



Kierunek: Automatyka i Robotyka

Specjalność: Informatyka Przemysłowa

Data urodzenia: 21.03.1993 r.

Data rozpoczęcia studiów: 23.02.2016 r.

Życiorys

Urodziłem się 21 marca 1993 roku w Lublinie. W 2012 roku rozpoczęłem studia inżynierskie I stopnia na Wydziale Mechatroniki Politechniki Warszawskiej, na kierunku Automatyka i Robotyka, na specjalności Informatyka Przemysłowa. Ukończył je 17 lutego 2016 roku. Następnie, tego samego roku rozpoczęłem studia II stopnia na tym samym wydziale, na specjalności Informatyka Przemysłowa.



POLITECHNIKA WARSZAWSKA

Wydział Mechatroniki

Praca magisterska

Ireneusz Szulc

Planowanie bezkolizyjnych tras dla zespołu robotów mobilnych

Promotor:
prof. nzw. dr hab. Barbara Siemiątkowska

Warszawa, 2018

PRACA DYPLOMOWA magisterska

Specjalność: Informatyka Przemysłowa

Instytut prowadzący specjalność: Instytut Automatyki i Robotyki

Instytut prowadzący pracę: Instytut Automatyki i Robotyki

Temat pracy: Planowanie bezkolizyjnych tras dla zespołu robotów mobilnych

Temat pracy (w jęz. ang.): Path planning for a group of mobile robots

Zakres pracy:

1. Projekt algorytmu wyznaczania trajektorii dla pojedynczego robota
2. Algorytm detekcji i zapobiegania kolizjom między robotami
3. Implementacja oprogramowania symulacyjnego
4. Przeprowadzenie testów symulacyjnych

Podstawowe wymagania:

1. Aplikacja powinna umożliwiać symulację ruchu robotów oraz definiowanie położenia przeszkód przez użytkownika.
2. Planowanie tras dotyczy robotów holonomicznych.

Literatura:

1. Mówiński K., Roszkowska E.: Sterowanie hybrydowe ruchem robotów mobilnych w systemach wielorobotycznych, Postępy Robotyki, 2016,
2. Siemiątkowska B.: Uniwersalna metoda modelowania zachowań robota mobilnego wykorzystująca architekturę uogólnionych sieci komórkowych, Warszawa 2009,
3. Silver D.: Cooperative Pathfinding, 2005

Słowa kluczowe: planowanie tras, systemy wielorobotowe

Praca dyplomowa jest realizowana we współpracy z przemysłem:

Nie

| | |
|---|---|
| <i>Ireneusz Szulc</i> Imię i nazwisko dyplomanta: | Imię i nazwisko promotora: <i>Barbara Siemiątkowska</i> |
| | Imię i nazwisko konsultanta: |
| Temat wydano dnia: | Termin ukończenia pracy: |

Zatwierdzenie tematu

| | |
|--|--|
| <i>B. Smi</i> Opiekun specjalności | <i>K. Siemiątkowska</i> Z-ca Dyrektora Instytutu |
|--|--|

Streszczenie

Planowanie bezkolizyjnych tras dla zespołu robotów mobilnych

TODO wyciągnąć z podsumowania i z intro

Zakres pracy:

- Projekt algorytmu wyznaczania trajektorii dla pojedynczego robota
- Algorytm detekcji i zapobiegania kolizjom między robotami
- Implementacja oprogramowania symulacyjnego
- Przeprowadzenie testów symulacyjnych

Podstawowe wymagania:

- Aplikacja powinna umożliwiać symulację ruchu robotów oraz definiowanie położenia przeszkód przez użytkownika.
- Planowanie tras dotyczy robotów holonomicznych.

Słowa kluczowe: planowanie tras, systemy wielorobotowe

Abstract

Path planning for a group of mobile robots

TODO tłumaczenie

Key words: cooperative path-planning, multi-agent systems



„załącznik nr 3 do zarządzenia nr 24/2016 Rektora PW

.....
miejscowość i data

.....
imię i nazwisko studenta

.....
numer albumu

.....
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
czytelny podpis studenta”

Spis treści

Spis treści

| | |
|--|-----------|
| 1 Wprowadzenie | 15 |
| 1.1 Cel i zakres pracy | 15 |
| 1.2 Założenia | 16 |
| 1.3 Zastosowanie koordynacji ruchu robotów | 16 |
| 1.4 Podobieństwo do gier RTS | 17 |
| 2 Wstęp teoretyczny | 19 |
| 2.1 Podstawowe pojęcia | 19 |
| 2.2 Kooperacyjne planowanie tras | 19 |
| 2.3 Metoda pól potencjałowych | 20 |
| 2.4 Rozproszone planowanie tras | 20 |
| 2.5 Planowanie uwzględniające priorytety | 22 |
| 2.6 Metoda Path Coordination | 22 |
| 3 Algorytmy oparte o A* | 25 |
| 3.1 Algorytm A* | 26 |
| 3.1.1 Zasada działania | 26 |
| 3.1.2 Funkcja heurystyczna | 26 |
| 3.2 Metody ponownego planowania | 28 |
| 3.2.1 Local Repair A* | 28 |
| 3.2.2 Algorytm D* | 29 |
| 3.2.3 D* Extra Lite | 29 |
| 3.3 Cooperative A* | 29 |
| 3.3.1 Trzeci wymiar - czas | 30 |
| 3.3.2 Tablica rezerwacji | 30 |
| 3.4 Hierarchical Cooperative A* | 32 |

| | | |
|----------|---|-----------|
| 3.5 | Windowed Hierarchical Cooperative A* | 32 |
| 3.6 | Podsumowanie | 34 |
| 4 | Opracowanie i implementacja algorytmów | 35 |
| 4.1 | Generator map | 35 |
| 4.2 | Wyznaczanie trajektorii dla pojedynczego robota | 39 |
| 4.3 | Detekcja i przeciwdziałanie kolizjom | 41 |
| 4.4 | Implementacja algorytmu WHCA* | 43 |
| 4.5 | Dynamiczny przydział priorytetów | 44 |
| 4.6 | Metoda pól potencjałowych | 44 |
| 5 | Oprogramowanie symulacyjne | 45 |
| 5.1 | Funkcjonalności aplikacji | 45 |
| 5.2 | Graficzny interfejs użytkownika | 46 |
| 5.2.1 | Metoda pól potencjałowych | 46 |
| 5.2.2 | Local-Repair A* | 47 |
| 5.2.3 | Windowed Hierarchical Cooperative A* | 48 |
| 5.3 | Wykorzystane technologie | 50 |
| 5.3.1 | Java 8 | 50 |
| 5.3.2 | JavaFX | 50 |
| 5.3.3 | Spring | 51 |
| 5.3.4 | jUnit i Test-driven development | 51 |
| 5.3.5 | Maven | 52 |
| 5.3.6 | IntelliJ IDEA | 52 |
| 5.3.7 | Pozostałe narzędzia i biblioteki | 52 |
| 5.4 | Struktura aplikacji | 53 |
| 5.4.1 | Wzorzec Model-View-Presenter | 53 |
| 5.4.2 | Wielowątkowość | 54 |
| 5.5 | Ograniczenia | 55 |
| 6 | Wyniki testów | 57 |
| 6.1 | Obszerne testy aplikacji | 57 |
| 6.2 | Screeny | 57 |
| 6.3 | Środowiska | 57 |
| 6.4 | Porównanie wyników | 61 |

| | |
|--------------------------------|-----------|
| 7 Podsumowanie | 63 |
| 7.1 Dyskusja wyników | 64 |
| Bibliografia | 65 |
| Wykaz skrótów | 69 |
| Spis rysunków | 71 |
| Spis tabel | 73 |
| Spis załączników | 75 |

Rozdział 1

Wprowadzenie

1.1 Cel i zakres pracy

Przedmiotem niniejszej pracy jest przegląd metod wykorzystywanych do planowania bezkolizyjnych tras dla wielu robotów mobilnych. Stanowi to również wstęp teoretyczny do zaprojektowania algorytmu i implementacji oprogramowania pozwalającego na symulację działania skutecznego planowania tras dla systemu wielorobotowego.

Praca skupia się na przypadkach, w których mamy do czynienia ze środowiskiem z dużą liczbą przeszkód (np. zamknięty budynek z licznymi ciasnymi korytarzami), aby uwypuklić problem blokowania się agentów często prowadzący do zakleszczenia. Często okazuje się, że należy wtedy zastosować nieco inne podejścia niż te, które sprawdzają się w przypadku otwartych środowisk, a które zostały opisane np. w pracach [8], [10]. W otwartych środowiskach z małą liczbą przeszkód wystarczające może się okazać np. proste ponowne planowanie wykorzystujące algorytm LRA* (por. 3.2.1) lub D* (por. 3.2.2).

W niniejszej pracy starano się znaleźć metody rozwiązuje zagadnienie, w którym znane są:

- pełna informacja o mapie otoczenia (położenie statycznych przeszkód),
- aktualne położenie i położenie celu każdego z robotów.

Szukany jest natomiast przebieg tras do punktów docelowych dla agentów. Zadaniem algorytmu będzie wyznaczenie możliwie najkrótszej bezkolizyjnej trasy dla wszystkich robotów. Należy jednak zaznaczyć, że priorytetem jest dotarcie każdego z robotów do celu bez kolizji z innymi robotami. Drugorzędne zaś jest, aby wyznaczone drogi były możliwie jak najkrótsze.

1.2 Założenia

Założenia i ograniczenia rozważanego problemu:

1. Każdy z robotów ma wyznaczony inny punkt docelowy, do którego zmierza.
2. Planowanie tras dotyczy mobilnych robotów holonomicznych.
3. Czas trwania zmiany kierunku robota jest pomijalnie mały.
4. Środowisko, w którym poruszają się roboty, jest dwuwymiarową przestrzenią zawierającą dużą liczbę przeszkód oraz wąskie korytarze (por. rys. 1.1).
5. Roboty "wiedzą" o sobie i mogą komunikować się ze sobą podczas planowania tras.
6. Każdy robot zajmuje w przestrzeni jedno pole. Na jednym polu może znajdować się maksymalnie jeden robot (por. rys. 1.1).
7. Planowanie tras powinno odbywać się w czasie rzeczywistym.



Rysunek 1.1: Przykładowe środowisko z dużą liczbą przeszkód (czarne kwadraty) i rozmieszczonymi robotami (kolorowe koła). Źródło: własna implementacja oprogramowania symulacyjnego

1.3 Zastosowanie koordynacji ruchu robotów

Koordynacja ruchu robotów jest jednym z fundamentalnych problemów w systemach wielorobotowych [1].

Kooperacyjne znajdowanie tras (ang. *Cooperative Pathfinding*) jest zagadnieniem planowania w układzie wieloagentowym, w którym to agenci mają za zadanie znaleźć bezkolizyjne drogi do swoich, osobnych celów. Planowanie to odbywa się w oparciu o pełną informację o środowisku oraz o trasach pozostałych agentów [11].

Algorytmy do wyznaczania bezkolizyjnych tras dla wielu agentów (robotów) mogą znaleźć zastosowanie w szpitalach (np. roboty TUG i HOMER do dostarczania sprzętu na wyposażeniu szpitala [23]) oraz magazynach (np. roboty transportowe w magazynach firmy Amazon - por. rys. 1.2).



Rysunek 1.2: Roboty Kiva pracujące w magazynie firmy Amazon. Źródło: [16]

1.4 Podobieństwo do gier RTS

Problem kooperacyjnego znajdowania tras pojawia się nie tylko w robotyce, ale jest również popularny m.in. w grach komputerowych (strategiach czasu rzeczywistego), gdzie konieczne jest wyznaczanie przez sztuczną inteligencję bezkolizyjnych dróg dla wielu jednostek, unikając wzajemnego blokowania się. Niestety brak wydajnych i skutecznych algorytmów planowania dróg można zauważać w wielu grach typu RTS (ang. *Real-Time Strategy*), gdzie czasami obserwuje się zjawisko zakleszczenia jednostek w wąskich gardłach (np. w grach *Age of Empires II*, *Warcraft III* lub nawet we współczesnych produkcjach) [4] (por. rys. 1.3). Ponadto, zauważalny brak ogólnie dostępnych bibliotek open-source do rozwiązania problemu typu *Cooperative Pathfinding* świadczy o potrzebie rozwoju tych metod.

Często algorytmy wykorzystywane w grach typu RTS (ang. *Real-Time Strategy*) zajmują się planowaniem bezkolizyjnych dróg dla układu wielu agentów w czasie rzeczywistym (będącego

przedmiotem niniejszej pracy), dlatego nic nie stoi na przeszkodzie, aby stosować je zamiennie również do koordynacji ruchu zespołu robotów mobilnych.



