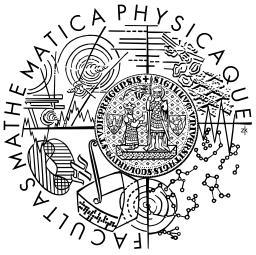


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Tomáš Karella

Evoluční algoritmy pro řízení heterogenních robotických swarmů

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Martin Pilát, Ph.D.

Studijní program: Informatika

Studijní obor: Programování a Softwarové Systémy

Praha 2017

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Poděkování.

Název práce: Evoluční algoritmy pro řízení heterogenních robotických swarmů

Autor: Tomáš Karella

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Martin Pilát, Ph.D., katedra

Abstrakt: Abstrakt.

Klíčová slova: klíčová slova

Title: Evolutionary Algorithms for the Control of Heterogeneous Robotic Swarms

Author: Tomáš Karella

Department: Department of Theoretical Computer Science and Mathematical Logic at Faculty of Mathematics and Physics

Supervisor: Mgr. Martin Pilát, Ph.D., department

Abstract: Abstract.

Keywords: key words

Obsah

Úvod **3**

1 Evoluční algoritmy **4**
1.1 Historie . . . 4
1.2 Co je evoluční algoritmus? . . . 4
1.3 Části evolučních algoritmů . . . 6
1.3.1 Reprezentace . . . 6
1.3.2 Populace . . . 7
1.3.3 Selekce rodičů . . . 7
1.3.4 Variační operátory . . . 8
1.4 Differciální evoluce . . . 10
1.5 Evoluční strategie . . . 11

2 Robotický Swarm **13**
2.1 Základní vlastnosti . . . 13
2.2 Použití . . . 15
2.3 Řízení robotických swarmů . . . 15

3 Simulátor **16**
3.1 Členění programu: . . . 16
3.1.1 SwarmSimFramework . . . 16
3.1.2 Vedlejší projekty . . . 16
3.2 SwarnSimFramework: . . . 17
3.2.1 Úvod: . . . 17
3.2.2 Hlavní třídy: . . . 17
3.2.3 Map . . . 18
3.2.4 Rozhraní . . . 19
3.2.5 Efektory a sensory: . . . 20
3.2.6 Entity . . . 21
3.2.7 Experiment: . . . 23
3.2.8 MultiThread . . . 23
3.2.9 RobotBrains . . . 23
3.2.10 Externí knihovny, NuGet . . . 24
3.2.11 Support třídy . . . 24
3.3 Ostatní projekty: . . . 25
3.3.1 SwarmSimVisu: . . . 25
3.3.2 SimpleNetworking: . . . 25
3.3.3 InterSection2d: . . . 25
3.4 Prostředí . . . 25
3.5 Implementace scénářů . . . 25
3.6 Roboti . . . 25
3.6.1 Sensory . . . 25
3.6.2 Efektory . . . 25
3.6.3 Experimenty . . . 25

4 Experimenty **26**
4.1 Úvod 26
4.2 Použité algoritmy 26
4.3 Experimenty 26
4.3.1 Wood Scene 26
4.3.2 Mineral Scene 27
4.3.3 Competitive Scene 27

Závěr **28**

Seznam obrázků **29**

Seznam tabulek **30**

Seznam použitých zkratek **31**

Přílohy **32**

Úvod

Využití robotických hejn(robotic swarms) patří mezi rentabilní metody pro řešení složitějších úkolů. Zdá se, že velký počet jednoduchých robotů dokáže plně nahradit komplexnější jedince. Dostatečná velikost hejna umožní řešení úloh, které by jednotlivec z hejna provést nesvedl. Navíc přináší několik výhod, díky kvantitě jsou odolnější proti poškození či zničení, neboli zbytek robotů pokračuje v plnění cílů. Dále výroba jednodušších robotů vychází levněji než komplexní jedinců, což přináší vhodnou výhodu pro práci v nebezpečném prostředí. Hejno také může pokrývat vícero různých úkolů než specializovaný robot, který bude při plnění úkolů lišících se od typu úloh zamýšlených při konstrukci mnohem více nemotorný a nejspíše pomalejší. Hejno pokryje větší plochu při plnění úkolů.

Existuje mnoho aplikací robotických hejn, vetšinou se používají k úlohů týkajících se průzkumu a mapování prostředí, hledání nejkratších cest, nasazení robotů v nebezpečných místech (**?**). Jako příklad můžeme uvést asistenci záchraným složkám při požáru (**?**). Mnoho projektů zabývající se řízením robotického hejna se inspiruje přírodou, používá se analogie k chování mravenců a jiného hmyzu (**?**). Objevují se i harwarové implementace chování hejn, zmiňme projekty Swarm-bots (**?**), Colias (**?**)

Elementárnost senzorů i efektorů jednotlivých robotů vybízí k použití genetického programování, jelikož prostor řešení je velmi velký a plnění cílu lze vhodně ohodnotit. Dokonce na toto téma také vzniklo několik vědeckých prací (**?**) (**?**).

Cíl práce

Všechny zmíněné práce používají pro tvorbu řídicích programů genetické programování (GP) pracují s homogenními hejny. Cílem této práce je vyzkoušet využití GP na generování chování hejna heterogenních robotů, tedy skupiny robotů, kde se objevuje několik druhů jedinců a společně plní daný úkol. V rámci práce byl sestaven program pro simulaci různých scénářů a pro jejich úspěšnosti v rámci GP. Byli zvoleny 3 odlišné scénáře, kde se objevují 2-3 druhy robotů.

Struktura práce

Rozdělení práce je následující. První kapitola je věnována obecnému úvodu do tématiky evolučních algoritmů, kde se více podrobněji věnuji Evolučním Strategiím a Diferenciální Evoluci, protože oba tyto postupy implementuji v programu pro řešení scénářů.

1. Evoluční algoritmy

1.1 Historie

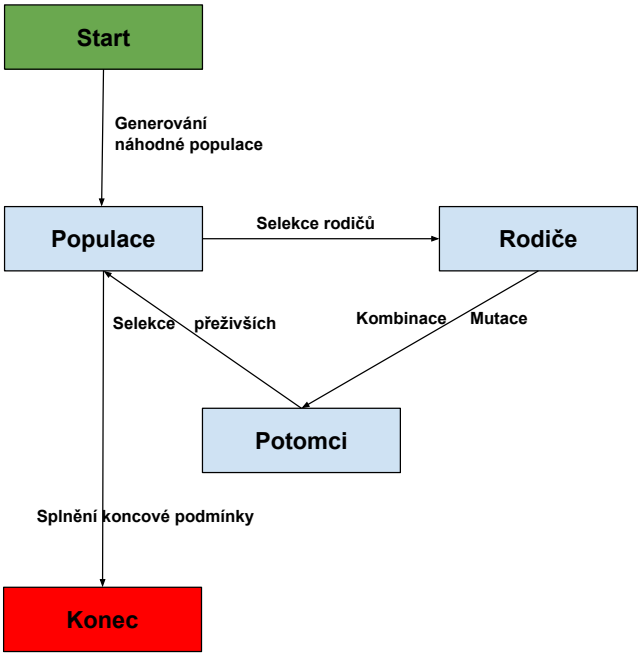
Začněme pohledem do historie Evolučních algoritmů na základě knih (**?**) a (**?**). Darwinova myšlenka evoluce lákala vědce už před průlomem počítačů, Turing vyslovil myšlenku *genetického a evolučního vyhledávání* už v roce 1948. V 50. a 60. letech nezávisle na sobě vznikají 4 hlavní teorie nesoucí podobnou myšlenku. Společným základem všech teorií byla evoluce populace kandidátů na řešení daného problému a jejich následná úprava způsoby hromadně nazývány jako genetické operátory, například mutace genů, přirozená selekce úspěšnějších řešení.

