionLibrary . . . 17
5.1.1. CollisionManager . . . 18
5.1.2. PRM . . . 19
5.1.3. PathReview . . . 20
5.2. InverseMap könyvtár . . . 21
5.2.1. InverseMap osztály . . . 21
5.2.2. UR5 . . . 21
5.3. Google OR-tools . . . 21
5.3.1. RoutingIndexManager . . . 22
5.3.2. RoutingModel . . . 22
5.3.3. Assignment . . . 23
5.4. SequencePlanning . . . 23
5.4.1. Workpiece . . . 24
5.4.1.1. Az osztály propertijei . . . 24
5.4.1.2. Az osztály függvényei . . . 24
5.4.2. DistanceMatrix . . . 25
5.4.3. Solver . . . 26
5.5. IntegarateServices . . . 27
5.5.1. Az osztály propertijei . . . 27
5.5.2. Az osztály függvényei . . . 27

6. Eredmények dec 3 **30**

7. Értékelés dec 3 **31**

Köszönetnyilvánítás **32**

Irodalomjegyzék **33**

Függelék **34**

HALLGATÓI NYILATKOZAT

Alulírott *Bodor Máté*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2019. december 2.

Bodor Máté
hallgató

Kivonat dec 6

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

1. fejezet

Bevezetés HIV

Az egyre növekvő ipari fejlődés következménye, hogy manapság egyre elterjedtebbé válnak a teljesen automatizált gyártósorok. Míg régen az ipari robotok alkalmazása kizárólag a tömeggyártásban volt jellemző, manapság már a kis sorozatszámú vagy akár az egyedi gyártásban is megjelennek. Ennek következményeként megnőtt az igény az egyre bonyolultabb, változatosabb ipari robotok és robotos gyártócellák alkalmazására. Ezek már nem csak egy egyszerű ipari folyamatokat hajtanak végre, hanem egyre összetettebb feladatokat látnak el, amelyeket már nem lehet előre beállítani a gyártósor beüzemelésekor. Ehelyett célszerű mesterséges intelligenciát használni a probléma megoldásához.

Az MI ilyen alkalmazásokban releváns ágai a cselekvéstervezés, a pályatervezés, és a gépi látás. Manapság, ezek a mesterséges intelligencia egyik legjobban fejlődő területei. Ehhez kapcsolódik szakdolgozat munkám, ami egy robotos pakolási feladat hatékony megoldása.

A robot által végrehajtandó feladat, hogy megmunkálandó munkadarabokat vesz fel egy tálcáról és azokat egy célhelyre teszi. A feladat nehézségét az jelenti, hogy a munkadarabok véletlenszerű pozícióban helyezkednek el a tálcán és hogy közel valós időben kell végrehajtani a műveletet. Ez egy nagyon összetett folyamat, ami több különálló feladatra bontható.

[KÉP]

A teljes problémát és a megoldási munkafolyamatot Tipary Bence írta le [HIV]. A szakdolgozatom során ennek a munkafolyamatnak egy részét valósítom meg.

Az ipari folyamat megoldása során meg kell határozni a munkadarabok pontos helyét a tálcán. Többféle érzékelőt is használható a feladat megoldására. Az egyik racionális döntés, a kamera használata, mert viszonylag olcsón beszerezhető és elegendően pontos eredményt ad. A kamera által készített képről matematikai eszközök segítségével már megállapítható a munkadarab térbeli elhelyezkedése a tálcán.

Nem elegendő az, ha tudjuk, hol találhatóak a munkadarabok a tálcán, mert a robotkar nem tudja minden pontban megfogni ezeket a munkadarabokat. Így a munkadaraboknak nem csak a helyét, hanem a munkadarabokhoz tartozó megfogási pontokat is meg kell állapítanunk. Ezeknek a lehetséges megfogási pontoknak a pontos térbeli helyét a kamaraképről szintén megállapíthatjuk.

A robotkarok csuklókból állnak, ezeknek a segítségével képes mozogni a kar. Mozgás közben a csuklók elfordulnak. A csukló kezdőállapotához képesti elfordulás egy csuklóállást ad meg. A csuklók különböző helyzetei egy robotkar konfigurációt

határoznak meg. Egy adott térbeli ponthoz, több robotkar konfiguráció is tartozhat. Nekünk meg kell határozni ezeket a robotkonfigurációkat.

Ezek közül konfigurációk közül még ki kell szűrnünk azokat, amik a valóságban valamilyen okból nem jöhetnek létre, pl.: A robot önmagával vagy a cella egy másik elemével ütközik.

A szűrt robotkar konfigurációkból lehet majd megállapítani a konfigurációk azon sorrendjét, amit egy jó megoldása a problémának és a lehető legrövidebb idő alatt tesz meg a robotkar.

A konfigurációk sorrendtervezése még nem a végleges megoldása a problémának. A megoldásban található, egymást követő konfigurációs párok közt még egy ütközés mentesen bejárható lehető legrövidebb utat kell találnunk.

Ezek közül a feladatom a munkadarabokhoz tartozó robotkonfigurációk meghatározása, a munkadarabok sorrendezése és az egyes megfogási és lerakási műveletekhez tartozó robot konfigurációk meghatározása és ütközésmenetes pályatervezés a robotkar számára úgy. A feladat célja, hogy a robotkar pályájának bejárási ideje és a teljes műveleti idő összege a lehető legalacsonyabb legyen.

Ez az egész ipari feladat egy nagyon leegyszerűsített képe, a valós probléma során sokkal több feladattal meg kell küzdeni. Például: ha egy elem olyan pozícióban ragadat, ahol nem tudja felvenni a kar, akkor el kell érnünk, hogy elmozduljon vagy ha az egyik munkadarab akadályozza a másik felvételét, akkor azt a másik előtt kell felvennünk.

A beszámolóban szeretnék egy elméleti áttekintés adni, arról a tudásról, ami szükséges volt a feladat megoldása során. Részletesen bemutatni az általam megoldott feladat részeket és a hozzájuk szükséges bemeneti paramétereket. Ezek után részletezem a megoldás matematikai modelljét. Majd kitérek az általam használt könyvtárakra és a feladat implementációjára. Végezetül elemzem a kapott eredményeket és értékelem, mennyire jól sikerült megoldani a feladatot, használható lenne-e éles környezetben is és bemutatom a projekt további fejlesztési lehetőségeit.

2. fejezet

Irodalmi áttekintés HIV

Ebben a fejezetben szeretném röviden bemutatni azt az elméleti hátteret ami szükséges a szakdolgozati munkám megértése szempontjából. Először bemutatom a robotkarok kinematikai modellezését, aztán a sorrendtervezés és végül a pályatervezés elméleti hátterét.

2.1. Robotok kinematikai modellezése

A robotok kinematikája a robotok mozgásának leírásával foglakozik, a mozgások dinamikai hátterének figyelembe vétele nélkül.

2.1.1. Feladat tér

A robotikában a feladat tér egy pontját egy M 4x4-es homogén transzformációs mátrixszal adjuk meg. Ez a mátrix a robotkar rögzítési pontjához kötött koordináta rendszer és a robotkar végberendezéséhez kötött koordináta rendszer között add meg egy transzformációt.

$$M = \begin{bmatrix} R_{3x3} & T_{3x1} \\ 0_{1x3} & 1 \end{bmatrix}$$

Az M mátrix egy 3x1-es T részmátrixa írja le a rögzítési ponthoz kötött Descartes koordináta rendszerben, hogy hol található a végpont koordinátarendszere. Azt, hogy a végponthoz viszonyított koordinátarendszer, hogy helyezkedik el a rögzítési ponthoz képest, azaz a két koordinátarendszer megfelelő tengelyei milyen szöget zárnak be egymással,egy 3x3-as R részmátrix írja le. Ezzel az R részmátrixal lényegében a robotkar végének állását írjuk le. [1]

2.1.2. Konfigurációs tér

A robotkar állását, konfigurációját megadhatjuk a robotkar egyes csuklóinak állásával. Ekkor C legyen az egyes csuklók állásának szögértéke. Egy 6 szabadságfokú robotkar esetén C hat darab értéket fog tartalmazni.

A konfigurációs teret szokás még csukló térnek nevezni.

[KÉP]

[1]

2.1.3. Denavit–Hartenberg paraméterek

A robotkar geometriai reprezentációját többféleképpen is meg tudjuk adni. Az egyik kényelmes módja, hogy minden egyes kartaghoz egy koordinátarendszert rögzítünk. Jacques Denavit és Richard Hartenberg írta le először egy konvencionális jelölésrendszert, hogy általánosítsa a kartagokhoz rendelt koordinátarendszerek relatív helyzetének leírását. A jelölésrendszer elemeit az 1955-ös publikációjuk alapján Danavit-Hartenberg paramétereknek (D-H paramétereknek) nevezzük őket.