Rysunek 1.3: Popularny problem zakleszczania się jednostek w wąskich gardłach występujący w grach typu RTS. Źródło: gra komputerowa *Age of Empires II Forgotten Empires*

Rozdział 2

Wstęp teoretyczny

2.1 Podstawowe pojęcia

Definicja 2.1.1. Robot holonomiczny

Robot holonomiczny to taki robot mobilny, który może zmienić swoją orientację, stojąc w miejscu.

Definicja 2.1.2. Przestrzeń konfiguracyjna

Przestrzeń konfiguracyjna to N -wymiarowa przestrzeń będąca zbiorem możliwych stanów danego układu fizycznego. Wymiar przestrzeni zależy od rodzaju i liczby wyróżnionych parametrów stanu. W odróżnieniu od przestrzeni roboczej, gdzie robot ma postać bryły, w przestrzeni konfiguracyjnej robot jest reprezentowany jako punkt.

Definicja 2.1.3. Zupełność algorytmu (ang. *Completeness*)

W kontekście algorytmu przeszukiwania grafu algorytm zupełny to taki, który gwarantuje znalezienie rozwiązania, jeśli takie istnieje. Warto zaznaczyć, że nie gwarantuje to wcale, że znalezione rozwiązanie będzie rozwiązaniem optymalnym.

2.2 Kooperacyjne planowanie tras

Spośród metod wykorzystywanych do kooperacyjnego planowania tras dla wielu robotów można wyróżnić dwie zasadnicze grupy [7]:

- **Zcentralizowane** - drogi wyznaczane są dla wszystkich agentów na raz (jednocześnie). Metody tego typu są często trudne do zrealizowania (gdyż do rozwiązania jest złożony problem optymalizacyjny) oraz mają bardzo dużą złożoność obliczeniową ze względu

na ogólną przestrzeń przeszukiwania. Struktura organizacyjna jest skoncentrowana - decyzje podejmowane są na podstawie centralnego systemu.

- **Rozproszone** (ang. *decoupled* lub *distributed*) - podejście to dekomponuje zadanie na niezależne lub zależne w niewielkim stopniu problemy dla każdego agenta. Dla każdego robota droga wyznaczana jest osobno, w określonej kolejności, następnie rozwiązywane są konflikty (kolizje dróg). Zastosowanie metod rozproszonych wiąże się najczęściej z koniecznością przydzielania priorytetów robotom, co stanowi istotny problem, gdyż od ich wyboru może zależeć zupełność algorytmu. Nie należy mylić tej metody z zagadnieniem typu *Non-Cooperative Pathfinding*, w którym agenci nie mają wiedzy na temat pozostałych planów i muszą przewidywać przyszłe ruchy pozostałych robotów [11]. W podejściu rozproszonym agenci mają pełną информацию na temat stanu pozostałych robotów, lecz wyznaczanie dróg odbywa się w określonej kolejności.

W systemach czasu rzeczywistego istotne jest, aby rozwiązanie problemu planowania tras uzyskać w krótkim, deterministycznym czasie, dlatego w tego typu systemach częściej używane są techniki rozproszone.

2.3 Metoda pól potencjałowych

Metoda pól potencjałowych (ang. *Artificial Potential Field* lub *Potential Field Technique*) polega na zastosowaniu zasad oddziaływanego między ładunkami znanych z elektrostatyki. Roboty i przeszkody traktowane są jako ładunki jednoimienne, przez co "odpychają się" siłą odwrotnie proporcjonalną do kwadratu odległości (dzięki temu unikają kolizji między sobą). Natomiast punkt docelowy robota jest odwzorowany jako ładunek o przeciwnym biegunie, przez co robot jest "przyciągany" do celu. Główną zasadę działania metody przedstawiono na rysunku 2.1.

Technika ta jest bardzo prosta i nie wymaga wykonywania złożonych obliczeń (w odróżnieniu do pozostałych metod zcentralizowanych). Niestety bardzo powszechny jest problem osiągania minimum lokalnego, w którym suma wektorów daje zerową siłę wypadkową. Robot zostaje "uwięziony" w takim minimum lokalnym, przez co nie jest w stanie dotrzeć do wyznaczonego celu. Do omijania tego problemu muszą być stosowane inne dodatkowe metody [14]. Metoda pól potencjałowych nie daje gwarancji ani optymalności, ani nawet zupełności.

2.4 Rozproszone planowanie tras

Najczęściej stosowanymi podejściami są metody oparte o algorytm A* lub jego pochodne. W celu wykonywania wydajnych obliczeń w algorytmach przeszukujących grafy, nawet w przypadku



Rysunek 2.1: Zasada działania metod pól potencjałowych. Dodatni ładunek q_{start} reprezentuje robota. Przyciągany jest w stronę ujemnego ładunku celu q_{goal} , zaś odpychany jest od dodatnio naładowanej przeszkody. Źródło: [22]

ciągłej przestrzeni mapy, stosuje się podział na dyskretną siatkę pól (por. rys. 2.2) [3].



Rysunek 2.2: Ciągła przestrzeń mapy zdyskretyzowana do siatki pól.
Źródło: edytor map z gry Warcraft III.

Popularne podejścia unikające planowania w wysoko wymiarowej zbiorowej przestrzeni konfiguracyjnej to techniki rozproszone i uwzględniające priorytety [1]. Pomimo, że metody te są bardzo efektywne, mają dwie główne wady:

- Nie są zupełne - nie dają gwarancji znalezienia rozwiązania, nawet gdy takie istnieje.
- Wynikowe rozwiązania mogą być nieoptymalne.



Rysunek 2.3: Sytuacja, w której żadne rozwiązanie nie zostanie znalezione, stosując planowanie uwzględniające priorytety, jeśli robot 1 ma wyższy priorytet niż robot 2. Źródło: [1]

2.5 Planowanie uwzględniające priorytety

Często używaną w praktyce metodą jest planowanie z uwzględnianiem priorytetów. W tej technice agenci otrzymują unikalne priorytety. Algorytm wykonuje indywidualne planowanie sekwencyjnie dla każdego agenta w kolejności od najwyższego priorytetu. Trajektorie agentów o wyższych priorytetach są ograniczeniami (ruchomymi przeszkodami) dla pozostałych agentów [2].

Złożoność ogólnego algorytmu rośnie liniowo wraz z liczbą agentów, dzięki temu to podejście ma zastosowanie w problemach z dużą liczbą agentów. Algorytm ten jest zachłanny i niezupełny w takim znaczeniu, że agentów zadowala pierwsza znaleziona trajektoria niekolidująca z agentami wyższych priorytetów.

Istotną rolę doboru priorytetów robotów w procesie planowania tras ukazuje prosty przykład przedstawiony na rysunku 2.3. Jeśli robot 1 (zmierzający z punktu S1 do G1) otrzyma wyższy priorytet niż robot 2 (zmierzający z S2 do G2), spowoduje to zablokowanie przejazdu dla robota 2 i w efekcie prawidłowe, istniejące rozwiązanie nie zostanie znalezione.

Układ priorytetów może mieć także wpływ na długość uzyskanych tras. Potwierdzający to przykład został przedstawiony na rysunku 2.4. W zależności od wyboru priorytetów, wpływających na kolejność planowania tras, otrzymujemy różne rozwiązania.

2.6 Metoda Path Coordination

Jedną z metod rozproszonego planowania tras z uwzględnianiem priorytetów jest *Path Coordination*, której idea przedstawia się w następujących krokach [1]:

1. Wyznaczenie ścieżki dla każdego robota **niezależnie** (np. za pomocą algorytmu A*)
2. Przydział priorytetów
3. Próba rozwiązywania możliwych konfliktów między ścieżkami. Roboty utrzymywane są na ich indywidualnych ścieżkach (wyznaczonych na początku), wprowadzane modyfikacje



Rysunek 2.4: a) Niezależne planowanie optymalnych tras dla 2 robotów; b) suboptimalne rozwiązanie, gdy robot 1 ma wyższy priorytet; c) rozwiązanie, gdy robot 2 ma wyższy priorytet.
 Źródło: [1]

pozwalały na zatrzymanie się, ruch naprzód, a nawet cofanie się, ale tylko **wzdłuż trajektorii** w celu uniknięcia kolizji z robotem o wyższym priorytecie.

Rozdział 3

Algorytmy oparte o A*

Kiedy pojedynczy agent staje przed zadaniem znalezienia drogi do wyznaczonego celu, prosty algorytm A* sprawdza się znakomicie. Jednak w przypadku, gdy wiele agentów porusza się w tym samym czasie, podejście to może się już nie sprawdzić, powodując wzajemne blokowanie się i zakleszczenie jednostek. Rozwiązaniem tego problemu jest kooperacyjne znajdowanie tras. Dzięki tej technice roboty będą mogły skutecznie przemieszczać się przez mapę, omijając trasy wyznaczone przez inne jednostki oraz schodząc innym jednostkom z drogi, jeśli to konieczne [11].

Zagadnienie znajdowania drogi jest również ważnym elementem sztucznej inteligencji zaimplementowanej w wielu grach komputerowych. Chociaż klasyczny algorytm A* potrafi doprowadzić pojedynczego agenta do celu, to jednak dla wielu agentów wymagane jest zastosowanie innego podejścia w celu unikania kolizji. Istniejące rozwiązania są jednak wciąż oparte o Algorytm A*, choć nieco zmodyfikowany. Chociaż A* może zostać zaadaptowany do ponownego planowania trasy na żądanie, w przypadku wykrycia kolizji tras, to jednak takie podejście nie jest zadowalające pod wieloma względami. Na trudnych mapach z wieloma wąskimi korytarzami i wieloma agentami może to prowadzić do zakleszczenia agentów w wąskich gardłach lub do cyklicznego zapętlenia akcji agentów [11].

Rozdział ten zmierza do zaprezentowania zasady działania oraz cech algorytmu WHCA* (por. 3.5), zaproponowanego przez Davida Silvera [11]. Zaczynamy jednak od przedstawienia samego algorytmu A* i wprowadzając kolejne modyfikacje (CA*, HCA*), przekształcimy go stopniowo w WHCA*. W rozdziale zaprezentowano także rozwiązania alternatywne takie, jak: D* (por. 3.2.2) i LRA* (por. 3.2.1).

3.1 Algorytm A*

3.1.1 Zasada działania

A* jest algorytmem heurystycznym służącym do przeszukiwania grafu w celu znalezienia najkrótszej ścieżki między węzłem początkowym a węzłem docelowym. Algorytm ten jest powszechnie stosowany w zagadnieniach sztucznej inteligencji oraz w grach komputerowych [24]. Opiera się na zapisywaniu węzłów w dwóch listach: zamkniętych (odwiedzonych) i otwartych (do odwiedzenia). Jest modyfikacją algorytmu Dijkstry poprzez wprowadzenie pojęcia funkcji heurystycznej $h(n)$. Wartość funkcji heurystycznej powinna określać przewidywaną drogę do węzła docelowego z bieżącego punktu. Pełny koszt $f(n)$ stanowi sumę dotychczasowego kosztu $g(n)$ oraz przewidywanego pozostałoego kosztu.

$$f(n) = g(n) + h(n) \quad (3.1)$$

gdzie:

$g(n)$ - dotychczasowy koszt dotarcia do węzła n , dokładna odległość między węzłem n a węzłem początkowym

$h(n)$ - heurystyka, przewidywana pozostała droga od węzła bieżącego do węzła docelowego

$f(n)$ - oszacowanie pełnego kosztu ścieżki od węzła startowego do węzła docelowego prowadzącej przez węzeł n

n - bieżący węzeł, wierzchołek przeszukiwanego grafu

W każdym kroku przeszukiwany jest węzeł o najmniejszej wartości funkcji $f(n)$. Dzięki takiemu podejściu najpierw sprawdzane są najbardziej "obiecujące" rozwiązania, co pozwala szybciej otrzymać wynik (w porównaniu do algorytmu Dijkstry). Algorytm kończy działanie w momencie, gdy napotka węzeł będący węzłem docelowym. Dla każdego odwiedzonego węzła zapamiętywane są wartości $g(n)$, $h(n)$ oraz węzeł będący rodzicem w celu późniejszego odnalezienia drogi powrotnej do węzła startowego po napotkaniu węzła docelowego (por. rys. 3.1).

Algorytm zwraca optymalny wynik (najkrótszą możliwą ścieżkę), ale w pewnych warunkach (por. 3.1.2).