Rechenberg a Schwefell (1965, 1973) představuje *Evoluční strategie*, metoda optimalizující parametry v reálných číslech, jejich použití pro letadlová křídla. Fogel, Owens, Walsh zveřejňují *evolutionary programming*(evoluční programování), technika využívající k reprezentaci kandidátů konečný automat(s konečným počtem stavů), který je vyvíjen mutací přechodů mezi stavy a následnou selekcí. *Genetické algoritmy* vynalezl Holand v 60. letech a následně se svými studenty a kolegy z Michiganské Univerzity implementoval, oproti ES a EP nebylo hlavním cílem formovat algoritmus pro řešení konkrétních problémů, ale přenos obecného mechanismu evoluce jako metody aplikovatelné v informatickém světě. Princip GA spočívá v transformaci populace chromozonů(př. vektor 1 a 0) v novou populaci pomocí genetických operátorů křížení, mutací a inverze. V 1975 v knize *Adaptation in Natural and Artificial Systems* (**?**) definoval genetický algoritmus jako abstrakci biologické evoluce spolu s teoretickým základem jejich používání. Ovšem někteří vědci používají pojem GA i ve významech hodně vzdálených původní Holandově definici. K sjednocení jednotlivých přístupů přispěl v 90. letehc Koza, dále jsou všechny zahrnuty jako oblasti *Evolučních algoritmů*. Dnes existuje řada konferencí a odborných časopisů sdružující pracovníky zabývající se touto oblastí. Zmiňme ty větší z nich, co se týče konferencí: GECCO, PPSN, CEC, EVOSTAR, časopisy: Evolutionary Computation, IEEE Transactions on Evolutionary Computation, Genetic Programming and Evolvable Machines, Swarm and Evolutionary Computation

1.2 Co je evoluční algoritmus?

Ač existuje mnoho variant evolučních algoritmů, jak jsme zmínili v krátké pohledu do historie, spojuje je společná myšlenak populace jedinců uvnitř prostředí s omezenými podmínkami. Jedinci, jinak také nazývání kandidáti, soutěží o zdroje daného prostředí, tím je docíleno přírodní selekce(, přežijí jen Ti nejlepší). Pokud budeme mít k dispozici kvalitativní funkci, kterou se snažíme maximalizovat. Pak nebude problém vytvořit náhodné jedince z definičního oboru přesně této funkce. Náhodně vzniklé jedince můžeme ohodnotit, tímto způsobem dáme vzniku abstraktu pro měření fitness(, z anglického fit nejvíce vhodný). Z vzniklých a ohodnocených jedinců lze zvolit ty nejlepší pro tvorbu nové generace jedinců. Tvorba nové generace probíhá kombinováním zvolených rodičů a mutacemi jedinců. Jako kombinaci uvažujeme operátor, který je aplikován na 2-více zvolených kandidátů(, proto se jim také říká rodiče,) a tvořící 1-více nových jedinců(,

také nazývány děti). Mutace je aplikována pouze na 1 jedince a její výsledkem je také pouze 1 jedinec. Tyto dvě operace aplikované na rodičovskou generaci vedou k vytvoření nových kandidátů(potomků,offsprings). I tato nová generace je ohodnocena fitness a dále soutěží se starými jedinci na základě fitness(, občas také v závislosti na stáří kandidáta,) o místo v nové generaci. Popsaný proces je opakován dokud není nalezen kandidát s dostatečně velkou fitness nebo narazíme na výpoční limit(, bylo dosaženo požadovaného počtu opakovaní apod). Základy Evolučního systému pohání 2 základní hnací síly: variační operátory, selektivní operátory. Variační operátory zajišťují potřebnou různorodost v populaci a tím tvoří nové cesty k úspěšnému kandidátovi. Oproti tomu selektivní operátory zvyšují průměrnou kvalitu řešení v celé populaci. Kombinací těchto operátorů obecně vede ke zlepšování fitness hodnot v následující generaci. Celkem jednodušše lze vidět, zda evoluce optimalizuje či nikoliv, stačí nám k tomu pozorovat zda se fitness v popolaci blíží více a více k optimálním hodnotám vzhledem k postupu v čase. Mnoho komponent zapříčiňuje, že EA se řadí ke stochastickým metodům, selekce nevybírá nejlepší jedince deterministicky i jedinci s malou fitness mají šanci být rodiči následující generace. Během kombinování jsou části rodičů , kterou budou z zkombinovány, také zvoleny náhodně. Podobně u mutací, část která bude změněna je taktéž určena náhodně, nové rysy nahrazující staré jsou generovány taktéž náhodně.



Ze schéma na obrázku můžeme vyčíst, že EA patří mezi algoritmy ëenerate and test (vygeneruj a otestuj)¨ Vyhodnocení fitness funkce poskytuje heuristický odhad kvality řešení, prohledávaní je řízen variací a selekcí. EA splňují charakteris-

tické rysy G&T algoritmů, zpracovávají zároveň celé kolekce kandidátů, většina EA míchá informace ze 2-více kandidátů a EA se řadí ke stochastickým metodám.

Různé dialekty evolučního programování, zmíněné v historickém okénku, splňují všechny tyto hlavní rysy a liší se pouze v technických detailech. Kandidáti jsou často reprezentování různě, liší se datové struktury pro jejich uchování i jejich zakódování. Typicky se jedná o řetězce nad konečnou abecedou v případě GA, vektory reálných čísel v ES, konečné automaty v EP a stromy v GP. Důvod těchto rozdílů je hlavně historický. Technicky však lze upřednostit jednu reprezentaci, pokud lépe odpovídá danému problému, tzn. kóduje kandidáta jednodušeji či přirozenější formou. Pro ilustraci zvolme řešení splnitelnosti(SAT) s n proměnnými, vcelku přirozeně sáhneme po použití řetězce bitů délky n a obsah i-tého bitu označuje, že i-tá proměnná má hodnotu (1)- pravda (0) - nepravda. Proto bychom použili jako vhodný EA Genetické Algortimy. Oproti tomu evoluce programu, který hraje šachy, by bylo vhodné použít derivační strom, kde by vrcholy odpovídali tahům na šachovnici. Přirozenější by tedy bylo použít GP.

Neopomeňme zmínit ještě dva důležité fakt. Za prvé kombinační a mutační operátory musí odpovídat dané reprezentaci, např. v GP kombinace pracuje se stromy(kombinuji podstromy..), zatímco v GA na řetězcích(prohazuji části řetězců). Za druhé oproti variacím musí fitness selekce záviset vždy pouze na fitness funkci, takže selekce pracuje nezávisle na reprezentaci.

1.3 Části evolučních algoritmů

Abychom vytvořili běhu schopný evoluční algoritmus, musíme specifikovat každou zmíněnou část a také inicializační funkci, která připraví první populaci. Pro konečný algoritmus ještě nesmíme opomenout a dodat koncovou podmínku.