Ebben a jelölésrendszerben négy paraméterre van szükségünk, hogy a egy koordinátarendszer relatív helyzetét leírjunk egy másik koordináta rendszerhez viszonyítva. A négy paraméterből kettő vonatkozik a csuklókra és kettő a kartagok leírására. A csuklóknál a hozzájuk kapcsolódó kartagok eltolása d és a csuklóállás szöge Θ . A kartagoknál a kartag hossza a és a kartag elcsavarodása α .

A jelölésrendszerben az i-edik csukló az i-edik kartag végén helyezkedik el, az i-edik és i+1-edik kartag között. A csukló eltolás d_i és a csuklószög Θ_i az i-1. csukló koordinátarendszere szerint van mérve, ezért a csukló indexe és a csukló paramétereinek indexe nem egyezik meg.

[KÉP]

[5]

2.1.4. Direkt kinematika

A direkt kinematika feladata nyílt kinematikai láncú robotkar esetén, hogy a kar csuklóállásainak értékéből, egy viszonyítási ponthoz tekintve, meghatározza a robotkar végpontjának helyzetét. A viszonyítási pont általában a robotkar rögzítési pontja, ami egy konstans eltolással megkapható a 0. kartagból. Általában a robotkar végére valamilyen eszközt szerelünk, hogy befolyásolni, vagy érzékelni tudja a környezetét. Ezt az eszközt végberendezésnek szokás nevezni. A végberendezés helyzete megkapható egy konstans transzformációval a robot utolsó kartagjához képest.

A számításhoz bemeneti paraméterként két dologra van szükség, egyik a robotkar fizikai paraméterei, a másik a robotkar csuklóállásainak értéke.

A direkt kinematikai probléma megoldása egy transzformáció egy a robotkar rögzítési pontjához kötött koordináta rendszerből a végberendezéshez kötött koordináta rendszerbe, azaz a felszerelt eszköz és a rögzítési pont között. Ezt a transzformációt egy 4x4-es homogén transzformációs mátrixszal szokás leírni. A transzformáció egyértelmű, mert egy robotkarkonfigurációhoz egy homogén transzformációs mátrixot rendel.

[5]

2.1.5. Inverz kinematika

Nyílt kinematikai láncú robotkar esetén, az inverz kinematika feladata, hogy egy homogén transzformációs mátrixszal leírt ponthoz megadja a hozzá tartozó robotkarkonfigurációk értékét, azaz feladat térbeli ponthoz meghatározza a hozzátartozó konfigurációs térbeli pontokat.

A feladat megoldásához két paraméter szükséges, a homogén transzformációs mátrix és a robotkar fizikai paraméterei.

Ez egy nem lineáris transzformáció, előfordulhat, hogy nincs megoldása vagy akár több megoldása is lehet. Ha a feladattérbeli pont a robotkar munkaterén kívül esik, akkor biztosan nincs megoldása az egyenletnek. Az inverz kinematikai transzformációt kétféle módszerrel lehet megoldani. Az egyik a zárt alakú megoldás, de ez csak bizonyos fizikai szerkezetű robotkarokra értelmezhető. A másik módszer a numerikus számítás, ez általánosan megadható.

2.1.5.1. Zárt alakú megoldás

A zárt alakú megoldás nagy előnye, hogy gyorsabb mint numerikus alak kiszámítása és az összes lehetséges megoldást megadja. Hátránya pedig, hogy nem minden robotkarra lehet megadni és hogy minden robotkarra más. Zárt alakú megoldás során a robotkar egyedi geometriai adottságait használjuk ki. Általában, zárt alakú megoldás csak speciális hat szabadsági fokú robotkarokra adható, ahol sok robotkar paraméter nulla. Elégséges feltétele, hogy létezzen zárt kinematikai megoldása egy hat szabadsági fokú robotkarnak:

- Három egymást követő csukló tengelye egy pontban metszi egymást.
- Három egymást követő csukló tengelye párhuzamos.

A zárt alakú módszert algebrai és geometriai úton oldhatjuk meg. A szakdolgozatom során használt UR5-ös robotkarnak létezik zárt alakú megoldása.

2.1.5.2. Numerikus megoldás

A zárt alakú megoldással ellentétben a numerikus alak megoldása nem függ a robotkartól. A hátránya a zárt alakú megoldáshoz képest, hogy lassabb és nem mindig találja meg az összes lehetséges megoldást. A numerikus alakú módszer megoldható szimbólum kiküszöbölési, folyamatos és iteratív úton.

[5]

2.2. Sorrendtervezés

Sorrendtervezés során célunk adott feladat végrehajtási sorrendjére egy megengedett és optimális megoldási sorrendet adnunk.

2.2.1. Sorrendtervezési modellek

A sorrendtervezési modellek azért hasznosak, mert segítségükkel egy mérnöki problémához általánosan ismert matematikai modellt rendelünk. Egy sorrendtervezési feladatot nehézsége szerint négy csoportba tudunk sorolni.

A nehézség megállapítására két paramétert tudunk használni, egyik a bemeneti paraméterek típusa, másik a kimeneti paramétereké. Egy sorrendtervezési feladat bemenet állhat egyszerű feladatokból, például a munkadarabok helyzete vagy összetett feladatokból, ilyen lehet a hegesztési varratok pályája. A kimeneti paraméter lehet a feladatok elvégzésének sorrendje vagy a feladat végrehajtásának ütközésmentes pályája.

A szakdolgozati feladatom során a sorrendtervezési fázisban egy egyszerű feladatokból álló problémát oldok meg aminek kimenetele a feladatok elvégzésének sorrendje. A következőkben ennek a csoportnak a matematikai modelljeit szeretném bemutatni.

[KÉP: Robotic Task Sequencing Problem: A Survey fig. 3 részlet]

2.2.1.1. TSP

A TSP-t (Travelling Salesman Problem) magyarul Utazó Ügynök Problémának hívják. A TSP során egy összefüggő súlyozott gráf csúcsai közt kell megtalálnunk a lehető legrövidebb kört úgy, hogy csak a gráf élein lépkedhetünk a csúcsok között. A kör hossza a bejárt élsúlyok összege lesz. Általában a gráfról feltételezzük hogy két pontja között, A és B, A-ból B-be vezető út súlya megegyezik a B-ből A-ba vezető út súlyával. Ez egy NP nehéz probléma, ami azt jelenti hogy nincs polinomiális lépésszámú megoldása. Nagyméretű TSP-t ezért csak valamilyen metaheurisztika segítségével tudjuk megoldani.

2.2.1.2. ATSP

Az Asymmetric TSP (ATSP) azaz Aszimmetrikus Utazó Ügynök Probléma az Utazó Ügynök Probléma egy olyan változata, amikor a gráf két pontja közt, A és B, nem feltétlen ugyan akkora az út súlya, ha A-ból megyünk B-be vagy B-ből megyünk A-ba.

2.2.1.3. GTSP

A GTSP a Genaralized TSP rövidítése, azaz Általános Utazó Ügynök Probléma. Ebben az esetben az összefüggő súlyozott gráf csúcsait csoportokba rendezzük és a gráfban egy olyan legrövidebb pályát kell találnunk ami minden csoportot legalább egyszer érint.

2.2.1.4. SSP

A Shortest Sequence Problem azaz a Legrövidebb Sorrend Problémája a TSP egy olyan változata amikor az összefüggő súlyozott gráfban, a kör helyett egy olyan legrövidebb utat kell találnunk ami az összes csúcsot érinti. Az SSP-nek is van aszimmetrikus és általános változata, ekkor a ATSP-hez illetve a GTSP-hez hasonlóan a kör helyett egy legrövidebb utat kell találnunk. Ezeknek jelölése ASSP és SSP++.

[1]

2.2.2. Megoldó eljárások

A megoldó eljárások segítségével próbáljuk megtalálni egy feladat optimális végrehajtását. Ezek lehetnek teljes keresési eljárások, amelyek bejárják a teljes keresési teret, de ezek nagy feladatokon nem jól alkalmazhatóak vagy heurisztikus megoldók, amelyek nem mindig találják meg a legjobb megoldást, de nagy feladatokon is alkalmazhatóak. Ebben a részben heurisztikus megoldókat mutatok be.

2.2.2.1. Lokális keresés

A számításelméletben a lokális keresés egy heurisztikus módszer nagy számításigényű optimalizációs feladatok megoldására. A lokális keresés segítségével megtalálhatjuk a közel legjobb megoldást egy probléma megoldásainak halmazából. Általánosságban egy lokális kereső algoritmus a megoldási térben, megoldásról megoldásra halad, lokális változásokat végrehajtva. Az algoritmusok addig futnak, amíg el nem érnek egy olyan megoldáshoz, amin már nem tudnak tovább javítani vagy amíg a keresés ideje le nem telik.