3.1.2 Funkcja heurystyczna

Od wyboru sposobu obliczania heurystyki zależy czas wykonywania algorytmu oraz optymalność wyznaczonego rozwiązania.



Rysunek 3.1: Ilustracja wyznaczania działania przez A*. Każdy odwiedzony węzeł wskazuje na swojego rodzica, co umożliwia późniejszą rekonstrukcję drogi. Źródło: [15]

Funkcja heurystyczna $h(n)$ jest dopuszczalna, jeśli dla dowolnego węzła n spełniony jest warunek

$$h(n) \leq h^*(n) \quad (3.2)$$

gdzie:

$h^*(n)$ - rzeczywisty koszt ścieżki od węzła n do celu

Innymi słowy, heurystyka dopuszczalna to taka, która nigdy nie przeszacowuje pozostałe do przebycia drogi. Wtedy algorytm A* zwraca optymalną (najkrótszą) ścieżkę [11].

A* opiera się na heurystyce, która ”kieruje” przeszukiwaniem. To od niej zależy, w kierunku których węzłów będzie podążało przeszukiwanie. Źle wybrana heurystyka może prowadzić do zbędnego odwiedzania dodatkowych węzłów.

Poniżej przedstawiono najczęściej wykorzystywane heurystyki będące oszacowaniem odległości między przeszukiwanym węzłem (x_n, y_n) a węzłem docelowym (x_g, y_g) na dwuwymiarowej mapie.

Heurystyka euklidesowa

Heurystyka wykorzystująca metrykę euklidesową wyraża dokładną odległość po linii prostej. Wymaga to jednak częstego obliczania pierwiastków (lub stosowania *look-up table*).

$$h(n) = \sqrt{(x_n - x_g)^2 + (y_n - y_g)^2} \quad (3.3)$$

Heurystyka Manhattan

W przypadku, gdy agent może poruszać się po mapie jedynie poziomo lub pionowo (nie na ukos) wystarczająca okazuje się metryka Manhattan (metryka miejska):

$$h(n) = |x_n - x_g| + |y_n - y_g| \quad (3.4)$$

Heurystyka metryki maksimum

Zastosowanie metryki maksimum (metryki Czebyszewa) może sprawdzić się np. dla niektórych figur szachowych:

$$h(n) = \max(|x_n - x_g|, |y_n - y_g|) \quad (3.5)$$

Heurystyka zerowa

Przyjęcie heurystyki równej $h(n) = 0$ sprawia, że algorytm A* sprowadza się do algorytmu Dijkstry.

3.2 Metody ponownego planowania

3.2.1 Local Repair A*

W algorytmie Local Repair A* (LRA*) każdy z agentów znajduje drogę do celu, używając algorytmu A*, ignorując pozostałe roboty oprócz ich obecnych sąsiadów. Roboty zaczynają podążać wyznaczonymi ścieżkami do momentu, aż kolizja z innym robotem jest nieuchronna (w lokalnym otoczeniu). Wtedy następuje ponowne przeliczenie drogi pozostałej do przebycia, z uwzględnieniem nowo napotkanej przeszkody.

Możliwe (i całkiem powszechnie [11]) jest uzyskanie cykli (tych samych sekwencji ruchów powtarzających się w nieskończoność), dlatego zazwyczaj wprowadzane są pewne modyfikacje, aby rozwiązać ten problem. Jedną z możliwości jest zwiększenie wpływu losowego szumu na wartość heurystyki. Kiedy agenci zachowują się bardziej losowo, prawdopodobne jest, że wydostaną się z problematycznego położenia i spróbują podążać innymi ścieżkami.

Algorytm ten ma jednak sporo poważnych wad, które szczególnie ujawniają się w trudnych środowiskach z dużą liczbą przeszkód. Wydostanie się z zatłoczonego wąskiego gardła może trwać bardzo długo. Prowadzi to także do ponownego przeliczania trasy w prawie każdym kroku. Wciąż możliwe jest również odwiedzanie tych samych lokalizacji w wyniku zapętleń.

3.2.2 Algorytm D*

D* (*Dynamic A* Search*) jest przyrostowym algorytmem przeszukiwania. Jest modyfikacją algorytmu A* pozwalającą na szybsze ponowne planowanie trasy w wyniku zmiany otoczenia (np. zajmowania wolnego pola przez innego robota). Wykorzystywany jest m.in. w nawigacji robota do określonego celu w nieznanym terenie. Początkowo robot planuje drogę na podstawie pewnych założeń (np. nieznany teren nie zawiera przeszkód). Podążając wyznaczoną ścieżką, robot odkrywa rzeczywisty stan mapy i jeśli to konieczne, wykonuje ponowne planowanie trasy na podstawie nowych informacji. Często wykorzystywana implementacja (z uwagi na zoptymalizowaną złożoność obliczeniową) jest wariant algorytmu *D* Lite* [6].

3.2.3 D* Extra Lite

Wartym uwagi jest także algorytm *D* Extra Lite* charakteryzujący się jeszcze korzystniejszą wydajnością niż *D* Lite*, co potwierdziły przeprowadzone obszerne testy w różnego rodzaju środowiskach (m.in. na mapach z gier komputerowych oraz w zawiłych labiryntach) [9].

D Extra Lite* służy do przyrostowego planowania najkrótszej ścieżki w dwuwymiarowej przestrzeni bez dokładnej wiedzy o środowisku. Wykorzystanie wyników z poprzednich iteracji oraz wczesne odrzucanie pewnych węzłów z drzewa przeszukiwania znacznie skracza czas potrzebny do wykonania ponownego planowania. *D* Extra Lite* jest nowatorskim algorytmem ogólnego przeznaczenia. Naturalnym jego zastosowaniem jest nawigacja robotów mobilnych [9].

3.3 Cooperative A*

Cooperative A* jest algorytmem do rozwiązywania problemu kooperacyjnego znajdowania tras. Metoda może być również nazywana czasoprzestrzennym algorytmem A* (*time-space A* search*). Zadanie planowania jest rozdzielone na serię pojedynczych poszukiwań dla poszczególnych agentów. Pojedyncze poszukiwania są wykonywane w trójwymiarowej czasoprzestrzeni i biorą pod uwagę zaplanowane ścieżki przez pozostałych agentów. Akcja wykonania postoju (pozostania w tym samym miejscu) jest uwzględniona w zbiorze akcji możliwych do wykonania. Po przeliczeniu dróg dla każdego agenta, stany zajętości pól są zaznaczane w tablicy rezerwacji (ang. Reservation table). Pozycje w tej tablicy są uważane jako pola nieprzejezdne i w efekcie są omijane podczas przeszukiwania przez późniejszych agentów [11].

Należy zaznaczyć, że planowanie dla każdego agenta odbywa się sekwencyjnie według przydzielonych priorytetów. Algorytm ten jest podatny na zmianę kolejności agentów. Odpowiedni dobór priorytetów może wpłynąć na wydajność algorytmu oraz jakość uzyskanego wyniku.

3.3.1 Trzeci wymiar - czas

Do rozwiązania problemu kooperacyjnego znajdowania dróg algorytm przeszukiwania potrzebuje mieć pełną wiedzę na temat przeskódeł oraz jednostek na mapie. Aby zapisać tą informację, potrzeba rozszerzyć mapę o trzeci wymiar - czas. Pierwotną mapę będziemy nazywać mapą przestrzenną, natomiast nową - czasoprzestrzenną mapą [11].

Zagadnienie sprowadza się do przeszukiwania grafu, w którym każdy węzeł ma przypisane 3 wielkości: położenie x, położenie y oraz czas. Podczas gdy zakres wielkości x i y jest znany i wynika z rozmiarów mapy oraz podziału jej wymiarów na dyskretne pola, to jednak określenie wymiaru czasu i jego granicy nie jest trywialnym zagadnieniem. Wymiar czasu możemy również zdyskretyzować i przyjąć, że krok czasu jest okresem, jaki zajmuje robotowi przejście z jednego pola na sąsiednie (poziomo lub pionowo). Natomiast górną granicą czasu powinna być maksymalna liczba ruchów potrzebna do dotarcia do celu przez ostatniego robota. Wybór za małej liczby może spowodować, że algorytm nie zdąży znaleźć drogi dla niektórych agentów, z kolei zbyt duża granica kroków czasu znaczco wydłuża obliczenia. Rozwiązanie tego problemu zostało opisane w późniejszym podrozdziale 3.5.

Wprowadzenie trzeciego wymiaru wprowadza także konieczność zmian w doborze odpowiedniej heurystyki odpowiedzialnej za oszacowanie drogi pozostałe do celu.

3.3.2 Tablica rezerwacji

Tablica rezerwacji (ang. *Reservation Table*) reprezentuje współdzieloną wiedzę o zaplanowanych ścieżkach przez wszystkich agentów. Jest to informacja o zajętości każdej z komórki na mapie w danym miejscu i określonym czasie [11]. Jak tylko agent zaplanuje trasę, każda komórka odpowiadająca ścieżce zaznaczana jest jako zajęta w tablicy rezerwacji.

W prostej implementacji tablica rezerwacji jest trójwymiarową kostką (dwa wymiary przestrzenne i jeden wymiar czasu). Każda komórka kostki, która jest przecinana przez zaplanowaną przez agenta ścieżkę, jest zaznaczana jako nieprzejezdna przez określony czas trwania. W ten sposób zapobiega to planowania kolizyjnych tras przez pozostałych agentów. (por. rys. 3.2)



Rysunek 3.2: Dwie jednostki kooperacyjnie poszukujące tras. (A) Pierwsza jednostka znajduje ścieżkę i zaznacza ją w tablicy rezerwacji. (B) Druga jednostka znajduje ścieżkę, uwzględniając istniejące rezerwacje pól, również zaznaczając ją w tablicy rezerwacji. Źródło: [11]

Jeśli tylko niewielka część z całej tablicy rezerwacji będzie markowana jako zajęta, wydajniej jest zaimplementować ją jako tablicę typu *hash table*. Daje to zaletę oszczędności pamięci poprzez pamiętanie jedynie współrzędnych (x, y, t) zajętych pól.

W ogólności poszczególni agenci mogą mieć różną prędkość lub rozmiary, zatem tablica rezerwacji musi mieć możliwość zaznaczenia dowolnego zajętego obszaru. Zostało to przedstawione na rysunku 3.3.



Rysunek 3.3: Tablica rezerwacji jest współdzielona między wszystkimi agentami. Jej rozmiar powinien być odpowiednio dopasowany do agentów o różnych prędkościach. Źródło: [11]

Niestety powyższy sposób wykorzystania tablicy rezerwacji w pewnych sytuacjach nie zapobiega zderzeniom czołowym jednostek zmierzających w przeciwnych kierunkach. Jeśli jedna jednostka zarezerwowała komórki (x, y, t) i $(x + 1, y, t + 1)$, nic nie stoi na przeszkodzie, aby kolejna jednostka mogła zarezerwować komórki $(x + 1, y, t)$ i $(x, y, t + 1)$. Ten problem może być rozwiązywany poprzez zajmowanie (rezerwowanie) dwóch sąsiednich komórek w tym samym czasie t podczas ruchu robota, a nie tylko jednej komórki.

3.4 Hierarchical Cooperative A*

Metoda *Hierarchical Cooperative A** (HCA*) wprowadza pewną modyfikację do algorytmu Cooperative A*. Modyfikacja ta dotyczy heurystyki opartej na abstrakcjach przestrzeni stanu [11]. HCA* jest także jednym z przykładów rozproszonego podejścia do planowania tras.

W tym podejściu abstrakcja przestrzeni stanu oznacza zignorowanie wymiaru czasu, jak również tablicy rezerwacji. Innymi słowy, abstrakcja jest prostą dwuwymiarową mapą z usuniętymi agentami. Abstrakcyjne odległości mogą być rozumiane jako dokładne oszacowania odległości do celu, ignorując potencjalne interakcje z innymi agentami. Jest to oczywiście dopuszczalna heurystyka (por. 3.1.2). Niedokładność heurystyki wynika jedynie z trudności związanych z interakcją z innymi agentami (jak bardzo agent musi zboczyć z pierwotnie zaplanowanej ścieżki w celu ominięcia innych agentów).

Do wyznaczenia heurystyki dla abstrakcyjnej przestrzeni stanu opisywane podejście wykorzystuje algorytm przeszukiwania *Reverse Resumable A** (RRA*). Algorytm ten wykonuje zmodyfikowane przeszukiwanie A* w odwrotnym kierunku. Przeszukiwanie zaczyna się w węźle docelowym agenta i kieruje się do początkowego położenia. Jednak zamiast kończyć w tym punkcie, przeszukiwanie jest kontynuowane do natrafienia na węzeł N , w którym znajduje się agent.