- Reprezentace(definici individuí)
- Ohodnocující funce(Fitness funce)
- Populace
- Selekce rodičů
- Variační operátory(kombinace, mutace)
- Selekce prostředí

1.3.1 Reprezentace

Při tvorbě EA nejdříve musíme propojit prostor původního problému s prostorem řešení, kde se bude vlastní evoluce odehrávat. K docílení propojení je většinou potřeba zjednošit či vytvořit abstrakci nad aspekty reálného světa(prostor problému), abychom vytvořili vhodný prostor řešení, kde mohou být jednotlivá řešení ohodnocena. Neboli chceme docílit, aby možná řešení mohla být specifikována a uložena, tak aby s nimi mohl počítač pracovat. Objekty reprezentující možné řešení v kontextu původního problému jsou názývána **fenotyp**, zatímco kódování na straně EA prostoru **genotyp**. Reprezentace označuje mapování z fenotypů na genotypy, používá se ve významu funkci z fenotypu na genotyp(encode)

i genotypu na fenotyp(decode) a předpokládá se, že pro každý genotyp existuje nejvýše jeden fenotyp. Například pro optimalizační problém, kde množinou řešení jsou celá čísla. Celá čísla tvoří množinu fenotypu a můžeme se rozhodnout pro reprezentaci v binárních číslech. Tedy 18 by byl fenotyp a 10010 jeho genotyp. Prostor fenotypů a genotypů se zpravidla velmi liší a celý proces EA se odehrává pouze v genotypovém prostoru, vlastní řešení dostanema rozkódováním nejlepšího genotypu po ukončení EA. Jelikož nevíme, jak vlastní řešení vypadá, je nanejvýš vhodné umět reprezentovat všechny možná řešení, v G&T bychom řekli, že generátor je kompletní.

- V kontextu původního problému jsou následující výrazy ekvivaletní: fenotyp, kandidát(na řešení), jedinec, individuum (Množina všech možných řešení = fenotypový prostor)
- V kontextu EA: getotyp, chromozon, jedinec, individuum (Množina kde probíhá EA prohledávání = genotypový prostor)
- Části individuí jsou nazývány gen, locus, proměnná a dále se dělí na allely, či hodnoty

1.3.2 Populace

Populace je multimnožina genotypů, slouží jako jednotka evoluce. Populace utváří adaptaci a změny, zatímco vlastní jedinci se nijak nemění, jen vznikají nový a nahrazují předešlé. Pro danou reprezentaci je definice populace velmi jednoduchá charakterizuje ji pouze velikost. U některých specifických EA má populace další prostorové struktury, definované vzdáleností jedinců nebo relacemi sousedních jedinců. Což by se dalo připodobnit, reálným populacím, které se vyvíjejí v různých geografických prostředích. U většiny EA se velikost populace nemění, což vede k soutěživosti mezi jedinci(,zůstanou ti nejlepší). Na úrovni populací pracují právě selektivní operátory. Například zvolí nejlepší jedince aktuální populace jako rodiče následující, nahradí nejhorší jedince novými. Rozmanitost populace, vlastnost které chceme zpravidal docílit, je měřena jako počet různých řešení v multimnožině. Neexistuje však jediné hledisko podle, kterého lze tuto vlastnost měřit, většinou se používá počet rozdílných hodnot fitness, rozdílných fenotypů či genotypů a také například entropie(míra neuspořádanosti). Ovšem musíme mít na paměti, že jedna hodnota fitness v populaci neznamená, že populace obsahuje pouze jeden fenotyp, stejně tak jeden fenotyp nemusí odpovídat jednomu genotypu apod.

1.3.3 Selekce rodičů

Rodičovská selekce, někdy také partnerská selekce, vybírá lepší jedince jako rodiče pro příští generaci. Jedinec se stává rodičem, pokud byl zvolen k aplikaci variačních operátorů a tím dal vzniknout novým potomkům. Společně s selekcí přeživších je rodičovská selekce zoodpovědná za zvedání kvality v populacích. Rodičovská selekce je v EA většinou pravděpodobnostní metoda, která dává jedincům s větší kvalitou mnohem větší šanci býti vybrán než těm nízkou. Nicméně i jedincům s nízkou kvalitou je často přidělena malá nenulová pravděpodobnost

pro výběr, jinak by se celé prohledávání mohlo vydat slepou cestou a zaseknout se na lokální optimu.

1.3.4 Variační operátory

Hlavní úlohou variačních operátorů(mutace, rekombinaci) je vytváření nových individuí ze starých. Z pohledu Generate Test algoritmů spadají variační operátory do právě do Generate části. Obvykle je dělíme na dva typy podle jejich arity, jedná se o unární(Mutace) a n-arní(rekombinace) operátory.

Mutace

Ve většině případá myslíme unárním variačním operátorem mutace. Tento operátor je aplikován pouze na jeden genotyp a výsledkem je upravený potomek. Mutace řadíme mezi stochastické metody, výstup(potomek) totiž závisí na sérii náhodných rozhodnutí. Však mutace není vhodné pojmenování pro všechny variační operátory, například pokud se jedná o heuristiku závislou na daném problém, která se chová systematicky hledá slabé místo a následně se jej snaží vylepšit, nejedná se o mutaci v pravém slova smyslu. Obecně by mutace měla mutace způsobovat náhodné a nezaujaté změny. Unarní variační operátory hrají odlišnou roli v rozdílných EA opět díky historicky oddělenému vývoji. Zatímco v GP se nepoužívají vůbec, v GA má velmi důležitou roli a v EP se jedná o jediný variační operátor. Díky variačním operátorům aplikovaným v jednotlivých evolučních krocích dostává prohledávací prostor topologickou strukturu. Existují teorie formulící, že EA(s dostatečným časem) naleznou globální optimum daného problému opírající se právě o tuto topologii, spoléhají na vlastnost, že může vzniknout každý genotyp reprezentující možné řešení. Nejjednodušší cesta ke splnění těchto podmínek vede právě přes variační operátory. U mutací tohoto například dosáhneme, pokud povolíme, aby mohly skočit kamkoliv, tzn. každá allela může být zmutována na jakoukoli další s nenulovou pravděpodobností. Většina vědecká společnosti považuje tyto důkazy za nepříliš použitelné v praktickém využití a proto tuto vlastnost většina EA neimplementuje.

Rekombinace

Rekombinace, také nazývána křížení, je binární variační operátor. Jak název napovídá, spojuje informace ze 2 rodičů(genotypů) do 1 nebo 2 potomků. Stejně jako mutace patří rekombinace k stochastickým operátorům. Rozhodnutí, jaké části budou zkombinovány a jakým způsobem tak bude docíleno závisí na náhodě. Role rekombinace se znovu liší v rozličných EA, v GP se jedná o jediné variační operátory, v EP nejsou použity vůbec. Rekombinační operátory s vyšší aritou(, používající více než 2 rodiče,) jsou možné a jednoduché na implementaci, dokonce několik studií potvrdilo, že mají velmi pozitivní vliv na celou evoluci, ale nemají tak hojné zastoupení, nejspíše proto, že neexistuje biologický ekvivalent. Rekombinace funguje na jednoduchém principu, slučuje 2 individua a může vyprodukovat potomky, které kombinují jejich výhodné vlastnosti, tedy potomek je úspěšnější než jeho rodiče. Tento princip podporuje fakt, že po 1000-letí se aplikací rekombinace na rostliny a zemědělská zvířata mnokrát podařilo vytvořit nové

jedince s vyšším výnosem či jinými výhodnými vlastnostmy(odolnosti pro škůdcům,atd). Evoluční algoritmy vytváří množství potomků náhodnou rekombinací a doufáme, že zatímco malá část nové generace bude mít nežádoucí vlastnosti, většina se nezlepší ani si nepohorší a konečně další malá část předčí jejich rodiče. Na naší planetě se sexuálně(kombinací dvou jedinců) rozmnožují pouze vyšší organismy, což vzbuzuje dojem, že rekombinace je nejvyšší forma reprodukce. Neopomeňme, že variační operátory jsou závislé na reprezentaci(genotypu) jedinců. Např. pro genotypy bitových řetězců může použít bitovou inverzi jako vhodný mutační operátor ovšem pokud bude genotyp strom musíme zvolit nějaký jiný.