A lokális kereső algoritmus először egy jó megoldást keres. Ezután ezen a megoldásokon kis változásokat hajt végre, hogy javítsa a megoldás teljesítményét. Ezeket a kis változásokat lépéseknek nevezzük. Egy megoldás szomszédságába azok a megoldások tartoznak amiket a megoldásból egy lépés alkalmazásával el tudunk érni. Hagyományos egy ilyen szomszédság mérete polinomiális. A kereső algoritmus ezekből a szomszédokból választ egyet a megoldás keresése során. A kereső algoritmus célja hogy optimizálja a megoldás célfüggvényének az értékét. A keresés során a lokális optimumok jelentik a legnagyobb nehézséget. A keresési algoritmus tervezésénél figyelnünk kell ezek elkerülésére. A lokális keresésnél egy alapszabály, hogy minél nagyobb a szomszédság számossága annál nehezebben ragad lokális optimumban a keresés.

Lokális keresést nagyon sokféle nagy számítási igényű feladaton alkalmaztak, beleértve az utazó ügynök problémát is. Egy teljes lokális keresés mindig talál megoldást, ha az létezik. Az optimális lokális keresési algoritmusok mindig megtalálják a keresett globális minimumot vagy maximumot.

[3] [4]

2.2.2.2. VLNS

A Very Large-Scale Neighborhood Search (VLNS) azaz Nagyon Nagyméretű Szomszédsági Keresés esetében egy megoldás szomszédsága exponenciális nagyságú. A nagyobb számú szomszédság miatt a keresés kisebb eséllyel ragad lokális optimumba és a lokális optimum is egy jobb megoldás mintha polinomiális nagyságú szomszédságban keresnénk. Általában a VLNS úgy van strukturálva, hogy a keresési függvény polinomiális idő alatt találja meg a legjobb lépést. VLNS segítségével hatékonyabban tudjuk megoldani az optimalizációs problémákat.

[3]

2.2.2.3. Hegymászó keresés

A hegymászó keresés egy egyszerű mohó algoritmus, ami mindig az aktuális legjobb érték felé lép. Ha az algoritmus a következő lépésben már nem tud javítani, akkor a keresés megáll. A hegymászó keresés nem hatékony algoritmus. Mivel a keresés nem járja be a teljes állapotteret, csak amíg az eredményen javítani tud, a keresés könnyen lokális optimumban ragadhat.

2.2.2.4. Szimulált lehűtés HIV

A szimulált lehűtés a hegymászó algoritmus továbbfejlesztett változata. A hegymászó algoritmussal a legnagyobb probléma, hogy könnyen egy lokális

optimumba juthatunk és befejeződik a keresés. Ez abból adódik, hogy a keresés során az állapottérnek csak minimális részét járjuk be. A szimulált lehűtés ezt próbálja meg kiküszöbölni. A hegymászó algoritmus mindig a legjobb lépést választja, ehelyett a szimulált lehűtés egy véletlen lépést választ. Ha a lépés javítja a célfüggvény értékét, akkor mindig végrehajtásra kerül. Ellenkező esetben az algoritmus a lépést csak P eséllyel teszi meg. A P valamilyen 1-nél kisebb szám, értéke exponenciálisan csökken a lépés értékének rosszaságával, azaz egy Δ E mennyiséggel, amivel a célfüggvény értéke romlott. A valószínűség a szimulált lehűtés másik T paraméterétől is függ. A T hőmérséklet csökkentésével is csökken a rossz lépés valószínűsége. A rossz lépések esélye a szimulált lehűtés indulásánál, amikor a T nagy, elég magas, a T csökkenésével viszont egyre valószínűtlenebbekké válnak. Ha a T értékét kellően lassan csökkentjük akkor matematikailag bebizonyítható, hogy a szimulált lehűtés elég hosszú idő alatt közel egy valószínűséggel megtalálja a keresett globális optimumot.

Az algoritmus a 80-as évek elején terjedt el. A szimulált lehűtést először a VLSI nyomtatott áramkörök tervezésekor használták. Azóta széleskörűen alkalmazzák a nagy számításigényű optimalizációs feladatokra.

2.2.2.5. Tabu keresés HIV

A tabu keresés lényegében egy rövidtávú memóriával rendelkező hegymászó keresés, pár szabállyal kiegészítve. A tabu keresés során addig folytatjuk a keresést amíg ki nem elégítjük a keresés feltételét, ez lehet egy célfüggvényérték vagy a futási idő korlátozása. A tabu keresési algoritmus, ha lehet mindig olyan lépést választ, amivel javít a célfüggvény értékén. Ha nincs ilyen lehetőség akkor rontó lépést is elfogad. Az algoritmus egy véges nagyságú FIFO memóriában tárolja el a már meglátogatott csomópontokat. Ha egy csomópont benne van a memóriájában akkor oda nem lép az algoritmus. Az algoritmus képes arra, hogy kitörjön lokális optimumból. Ennek hatékonysága a FIFO méretétől függ, addig elkerüli az algoritmus a lokális optimumot amíg a csomópont a memóriában van.

A tabu keresés algoritmust a 1989 években írta le Fred W. Glover. A keresési eljárást azóta széleskörűen alkalmazták a logisztika, orvosbiológiai elemzés, ütemezés és egyéb nagy számítási teljesítményű optimalizációs feladatnál.

[4]

2.3. Pályatervezés

Pályatervezés során két pont között szeretnénk egy optimális hosszúságú ütközésmentes utat találni. Erre többféle algoritmus létezik, ezek közül szeretnék párát röviden bemutatni.

Pályatervezés során a konfigurációs térben tervezzük meg a robot mozgását. Jelölje a robot munkaterét C. C teret két altérre tudjuk osztani. Legyen az egyik altér C_{obs}, ez az a térrész ahová a robot nem tud eljutni ütközésmentesen. A C_{free} = C - C_{obs}, azaz a robotkar által ütközésmentesen bejárható térrész.

2.3.1. Kombinatorikus tervezés

A kombinatorikus tervezésnél egy útvonaltérkép segítségével határozzuk meg az optimális utat. Az útvonaltérkép egy gráf C_{free}-ben, aminek minden csúcsa C_{free}-ben helyezkedik el és a csúcsok közötti élek ütközésmentes utak C_{free}-ben. A gráf élei súlyozva vannak aszerint, hogy a két csúcs milyen távolságra helyezkedik el egymástól. Pályatervezéskor a útvonaltérkép két pontja között egy minimális súlyösszegű utat keresünk.

Az útvonaltérkép elkészítésére sok különböző eljárás létezik.

Az eljárás előnye, hogy kis szabadságfokú problémákra nagyon jó megoldást add. Hátránya viszont, hogy a szabadsági fokok növekedésével nagyon gyorsan kezelhetetlenné válik a feladat mérete, ezért sok dimenziós robotkarok pályatervezésére alkalmatlan.

2.3.2. Mintavétel alapú tervezés

A mintavétel alapú tervezésnél nem karakterizáljuk előre a C teret. A C_{free} térrész felfedezéséhez egy ütközésdetektáló algoritmusra lesz szükségünk. Az eljárás előnye, hogy sok dimenziós terek esetén sokkal gyorsabb mint egy kombinatorikus tervezési eljárás, hátránya, hogy nem minden esetben talál megoldást és ha talál, akkor sem biztos, hogy az optimálisat találja meg. A következőkben két mintavétel alapú eljárást fogok bemutatni.

2.3.2.1. PRM HIV

A Probabilistic Road Map (PRM) azaz Valószínűségi Útvonaltérkép egy mintavétel alapú tervezési eljárás. Az eljárás során random mintákat választunk ki a C térből, ha a minta C_{free}-be esik, akkor egy csúcsként értelmezzük. Aztán ezt a csúcsot próbáljuk meg összekötni a közeli csúcsokkal egy lokális tervező segítségével. A lokális tervező megvizsgálja, hogy az él a két csúcs között ütközésmentesen bejárható-e. Ha igen, akkor felvesszük az élet a csúcsok közé, ha nem akkor eldobjuk. Azt hogy melyik csúcsokat tekintjük közelinek, többféle képen definiálhatjuk, lehet a k db legközelebbi csúcs vagy egy bizonyos távolságon belül lévő összes csúcs. A gráfba addig veszünk fel csúcsokat amíg elég sűrű nem lesz az egy jó pálya megtervezéséhez.

A PRM előnye, hogy alkalmazható nagy dimenziójú feladatokra, nagy valószínűséggel teljes megoldást ad és a generált térkép a C térben több pályatervezésre is felhasználható. Hátránya, hogy nem jól alkalmazható szűk átjárókban. Elegendően sok idő alatt közel 1 valószínűséggel optimális és teljes megoldást.