Algorytm HCA* jest więc taki, jak algorytm CA*, ale z bardziej wyszukaną heurystyką, która używa RRA* do obliczania abstrakcyjnych odległości na żądanie.

3.5 Windowed Hierarchical Cooperative A*

Problematyczne zagadnienie związane z wyżej wspomnianymi algorytmami jest takie, że kończą one działanie w momencie, gdy agent osiąga swój cel. Jeśli agent znajduje się już w miejscu docelowym, np. w wąskim korytarzu, to może on blokować części mapy dla innych agentów. W takiej sytuacji agenci powinni kontynuować kooperację z pozostałymi jednostkami, nawet po osiągnięciu swoich celów. Może to zostać zrealizowane np. poprzez usunięcie się z wąskiego gardła w celu przepuszczenia pozostałych agentów, a następnie powrót do docelowego punktu [11].

Kolejny problem związany jest z wrażliwością na kolejność agentów (przydzielone priorytety). Chociaż czasem możliwy jest skuteczny, globalny przydział priorytetów [7], to jednak dobrym rozwiązaniem może być dynamiczne modyfikowanie kolejności agentów. Wtedy rozwiązania mogą zostać znalezione w tych przypadkach, w których zawiodło przydzielanie niezmiennych priorytetów [11].

Rozwiązaniem powyższych kwestii jest zamknięcie algorytmu przeszukiwania w oknie cza-

sowym. Kooperacyjne planowanie jest ograniczone do ustalonej głębokości. Każdy agent szuka częściowej ścieżki do celu i zaczyna nią podążać. W regularnych okresach (np. gdy agent jest w połowie drogi) okno jest przesuwane dalej i wyznaczana jest nowa ścieżka.

Aby zapewnić, że agenci podążają do prawidłowych punktów docelowych, ograniczana jest tylko głębokość przeszukiwania kooperacyjnego (związanego z wieloma agentami), podczas gdy przeszukiwanie abstrakcyjnych odległości (heurystyki opisanej w podrozdziale 3.4) odbywa się bez ograniczeń głębokości. Okno o rozmiarze w może być rozumiane jako pośrednia abstrakcja, która jest równoważna wykonaniu w kroków w rzeczywistym środowisku (z uwzględnieniem pozostałych agentów) a następnie wykonaniu pozostałych kroków zgodnie z abstrakcją (bez uwzględnienia innych agentów). Innymi słowy, pozostały agenci są jedynie rozważani dla w pierwszych kroków (poprzez tablicę rezerwacji) a dla pozostałych kroków są ignorowani [11].

Rozmiar okna jest wielkością ustalaną arbitralnie. Rozmiar okna powinien być przyjęty jako czas trwania najdłuższego przewidywanego wąskiego gardła (zatoru). Dobrą praktyką jest przyjęcie wartości równej liczbie agentów na mapie, gdyż to właśnie z ich powodów mogą wystąpić ewentualne zmiany w zaplanowanej trasie.

Porównanie HCA* i WHCA*

Algorytm HCA* wybiera ustaloną kolejność agentów i planuje trasy dla każdego agenta po kolei, unikając kolizji z poprzednio wyznaczonymi ścieżkami. Natomiast użycie przeszukiwania z przesuwany oknem w WHCA* poprawia skuteczność algorytmu oraz przyspiesza proces wyznaczania rozwiązania [12].

Fakt wykonywania przeszukiwania w oknie oznacza, że planowanie algorytmem WHCA* wykonywane jest zawsze z ustaloną liczbą kroków w przyszłości i wybierany jest najbardziej obiecujący węzeł na granicy tego okna [13]. W metodach Hierarchical Cooperative A* oraz Cooperative A* wybór granicy wymiaru czasu (głębokości przeszukiwania w liczbie kroków) stanowi balans pomiędzy wydajnością a zupełnością algorytmu.

W obu podejściach HCA* i WHCA* zastosowano dodatkowy algorytm przeszukiwania wstecz (RRA*) wspomagający heurystykę. Służy on wyznaczeniu dokładnej odległości z węzła do celu, pomijając wpływ innych agentów. Jest to często heurystyka wysokiej jakości (prawie idealne oszacowanie), gorzej sprawdza się jedynie w środowiskach z dużą ilością wąskich gardeł i zatorów [13].

Chociaż takie przeszukiwanie wstecz prowadzi do początkowego wykonania większej ilości obliczeń (wykonanie pełnego przeszukania A* z punktu docelowego do punktu startowego, jak również do innych węzłów), to jednak koszt obliczeniowy w kolejnych krokach jest już zdecydowanie niższy [13].

3.6 Podsumowanie

Większość popularnych algorytmów wykorzystywanych do planowania tras dla wielu robotów mobilnych (agentów) opiera się o A*.

Kooperacyjne planowanie tras jest ogólną techniką koordynacji dróg wielu jednostek. Znajduje zastosowanie, gdzie wiele jednostek może komunikować się ze sobą, przekazując informację o ich ścieżkach. Poprzez planowanie wprzód w czasie, jak również i w przestrzeni, jednostki potrafią schodzić sobie z drogi nawzajem w celu uniknięcia kolizji. Metody kooperacyjnego planowania są bardziej skuteczne i znajdują trasy wyższej jakości niż te uzyskane przez A* z metodą *Local Repair*.

Wiele z udoskonaleń przestrzennego algorytmu A* może być również zaadaptowane do czasoprzestrzennego A*. Ponadto, wprowadzenie wymiaru czasu otwiera nowe możliwości do rozwoju algorytmów znajdowania dróg.

Najbardziej obiecującym pod względem skuteczności algorytmem wydaje się być metoda WHCA*.

Aby wydajnie prowadzić obliczenia, zakłada się, że każdy ruch robota trwa tyle samo. Wprowadzi to upraszczające, błędne założenie, że ruch robota na pole w kierunku poziomym lub pionowym trwa tyle samo, co na ukos.

W wielu przypadkach metody do planowania bezkolizyjnych tras w systemach wieloagentowych mogą być wykorzystywane zamiennie zarówno do wyznaczania trajektorii robotów mobilnych, jak i w grach komputerowych, np. strategiach czasu rzeczywistego do planowania tras wielu jednostek.

Zaprezentowane algorytmy mogą znaleźć zastosowanie również w środowiskach z ciągłą przestrzenią oraz w dynamicznych środowiskach, w których to ścieżki muszą być przeliczane po wykryciu zmiany na mapie.

Rozdział 4

Opracowanie i implementacja algorytmów

Na potrzeby stworzenia oprogramowania symulacyjnego do planowania bezkolizyjnych tras dla wielu robotów mobilnych należało opracować lub zaimplementować niezbędne algorytmy.

W tym rozdziale opisano algorytmy, które znalazły zastosowanie w stworzonej aplikacji symulującej ruchy robotów mobilnych. Należą do nich:

- Generator labiryntów (por. 4.1)
- Algorytm A* (por. 4.2)
- Algorytm LRA* (por. 4.3)
- Algorytm WHCA* (por. 4.4)
- Procedury obsługi dynamicznego przydzielania priorytetów oraz rozmiaru okna czasowego (por. 4.5)

4.1 Generator map

W rozdziale 6 opisano wyniki testów przeprowadzonych na losowo wygenerowanych środowiskach. Do ich automatycznego utworzenia wykorzystano własny generator map, który zapewnia im pewne pożądane własności opisane poniżej.

Zastosowanie zwykłego losowania położenia przeszkołd na mapie mogłoby spowodować, że do niektórych obszarów na mapie nie udałoby się znaleźć drogi, nawet mimo braku istnienia pozostałych agentów. Takie środowisko nie miałoby sensownego zastosowania w praktyce, w kooperacyjnym znajdowaniu tras.

Zależy nam, aby uzyskać środowisko cechujące się dużą liczbą przeszkód i wąskimi korytarzami, aby uwypuklić problem występowania wąskich gardeł. Jednocześnie chcemy jednak, aby istniała możliwość przejścia między dwoma dowolnymi punktami na mapie. Do rozwiązania problemu wygenerowania takich labiryntów posłużymy się teorią grafów i pojęciem grafu spójnego.

Definicja 4.1.1. Graf spójny

Graf spójny spełnia warunek, że dla każdej pary wierzchołków istnieje łącząca je ścieżka.

Układ pól na mapie będziemy reprezentować jako graf. W naszym przypadku wszystkie przejezdne pola na mapie są wierzchołkami grafu. Natomiast połączenia między sąsiednimi przejezdnymi polami (między którymi istnieje możliwość bezpośredniego przejścia) są krawędziami w grafie.

Aby zapewnić, że powstały graf będzie spójny, na początku losujemy jedno pole będące ziarnem rozrostu labiryntu, a następnie do takiego podgrafa dołączamy kolejne wierzchołki, łącząc je drogą na mapie. Krok ten powtarzamy do momentu, aż wszystkie wierzchołki zostaną dołączone. W tym celu wykorzystamy dwie pomocnicze listy wierzchołków:

- lista wierzchołków *odwiedzonych* - zawiera wierzchołki (pola) należące już do grafu tworzącego labirynt. Między wszystkimi wierzchołkami z listy *odwiedzonych* istnieje łącząca je droga - tworzą one graf spójny.
- lista wierzchołków *nieodwiedzonych* - zawiera nieodwiedzone wierzchołki (pola), które nie zostały jeszcze dołączone do labiryntu.

Kolejne kroki proponowanego algorytmu przedstawiają się następująco:

1. Inicjalizacja pustych list wierzchołków: *odwiedzonych* i *nieodwiedzonych*.
2. Zapełnienie całej mapy przeskodami. Następnie, zaznaczenie co drugiego pola (wzdłuż każdego z wymiarów) jako wolne (por. rys. 4.1a) i dodanie ich do listy *nieodwiedzonych*.
3. Wylosowanie ziarna rozrostu labiryntu z listy *nieodwiedzonych*, przeniesienie go na listę *odwiedzonych*.
4. Łączenie kolejnych wierzchołków nieodwiedzonych z wierzchołkami odwiedzonymi. Dopóki lista *nieodwiedzonych* nie jest pusta:
 - (a) Wylosowanie wierzchołka z listy *nieodwiedzonych*.
 - (b) Znalezienie najbliższego dla niego (w sensie metryki miejskiej) sąsiada z listy *odwiedzonych*.



Rysunek 4.1: Kolejne etapy generowania labiryntu: (a) Zaznaczenie co drugiego pola jako wolne i wybór ziarna rozrostu labiryntu. (b) Wylosowanie i łączenie kolejnego wierzchołka poprzez "wyburzanie" przeszkód na drodze (c) Wynikowa mapa pochodząca z generatora

- (c) Połączenie wierzchołków drogą poprzez "wyburzanie" przeszkód (zaznaczania pola jako wolne), przesuwając się w kierunku wierzchołka docelowego, najpierw wzdłuż osi poziomej, następnie wzdłuż osi pionowej (por. rys. 4.1b).

Pojedyncze dołączanie kolejnych wierzchołków do rozrastającego się grafu labiryntu zapewnia, że wynikowy graf również będzie grafem spójnym.

Algorytm 4.1.1 przedstawia pseudokod, obrazujący szczegóły działania zaprojektowanej metody.

Wyrażenie $x++$ oraz $x--$ oznacza odpowiednio postinkrementację oraz postdekrementację (dokonanie zwiększenia lub zmniejszenia o 1 po wykorzystaniu wartości zmiennej w wyrażeniu). Funkcja **ZNAJDŹ NAJBLIŻSZEGO SĄSIADA** wyszukuje sąsiada dla pola *nowePole* z listy *odwiedzone*. Wynik wyszukiwania jest najbliższy w sensie metryki miejskiej (metryki Manhattan).

Kolejne etapy generowania labiryntu zobrazowano na rysunku 4.1.

Na rysunku 4.2 przedstawiono przykładowy labirynt wygenerowany opisanym algorytmem.

Wygenerowane w ten sposób mapy mają jeszcze jedną właściwość - w takim środowisku robot nie ma nigdy możliwości wykonania ruchu ukośnego (innego niż w poziomie lub pionie). Pozwala to wprowadzić pewne ograniczenia do niektórych algorytmów planowania (por. 3.3), które przyspieszają wykonywanie obliczeń ze względu na możliwość założenia jednakowego czasu trwania wszystkich ruchów robotów. Nie będziemy jednak tego zakładać na tym etapie, gdyż chcemy, aby opracowana metoda planowania tras mogła także działać w środowiskach innego typu.