Selekce prostředí

Selekce prostředí, někdy také nazývána selekce přeživších, jak název napovídá má blízko k selekci rodičů, odlišuje jednotlivá individua na základě jejich fitness hodnoty. Oproti rodičovské selekci ji však používáme v jiné části evolučního cyklu, selekce prostředí spouštíme hned po vytvoření nových potomků. Jelikož velikost populace kandidátů bývá skoro vždy konstatní, tak je nutné rozhodnout které kandidáty zvolit do další generace. Toto rozhodnutí většinou závisí na fitnes kandidátů, také se však hledí na věk daného kandidáta(,generace v které vznikl). Na rozdíl od rodičovské selekce, která bývá stochastická, bývá selekce prostředí deterministickou metodou. Uveďme dvě nejběžnější metody, obě kladou největší důraz na fitness, první vybírá na nejlepší segment z množiny nových potomků i původní generace nezávisle, druhá dělá totéž jen z množiny nových potomků. Selekce prostředí je uváděna i pod názvem záměná strategie(replacement strategy), selekce prostředí(přeživších) se více používá pokud množina nových potomků je větší než velikost generace a záměna, když nových potomků je velmi málo.

Incializace populace

Inicializace populace zastává důležitý úkol vytvořit první generaci kandidátů. Zpravidla bývá ve většině EA její implementace velmi jednoduchá, první generace je vygenerována čistě náhodně. Principiálně by se zde dalo využít nějaké heuristiky k vytvoření vyšší fitness v první generaci, ovšem musela by zohledňovat řešený problém. Jestli tento postup stojí za výpočetní čas ne velmi záleží na konkrétní aplikaci EA. Ovšem existují obecná doporučení, která se touto otázkou zabývají.

Ukončovací podmínka

Rozlišme dvě varianty vhodné ukončovací podmínky. Pokud řešený problém má známou optimální hodnotu fitness, potom v ideální případě ukončovací podmínkou je řešní s touto fitness. Pokud jsme si vědomi, že náš model oproti modelovanému prostředí obsahuje nutná zjednoduššení nebo by se v něm mohly vyskytovat nežádoucí šumy, lze se spokojit s řešením, které dosáhlo optima fitness s přesností $\epsilon > 0$. Nicméně, EA díky své stochastičnosti většinou nemohou garantovat dosažení takovéto hodnoty fitness, takže ukončovací podmínka by nemusela být nikdy splněna a algoritmus by běžel věky. Kdybychom vyžadovali konečnost algoritmu, tak musíme rozšířit ukončovací podmínku. Zatímto účelem se nejběžněji používají následující rozšíření:

1. Byl překročen čas vyhrazený pro počítání na CPU
2. Celkový počet evaluací fitness přesáhl svůj předem daný limit
3. Zlepšení fitness v dané časovém bloku(, měřenému počtem generací, či evaluací) klesla pod přípustný práh
4. Rozmanitost v populaci nedosahuje předem určených hodnot

Prakticky ukončovací podmínka bývá disjukce: dosáhli jsme optima or podmínka X ze seznamu. Může se stát, že optimum známé není. Pak se používá pouze zmíněných podmínek nebo se smíříme s nekonečností algoritmu.

1.4 Differciální evoluce

Differrenciální evoluce(DE) se objevila v roce 1995, kdy Storn a Price vydali technický report TODO:citace popisující hlavní koncept skrývající se za pojmem DE N̈ew heuristic approach for minimizing possibly nonlinear and nondifferentiable continuous space functions.̈ Tento evoluční algoritmus popíši trochu více, protože je klíčovým algoritmem mého řešení.

Charakteristické vlastnosti

DE dostala své jméno hlavně díky změnám obvyklých reprodučkních operátorů v EA. Differciální mutace, jak jsou mutace v DE nazývány, z dané populace kandidátů vektorů v $\mathbb{R}^n$ vzniká nový mutant $\overline{x}'$ přičtením pertubačního vektoru(pertubation vector) k existujícímu kandidátovi.

$\overline{x}' = \overline{x} + \overline{p}$

Kde pertubační vektor $\overline{p}$ je vektor rozdílů dvou náhodně zvolených kandidátů z populace $\overline{y}a\overline{z}$.

$\overline{p} = F(\overline{y} - \overline{z})$

Škálovací faktor $F > 0$, $F \in \mathbb{R}$, který kontroluje míru mutace v populaci. Jako rekombinační operátor v DE slouží uniformní křížení, pravděpodobnost křížení je dáne $C_r \in [0,1]$, která definuje šanci jakékoli pozice v rodiči, že odpovídající allela potomka bude brána z 1. rodiče. DE také upravuje křížení, neboť 1 náhodná allela je brána vždy z 1. rodiče, aby nedocházelo k duplikaci 2. rodiče.

V hlavních implementacích DE reprezentují populace spíše listy, odpovídají lépe než množiny, umožňují referenci i-tého jedince podle pozice $i \in 1,..,\mu$ v listu. Pořadí kandidátů v této populaci $P =< \overline{x_1}...\overline{x_i}...\overline{x_\mu} >$ není závislé na hodnotě fitness.Evoluční cyklus začíná vytvořením populace vektorů mutantů $M =< \overline{v_1}...\overline{v_\mu} >$. Pro každého nového mutanta $\overline{v_i}$ jsou zvoleny 3 vektory z P, base vektora a dva definující pertubační vektor. Po vytvoření populace mutantů V, pokračujeme tvorbou populace $T =< \overline{u_1},....\overline{u_\mu} >$, kde $\overline{u_i}$ je výsledek aplikace křížení $\overline{u_i}$ je výsledek křížení $\overline{v_i}$ a $\overline{x_i}$(všimněme si, že je zaručené, že nezreplikujeme $\overline{x_i}$). Jako poslední aplikujeme selekci na každý pár $\overline{x_i}$ a $\overline{u_i}$ a do další generace vybereme $\overline{u_i}$ pokud $f(\overline{u_i}) \leq f(\overline{x_i})$ jinak $\overline{x_i}$.

DE algoritmus upravují 3 parametry, ?scalovací? faktor F, velikost populace μ (obvykle značen NP v DE literatuře) a pravděpodobnost křížení C_r. Na C_r lze také pohlížet jako na míru mutace, tzn. allela nebude oddděna od mutanta.

Vlastnosti Differciální Evoluce:

Reprezentace:	vektor $\mathbb{R}^n$
Rekombinace:	uniformní křížení
Mutace:	differeční mutace
Rodičovská selekce:	uniformní náhodné selekce 3 nezbytných vektorů
Selekce prostředí:	deterministická selekce elity (dítě vs rodič)

Varianty DE:

Během let, vzniklo a bylo publikováno mnoho variant DE. Jedna z modifikací zahrnuje možnost base vektoru, když vytváříme mutatnské populace M. Může být náhodně zvolen pro každé v_i, jak bylo řečeno, ale také lze využít jen nejlepšího vektoru a nechat změny na pertubačním vektoru. Další možnost otevírá použitím více pertubačních vektorů v mutačním operátoru. Což by vypadalo následovně:

$\overline{p} = F(\overline{y} - \overline{z} + \overline{y}' - \overline{z}')$, kde $\overline{y}, \overline{z}, \overline{y}', \overline{z}'$ jsou náhodně vybrány z původní populace.

Abychom odlišili různé varianty, používá se následující notace DE/a/b/c, kde **a** specifikuje base vektor(rand, best), **b** specifiuje počet differečních vektorů při vzniku pertubačního vektoru, **c** značí schéma křížení (bin=uniformní). Takže podle této notace by základní verze DE byla zapsána takto: DE/rand/1/bin.