2.3.2.2. RRT HIV

Az RRT (Rapidly Exploring Random Trees) magyarul Gyorsan Felfedező Véletlenszerű Fák módszere. Az eljárás során egy q_0 pontból szeretnénk eljutni egy q_g pontba úgy, hogy egy fa gráfot építünk. Az RRT egy iteratív eljárás, addig ismétlődik, amíg el nem érünk a q_g pontban. Minden iterációban felveszünk egy q_i pontot és megpróbáljuk ütközésmenetesen összekötni az addig lerakott q_n, legközelebbi ponttal. Ha sikerül, akkor felvesszük az élet és q_i is fa része lesz. Ha nem sikerül q_i-t ütközésmentesen összekötni q_n-el akkor felveszünk egy q_i' csúcsot

az ütközéshez lehető legközelebb és azt kötjük be a fába. Az eljárás minden m-edik iterációjában $q_i = q_g$. Ha sikerül q_g bekötnünk a fába leáll az eljárás.

Ez csak egy lehetséges eljárás a RRT megvalósítására. Az RRT-nek számos további továbbfejlesztett változata létezik a hatékonyabb keresés érdekében.

Az RRT előnye hogy kis résekben is hatékony és könnyű implementálni. Hátránya, hogy minden egyes pályatervezéshez új keresést kell indítanunk, egyes esetekben nagyon lassan talál megoldást.

3. fejezet

Feladat definíció nov 15

A szakdogalzati feladatom egy közel valós idejű sorrend és pályatervezési általános megoldó készítése, robotos pakolási feladatokhoz. A pakolási feladatot egy darab nyílt kinematikai láncú robotkar hajtja végre. A robotos munkaállomás terve előre adott. A pakolási feladat munkadarabjai egy sík felületű tálcán helyezkednek el. Ez a tálca egy rázóasztalra van erősítve, ennek segítségével a munkadarabokat jól el lehet különíteni, az esetek nagy részében így nem fedik egymást, a robotkar hozzájuk tud férni. A munkadarabok célhelyzete előre meghatározott. A pakolási feladat megoldása során tetszőleges sorrendben fel kell vennünk az egyes munkadarabokat és a célhelyzetükbe kell vinnünk őket. A pakolási feladatnak vége, ha az összes munkadarabot eljuttatjuk a célhelyzetébe. Ekkor a tálcán nem találhatóak munkadarabok. Ha a tálcára új munkadarabokat helyezünk, akkor azoknak a pakolását egy következő feladatnak tekintjük.

A megoldó a szükséges bemeneti paraméterek magadása után, ilyen például a munkadarabok helyzete, egy a robotkar által ütközésmentesen bejárható pályát ad vissza. Ez az út időben egy közel optimális megoldása a munkadarabok célba juttatásának.

Mivel közel valós időben szeretnénk végrehajtani a pakolási feladatot, ezért célunk egy olyan megoldó tervezése, hogy a megoldó futásideje és a talált pálya bejárási idejének összege a lehető legkisebb legyen.

A feladat megoldás három részre bontható, ez a három:

- inverz kinematika,
- sorrendtervezés és robotkarkonfigurációs lista választás,
- ütközésmentes pályatervezés.

[KÉP]

3.1. Bemeneti paraméterek

A feladat megoldása során szükségesek bizonyos bemeneti paraméterek. Ezek az értékek lehetnek előre adottak vagy az érzékelők adatai alapján generáltak. Az adott, konstans érték, lehet például a környezet modellje vagy a robotkar egyes paraméterei.

3.1.1. Munkadarabok száma

A feladatom során a munkadarabok száma előre adott. Legyen ezen munkadarabok számának jelölése NW.

3.1.2. Munkadarabok kiinduló- és célhelyzete

A munkadarabok kiindulási helyzete egyes érzékelők adatai alapján meghatározhatók. Az egyik leghatékonyabb módszer kamera segítségével meghatározni az egyes munkadarabok helyzetét.

Nem elegendő egy munkadarab pontos helyének leírásához egy a robotkar rögzítési pontját origónak tekintő Descartes koordináta rendszert használni. Mivel a munkadarabok nem pontszerű testek, így nem elég tudni csak a munkadarabok tengelyekhez mért távolságát. Egy munkadarab több félé pozícióban helyezkedhet el aszerint, hogy hány stabil egyensúlyi helyzete van.

A munkadarabok pontos helyének leírásához az elméleti összefoglalóban már említett feladat teret használjuk. Ekkor a térben az egyes munkadarabok helyzetét egy homogén transzformációs mátrix írja le. Az egyes munkadarabok kiinduló helyéhez egy ilyen mátrixot rendelhetünk bemeneti paraméterként. Legyen ennek a kiindulóhely mátrixnak az i-edik munkadarabhoz rendelt jele WS_i.

A munkadarabok célhelyzete előre meghatározott. Megoldandó feladattól függ, hogy az egyes munkadarabokhoz ugyanazt vagy egyedi célhelyzetet rendeljünk. Az általánosabb használhatóság miatt, minden egyes munkadarabhoz célszerű egyedi értéket rendelnünk. A kiindulóhelyzethez hasonlóan a célhelyzetet is faladat térben adjuk meg. Legyen az i-edik munkadarab célhelyzet mátrixának jele WE_i.

3.1.3. Robotkar kiinduló- és végállapota

A hagyományos nyílt hurkú ipari robotok mozgásukat előre beprogramozott ciklus szerint végzik így a kiinduló- és végállapotuk megegyezik. Esetünkben ez nem mindig van így, ezért a robotkar mozgásának tervezésekor szükségünk van a robotkar kezdeti és végállapotára.

A robotkar egy állását az egyes csuklók szögeinek megadásával jellemezhetjük. Legyen egy n csuklóból álló robotkar egyes csuklóinak szöge r_1, r_2, r_3, ..., r_n. Ekkor a csuklók szögei egyértelműen meghatároznak egy R robotkar konfigurációt, azaz $R = [r_1, r_2, r_3, ..., r_n]$. A robotkar kiinduló és végállapotát is egy ilyen R robotkar konfigurációval adhatjuk meg. Legyen a kiindulóállapotát leíró robotkar konfiguráció R_s és a végállapot leíró pedig R_e.

Abban az esetben ha R_s és R_e megegyezik a robotkar mozgása során pályáról, abban az esetben ha különböznek útról beszélhetünk.

3.1.4. Robotkar és a cella modellje

A robotkarmozgását leíró kinematikai számításoknál szükségünk van a robotkar modelljére. A robotkar térbeli leírását úgy kapjuk meg, hogy a robotkar, különálló, külön mozogni képes részeit egy térbeli testtel közelítjük. Nem elég tudnunk ezeket a térbeli testeket. Szükségünk van még egy szabályrendszerre amely leírja a testek elhelyezkedését, mozgását és egymással való kapcsolatukat.

Egyes robotkaros problémák megoldása során szükségünk lehet a robotkar munkaterében lévő környezet leírására. Itt is a különálló részeket egy térbeli testtel közelítjük és egy szabályrendszerrel leírjuk az egyes testek elhelyezkedését és környezetükkel való kapcsolatát.

Esetünkben a robotkar, munkadarabok és munkaasztal leírására van szükség.

3.1.5. Robotkar általános paraméterei

Ahhoz hogy a robotkar mozgását vizsgáljuk szükséges tudnunk a mozgását befolyásoló paramétereket. Egy robotkar csuklókból és a csuklókat összekötő kartagokból áll. Az utolsó kartag végére álatlában végberendezést rögzítenek, amivel a robot tudja érzékelni vagy befolyásolni környezetét. Robotkaros pakolási feladatoknál általában ez a munkadarabok megfogásához szükséges eszköz.

A tervezés és modellezés során a robotkar gyári adatait használjuk. Fontos azonban kiemelni, hogy a robotkarok rendelkeznek a gyártási pontatlanságokból adó eltérésekkel. Ezen eltérések nagyban befolyásolhatják a robotkar mozgását. Ezért éles használat előtt korrigálni kell ezen adatokat.

A tervezés során a SZTAKI-ban megtalálható UR5-ös robotkart vettem alapul. De a feladat megoldása során törekedtem az általános felhasználhatóságra. Egy robotkart nagyon sok paraméterrel jellemezhetünk. A megoldás során azonban eltekintünk bizonyos paraméterektől, amik nem, vagy kis mértékben befolyásolják a robot mozgását. Ilyen például a robotkar terhelhetősége és az ebből származó mozgás változás.

3.1.5.1. A kar csuklóinak száma

A robotkar csuklóinak száma nagyban befolyásolja egy robot kar mozgási képességeit. Egy egy csuklóból álló robotkar egy tengely mentén képes mozgást végezni. A robotkar csuklószámának növelésével nő a robotkar mozgásának bonyolultsága, több tengely, dimenzió mentén képes mozogni a kar. Jelölje a robotkar csuklóinak számát NJ. UR5-ös robot esetében az NJ értéke 6.