Algorytm 4.1.1 Generowanie labiryntu

```
1: function GENERUJLABIRYNT(w, h) // w - szerokość mapy, h - wysokość mapy
2:   mapa[][] ← nowa tablica w × h wypełniona wartościami ZABLOKOWANE
3:   nieodwiedzone ← Ø, odwiedzone ← Ø // inicjalizacja pustych list
4:   for x ← 0; x < w; x ← x + 2 do // co drugi indeks szerokości
5:     for y ← 0; y < h; y ← y + 2 do // co drugi indeks wysokości
6:       mapa[x][y] ← WOLNE
7:       dodaj mapa[x][y] do nieodwiedzone
8:     end for
9:   end for
10:  ziarno ← losowe pole z nieodwiedzone
11:  dodaj ziarno do odwiedzone // przeniesienie do odwiedzone
12:  usuń ziarno z nieodwiedzone
13:  while nieodwiedzone ≠ Ø do // dopóki nieodwiedzone nie jest pusta
14:    nowePole ← losowe pole z nieodwiedzone
15:    sasiad ← ZNAJDŹNAJBŁIŻSZEGOSĄSIADA(odwiedzone, nowePole)
16:    WYBURZDROGĘ(mapa, nowePole, sasiad) // przeniesienie do odwiedzone
17:    dodaj nowePole do odwiedzone
18:    usuń nowePole z nieodwiedzone
19:  end while
20:  return mapa[]
21: end function
22:
23: function WYBURZDROGĘ(mapa, poleZ, poleDo)
24:   x ← poleZ.x
25:   y ← poleZ.y
26:   while x < poleDo.x do // przesuwanie w prawo
27:     mapa[x++][y] ← WOLNE // zwiększenie x
28:   end while
29:   while x > poleDo.x do // przesuwanie w lewo
30:     mapa[x--][y] ← WOLNE // zmniejszenie x
31:   end while
32:   while y < poleDo.y do // przesuwanie w dół
33:     mapa[x][y++] ← WOLNE // zwiększenie y
34:   end while
35:   while y > poleDo.y do // przesuwanie w górę
36:     mapa[x][y--] ← WOLNE // zmniejszenie y
37:   end while
38: end function
```



Rysunek 4.2: Przykładowy labirynt rozmiaru 75×75 pochodzący z zaprojektowanego generatora map

4.2 Wyznaczanie trajektorii dla pojedynczego robota

Wiele technik planowania tras, mimo iż może posłużyć do koordynacji ruchu wielu robotów, korzysta z metod indywidualnego planowania dla każdego agenta z osobna. Przykładem takiego podejścia jest *Local-Repair A**, w którym to głównym algorytmem przeszukującym jest A*. Z tego powodu przed przystąpieniem do opracowania bardziej skomplikowanych metod, należy najpierw zająć się implementacją algorytmu poszukiwania najkrótszej drogi na dwuwymiarowej mapie z jednym robotem. Wymagane jest to nawet dla metody WHCA*, która pomimo, że operuje na węzłach określonych w czasie i przestrzeni, to jednak w obliczaniu samej heurystyki wykorzystuje przestrzenny algorytm A*. Ogólna zasada działania algorytmu A* wraz z kluczowymi pojęciami została opisana w rozdziale 3.1.

Pseudokod 4.2.1 oraz opisy użytych pomocniczych funkcji ukazują szczegółowo implementacji metody wyznaczania najkrótszej drogi na przestrzennej mapie. Algorytm oparty jest o A*, jednak posiada własne, niewielkie modyfikacje.

Algorytm 4.2.1 Algorytm A*

```
1: function ZNAJDZDROGĘ(mapa, start, cel)
2:   closed  $\leftarrow \emptyset$                                 // pusta lista zamkniętych
3:   open  $\leftarrow \{start\}$                             // lista otwartych zawiera punkt startowy
4:   for wezel  $\in$  mapa do
5:     wezel.g  $\leftarrow \infty$                          // domyślnie nieskończony koszt - odległość od startu
6:   end for
7:   start.g  $\leftarrow 0$                              // Zerowy koszt przejścia do węzła startowego
8:   while open  $\neq \emptyset$  do                  // dopóki lista otwartych nie jest pusta
9:     obecny  $\leftarrow$  ZNAJDŹMINF(open)           // Szukamy pola o najniższej wartości f
10:    if obecny == cel then
11:      return ZBUDUJŚCIEŻKE(cel)                // Znaleziono ścieżkę
12:    end if
13:    dodaj obecny do closed                   // Przesunięcie z open do closed
14:    usuń obecny z open
15:    for sasiad  $\in$  SĄSIEDZI(obecny) do      // Dla każdego sąsiada aktualnego pola
16:      if mapa[sasiad.x][sasiad.y] == ZABLOKOWANE or
17:        not PRZEJŚCIEPOPRAWNE(obecny, sasiad) then
18:          continue                                // Ignoruj niepoprawne pola lub przejścia
19:        end if
20:        nowyKoszt  $\leftarrow$  obecny.g + KOSZTPRZEJŚCIA(obecny, sasiad)
21:        if nowyKoszt < sasiad.g then            // znaleziono korzystniejsze połączenie
22:          usuń sasiad z open                  // konieczność ponownego przeliczenia
23:          usuń sasiad z closed                // jeśli sasiad  $\in$  open lub sasiad  $\in$  closed
24:        end if
25:        if sasiad  $\notin$  open  $\wedge$  sasiad  $\notin$  closed then
26:          sasiad.g  $\leftarrow$  nowyKoszt             // zapisanie nowego połączenia
27:          sasiad.h  $\leftarrow$  HEURYSTYKA(sasiad, cel)
28:          sasiad.parent  $\leftarrow$  obecny           // pole obecny rodzicem dla pola sasiad
29:          dodaj sasiad do open
30:        end if
31:      end for
32:    end while
33:    return  $\emptyset$                                 // Przeanalizowano wszystkie węzły, brak istniejącej ścieżki
34: end function
```

Wykorzystane zostały pomocnicze funkcje:

- ZNAJDŹMINF(*lista*) - Funkcja zwraca z listy pole o najniższej wartości *f* (sumie kosztu i heurystyki);
- ZBUDUJŚCIEŻKE(*cel*) - Funkcja zwraca ścieżkę z punktu startowego do punktu *cel* zbudowaną na podstawie przechodzenia wstecz od punktu *cel* po kolejnych rodzicach węzłów, aż do dotarcia do węzła bez rodzica (punktu startowego).

- **SĄSIEDZI**(*pole*) - Funkcja zwraca zbiór pól bezpośrednio sąsiadujących (dla których istnieje możliwość przejścia) ze wskazanym polem. Jest to zazwyczaj zbiór ośmiu sąsiadujących pól. Na granicach mapy należy uwzględniać warunki brzegowe.
- **PRZEJŚCIEPOPRAWNE**(*poleZ, poleDo*) - Funkcja zwraca prawdę wtedy i tylko wtedy, gdy istnieje możliwość przejścia z *poleZ* do *poleDo*. Gdy wykonywany jest ruch ukośny, ale na przynajmniej jednym polu sąsiadującym z *poleZ* i *poleDo* znajduje się przeszkoda, to taki ruch jest niepoprawny. Ruch ukośny agenta z punktu (x_1, y_1) do (x_2, y_2) możliwy jest tylko w przypadku, gdy na żadnym z czterech pól: $(x_1, y_1), (x_2, y_1), (x_1, y_2), (x_2, y_2)$ nie znajduje się przeszkoda. W rzeczywistym środowisku zabezpiecza to robota przed uderzaniem o wystający róg ściany.
- **KOSZTPRZEJŚCIA**(*poleZ, poleDo*) - Funkcja zwraca koszt przejścia z *poleZ* do *poleDo*. Jest to odległość w sensie metryki euklidesowej.
- **HEURYSTYKA**(*poleZ, poleDo*) - Funkcja zwraca przewidywaną długość pozostałej drogi od *poleZ* do *poleDo*. Jest to również odległość euklidesowa.

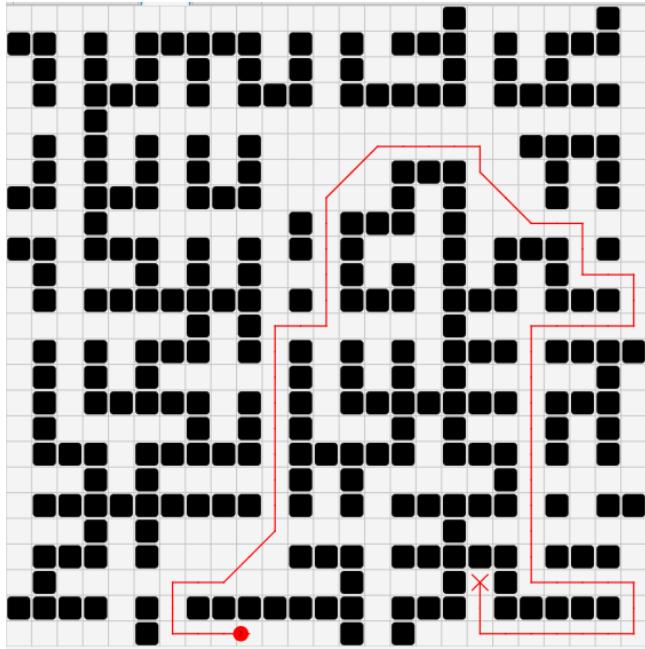
Potwierdzeniem poprawności zaimplementowania algorytmu wyznaczania najkrótszej ścieżki jest przykładowa droga przedstawiona na rysunku 4.3 pochodzący z aplikacji. Warto zauważyć, że możliwy jest ruch ukośny robota, ale nie taki, który powodowałby kolizję z wystającym rogiem ”ściany”.

4.3 Detekcja i przeciwdziałanie kolizjom

Do zademonstrowania prostego przeciwdziałania kolizjom robotów wykorzystano metodę LRA* (*Local-Repair A**). Należy zaznaczyć, że jest to bardzo uproszczone podejście i stanowi wstęp do bardziej złożonego algorytmu. Dużo skuteczniejsza w zapobieganiu kolizji jest metoda WHCA* (por. 4.4) z dynamicznym przydziałem priorytetów (por. 4.5), która została opisana w późniejszych rozdziałach.

Zasada działania LRA* mocno opiera się na standardowym algorytmie A*. Jest to metoda korekcji tras w przypadku lokalnie wykrytych kolizji.

Planowanie trajektorii odbywa się dla każdego agenta niezależnie, stosując algorytm A*. Przy wyznaczaniu najkrótszej drogi uwzględniane są (jako zajęte pola) zarówno wszystkie przeszkody statyczne, jak i aktualne pozycje pozostałych robotów. Wyznaczona ścieżka dzielona jest na pojedyncze akcje agenta (ruchy robota), które zostają zakolejkowane na liście akcji do wykonania przez robota. Jeśli droga do celu nie mogła zostać wyznaczona, próba ta zostanie powtórzona w kolejnym kroku.



Rysunek 4.3: Przykładowa ścieżka wyznaczona przez zaimplementowany w aplikacji algorytm A*. Kolorowe koło reprezentuje robota, kolorowe koło - robota, linia łamana - wyznaczoną ścieżkę, czarne kwadraty - przeszkody.

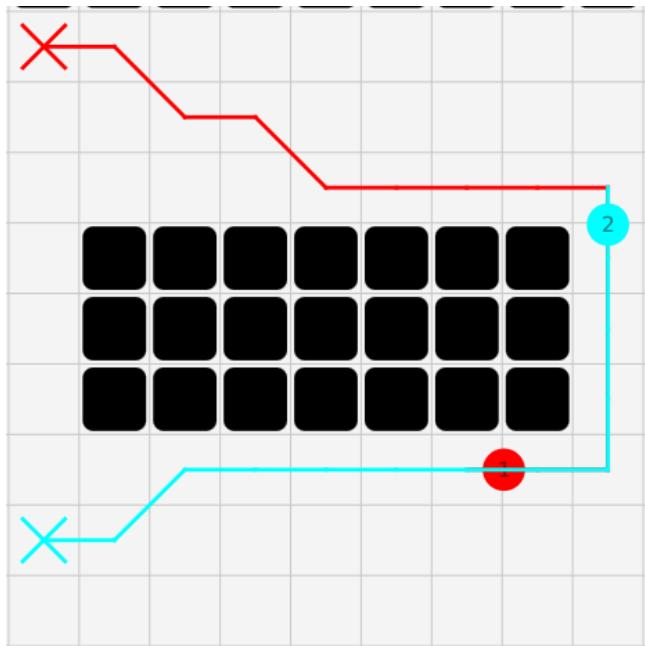
W każdym kroku symulacji agenci poruszają się wzdłuż wyznaczonych tras. Co prawda w kolejnych krokach ścieżki te mogą być już nieaktualne, jednak agenci wciąż nimi podążają, aż do osiągnięcia punktu docelowego lub wykrycia kolizji. Wykrycie kolizji w tym przypadku nie oznacza faktycznego zderzenia (przebywania więcej niż jednego robota na tym samym polu), a jedynie możliwość wystąpienia takiej niepożądanej sytuacji w następnym kroku symulacji.