1.5 Evoluční strategie

Evoluční strategie(Evolution Strategies) byly představeny na počátku 60. let pány Rechenbergem a Schwefelem (**?**), oba jmenovaní pánové se zabývali optimalizací tvarů křídel na Technické univerzitě v Berlíně. Jedná se o evoluční algoritmus cílící na optimalizaci vektorů reálných čísel. Objevili si dvě schémata (1+1) a (1,1). Starší (1+1) (one plus one ES) vytvářejí potomka mutací a to přičtením náhodných nezávislých hodnot ke složkám rodičovského vektoru, následně je potomek přijat, pokud získal lepší fitness než jeho rodič. Jako další vznikl (1,1) (one comma one ES), které neimplementovalo elitismus, tzn. měnila vždy rodiče potomkem. Mutační funkce bere náhodná čísla z Normální rozdělení s se střední hodnotou 0 a odchylkou σ. σ se také nazývá velikost mutačního operátoru(mutation step size). Vylepšení původního algoritmu bylo představeno v 70. letech, jedná se o koncept multisložkových ES, které se skládají ze μ jedinců(velikost populace) a λ jedinců vygenerovaných během jednoho cyklu. Opět tento koncept existuje ve 2 verzích (μ,λ) a ($\mu + \lambda$). Zatímco bežné rekombinační schéma zahrnuje dva rodiče a jednoho potomka, intermediate křížení, které se u ES používá, zprůměruje hodnoty z rodičovských alel. Tímto způsobem můžeme používat i rekombinační operátory s použitím více než 2 rodičů, tyto operátory se nazývají globalní rekombinace. Konkrétně se na potomkovi podílí λ rodičů. V praxi se preferuje (μ,λ) před ($\mu + \lambda$), protože zahazuje všechny rodiče tím pádem se snadno nezastaví na lokálním optimu, (μ,λ) se zvládá lépe adaptovat i při hledání pohyblivého optima. Pro typické použití se používá poměr 1:4 a 1:7.

Vlastnosti Evolučních strategií:

Reprezentace:	vektor $\mathbb{R}^n$
Rekombinace:	intermediary křížení
Mutace:	přičítání náhodných hodnot z norm. rozdělení
Rodičovská selekce:	uniformní náhodná
Selekce prostředí:	(sigma,lambda) nebo (sigma + lambda)

Následuje krátký jednoduchý příklad ES v jazyku Pythonu pro lepší pochopení, jsem použil kód ze stránky (**?**). Cílem tohoto velmi jednoduché programu je nalézt vektor čísel, který se co nejvíce blíží vektoru solution

```
solution = np.array([0.5, 0.1, -0.3])
#fitness funkce, kter pocita vzdalenost
#mezi vlozenym vektorem a solution
def f(w): return -np.sum((w - solution)**2)

npop = 50
sigma = 0.1
# learning rate
alpha = 0.001
w = np.random.randn(3)
for i in range(300):
# vygenerujme npop mutaci k aktualnimu reseni
  N = np.random.randn(npop, 3)
  # vektor R slouzi k uchovani fitness hodnot
  R = np.zeros(npop)
  # Pro kazdou vyrobenou mutaci
  for j in range(npop):
  # Aplikujeme mutaci na aktualniho jedince
    w_try = w + sigma*N[j]
    # Ohodnotime uspesnost zmutovaneho jedince
    vzhledem k fitness funkci
    R[j] = f(w_try)
  A = (R - np.mean(R)) / np.std(R)
  # upravime aktualni reseni mutacemi,
  # cim uspesnejsi mutace,
  # tim vetsi vliv do aktualniho jedince.

  w = w + alpha/(npop*sigma) * np.dot(N.T, A)
```

2. Robotický Swarm

V češtině se také používá výraz Rojová Robotika nebo Robotický Roj, v angličtině je známý pod pojmem Swarm Robotics. Myšlenka Robotického Swarmu pochází podobně jak u Genetických Algoritmů z inspirace matkou Přírodou. Podle souhrnu (**?**) popíši základní myšlenku RS.

2.1 Základní vlastnosti

Motivací pro použití RS může být chování živočichů na Zemi, když se zaměříme na skupiny živočichů jako jsou mravenci, včely, ryby a dokonce i někteří savci. Pokud bychom vložili do prostředí jednotlivce z některé ze zmíněných skupin, nebude schopen konkurovat nepřátelskému prostředí a nejspíše příliš dlouho nepřežije. Na druhou stranu, když budeme uvažovat celé společenství, tak se nám ze slabého jedince stane velmi adaptivní, odolný a rychle se vyvíjející roj. Podobnému účinku bychom se chtěli přiblížit v RS. Pro relativně jednoduchého robota, který není schopen plnit obtížný úkol, se pokusíme použít vícero robotů stejného typu, kteří společně zadaný úkol vyřeší. Navíc chceme těžit ze všech výhod hejna.

Jako nejčastější výhody RS oproti jednomu robotovi se nejčastěji uvádějí:

1. Paralelnost - Díky malé ceně jedince, si můžeme dovolit velkou populaci jedinců. Malou cenou jedince v ES myslíme jednoduchý robot s malou pořizovací cenou, v kontextu živočichů můžeme uvažovat množství energie, jídla pro tvorbu takového jedince. Velká populace nám umožňuje řešit vícero úkolů na ráz, také na velké ploše. Zvláště pro vyhledávací úkoly ušetříme nemalé množství času.

2. Škálovnatelnost - Změna velikosti populace hejna neovlivniví chování ostatních jedinců. Samozřejmě plnění úkolu bude rychlejší resp. pomalejší, ale původní hejno bude stále plnit původní úkol. Tím pádem můžeme celkem snadno upravovat velikost populace bez větší obtíží. V přírodě můžeme pozorovat, že smrt pár jednotlivých mravenců dělníků znatelně neovlivní práci celého mraveniště. Nově narození mravenci se mohou vydat do práce, zatímco zbytek mraveniště nemění činnost.

3. Houževnatost - Související se škálovatelností, jen v tomto případě máme na mysli necílenou změnu populace. Jak v předchozím příkladu u smrti mravenců, část robotů ES může selhat z rozličných důvodů. Zbytek hejna však bude pokračovat k cíli i když ve výsledku jim bude jeho dosáhnutí trvat o něco déle. Což se nám může vyplatit v nebezpečných prostředích.

4. Ekonomické výhody - Cena návrhu a konstrukce jednoduchých robotů hejna vyjde většinou levněji než jeden specializovaný robot schopný uspokojit stejné požadavky. V dnešním světě výroba ve velkém množství vychází mnohem levněji než tvorba jednoho drahého konkrétního robota.

5. Úspora energie - Díky menší velikosti a složitosti jednotlivých robotů vyžadují mnohem menší množství energie. Což má za důsledek, že si u nich můžeme dovolit energetickou rezervu na delší čas. Navíc když je pořizovací

cena jednoho robota menší než náklady na dobití, tak díky škálovatelnosti můžeme pouze připojit nové roboty, což u drahého robota jde málokdy.

6. Autonomie a Decentralizace - V kontextu RS musí každý jediden hejna jednat autonomně, jedinci nejsou řízeny žádnou autoritou. Takže umí pracuje i při ztrátě komunikace. Opět se vychází z chování živých organismů. Pokud se chovají jedinci hejna dostatečně kooperativně. Tak mohou pracovat bez centrálního řízení, důsledkem toho se stává celé hejno ještě flexibilnější a odolnější, hlavně v prostředích s omezenou komunikací. Navíc hejno mnohem rychleji reaguje na změny.

Mimo RS existuje i řada jiných přístupů, které se inspirovaly životem hejn v přírodě. Občas jsou zaměňovány za RS, nejčastěji se jedná o multi-agentní systémy a sensorové sítě(sensor networks). V následující tabulce jsou popsány jejich nejklíčovější vlastnosti.

2.2 Použití

Existuje několik vědeckých prací, které studují a navrhují použití RS v reálném nasazení. Některé jsem zmínil už v úvodu této práce jako například hasičům asistující roboty (**?**). RS se ukázala také jako dobrá aplikace u ekologický pohrom, španělští vědci testovali jejich použití při úniku ropy (**?**), či hledání centra radiace (**?**). Některé neskončili u simulací a také využívali RS u fyzických robotů, u robotů na vodním povrchu (**?**).