3.1.5.2. Denavit-Hartenberg paraméterek

A Denavit-Hartenberg(D-H) paramétereket a robot kinematikai leírására használjuk, segítségükkel könnyen átválthatjuk a robotkar csuklóállásaival leírt pont helyét világkoordináta rendszerbe. A feladat során az inverz kinematikánál és a pályatervezésnél használom fel. UR5-ös robot esetén ezek a paraméterek: [Táblázat]

3.1.5.3. Sebesség- és gyorsulásparaméterek

A robotkar mozgását a kar csuklóinak paramétereivel jellemezhetjük. A feladat megoldása során két ilyen csuklóparamétert használtam fel, ezek a paraméterek a csukló szöggyorsulása és szögsebessége.

Általánosságban a robotkar csuklója mozgása során először maximálisan gyorsul amíg el nem éri a maximális sebességét, ezt követően tartja ezt a sebességét, majd fékez és megáll. A fékezés értéke megegyezik a maximális gyorsulásával. Ezért egy

csukló szöggyorsulását és szögsebességét a csukló maximális szöggyorsulásával és szögsebességével közelítettem.

Tapasztalatok szerint az UR5-ös robotnál ezek az értékek jó közelítést adnak. Ha valamiért a robotkar nem tudná tartani ezeket az értékeket, például túlterheljük a kart, akkor a robot mozgása leáll.

UR5-ös robotkar esetén ezek az értékek: [Táblázat]

3.1.5.4. Megfogó paraméterei

A robotkar végén elhelyezkedő eszköz esetünkben egy a pakolási feladat munkadarabjait megfogni képes megfogó. A munkám során két paraméterét használta fel, egyik a megfogó eszköz hossza, jelölje Gh a másik a megfogó pofák egymástól való távolsága ez legyen Gw. Ezeket az adatokat a pályatervezésnél és az inverz kinematikánál használom fel.

3.2. Megoldási terv

A szakdolgozat során három részfeladatot kell megoldanom, ezeket a fejezet bevezetésében már említettem. Az egyes részfeladatok kimenetele lesz az azt követő részfeladat egyik bemenete. A szakdolgozatom során megtervezem és egymásba fűzöm ezeket a feladatrészeket. A következőkben szeretném definiálni a részfeladatokat.

3.2.1. Inverz kinematika

Az első megoldandó részfeladat a munkadarabok kiinduló és célhelyzetének átszámítása robotkar által értelmezhető vonatkoztatási rendszerbe. Ekkor egy munkadarab világkoordináta rendszerbeli elhelyezkedését kell átváltanunk robotkonfigurációba. Egy ilyen transzformáció nem egyértelmű, egy világkoordinátához több robotkar konfiguráció is tartozhat.

Az inverz kinematikai számítás bemenete egy homogén transzformációs mátrix és a robotkar fizikai paraméteri. A kimeneti paramétere pedig egy lehetséges robotkar konfigurációkat tartalmazó lista. Ebben a listában azok a robotkar konfigurációk vannak benne, amiket egy ideális, kiterjedés nélküli robotkar képes felvenni, a képzetes gyökkel és nagy számítási hibával rendelkező konfigurációkat nem tartalmazza. A inverz kinematika kimenete ez a lista, melynek jele legyen IC.

Az inverz kinematikai transzformáció azonban nem vizsgálja a környezet és robot kölcsönhatását. Ahhoz, hogy megkapjuk a lehetséges robotkar konfigurációkat, a listából ki kell szűrnünk azokat a konfigurációkat ahol a robotkar ütközik önmagával vagy a környezettel.

[KÉP]

3.2.2. Ütköző robotkonfigurációk kiszűrése

Az inverz kinematikai transzformáció azonban nem vizsgálja a környezet és robot kölcsönhatását. Ahhoz, hogy megkapjuk a lehetséges robotkar konfigurációkat, az IC listából ki kell szűrnünk azokat a konfigurációkat ahol a robotkar

ütközik önmagával vagy a környezettel. Az ütközésmentes robotkonfigurációk megállapításához szükségünk van bemeneti paraméterként az IC listára és a robotkar és cella modelljére. A folyamat kimeneteli paramétere pedig az ütközésmentes robotkonfigurációkat tartalmazó IF lista.

3.2.3. Sorrendtervezés és robotkonfigárációs lista választás

A sorrendtervezés és a robotkonfiguráció választás két különálló feladat, de esetünkben ezek összekapcsolódnak. A részfeladat megoldása során törekszünk a időben legrövidebb sorrend kiválasztására. Mivel egy munkadarabhoz több robotkarkonfiguráció is tartozik, nem tudjuk munkadarabok sorrendjét a robotkarkonfigurációk figyelembe vétele nélkül meghatározni.

A részfeladat megoldása során egy döntést kell hoznunk, hogy melyik szabályos robotkarkonfigurációs sorrendet választjuk. A pakolási feladatot szeretnénk időben a legrövidebb idő alatt végrehajtani, ezért azt a sorrendtervezési és robotkarkonfigurációs lista választási eljárást keressük ami futási időben és megoldás idejét tekintve időben a lehető legkedvezőbb. Mivel nagyon sok jó robotkonfigurációs lista létezik a feladat megoldására és közel valós időben szeretnénk megoldani a pakolási feladatot, célszerű valamilyen heurisztikát használni a számunkra legkedvezőbb robotkonfigurációs lista kiválasztásához.

A folyamat bemeneti paramétere az IF robotkonfigurációs lista és a robot sebesség- és gyorsulásparaméterei. A részfeladat kimenetele a döntés eredménye lesz. Legyen a kiválasztott robotkarkonfigurációs lista C és a lista i-edik eleme C_i.

[KÉP]

3.2.4. Pályatervezés

A pályatervezés során két robotkarkonfiguráció között szeretnénk egy időben hatékony ütközésmentes pályát találni.

A pálytervezéshez bemenetként szükségünk van a robot és környezet modelljére és a robot paramétereire. A pályatervezést mindig ugyan annak a robotnak a munkaterében szeretnénk végrehajtani.

[KÉP]

Olyan megoldást kell keresni ami gyors, mivel közel valós időben szeretnénk végrehajtani a pakolási feladatot. Ezért nem feltétlenül az időben optimális pályát keressük, hanem egy olyan eljárást aminek a véghajtási ideje és a talált pálya bejárási ideje együtt a lehető legkisebb. Esetünkben a pályatervezést a C lista minden egyes egymást követő elempárjára végre kell hajtanunk, ez $NW * 2 + 1$ darab tervezést jelent. A pályatervezési eljárás kimenete legyen egy lista amiben az egyes tervezések eredményit egymás után fűzzük.

Legyen ennek a listának neve P és a lista i-edik eleme P_i.

4. fejezet

Megoldás nov 26

4.1. Munkafolyamat

4.2. Inverz kinematika

4.3. Sorrendtervezés és konfiguráció választás

4.4. Pályatervezés

5. fejezet

Implementáció HIV

Az alábbi fejezetben szeretném bemutatni az általam elkészített, használt könyvtárakat és az elkészült program szerkezetét.

A felhasznált könyvtárak egy részét a SZTAKI bocsájtotta rendelkezésemre másik része pedig nyílt forráskódú és licence alapján szabadon felhasználható.

A fejlesztést C#-ban hajtottam végre, és Visual Studio 2019-es IDE használtam hozzá. Ezen kívül használtam még a RoboDK-t api-t a teszt feladatok generálásához. Ezt Python nyelven tettem meg és a Visual Studio Code nevű szövegszerkesztőtben készítettem el. Az egyes robotkarállások térbeli ellenőrzésére a RoboDK-t és a CellVisualizer-t, ami a SZTAKI-tól kapott CollisionLibrary része, használtam fel.

A program bemeneti paraméterei a robot és cella modellt leíró STL fájlok és a elvégzendő pakolási feladat leírását tartalmazó fájl. A feladatot egy txt-ből olvassa be a program. A txt fájl minden sora egy összefüggő adatot tartalmaz, az adat egyes részeit tabulátor választja el. A txt fájl felépítése a következő:

- A robotkar kezdő robotkonfigurációjának szögei radiánban.
- A robotkar cél robotkonfigurációjának szögei radiánban.
- A munkadarabok felvételi és lerakási pontja. Egy munkadarabhoz két sor tartozik, az első sor a munkadarab felvételi pontjának homogén transzformációs mátrixa, a második sor pedig a munkadarab lerakási pontjának homogén transzformációs mátrixa.

A szakdogalzat feladatom elkészítése során először továbbfejlesztettem az önálló laboratóriumi munkámat, majd a különálló könyvtárakat egymásba fűztem, hogy megoldják a pakolási feladatot. Az IntegrateServices osztály végzi el a három különálló könyvtár egymásba fűzését. Ezek a könyvtárak a CollisionLibrary, InverseMap, SequencePlanning. A könyvtárak függőségi diagramja a következő:

Először az általam felhasznált könyvtárakat szeretném röviden bemutatni, amikre az én munkám építkezik és ezek után az általam létrehozott könyvtárat illetve osztályt.