Detekcja potencjalnej kolizji sprowadza się do przeanalizowania dla każdego robota jego zaplanowanego położenia z następnego kroku. Jeśli robot nie ma zaplanowanej żadnej ścieżki, oznacza to, że jego położenie w następnym kroku będzie takie samo, jak w obecnym. Uznaje się, że kolizja może wystąpić w dwóch przypadkach:

- w przypadku, gdy planowane położenie robota w następnym kroku jest takie samo jak aktualne położenie innego robota;
- w przypadku, gdy planowane położenie robota w następnym kroku jest takie samo jak planowane położenie innego robota w następnym kroku.

Sprawdzenie obu tych przypadków jest konieczne, aby wykrywać także zderzenia czołowe robotów zmierzających w przeciwnych kierunkach.

Skutkiem tak wykrytej potencjalnej kolizji jest wykonanie ponownego planowania (tylko dla jednego robota lub dla wielu) trajektorii. Uwzględnienie aktualnego położenia sąsiadującego



Rysunek 4.4: Przykład powtarzających się cykli planowanych akcji uzyskanych przez metodę LRA*. Dwa roboty "zderzą się" w wąskim przejściu po prawej stronie i wyznaczają trajektorie przechodzące przez przejście po lewej stronie, gdzie ponownie dojdzie do kolizji.

robota zapobiega zderzeniu się z nim, gdyż pole, które on zajmuje jest już podczas ponownego planowania rozpatrywane jako nieprzejezdne.

Przy takim podejściu priorytety robotów nie mają znaczenia, zaś samo planowanie może odbywać się dla agentów równolegle. W celu poprawy wydajności technikę tą można połączyć z algorytmem D* Lite, D* Extra Lite lub RRA*.

Łatwo zauważyc, że metoda ta nie zapobiega tworzeniu się cykli planowanych akcji - powtarzających się w nieskończoność sekwencji tych samych ruchów, które nie doprowadzają agenta do celu (por. rys. 4.4).

4.4 Implementacja algorytmu WHCA*

TODO rozwiązuje bottleneck - nawet radzi sobie dla większej liczby. Roboty wykazują inteligentne zachowanie, potrafią kooperować ze sobą, schodzić sobie z drogi, nawet, gdy już dotarły do swojego celu

WHCA - własny, schemat blokowy, pseudokod wiele trzeba było zrobić samemu, bo nic nie było podane zaprojektowany własny algorytm planowanie tylko części ścieżki (długości okna) koszt pozostania w miejscu - zerowy po przejściu algorytmu, wyłanianie najlepszego rozwiązania, 2 kryteria

rozszerzona wersja o dynamiczne przydzielanie priorytetów i rozszerzanie okna

4.5 Dynamiczny przydział priorytetów

TODO metoda przydziału / zmiany priorytetów - zwiększenie i rekalkulacja w przypadku braku znalezienia trasy, nie dotyczy np LRA

zwiększenie priorytetu w każdym kroku, jeśli nie znaleziono trasy, z próbą ponownego znalezienia w następnym kroku zwiększenie okna czasowego do max z priorytetów

przykłady rozwiązywanych problemów

należy wykrywać kolizje. kolizje też mogą się zdarzyć, bo np robot który planuje drogę wcześniej nie uwzględnia że następnemu robotowi nie uda się wyznaczenie trajektorii.

4.6 Metoda pól potencjałowych

TODO że nie wyszło , bo minima lokalne zalety: real time - potrzeba mało obliczeń, szybka metoda siła od celu robota jest stała (niezależna od odległości) i tak wyszło chujowo efekt wallhacka

Rozdział 5

Oprogramowanie symulacyjne

Na potrzeby pracy zostało stworzone oprogramowanie symulacyjne, które posłużyło do przeprowadzenia testów skuteczności algorytmów planowania tras oraz wizualizacji ich działania. W tym rozdziale opisano techniczne rozwiązania wykorzystane podczas tworzenia oprogramowania. W aplikacji zostały zaimplementowane algorytmy planowania tras opisane w rozdziale 4. Prezentacja ich działania odbywa się poprzez wizualizację ruchu robotów mobilnych w czasie rzeczywistym.

5.1 Funkcjonalności aplikacji

Aplikacja umożliwia dowolne definiowanie przez użytkownika środowiska, w którym poruszają się roboty. Obejmuje to:

- wybór rozmiaru mapy - dowolną wysokość oraz szerokość. Mapa nie musi być kwadratowa.
- możliwość wygenerowania mapy za pomocą generatora labiryntów (por. 4.1) lub manualnego umieszczania przeszkód na mapie za pomocą myszki,
- wybór liczby robotów i dokonanie ich automatycznego rozmieszczenia na mapie (w losowych polach z pominięciem pól zajętych). Użytkownik ma także możliwość manualnego dodawania i usuwania robotów.

Aplikacja przeprowadza symulację ruchu robotów w czasie rzeczywistym. W oprogramowaniu zostały zaimplementowane trzy algorytmy planowania ruchu robotów. Są to:

- Metoda pól potencjałowych (por. 4.6)
- Local-Repair A* (por. 4.3)

- Windowed Hierarchical Cooperative A* (por. 4.4)

Wizualizacja każdego z tych algorytmów dostępna jest na osobnej zakładce w aplikacji.

5.2 Graficzny interfejs użytkownika

Graficzny interfejs użytkownika stanowi desktopowa aplikacja okienkowa, której głównym elementem jest panel zakładek. Każda z zakładek reprezentuje wizualizację osobnego algorytmu planowania ruchu robotów.

Z myślą o spopularyzowaniu opracowanych algorytmów, elementy interfejsu użytkownika (takie, jak etykiety lub przyciski) posiadają tekst w języku angielskim.

Pod każdą z zakładek aplikacji układ wizualny jest podobny. Poniżej opisano graficzny interfejs użytkownika wspólny dla wszystkich zakładek.

Po prawej stronie znajduje się panel przycisków akcji oraz panel rozwijanych właściwości z konfiguracją parametrów mapy oraz symulacji. Natomiast po lewej stronie wyświetlany jest obecny stan mapy i położenia robotów mobilnych. Mapa wyświetlana jest z zaznaczeniem linii siatki pól obrazujących liczbę pól na mapie. Przeszkody wyświetlane są jako czarne kwadraty, zaś roboty jako kolorowe wypełnione koła. Kliknięcie prawym przyciskiem myszy (również z możliwością ciągłego przytrzymania) powoduje wstawienie lub usunięcie przeszkody na mapie w odpowiednich polach. Punkty docelowe, do których podążają roboty wyświetlane są jako krzyżyki w kolorze takim samym, jaki został przypisany do robota. Liczba robotów nie jest ograniczona przez aplikację. Aby odróżnić roboty między sobą, każdy robot otrzymuje inny kolor wynikający z równego podziału palety HSV według barwy (ang. *hue*) na liczbę robotów.

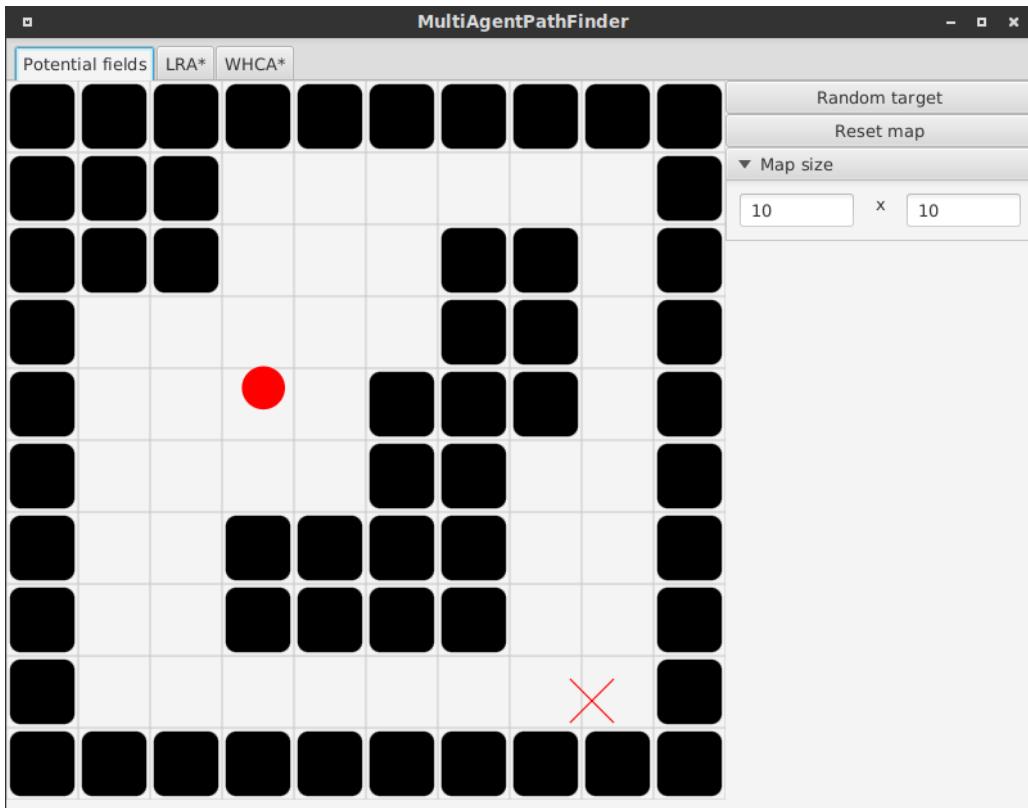
Okno aplikacji jest skalowalne i w pełni responsywne. Po zmianie rozmiaru okna przez użytkownika rozmiar mapy dopasowuje się do maksymalnego obszaru, jaki może zająć (z zachowaniem proporcji wymiarów mapy).

Symulacja ruchu robotów rozpoczyna się jak tylko zostanie wyznaczony punkt docelowy dla robota. Dzięki temu, że wykonywanie obliczeń planowania tras zostało przeniesione do osobnych wątków, uzyskano płynność animacji ruchu robotów mobilnych. Uniknięto również problemu braku odpowiedzi interfejsu użytkownika podczas planowania trajektorii.

5.2.1 Metoda pól potencjałowych

Na pierwszej zakładce "Potential fields" w oknie aplikacji przedstawiona jest wizualizacja metody pól potencjałowych.

Na panelu konfiguracji parametrów, użytkownik może wybrać dowolny rozmiar mapy, podając szerokość i wysokość wyrażone w liczbie pól. Po naciśnięciu przycisku "Reset map" tworzona



Rysunek 5.1: Zrzut ekranu aplikacji w trakcie wizualizacji metody pól potencjałowych.

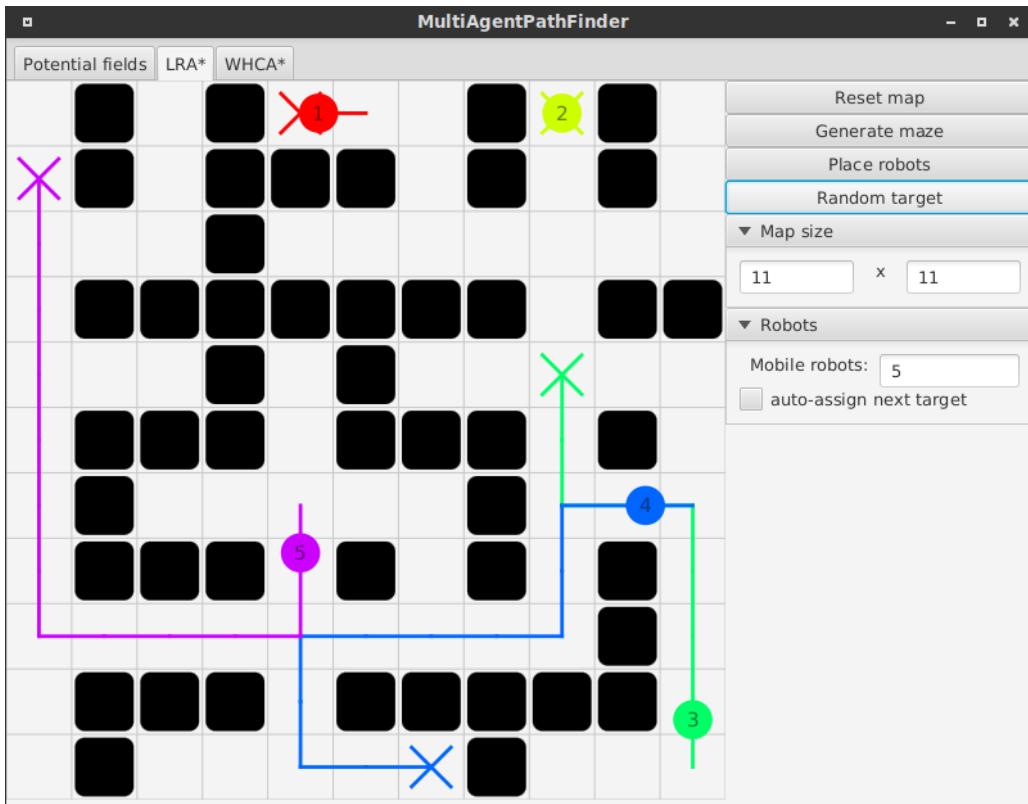
jest nowa mapa o zadanych rozmiarach, natomiast pozycja początkowa robota zostaje wylosowana. Naciśnięcie przycisku "Random target" powoduje wyznaczenie losowego punktu na mapie jako cel dla robota i rozpoczęcie podążania za nim.