2.3 Řízení robotických swarmů

Chování swarmů se řadí mezi velmi obtížné úkoly pro svět informatiky. Pro reprezentaci chování se využívá neuronových sítí, které se optimalizují pomocí nastavování vah jednotlivých perceptronů. Neboť se jedná o velký prostor vstupních informací ze sensorů a prostor pro interakci s prostředím je taktéž velmi rozsáhlý. Přímé prohledávání takto obřího prostoru nepřichází v úvahu, proto v poslední získavají na oblibě evoluční algoritmy. Mezi nejvíce používané patří Evoluční strategie.

TODO: Rozepsat

3. Simulátor

3.1 Členění programu:

3.1.1 SwarmSimFramework

Vlastní framework, definice rozhraní, vlastní kód simulace, příklady scénářů

3.1.2 Vedlejší projekty

- SwarmSimVisu - Visualizace průběhu simulace, prohlížení vygenerovaných chování.
- SimpleNetworking - Umožňuje provádět simulaci mapy na vzdáleném stroji přes TCP protokol.
- Intersection2D - Implementace jednoduchých průsečíků v 2D prostoru (přímky, úsečky, kružnice).

3.2 SwarnSimFramework:

3.2.1 Úvod:

Map je centrální třídou projektu, zajištuje vlastní průběh simulace. Uchovává jednotlivé entity(, roboty, překážky, palivo, minerály), volá vyhodnocení akcí pro aktivní entity(roboty) včetně počítání kolizí a interakcí s mapou(přesuny pasivních entit, vysátí paliva, atd..). Potomci třídy entita reprezentuje objekt v mapě, všechny jsou odděny od stejného předka. V celém projektu se vyskytují 2 tvary entit, konkrétně se jedná o úsečku(sensory, efektory) dále o kruh(roboti, překážky, minerály). Robot zastává pozici aktivní entity pohybující se na mapě a interagující s ostatními entitami. Ke komunikaci, pohybu a změnám se v mapě používá robot efektory a sensory. Sensory se používají ke čtení informací z mapy, tedy vrací vektor čísel representující rozdílné vlasnosti mapy(vzdálenostní sensory, rádiové etc.), zatímco efektory mají opačný účel, tedy ovlivňovat mapu a entity a přijímají vektor čísel jako nastavení konkrétního efektory. Sensor a Efektor jsou implementací rozhraní ISensor a IEffector(viz. další kapitola). Pro simulování chování robotů slouží tzv. "mozek", v programu IRobotBrain, který má za úkol z vektoru ze všech sensorů vytvořit vektor pro všechny efektory a tímto způsobem řídit chování robota.

Vývin jednotlivých mozků je řízen přes experiment, což je pojem implementován v projektu různými způsoby dle požadavků uživatele. (MultiThreadExperiment, Experiment). Jejich společným cílem je nastavení parametrů dané mapy(počet překážek, druhy robotů..), iterace přes simulační kroky, průběh generací mozků, změnu mozků(differenciální evoluce, mutace..). Následně ukládání mezivýsledků a zobrazování postupu daného experimentu.

3.2.2 Hlavní třídy:

- Sensors - sensory, které čtou data z mapy
- Efektory - interakce s mapou a ostatními entitami
- Entities - Reprezentace jednotlivých entit, které se vyskytují na mapě. Všechny entity jsou odděděny od abstraktního předka **Entity**.
- Experiments - jednotlivé parciální evoluce pro řešení daného úkolu, které počítají s visualizací
- Map - Reprezentace 2D prostředí, kde se všechny entity pohybují, zajištuje kontrolu kolizí a celý průběh simulací
- MultiThreadExperiment - jednotlivé parciální experimenty, optimalizované pro běh na více vláknech bez GUI.
- RobotBrains - Reprezentace jednotlivých mozků implementující interface IRobotBrain
- Robots - konkrétní reprezentace robotů

3.2.3 Map

Reprezentace 2D prostředí simulace. Mapa je daná obdélníkem o dané velikosti při konstrukci. Během konstrukce také dostává všechny entity ve výchozích pozicích, co se budou v prostředí vyskytovat, také vytvoří jejich klony, aby později bylo možné vrátit mapu do počátečního stavu. Existují 4 základní typy entit v mapě. Jedná se o Robots - aktivní entity(IRobotEntity), na které je při každém kroku mapy, zavolána nejdříve PrepareMove() a dále Move() v náhodném pořadí, aby žádný robot nebyl upřednostěn. PasiveEntities pasivní entity, buď překážky nebo jiné nezpracované materiály. (CircleEntity), FuelEntities- palivo vyskytující se v mapě, pokud je spotřebováno je odebráno z tohoto seznamu. Jako poslední RadioEntities, což je vrstva rádiových signálů, které se počítají jen pro speciální kolize.

MakeStep() je metoda provádějící jeden krok simulace. Mapa charakterizují 4 krajní body A,B,C,D, také aktuální cyklus(počet zavolaných MakeStep()).

- Kolize:
  - Pro CircleEntity vrací bool, zda s něčím koliduje.
  - pro LineEntity vrací průsečík s nejbližším objektem mimo fuel na něj je speciální metoda.
  - pro CircleEntity reprezentující rádiový sensor, vrací slovní všech průsečíku s rádiovými signály.
  - pro CircleEntity existuje metoda CollisionColor, která vrací všechny průsečíky v dosahu CircleEntity.
- SceneMap - konkrétní mapy pro jednotlivé experimenty, zatím jsou dispozici dvě vzorové MineralScene a WoodScene
- Intersection - struktura pro jednotlivé druhy průsečíků z kolizí

3.2.4 Rozhraní

- IEffector - definuje efektor, který ovlivňuje pohyb a interakce robota s mapou.
  - k jeho použití slouží funkce Effect(float[] settings, RobotEntity robot, Map.map map). Settings určuje, jakým způsobem ovlivňuje robota a danou mapu. Před 1. použitím efektoru je nutné robota připojit, pomocí funkce ConnectToRobot(RobotEntity robot), která nastaví normalizační funkce(= rozsahy a transformace hodnot přicházející od robota)

- ISensor - definuje sensor, který čte prostředí simulace
  - k jeho použití slouží funkce float[] Count(RobotEntity robot, Map.Map map), která dle pozice robota vrátí informace, přečtené z mapy. Druh informací se liší konkrétními implementacemi. Před první použitím jiného robota je nutné analogicky jako u efektoru robot připojit pomocí funkce ConnectToRobot(RobotEntity robot).

- IRobotBrain - definuje mozek robot, tzn. jeho chování. Slouží k transformaci vektoru přicházejího ze sensorů na vektor vstupující do efektorů. K tomuto účelu slouží fce float[] Decide(float[] readValues). Dále každý mozek lze ohodnotit hodnotou Fitness, dle jeho úspěšnosti v simulaci. Každý mozek má vstupní a výstupní velikost (IoDimension), rozsahy hodnot pro výstup a vstup (InOutBounds), převodní funkci z interní hodnot počítání vstupu na hodnoty výstupní (Activation func), umí vytvořit svou čistou kopii (GetCleanCopy), případně se (de)serializovat (z)do json formátu.

- IExperiment - definuje průběh experimentu, ale je vhodný pro visualizační řešení. Obsahuje mapu na které je simulace prováděná. Každé volání MakeStep() provede nejmenší krok simulaci(jeden pohyb každé entity). Experiment musí být inicializován metodou Init(). Pokud experiment dosáhl svého cíle, FinnishedGeneration je nastaven na true.