5.1. CollisionLibrary

[Önlab / TDK hivatkozás]

A CollisionLibrary egy C#-ban íródott keretrendszer nyílt kinematikai láncú robotkarok ütközésvizsgálatára és pályatevezésére. Ezt a könyvtár a SZTAKI-ban található meg és az ő engedélyükkel használtam.

5.1.1. CollisionManager

A CollisionManager osztály segítségével lehet ütközésvizsgálat során vizsgált objektumokat, az az a robotkar és cella modellje, kezelni és ütközésvizsgálatot végrehajtani.

Szakdolgozatom során a pályatervezésnél és az inverz kinematika kimenetének szűrésénél használom fel.

Először bemutatom az osztály használatához szükséges osztályokat és fontosabb függvényeiket. Majd a CollisionManager-t és egy példát az inicializációjára.

RobotGripper

A robotkarra szerelhető végberendezések egyik fajtája a megfogó. Ez az osztály implementálja a megfogó modelljét.

public RobotGripper(): Az osztály konstruktora paraméter nélkül hívható, inicializálja a megfogót.

UR5

Az UR5 osztály implementálja az UR5-ös robotkar modelljét. Segítségével lehet a robotkar paramétereit beállítani.

public UR5(GripperModel gripper, double[ ] translation, double[ ] rotation): A konstruktor első paramétere a robotkarra szerelt végberendezés, második paraméterrel a robotkar eltolását tudjuk megadni a munkatér origójához képest, harmadik paraméterrel az egész robotkart tudjuk elforgatni a modell egyes koordináta tengelyi körül.

PQPCollisionLib

Az útkőestérvezés matematikai modelljét megvalósító osztály. Segítségével lehet két osztály ütközését és távolságát vizsgálni.

public PQPCollisionLib(): Az osztály paraméter nélküli konstruktora.

Mesh

A Mesh osztály egy geometriai test térhálóját tárolja és kezeli.

pulic Mesh(string stlFile): A Mesh osztály konstruktorának az STL fájl elérési útját kell megadni. A konstruktor betölti a fájlt és inicializálja objektumot.

CollObj

A CollObj osztályban tárolódnak el a robot modell és cella egyes különálló részei. Például a robotkar egyes kartagjai egy egy CollObj-ek.

public CollObj(Mesh mesh): A konstruktor paraméterként egy Mesh-t vár és inicializálja az objektumot.

CollisionRule

Az osztály segítségével két CollObj között lehet felvenni ütközési szabályt.

public CollisionRule(CollObj a, CollObj b, bool collCheckActive): A konstruktor első két paraméterrként egy egy CollObj-et, harmadik paraméter a szabály, ha a bool értéke true, akkor vizsgálja köztük az ütközést, ha false akkor nem.

Config

Az osztály segítségével a robotkar egy konfigurációját kezelhetjük.

public Config(double[] joints): Az osztály konstruktora a robotkar csuklóállásainak segítségével inicializálja az objektumot.

Az osztály általam használt függvényei

public CollisionManager(ICollisionLibrary library, RobotModel robotModel): Az osztály konstruktora két paramétert vár, az első az ütközésvizsgálatra használt osztály egy példánya, a második a robot modellje.

public CollObj getCollisionObject(string name): A CollisionManager által kezelt CollObjecteket lehet elérni név alapján.

public void addCollisionRule(CollisionRule newCollisionRule): Segítségével lehet új ütközési szabályokat felvenni az osztályhoz.

public bool checkConfigurationBool(Config configuration): A függvény segítségével lehet vizsgálni, hogy egy robotkarállás ütközik-e.

A CollisionManager beállítása

5.1.2. PRM

A PRM osztály a CollisonLibrary része. Az irodalmi áttekintésben bemutatott PRM pályatervezési algoritmus megvalósítása. A PRM során egy keresési térképet építünk fel, amit ezt követően több is fel tudunk használni a pályatervezéshez. Az osztály segítségével létrehozhatunk, lementhetünk, betölthetünk ilyen térképeket és ütközésmentes pályatervezést valósíthatunk meg.

PRMParameters

Az osztály segítségével a PRM keresés paramétereit lehet beállítani és kezelni.
Az osztály fontosabb propertijei:

NumberOfNodes int: A PRM gráf csúcsainak számát adja meg.

NumberOfNeighbours int: Megadja, hogy a PRM gráf csúcsát hány darab hozzá legközelebbi csúccsal kössünk össze.

TCP double: A robotkarra szerelt megfogó két pofája közti távolság milliméterben megadva.

MinimumDegrees double[ ]: A robotkar csuklóinak minimális csuklószögei.

MaximumDegrees double[ ]: A robotkar csuklóinak maximális csuklószögei.

A PRM osztály általam használt függvényei

public void Build(): Felépít egy új pályatérképet a PRMParameters értékei alapján.

public void Save(string fileName): Elmenti a pályatérképet a fileName nevű fájlba.

public void Load(fileName): Betölti a pályatérképet a fileName nevű fájlból.

public List< Config > Execute(Config from, Config to): A függvény tervez egy ütközésmentes pályát a from robotkonfigurációból a to robotkonfigurációba, visszatérési értéke a pályatérkép bejárt csúcsainak robotkonfigurációit bejárási sorrendben tartalmazó lista.

5.1.3. PathReview

A PathReview könyvtárban különböző pályasimítási eljárások találhatóak. A PRM keresés csak a térképben felépített gráf csúcsaiban keresi meg a legrövidebb utat. Így ez a talált út még nem optimális. A simítási eljárásokkal ezt az utat tudjuk lerövidíteni és hatékonyabbá tenni.

Pályasimítási eljárások osztályai

A pályasimítási eljárásokat egy egy osztály implementálja. Ezek az osztályok egy közös IPathReview interface-t valósítank meg, ezáltal egységesen használhatóak. Az osztályoknak ugyan úgy használható a konstruktora és az execute függvénye.

public PathReview(ICollisionManager collisionManager, List<Config> path) A simítóeljárásokat implementáló osztályoknak konstruktorban egy ICollisionManager interface-t implementáló osztály példányát és egy utat kell megadni.

List< Config > Execute() A függvény a konstruktorban megadott pályát módosítja az osztályban implementált eljárás szerint. Viszatérési értéke a pálya robotkonfigurációit bejárási sorrendben tartalmazó lista.

5.2. InverseMap könyvtár

Az InverseMap könyvtár az irodalmi áttekintésben bemutatott inverz kinematikai folyamatra épül. Elérhető benne néhány robot zárt alakjának megoldása, köztük az általam tesztelésre használt UR5-ös roboté is. A könyvtárat a SZTAKI bocsájtotta rendelkezésemre.

5.2.1. InverseMap osztály

Az InvereMap egy absztrakt osztály, a különböző robotok inverz kinematikai modelljét implementáló osztályok ősosztálya. Segítségével közös interface-en keresztül használhatóak az egyes inverz kinematikai osztályok. Fontosabb függvényei:

public void SetInputVariable(string key, double val): Ennek a függvénynek a segítségével lehet beállítani inverz megoldó paramétereit, például: Robotkar D-H paramétereit, a feladattérbeli pont paramétereit, stb. .

public List< List< double > > GetOutputVariablesList(string[ ] keys, int[ ] branches): A függvény segítségével lehet lekérni az a barnches int tömbben megadott megoldáságak keys tömbben felsorolt paraméterit.

5.2.2. UR5

Az UR5-ös robotkar inverz kinematikáját megvalósító osztály.

public UR5(): Az osztály paraméter nélküli konstruktora.

public void Calculate(): Az inverz kinematikai számítást elvégző függvény.

5.3. Google OR-tools

A Google OR-Tools egy nyílt forráskódú keretrendszer optimalizációs problémák megoldására.

A kertrendszer C++-ban készült a minél hatékonyabb futás érdekében, de elérhető még másik három népszerű programozási nyelven, ezek a Python, C# és

Java. A keretrendszer egyegységes interfészt add, amelynek segítségével le tudjuk írni a matematikai problémánkat.

A keretrendszer hatékony megoldást kínál több matematikai problémára. Ezek közt található a VRP azaz Vehicle Routing Problem. A VRP általánosan megfogalmazható úgy, hogy mi az optimális útja egy flotta járműnek, hogy kiszállítsa a vevőknek a csomagokat. Ennek egy speciális változata, amikor csak egy "járművel" rendelkezünk a TSP.

Ahhoz hogy a megoldó megoldja a problémánkat egy adatmodellt kell építenünk, VRP és TSP esetében ez egy négyzetes mátrix. A négyzetes mátrixban kell leírnunk a GTSP gráf egyes csúcsai közötti élsúlyoknak értékét.

A továbbiakban bemutatom az OR-tools általam használt osztályait és azok fontosabb függvényit.