Klikając lewym przyciskiem myszy na mapie, Użytkownik może manualnie wskazać punkt docelowy dla robota. Planowanie i symulacja ruchu robota odbywa się w trybie ciągłym. Robot nie jest ograniczony zdyskretyzowaną siatką pól na mapie, jego przestrzeń poruszania się jest ciągła.

5.2.2 Local-Repair A*

Pod kolejną zakładką "LRA*" w oknie aplikacji dostępna jest wizualizacja metody algorytmu bazującego na A* z lokalną detekcją i rozwiązywaniem kolizji.

Na panelu konfiguracji parametrów w sekcji "Map size", użytkownik może wybrać dowolny rozmiar mapy, podając szerokość i wysokość wyrażone w liczbie pól. Po naciśnięciu przycisku "Reset map" tworzona jest nowa, pusta mapa o zadanych rozmiarach. W sekcji "Robots" użytkownik wybiera liczbę robotów (do automatycznego rozmieszczenia) oraz ma możliwość zaznaczenia opcji "auto-assign next target", co skutkuje automatycznym wyznaczaniem kolejnego punktu docelowego, dla robota, który dotarł do poprzedniego celu. Przycisk "Generate maze"



Rysunek 5.2: Zrzut ekranu aplikacji w trakcie wizualizacji metody Local-Repair A*.

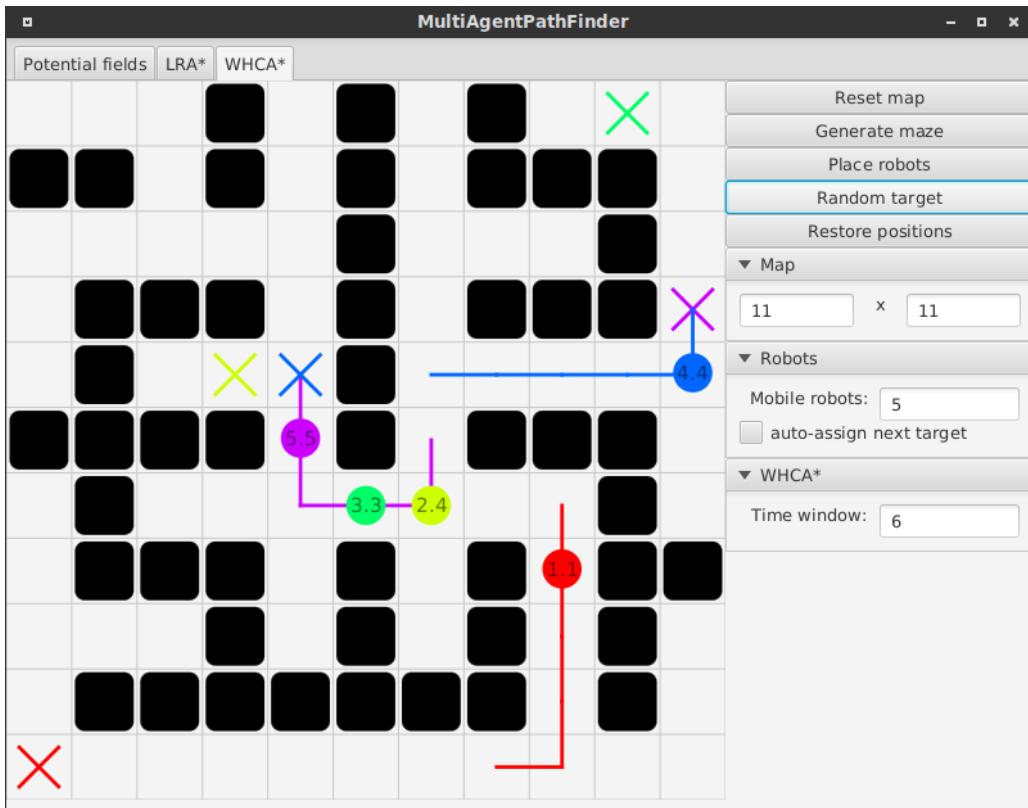
pozwala na wygenerowanie labiryntu przy użyciu generatora map (por. 4.1). Przycisk "Place robots" usuwa wszystkie roboty z mapy i umieszcza na niej wybraną przez użytkownika liczbę robotów w losowych miejscach na mapie (za wyjątkiem zajętych przez przeszkody oraz inne roboty). Przycisk "Random target" losuje punkty docelowe dla wszystkich robotów. Każdy z agentów otrzymuje inny punkt docelowy (nie będący przeszkodą), do którego zmierza. Nadanie punktów docelowych powoduje rozpoczęcie planowania trajektorii i symulację ruchu agentów.

Klikając lewym przyciskiem myszy na mapie, Użytkownik może utworzyć nowego robota. Nowemu agentowi zostaje przydzielony punkt docelowy w miejscu, w którym lewy przycisk myszy został zwolniony. Dodatkowo pokazywane są zaplanowane ścieżki dla każdego robota. Wyświetlane są jako linie łamane w kolorze takim samym, jaki odpowiada robotowi. Nad robotem, w jego aktualnym położeniu wyświetlany jest jego numer (identyfikator).

5.2.3 Windowed Hierarchical Cooperative A*

Pod ostatnią zakładką "WHCA*" w oknie aplikacji dostępna jest wizualizacja metody kooperacyjnego planowania tras dla wielu robotów z planowaniem odbywającym się w ograniczonym oknie czasowym.

Interfejs użytkownika jest praktycznie taki sam, jak w przypadku opisanej metody LRA*



Rysunek 5.3: Zrzut ekranu aplikacji w trakcie wizualizacji metody Windowed Hierarchical Cooperative A*.

(por. 5.2.2). Różni się obecnością dodatkowej sekcji "WHCA*" na panelu konfiguracji z polem "Time window", które pozwala na zmianę wielkości okna czasowego dla algorytmu WHCA*. W wyniku dynamicznego przydzielania priorytetów wielkość ta może zostać zwiększana automatycznie podczas symulacji. Dodatkowy przycisk "Restore positions" pozwala na cofnięcie stanu robotów do momentu, tuż przed rozpoczęciem symulacji (zaraz po przydzieleniu punktów docelowych). Pozwala to na ponowną obserwację przebiegu symulacji dzięki przywróceniu położenia i stanu agentów.

Nad robotem, w jego aktualnym położeniu wyświetlany jest jego numer (identyfikator) oraz priorytet (oddzielony kropką). Priorytet może być zmienny w trakcie symulacji (w przeciwieństwie do identyfikatora). Początkowo robot zawsze ma priorytet równy identyfikatorowi. Większa wartość priorytetu względem pozostałych oznacza pierwszeństwo uwzględniane podczas planowania tras.

5.3 Wykorzystane technologie

Do stworzenia aplikacji wykorzystano wiele technologii, narzędzi deweloperskich oraz technik opisanych w poniższych podrozdziałach.

Aplikacja była rozwijana i testowana na systemach operacyjnych z jądrem Linux (Arch Linux oraz Debian 9), jednak wszystkie zastosowane narzędzia oraz technologie są wieloplatformowe i z powodzeniem mogą być uruchomione na innych systemach opearcyjnych (np. Windows, macOS)

5.3.1 Java 8

Aplikacja w całości została napisana w języku Java. Java jest językiem programowania ogólnego przeznaczenia, zorientowanym obiektowo. Posiada obsługę wyjątków oraz naturalne wsparcie dla wielowątkowości, pozwala na szybkie tworzenie wysokopoziomowych programów. Aplikacje napisane w języku Java kompilowane są do kodu bajtowego, który uruchamiany jest przez wirtualną maszynę Javy (JVM), niezależnie od architektury urządzenia.

Dzięki wieloplatformowości stworzony program może być uruchamiany na każdym systemie operacyjnym z wirtualną maszyną Javy.

W projekcie aplikacji wykorzystano język Java w wersji 8. Pozwoliło to na użycie w projekcie takich elementów języka, jak: wyrażenia lambda i interfejsy funkcyjne (skracające zapis i zwiększające czytelność kodu) oraz strumieni (stream API) - do szybkiego i przejrzystego przekształcania i filtracji danych.

5.3.2 JavaFX

JavaFX jest platformą do tworzenia aplikacji desktopowych (okienkowych) pisanych w języku Java. Od wersji 8 JavaFX została włączona do platformy Java Standard Edition. Jest to nowsze, obecnie zalecane przez firmę *Oracle* rozwiązanie do tworzenia aplikacji okienkowych. Nie zaleca się natomiast korzystania z bibliotek *AWT* lub *Swing* [25].

Do zbudowania widoków aplikacji wykorzystano specjalny format plików FXML, w którym to JavaFX przechowuje informację o właściwościach komponentów interfejsu użytkownika oraz o ich wzajemnych relacjach.

5.3.3 Spring

Spring Framework

Spring jest frameworkiem, który umożliwia w aplikacjach Javy zastosowanie wzorca architektonicznego odwrócenia sterowania (IoC - ang. *Inversion of Control*), a w szczególności wstrzykiwania zależności (ang. *Dependency Injection*). Pozwala to uniknąć występowania bezpośrednich zależności pomiędzy komponentami oraz umożliwia automatyczne dostarczanie i zarządzanie cyklem życia komponentów aplikacji.

W zaprojektowanej aplikacji Spring jest wykorzystywany m.in. do automatycznego dostarczania współdzielonych danych o parametrach symulacji oraz do zarządzania cyklem życia klas prezenterów i widoków z biblioteki JavaFX.

Spring Boot

Spring Boot jest rozwiązaniem przyspieszającym proces konfiguracji, tworzenia oraz uruchamiania aplikacji opartych na Spring Framework. Jest to zestaw wstępnie skonfigurowanych komponentów, dzięki którym jeszcze łatwiejsze staje się dołączenie nowych bibliotek zewnętrznych do projektu. Celem jest pozbycie się zbędnych konfiguracji w plikach XML a zastąpienie ich domyślnym zestawem konfiguratorów, gdyż większość komponentów w aplikacji zazwyczaj konfigurowana jest w typowy, powtarzalny sposób [20].

Spring Boot dostarcza do aplikacji symulacyjnej wiele zależności do bibliotek (np. logback) oraz dostarcza konfigurację dla systemu budowania Maven.

Spring Boot JavaFx Support

Spring Boot JavaFx Support jest niewielką biblioteką umożliwiającą użycie Spring Boot w jednym projekcie w połączeniu z JavaFx.

5.3.4 jUnit i Test-driven development

jUnit jest frameworkiem do wykonywania testów jednostkowych dla programów napisanych w Javie. Testy jednostkowe weryfikują poprawność działania pojedynczych komponentów aplikacji. Uruchamiane są automatycznie podczas budowania projektu.

W projekcie zastosowano podejście TDD (ang. *Test-driven development*) dla procesu rozwoju oprogramowania. Dotyczy to w szczególności rozwoju logiki silnika planowania tras dla algorytmu A* i WHCA*. Algorytm WHCA* jest na tyle złożony, że zdecydowano się najpierw napisać szczególne przypadki testowe, które określały jakie dane wyjściowe (zaplanowane trajektorie) są oczekiwane przy zadanych danych wejściowych. Dopiero po takim pokryciu testami

przystąpiono do implementacji algorytmu. Warunkiem poprawności zaimplementowanego algorytmu było, aby wszystkie testy jednostkowe wykonały się prawidłowo. Na tym właśnie podejściu opiera się proces TDD (ang. *Test-driven development*), który jest naturalnie wspierany przez framework jUnit.