3.2.5 Efektory a sensory:

Effectors:

- obsahuje konkrétní implementace efektorů, všechny třídy jsou odděděny od IEffector. Jedná se o efektory určené pro vzorové scénáře. Mohou být rozšířené skrz IEffector.

- MineralRefactor - slouží k přeměně minerálů (RawMaterialEnitity) na palivo. Refaktoruje entitu na vrcholu kontejneru robota.
- Picker - implementovaný jako LineEntity, slouží ke zvedání entit, které se protínají s jeho úsečkou. Dále umí na úsečku pokládat entity z vrcholu zásobníku.
- RadioTransmitter - umí vysílat rádiové různé rádiové signály dle nastavení Effect
- TwoWheelMotor - pohybuje s robotem, dle nastavení rychlostí koleček. Fyzikální model, lze najít, zde `http://rossum.sourceforge.net/papers/DiffSteer/DiffSteer.html`.
- Weapon - dle nastavení může působit poškození robotům, protínající úsečku jeho působnosti.(LineEntity)
- WoodRefactor - slouží k přeměně RawMaterialEnitity, pokud protínají úsečku jeho působnosti(LineEnitity), přímo na mapě. Přeměněná entita tedy nemusí být v kontejneru.

Sensors:

- obsahuje konkrétní implementace sensorů, všechny třídy jsou odděděny od ISensor. Jedná se o sensory pro vzorové scénáře. Mohou být rozšířené skrz ISensor.

- FuelLInSensor - Sensor, který vrací vzdálenost od Fuel, pokud úsečka (LineEntity) nějaké na mapě protíná.
- LineTypeSensor - Sensor, který vrací vzdálenost od libovolné Enity(mimo fuel, rádiové signály) a jeho typ(EntityColor). Pokud nějakou na mapě protíná(LineEntity).
- LocatorSensor - Sensor, který vrací aktuální polohu robota a jeho orientaci vzhledem ke středu robota.
- MemoryStick - Sensor a Efektor v jednom, slouží k zapisování float do paměti. Pokud k němu přistupuji jako k sensoru vrací uložené hodnoty, pokud jako k effektoru, tak ukládá zapisované hodnoty.
- RadioSensor - Sensor, který vrací přečtené signály z okolí a průměr z jejich umístění. Implementován jako CircleEntity.
- TouchSensor - Sensor, který vrací jen binární hodnotu, zda protíná nějakou entitu nebo nikoliv. Implementován jako CircleEntity.

- TypeCircleSensor - Sensor, který vrací binární hodnotu pro každý druh entity(Entity Color), která říká, zda je daná entita v jeho okolí či nikoliv.

3.2.6 Entity

abstract class Entity

Reprezentuje společného předka a implementuje zakladní společné vlastnosti a metody pro všechna entity pasivní i nepasivní. Definuje vlastnost Color určující účel entit v mapě.

- ObstacleColor
- RawMaterialColor
- FuelColor
- RobotColor
- WoodColor

Dále jsou od Entity odděleny základní dva tvary entit abstraktní třídy CircleEntity a LineEntity, které přidávají konkrétní implementace pohybových funkcí a přidávají některé další vlastnosti.

CircleEntity:

Přepisuje metody MoveTo, RotateRadians pro pohybování kruhu. Přidává vhodné konstruktory.

LineEntity:

Přepisuje metody MoveTo, RotateRadians pro pohybování úsečkou. Přidává vhodné konstruktory.

RobotEntity

Potomek třídy CircleEntity, který tvoří základ pro jednotlivé roboty. Uchovává konkrétní instance efektorů a sensorů, zajišťuje komunikaci mezi nimi a mozkem(i převody jednotlivých rozsahů. Přidává další vlastnosti jako životy, množství paliva, číslo týmu, kontejner(možnost přesouvat a uchovávat ostatní instance třídy Entity).
Některé důležitější metody:

- List<CircleEntity> ContainerList() - robot může mít kontejner na CircleEntities, dané kapacity při vytváření robota, tata metody vratí celý jeho obsah
- PrepareMove(Map.Map map) - na dané mapě provede výpočet na všech sensorech a dané hodnoty předá mozku na zpracování, uloží vstup pro efektory z mozku.
- Move(Map.Map map) - spustí všechny efektory na základě vektoru vypočítaného v předchozí metodě.
- Metody spojené s kontejnerem - PushContainer, PopContainer, PeekContainer

Ostatní CircleEntity:

- FuelEntity - pasivní entita, která reprezentuje nádobu s palivem
- ObstacleEntity - pasivní entita, reprezentující překážky
- RadioEntity - pasivní entita, reprezentující rádiový signál s danou informací
- RawMaterialEntity - pasivní entita, reprezentující nezpracovaný materiál (strom, minerál)
- WoodEntity - pasivní entita, reprezentující zpracovaný materiál vytěžené dřevo

Příklady robotů:

- ScoutCuttorRobot - robot, který je určen pro scénář těžení stromů, obstarává kácení
- ScoutCuttorRobotWithMemory - stejný jako předchozí jen má navíc paměťový slot
- WoodWorkerRobot - robot ze scénáře těžení stromů, obstarává přesun pokáceného dřeva
- WoodWorkerRobotMem - stejný jako předchozí jen má navíc paměťový slot

3.2.7 Experiment:

Experimenty jsou určeny pro nastavení vývoje mozků. Jedná se o počet iterací jedné simulace mapy, velikost populací, algoritmus, který vytváří nové generace, ohodnocení fitness pro jednotlivé mozky. Jejich hlavním cílem poskytovat třídu, kterou používá GUI nebo konzole a v ní běží všechny simulace. Tomuto konceptu jsou v programu věnovány dvě třídy Experiment(GUI) a MultiThreadExperiment(výkon).

Experiments:

Základem experimentů je abstraktní třída Experimet<T>, kde T je potomek IRobotBrain typ mozku, který chceme vyvíjet. Obsahuje spoustu proměnných pro nastavení prostředí a evoluce. Viz. komentáře u dané třídy. Třída je uzpůsobena na jednotlivé kroky evoluce, aby po nich mohla přijít na řadu visualizace. Slouží výhradně k testování prostředí a výsledných mozků z experimentů.

Jednotlivé experimenty:

TODO: fix nové experimenty s ES

3.2.8 MultiThread

Základem MT experimentů je bstract class MultiThreadExperiment<T>, kde T je potomek IRobotBrain druh mozku, který vyvíjí. Tato třída obsahuje základní nastavení evoluce. (velikost populace, jméno, počet iterací atd..). Před spuštěním fce Run() je potřeba připravit Mapu a modely mozků, robotů pomocí přetížení abstraktní metody Init(). Funkce Run() - pouští jednotlivé členy aktuální populace každou na jiném vlákně, jejich ohodnocení je implementována pomocí abstraktní metody CountFitness(map). Takto pokračuje napříč všemi generacemi až do poslední. Během běhu serializuje nejlepší mozky, graf(,pokud PC obsahuje GNUplot, tak i vykresluje), všechny mozky(ve zvolených generacích) .
Složky Mineral Scene a WoodScene obsahují vzorové příklady experimentů pro scénáře WoodScene a MineralScene.

3.2.9 RobotBrains

Třídy definující chování robotů. Základním principem je funkce, která přijme vektor float hodnot a z něj vytvoří jiný vektor float. Vstupní hodnoty předává robot ze sensorů a výstupní hodnoty jsou použity pro nastavení efektorů. Projekt obsahuje 3 základní mozky:

- FixedBrain - mozek, který ignoruje vstup a vrací daný výstup
- Perceptron - základní prvek neuronových sítí (vážený součet) lib. vstup a jeden výstup
- SingleLayerNeuronNetwork - neuronová síť tvořená z perceptronů.