5.3.1. RoutingIndexManager

A RoutingIndexManager osztály feladata számon tartani a keresés indexelését, a csomópontok és a járművek maximális számát és az egyes járművek kezdő és véghelyének indexét.

public RoutingIndexManager(int num_nodes, int num_vehicles, int[] starts, int[] ends): Az osztály konstruktorának meg kell adni a keresési gráf csomópontjainak számát, a feladat járműveinek számát, a mi esetünkben 1, a járművek kiinduló és cél helyének indexét a négyzetes mátrixban.

public int IndexToNode(long index): A keresés indexét váltja át a négyzetes mátrix tagjának indexévé.

5.3.2. RoutingModel

A RoutingModel feladata a GTSP probléma megoldása.

public RoutingModel(RoutingIndexManager manager): Az osztály konstruktorának egy RoutingIndexManager példányt kell megadnunk.

AddDisjunction(int [] indexes): A függvény segítségével tudjuk beállítani a GTSP csoportjait, az indexes tartalmazza a csoport négyzetes mátrixban lévő helyét.

public int RegisterTransitCallback(LongLongToLong callback): A függvény segítségével tudjuk beregisztrálni a callback eseményünket. A callback esemény pedig a négyzetes mátrix egy sor és oszlop indexéhez tartozó súlyértéket ad vissza. Például:

SetArcCostEvaluatorOfAllVehicles(int transitCallbackIndex): Az összes járműnek beállítja a költségszámító callback függvényét.

public IntVar NextVar(long index): Visszatér a következő csúcs indexével.

public bool IsEnd(long index): Igazzal tér vissza ha az index az út utolsó eleme.

5.3.3. Assignment

Ez az osztály felelős a sorrendtervezés eredményének kezeléséért.

public long Value(IntVar var): A megoldás egy var-al indexelt elemét adja vissza.

5.4. SequencePlanning

A SequencePlaning könyvtár felelős a sorrendtervezési feladat megoldásáért. A könyvtárat a Google OR-Tools felhasználásával hoztam létre.

Az OR-Tools mellett még a GLKH [2] megoldó jött fel lehetőségként. A GLKH a GTSP problémák megoldására kifejlesztett környezet. Előnye, hogy a egy GTSP problémát nagyon gyorsan oldhat meg és közel optimális megoldást adhat. Hátránya hogy, a GTSP problémán kívül nehéz bővíteni a modellt, plusz korlátokat nem tudnánk felvenni. Az OR-Tools egy általánosabb keretrendszer, könnyen felépíthető benne a GTSP modell és bővíthető új korlátokkal, mint a precedencia korlát. Hátránya viszont, hogy nem a GTSP problémára kifejlesztett keretrendszer, ezért lassabban találhat megoldást. A bővíthetőség miatt az OR-Toolst választottam.

A könyvtár első változatát az önállólaboratóriumi munkám során készítettem el. A könyvtár akkori változata C++-ban íródott, a szakdolgozati munkámat pedig C# szerettem volna megírni, részben azért mert a számomra szükséges könyvtárak C#-ban voltak elérhetők, részben meg azért mert a SZTAKI kutatócsoportja, akiknél a szakdolgozatomat készítettem C#-ot használ így számukra is előnyös, hogy egységes marad a kódbázisuk. Ahhoz hogy az önálló laboratóriumi munkámat fel tudjam használni a szakdolgozat során meg kellett oldani az eltérő programozási nyelvből adódó problémát. A probléma megoldására két lehetőséget láttam.

Az első lehetőség, hogy egy becsomagoló C# osztályt készítek az önálló laboratóriumi munkámhoz, és majd azon keresztül érem el. A neten elérhető számos szabadon elérhető program, ami képes legenerálni ezt a becsomagoló osztályt. Az önálló laboratóriumi munkám egyik hiányossága az, hogy bár a programot a letöltött Google OR-Tools környezetben lehet fordítani parancssorból és hatékonyan elvégzi a sorrendtervezést, önállóan nem fordul, nem lehet külső könyvtárként használni. Így becsomagolás előtt még ezt a problémát is meg kell oldani. Ezen felül a C++ kódot ki kell egészíteni egy interfésszel, amin keresztül elérhető a sorrendtervező külső alkalmazásokból és a kódot refaktorálni kell, és megváltoztatni pár helyen a kódot, hogy általánosabban is felhasználható legyen a sorrendtervezés.

Ennek a lehetőségnek előnye, hogy az önálló laboratóriumi munkám jó alapot ad, nem sok idő elvégezni a kódon a változtatásokat. Hátránya pedig, hogy nem lesz egységes a kódbázis és időbe telik megoldani a kód könyvtárként kezelését és becsomagolását.

A másik lehetőség, hogy újraírom C#-ban az önálló laboratóriumi munkámat, így könnyen tudom integrálni a szakdolgozatomi munkámba. A C# Nuget csomagkezelő rendszerében megtalálható a Google OR-tools és a hozzá szükséges

könyvtárak. Ennek köszönhetően a csomag feltelepítése után már nem kell azzal foglalkoznom, hogy hogyan fogom az általam írt könyvtárat kívülről elérni. Ebben a lehetőségben a feladatom újra implementálni az önnálló laboratóriumi munkámat.

Ennek a hátránya, hogy az újra implementálás időbe telik. Előnye pedig, hogy egységes lesz a kódbázis, tisztább lesz az újraírt kód.

Én a második lehetőséget választottam, úgy ítéltem meg, hogy minőségibb a munkám, ha egységes és átlátható a kódbázis. A másik indok amiért így döntöttem, hogy átláttam a munka mennyiségét és ezáltal könnyebben beosztottam az időmet.

A SequencePlaning könyvtár végzi a robotos pakolási feladatok sorrendtervezését. A megvalósítás során három osztályt implementáltam:

- Workpiece
- DistanceMatrix
- Solver

[osztálydiagram]

5.4.1. Workpiece

A Workpiece osztály egy munkadarab felvételi és lerakásipontjának helyét tárolja el robotkonfigurációkban.

5.4.1.1. Az osztály propertijei

GraspConfigs List< List <double> > A munkadarab felvételi robotkonfigurációinak gettere és settere.

PutawayConfig List< List <double> > A munkadarab lerakási robotkonfigurációinka settere és gettere.

5.4.1.2. Az osztály függvényei

public Workpiece(List< List <double> > graspConfigs, List< List <double> > putawayConfigs): Az osztály konstruktora, beállítja a felvételi és lerakási robotkonfigurációkat.

public Workpiece(): Az osztály paraméter nélküli konstruktora.

public void AddGraspConfig(List <double>): A felvételi robotkonfigurációkhoz hozzáad egy elemet.

public void AddGraspConfigs(List< List <double> >): A felvételi robotkonfigurációkhoz hozzáad egy robotkonfigurációkat tartalmazó listát.

public void AddPutawayConfig(List <double>): A lerakási robotkonfigurációhoz hozzáad egy elemet.

public void AddPutawayConfigs(List< List <double> >): A lerakási robotkonfigurációhoz hozzáad egy robotkonfigurációkat tartalmazó listát.

5.4.2. DistanceMatrix

A matematikai GTSP modellt a DistanceMatrix osztály implementálja. Feladata megállapítnai az egyes élsúlyokat, megadni az egyes csoportokat, felépíteni a keresés során használt négyzetes mátrixot. A DistanceMatrix konstruktorában egy Workpiece listát és a robotkar kiinduló és célkonfigurációját kapja meg, ezek alapján hozza létre a négyzetes mátrixot. A DistanceMatrix példányból propertiken keresztül érhetők el az egyes paraméterek, ezek:

- Starts int[ ] - A robotkarok, estünkben robotkar kiinduló konfigurációjának indexe a négyzetes mátrixban egy tömbben tárolva.
- Ends int [ ] - A robotkar, robotkarok célkonfigurációjának indexe a négyzetes mátrixban.
- MatrixLength int - A négyzetes mátrix egy dimenziójának mérete.
- Distance List< List< long > > - A négyzetes mátrix, két dimenziós Listában eltárolva.
- Disjunctions List< long[ ] > - A GTSP csoportjainak index tömbjét tartalmazó lista.

Az osztály fontosabb függvényei

public double CalculateWeight(List<double> from, List<double> to): Az élsúlyok számítása a CalculateWeight függvényel történik. A függvény a bemenetén kapott két robotkonfiguráció között határozza meg a robotkar csuklóinak maximális mozgási idejét. Azért a maximálisra van szükségünk, mert a robotkar mozgása során a csuklók egyszerre mozognak, így a kar mozgásideje a legtovább mozgó csukló mozgásideje lesz. A mozgásidő számítására a Megoldás című fejezetben leírt trapézmodell implementációját használom.

private void CreateMatrix(): A négyzetes mátrixot egy distance List< List< long > >-ban tárolom el, a distance a négyzetes mátrixot sorfolytonosan tárolja el, azaz a belső List< long > a mátrix egy sorát írja le. Distance-t a CreateMatrix függvény segítségével építem fel. A CreateMatrix függvény soronként építi fel a distance-t. A distance oszlop indexe a külső List indexének felel meg és a sor indexe a belső List indexének. A distanceben a külső oszlop indexén keresztül a négyzetes mátrix egy sorát kapjuk meg. Minden sor és az oszlop indexhez egy-egy robotkonfiguráció tartozik. A distance egy oszlop és egy sor indexével egy súly értéket kapunk az oszlop index robotkonfigurációjából a sor index robotkonfigurációjába. Ugyanazon értékű sor és oszlop index ugyan arra a robotkonfigurációra hivatkozik. Az indexek növekvő sorrendje egy robotkonfigurációs sorrendet ad, ez a sorrend:

- A 0. indexű elem a robotkar kiinduló konfigurációja

- Ezt követik az egyes munkadarabok felvételi robotkonfigurációi, a konstruktorban kapott Workpiece lista sorrendje szerint.
- Utána az egyes munkadarabok lerakási robotkonfigurációi, a konstruktorban kapott Workpiece lista sorrendje szerint.
- Végül az utolsó indexű elem a robotkar cél konfigurációja.

Az itt felsorolt négy csoporthoz különböző módon de a csoportok tagjainak egységes logika mentén lehet meghatározni a hozzátartozó distancebeli sor élsúlyait. Az egyes csoportoknak külön függvényeket hoztam létre amivel megállapítható a hozzájuk tartozó sor súlyai. A függvényeket megfelelő sorrendben meghívva pedig felépítjük a distance-t. Egy ilyen függvényre példa:

A függvény egy robotkonfigurációhoz végigmegy az összes sorindexhez tartozó robotkarkonfigurációkon és amelyek között fut él a Megoldás című fejezetben bemutatott GTSP gráfban, ott kiszámítja a súlyértéket, amúgy meg a legnagyobb lehetséges súlynak megfelelő értéket rak be, ami több nagyságrenddel nagyobb mint egy súlyérték, így ezt az élet soha nem fogja választani a megoldó.

void CreateDisjunctions(): Elkészíti a GTSP gráf csoportjainak listáját.

public List<double> ElementAt(int index) Visszaadja a négyzetes mátrix egy indexéhez tartozó robotkonfigurációt.

5.4.3. Solver

A Solver osztály feladata a keresés paramétereinek beállítása, callback függvény előállítása a Google OR-Tools számára, és a keresés futtatása majd az eredmény elmentése. A keresés során kétféle metaheurisztikát lehet beállítani a Solver osztállyal, az egyik egy egyszerű hegymászó keresés a másik pedig tabu keresés.

Az osztály propertijei

public RouteTime double: Visszaadja a tervezett sorrend robotkarral történő bejárásának idejét.

public Solution List<List<double»: A sorrendtervezés megoldását tartalmazó lista.

Az osztály függvényei

public Solver(IDistanceMatrix matrix): Az osztály konstruktora egy négyzetes mátrixot vár.

public void SetHilClimbingSerach(): A következő tervezést a hegymászó algoritmussal futtatja.

public void SetTabuSearch(int nanos): A következő tervezést a tabu algoritmussal futtatja, nanos-ban (ns) megadott ideig.

public void Solve(): Futtatja a sorrendtervezést.

5.5. IntegarateServices

Az IntegrateSerices osztály feladat a SequencePlanning, CollisionLibrary és InverseMap könyvtárak megfelelő használata, ezzel az egész munkafolyamat megvalósítása. Az IntegrateServices függvényein keresztül egy interface-t ad számunkra amelyen keresztül elérhetőek, beállíthatóak az egyes részfolyamatok, inverz kinematika, sorrendtervezés, pályatervezés, vagy az egész folyamat egyben.

A program gyorsaságának és helyességének tesztelésekor az egész folyamatot hajtottam végre. Egy teszteset lefutás a következő szekvenciadiagram

A Megoldás című fejezetben leírt, két prm történő pályatervezést a következőféleképpen implementáltam:

5.5.1. Az osztály propertijei

Workpices List< Worpiece > : Az osztály munkadarabjainak gettere és settere.

StartConfig List< double > : A robotkar kezdőpontjának gettere és settere.

EndConfig List< double >: A robotkar végpontjának gettere és settere.

Sequence List< List< double > >: A sorrendtervezés megoldásának gettere.

SequenceTime double : A sorrendtervezés megoldásának a robotkarra történő bejárásának ideje.

CollisionfreePath : List< Config >: Az ütközésmentes pályatervezés megoldása.

PathTime double : Az ütközésmentes pálya bejárási ideje.

5.5.2. Az osztály függvényei

public IntegarateServices(): Az osztály paraméter nélküli konstruktora.

public IntegrateServices(List<double> startConfig, List<double> endConfig): A konstruktor beállítja a robotkar kezdő és véghelyének értékét.

public AddWorkpieces(List< Worpiece >) : A munkadarabokhoz egy munkadarab listát ad hozzá.

public AddWorkpieces(List<double[][]> graspLocationMatrixes, List<double[][]> putawayLocationMatrixes) : A munkadarabokhoz egy munkadarab listát ad hozzá. A bemeneti paraméterek listájának homogén transzformációs mátrixain elvégzi az inverz kinematikát és munkadarabokat hoz létre belőlük.

public AddCollisionfreeWorkpieces(List< Worpiece >) : A bemeneti paraméterként megadott listán elvégzi az ütköző robotkonfigurációk szűrését és hozzáadja az osztály által tárolt munkadarabokhoz.

public AddCollisionfreeWorkpieces(List<double[][]> graspLocationMatrixes, List<double[][]> putawayLocationMatrixes) : A munkadarabokhoz egy munkadarab listát ad hozzá. A bemeneti paraméterek listájának homogén transzformációs mátrixain elvégzi az inverz kinematikát, ütköző konfigurációk szűrését és munkadarabokat hoz létre belőlük.

public void SetHilClimbingSerach(): A következő sorrendtervezést a hegymászó algoritmussal futtatja.

public void SetTabuSearch(int nanos): A következő sorrendtervezést a tabu algoritmussal futtatja, nanos-ban (ns) megadott ideig.

public void SequencePlanning(): Futtatja a sorrendtervezést.

public void BuildOpenArmPRM(): A függvény segítségével nyílt pofa állású megfogóval építünk egy PRM keresési térképet.

public void SaveOpenArmPRM(string fileName): Lementi a nyílt pofa állású megfogó keresési térképét.

public void LoadOpenArmPRM(string fileName): Betölti a nyílt pofa állású megfogó keresési térképét.

public void BuildClosedArmPRM(): A függvény segítségével zárt pofa állású megfogóval építünk egy PRM keresési térképet.

public void SaveClosedArmPRM(string fileName): Lementi a zárt pofa állású megfogó keresési térképét.

public void LoadClosedArmPRM(string fileName): Betölti a zárt pofa állású megfogó keresési térképét.

private List<Config> GetSmoothedPath(Config a, Config b, IMultiQueryPathPlanner prm): Az a-ból b-be egy ütközésmentes simított pályát tervez a prm-segítségével.

public void PathPlanning(): Futtatja a ütközésmentes pályatervezési eljárást.

6. fejezet

Eredmények dec 3

Eredmények

7. fejezet

Értékelés dec 3

Értékelés

Köszönetnyilvánítás

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

Irodalomjegyzék

[1] Sergey Alatartsev – Sebastian Stellmacher – Frank Ortmeier: Robotic task sequencing problem: A survey. *Journal of Intelligent & Robotic Systems*, 80. évf. (2015. Nov) 2. sz., 279–298. p. ISSN 1573-0409.
URL https://doi.org/10.1007/s10846-015-0190-6.

[2] Keld Helsgaun: Solving the equality generalized traveling salesman problem using the lin–kernighan–helsgaun algorithm. *Mathematical Programming Computation*, 7. évf. (2015. Sep) 3. sz., 269–287. p. ISSN 1867-2957.
URL https://doi.org/10.1007/s12532-015-0080-8.

[3] Sébastien Mouthuy – Pascal Van Hentenryck – Yves Deville: Constraint-based very large-scale neighborhood search. *Constraints*, 17. évf. (2012. Apr) 2. sz., 87–122. p. ISSN 1572-9354.
URL https://doi.org/10.1007/s10601-011-9114-7.

[4] Stuart J. Russell – Peter Norvig: *Artificial Intelligence: A Modern Approach (2nd Edition)*. 2002. December, Prentice Hall. ISBN 0137903952. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0137903952.

[5] Bruno Siciliano – Oussama Khatib: *Springer Handbook of Robotics*. Berlin, Heidelberg, 2007, Springer-Verlag. ISBN 354023957X.

Függelék