Pro SingleLayredNetwork je připravený evoluční algoritmus Differenciální evoluce, definovaná dle `https://en.wikipedia.org/wiki/Differential_evolution`.

3.2.10 Externí knihovny, NuGet

- Intersection2D - implementace jednoduchých průsečíků mezi kruhem, přímkou
- MathNet.Numerics - pokročilé matematické funkce, používané v evolučních algoritmech, normální rozdělení apod.
- Newtonsoft.Json - serializace do jsonu
- System.Numerics - reprezentace Vektorú

3.2.11 Support třídy

- ActivationFuncs - funkce pro převod hodnot ze sensorů do efektorů
- GNUPlot - knihovna pro ovládání programu GNUPLOT
- RandomNumber - statická třída pro volání náhodných tříd
- SupportClasses - ostatní pomocné třídy

3.3 Ostatní projekty:

3.3.1 SwarmSimVisu:

Motivací pro tento projekt je sledování, ladění chyb a v neposlední řadě také pozování vyvinutých mozků a chování dokončených experimentů. Pro debugovací účely umí pouštět jednotlivé potomky třídy Experiment, kde lze sledovat průběh experimentů. Na kontrolu vyvinutých mozků je k dispozici Experiment "Testing Brain", kde mohu připravit experiment simulující průběh mapy, kde se pohybují mnou zvolené entity s nahraným serializovaným mozkem. Vykreslování probíhá přes třídu MapCanvas, kde je použita externí třída D2dControl.D2dControl. Centrální třídou je MainWindow, které spouští jednotlivé Experimenty. Instance třídy InfoWindow slouží pro krátké informativní zprávy. Pro sestavení vlastního "Testing Brain"experimentu jsou k dispozici dvě okna BrainSelectionWindow(nastavení globální parametrů simulace), BrainRobotConnectionWindow(připravení mozků a robotů).

3.3.2 SimpleNetworking:

Jedná se o projekt, který umožňuje spouštět simulaci jednotlivých generačních cyklů na vzdálených počítačích skrze TCP. Oproti normálním konvencím se jedná o jednoho clienta, který pouští na serverch simulaci a přijímá jejich výsledek. K tomuto jsou připraveny dvě třídy ClientTcpCommunicator a ServerTcpCommunicator, kde lze nastavit, zda se má odesílat mozek k ohodnocení či vygenerovat zcela nový. Po navázání spojení s dostupnými servery se odesílá mapa, dále mozek(nebo se generuje), pak běží výpočet na serveru a zpět je odeslána fitness mozku(ů) či celý serializovaný mozek. Pro pomocné třídy a funkce nad TCP slouží statická třída TcpControl.

3.3.3 InterSection2d:

Jedná se jednoduché průsečíky kruhu, přímek, úseček v 2D prostoru. Projekt cílí na rychlost, neboť se jedná o nejvíce volané třídy z celého projektu.

3.4 Prostředí

3.5 Implementace scénářů

3.6 Roboti

3.6.1 Sensory

3.6.2 Efektory

3.6.3 Experimenty

4. Experimenty

4.1 Úvod

Všechny práce zmíněné v úvodní kapitole, sice používají evolučních algoritmů k vytvoření řízení chování robotického swarmu s pouze jedním druhem robotů. Cílem této práce je zmapovat použití jednoduchých evolučních algoritmů i pro heterogenní swarmy. Následující části mají přiblížit obecný postup při hledání optimálního chování hejn, stručně popsat jednotlivé experimenty a poté se věnovat podrobněji každému experimentu zvlášť. Nejvíce se rozepíši o prvním experimentu, protože ten sloužil jako testující pro rozličné postupy a podle něj jsem přistupoval i k ostatním experimentům.

4.2 Použité algoritmy

Vybral jsem dva evoluční algoritmy, které se zdáli při prvotních testech nejvíce perspektivní. Při evoluci homogenních robotů se nejčastěji používají Evoluční Strategie jako jeden z optimalizačních algoritmů, také proto jsem je zvolil jako jeden z využívaných algoritmů. Druhá volba padla na o něco méně agresivní optimalizaci v podobě Diferenciální Evoluce. Oba algoritmy jsou popsány v kapitole o evolučních algoritmech.

Při prvotních zkušebních optimalizací se ukázalo, že optimalizovat rovnou celý cíl jednotlivých scénářů není příliš slibné. Po konzultaci s vedoucím, jsem se rozhodl rozdělit vždy cíl na jednotlivé podúkoly hlavního cíle scénaře. Jednotlivé podúkoly neřeší vždy všichni roboti najednou, ale některé jen homogenní skupina. Nicméně nejpozději v posledním podúkolu už jsou evolvovány společně.

Zkoušel jsem také testovat, roboty s paměťovými sloty oproti těm bez. Úspěnost a konvergence robotů s pamětí se znatelně konvergovali rychleji. Roboty bez slotů se nebyly schopni přiblížit k výsledkům těm s pamětí ani při velkém počtu generací. Z tohoto důvodu všichni roboti mají připojeno alespoň 10 paměťových slotů.

4.2.1 Diferenciální Evoluce

4.2.2 Evoluční strategie

4.3 Experimenty

Pro testování jsem zvolil tři rozličné scénáře. Hlavní motivací bylo jednotlivé úkoly pro hejno udělat udělat komplexnější, aby každá skupina robotů uměla řešit pouze část ze zadání úkoli. Také jsem se snažil, aby se scénáře blížili reálným situacím v dnešním každodenním světě.

Pracovní názvy

1. Wood Scene
2. Mineral Scene

3. Competitive Scece

4.3.1 Wood Scene

Tento scénář je analogií pro kácení lesa, kdy se roboti snaží maximalizovat množství zpracované dřevo na předem vyznačené ploše. První robot plní úkol objevovávání a kácení stromu, ale neumí je převážet, v mém frameworku se nazývá Wood Scout. Oproti tomu druhý robot má vlastní kontajner na objekty, také je umí zvedat a následně pokládat. Ve frameworku pojmenovaný Wood Worker. Ovšem neumí stromy zpracovávat. Jedná se tedy o úkol typu najdi označ a převez.

4.3.2 Mineral Scene

Jedná se o scénář reprezentující sběr surovin pro výrobu paliva a jeho následné využití. Figurují zde 3 rozliční roboti, všichni potřebují pro pohyb dané množství paliva. Úspěšnost daného hejna se měří množstvím paliva. Nejmenší robot(Mineral scout) disponuje pouze sensory k exploraci prostředí a rádiovým vysílačem pro komunikaci se skupinou. Robot prostřední velikosti(Mineral Worker) se pohybuje o něco pomaleji než Mineral Scout, ale umí přesouvat objekty i více najednou. Robot pro přeměnu minerálu(‚suroviny na výrobu paliva,) označen ve frameworku jako Mineral Refactor se přemisťuje nejpomaleji, má možnost přeměnit minerál na palivo. Tento scénář si bere jako inspiraci strategické hry a hypotetické přežití robotů na cizí planetě, kde si budou muset obstarat vlastní nerostné suroviny pro běh.

4.3.3 Competitive Scene

Poslední ze scénářů se týká soutěže dvou týmů(hejn) ve kterých figurují jeden malý průzkumný robot(Competitive Scout) a jeden vetší bojový robot(Competitive Fighter). Úspěšnost týmu je dána zachovanými jednotkami zdraví robotů a uděleným poškozením do nepřátelské skupiny robotů. Competitive Scout se pohybuje značně rychleji než Competitive Fighter, ale uděluje menší poškození. Což lze opět vztáhnout na chování rozdílných skupin nepřátel např. ve strategických hrách, kde se jejich chování adaptuje, co nejlépe na dané prostředí.

4.4 WoodScene

Závěr

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

Přílohy