Bibliotekę jUnit wykorzystano także do wykonania testów skuteczności algorytmów i wszystkich pozostałych testów opisanych w rozdziale 6. Pomimo, że nie jest to typowym zastosowaniem biblioteki jUnit, to jednak umożliwia to wykonanie fragmentów logiki w innych, niestandardowych trybach pracy bez zmiany działania głównej aplikacji w normalnym trybie.

5.3.5 Maven

Apache Maven jest narzędziem do automatyzacji procesu budowania aplikacji napisanych w języku Java. Przy pierwszym wykorzystaniu zadeklarowanych bibliotek Maven automatycznie pobiera biblioteki ze swojego repozytorium i rozwiązuje wszystkie brakujące zależności do nich. Użytkownik zatem nie musi martwić się o brakujące biblioteki i o manualne dołączanie ich do projektu.

Narzędzie to zostało użyte w aplikacji do komplikacji ze źródeł oraz uruchamiania. Konfiguracja procesu budowania dla Maven jest wspomagana przez Spring Boot. Uruchomienie programu z kodów źródłowych następuje po wykonaniu w systemie polecenia: *mvn spring-boot:run*. Do komplikacji i uruchomienia całej aplikacji wystarczy zatem, aby w systemie operacyjnym było zainstalowane środowisko Java JDK oraz *Apache Maven*. Wszystkie potrzebne biblioteki zostaną pobrane automatycznie przez sieć Internet.

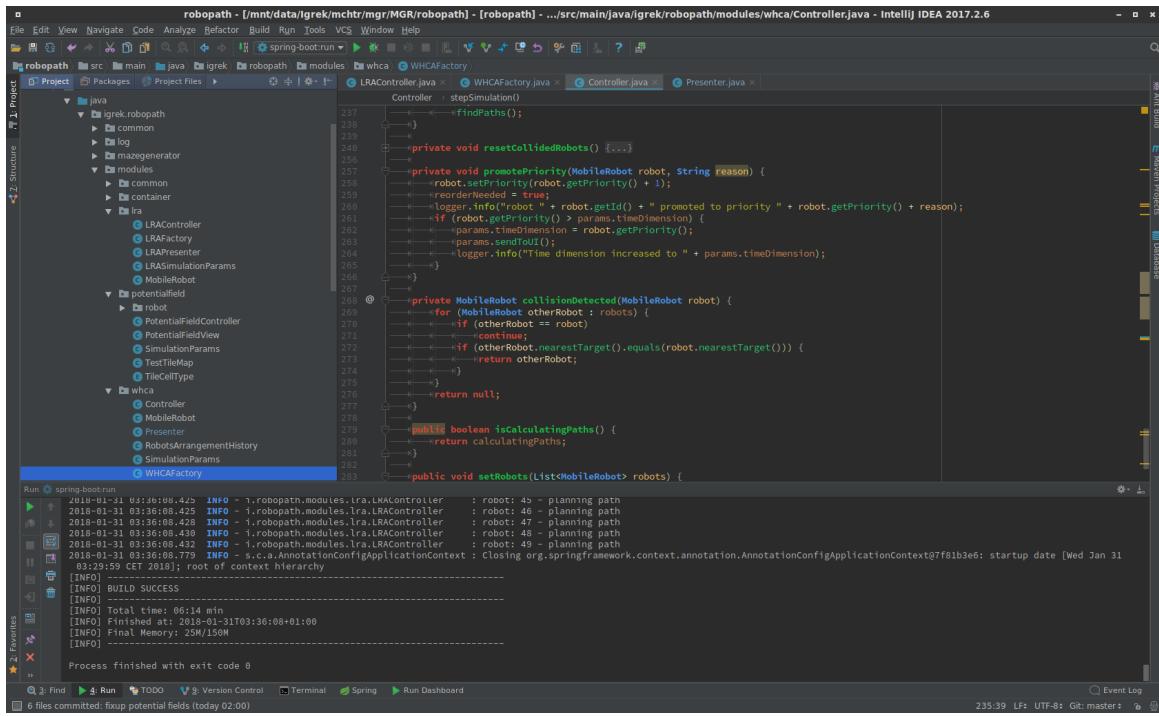
5.3.6 IntelliJ IDEA

IntelliJ IDEA jest zintegrowanym środowiskiem deweloperskim przeznaczonym głównie do rozwoju aplikacji w języku Java. Podczas opracowywania oprogramowania korzystano z wersji IntelliJ IDEA Ultimate 2017.2.6, z licencji studenckiej. Zrzut ekranu podczas pracy ze środowiskiem zaprezentowano na rysunku 6.5. Środowisko to zapewnia integrację ze wszystkimi technologiami wymienionymi w tym podrozdziale.

5.3.7 Pozostałe narzędzia i biblioteki

Guava

Guava jest biblioteką rozwijaną przez Google dostarczającą do Javy m.in. nowe typy kolekcji (struktur danych). W rozwijanej aplikacji wykorzystano ją m.in. do przetwarzaniałańcuchów tekstowych.



Rysunek 5.4: Zrzut ekranu środowiska deweloperskiego IntelliJ IDEA Ultimate

git

Do śledzenia i zapisywania zmian wykorzystano w projekcie system kontroli wersji *git*. Kody źródłowe powstałego programu są ogólnie dostępne w repozytorium git na portalu GitHub: *TODO* adres Github, czy wolno?

Logback

Do zapisu dziennika zdarzeń oraz błędów w aplikacji została wykorzystana biblioteka *Logback*, która jest domyślnie dostarczana do aplikacji i skonfigurowana dzięki *Spring Boot*.

5.4 Struktura aplikacji

5.4.1 Wzorzec Model-View-Presenter

Bardzo istotnym aspektem projektowanego oprogramowania okazała się być jego architektura. Przed będącym przedmiotem tego rozdziału symulatorem ruchu robotów zostało postawione wymaganie możliwości wykonania symulacji zarówno w trybie graficznej wizualizacji w czasie rzeczywistym, jak i w trybie przeprowadzenia obszernych testów algorytmów w celu zebrania potrzebnych danych statystycznych.

Oprogramowanie zostało zorganizowane w architekturę warstwową. Główny szkielet aplikacji zbudowany jest w oparciu o wzorzec architektoniczny MVP (ang. *Model-View-Presenter* - Model-Widok-Prezenter), będący pochodną wzorca MVC (ang. *Model-View-Controller*). Takie podejście zapewnia separację głównej logiki dziedziny programu od warstwy interfejsu użytkownika [21].

Wybrano podejście MVP, gdyż (w porównaniu do MVC) oferuje ono łatwiejszą testowalność kodu, możliwość "odczepiania" i wymieniania poszczególnych warstw (w celach testowania). Zapewnia to także większą separację modelu z widokiem, które nie muszą "wiedzieć" o sobie, a komunikują się jedynie za pośrednictwem prezentera.

Każdą zaimplementowaną metodę symulacji ruchu robotów zamknięto w osobnych modułach (pakiety). Każdy z tych modułów posiada swoje osobne klasy pochodzące z architektury MVP:

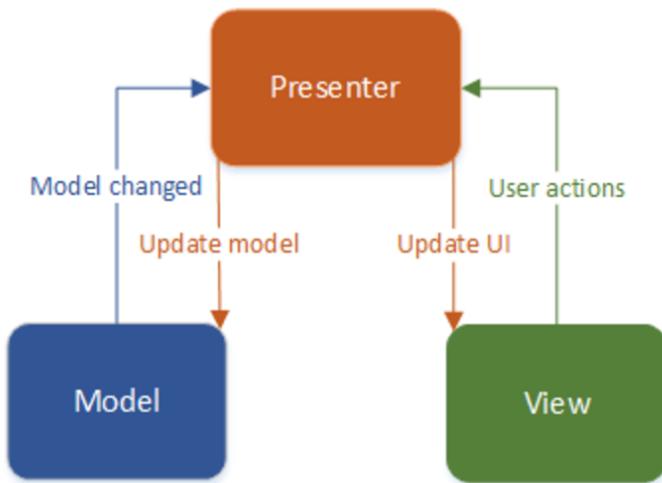
- **Model** jest zbiorem klas zajmujących się główną logiką dziedziny aplikacji. Model reprezentowany jest m.in przez klasy odpowiedzialne za reguły poruszania się robotów, obsługę wykonywania kolejnych kroków symulacji, algorytmy planowania trajektorii, kolejkowanie zaplanowanych ruchów, działania na wektorach (w metodzie pól potencjalnych) oraz reprezentację pól na mapie.
- **Widok** jest reprezentacją interfejsu użytkownika, który wyświetla dane i kieruje informacje o akcjach użytkownika (zdarzeniach) do prezentera. W tym wypadku instancje widoków generowane są automatycznie przez mechanizmy JavaFX na podstawie ich definicji zapisanych w plikach formatu FXML.
- **Prezenter** deleguje dane otrzymane z modelu do widoku, uprzednio przygotowując je do wyświetlenia. Jest w nim zawarta logika warstwy prezentacji.

Taka separacja warstw prezentacji od warstwy logiki umożliwia łatwe "odpięcie" logiki od interfejsu użytkownika i wykorzystanie jej do wykonania testów skuteczności metod planowania. Pozwala to wtedy na wykonanie "przyspieszonej" symulacji poprzez sekwencyjne uruchamianie kroków symulacji tak szybko, jak to możliwe, zamiast oczekiwania na wyzwolenie ich przez kolejne cykle zegara. Klasy modelu były projektowane właśnie z myślą o możliwości wykorzystania ich zarówno w wizualizacji w czasie rzeczywistym, jak i w przyspieszonej symulacji w celu przeprowadzenia testów.

5.4.2 Wielowątkowość

Aplikacja uruchamia planowanie trajektorii w osobnych wątkach w tle, aby nie wpływać na wątek interfejsu użytkownika i uzyskać możliwie płynne animacje. Wymaga to synchronizacji

MVP



Rysunek 5.5: Relacja pomiędzy poszczególnymi elementami wzorca architektonicznego Model-View-Presenter. Źródło: [21]

między wątkami, jednak język Java zapewnia stosunkowo łatwą obsługę wątków i synchronizacji sekcji krytycznych. Wątki obliczeń oraz odświeżania interfejsu użytkownika są powtarzane w stałych, zadanych cyklach zegara, które nie są ze sobą zsynchronizowane (nie czekają na siebie), co daje efekt symulacji w czasie rzeczywistym, niezależnie od stanu ukończenia obliczeń.

5.5 Ograniczenia

W aplikacji zostały wprowadzone pewne uproszczenia, które m.in. przyspieszają i ułatwiają proces obliczeniowy planowania trajektorii dla robotów.

1. Założono, że ruch ukośny robota o jedno pole (po przekątnej) trwa tyle samo, co ruch poziomy lub pionowy. Wprowadzono takie przybliżenie ze względu na możliwość ujednolicenia jednostki wymiaru czasu w tablicy rezerwacji pól przez agentów.
2. Nie uwzględniono czasu obrotu robota podczas zmiany kierunku jazdy. Założono, że czas ten jest zerowy, gdyż celem zaprojektowanego algorytmu było skupienie się na rozwiązaniu innego zagadnienia - problemu zakleszczeń w wąskich gardłach.

Rozdział 6

Wyniki testów

6.1 Obszerne testy aplikacji

TODO obszerne testy, porównanie metod: LRA*, WHCA* przy tych samych warunkach początkowych, porównanie czasu wykonania, porównanie tego samego algorytmu w zależności od parametru (np. okna czasowego); badanie skuteczności, długości tras, czasu wykonania, do przeprowadzenia testów wykorzystano bibliotekę jUnit, która co prawda służy do wykonywania testów jednostkowych sprawdzających poprawność pojedynczych komponentów aplikacji nie działa wycofywanie się obu robotów (zawsze jeden czeka) metodyka przeprowadzenia testów 3 typy środowisk testowych (rozmiar, roboty, screeny) histogram liczby kroków potrzebnych do rozwiązania potwierdzenie oczekiwania (poprawności) - nigdy nie było tak, żeby LRA był lepszy zwykłego A* nawet nie warto testować potential field - nawet nie warto testować, raczej jako ciekawostka, nie potrafi doprowadzić do celu nawet jednego robota przeprowadzenie testów jest trudne i wymaga losowania środowisk i warunków i wyciągania statystyki

6.2 Screeny

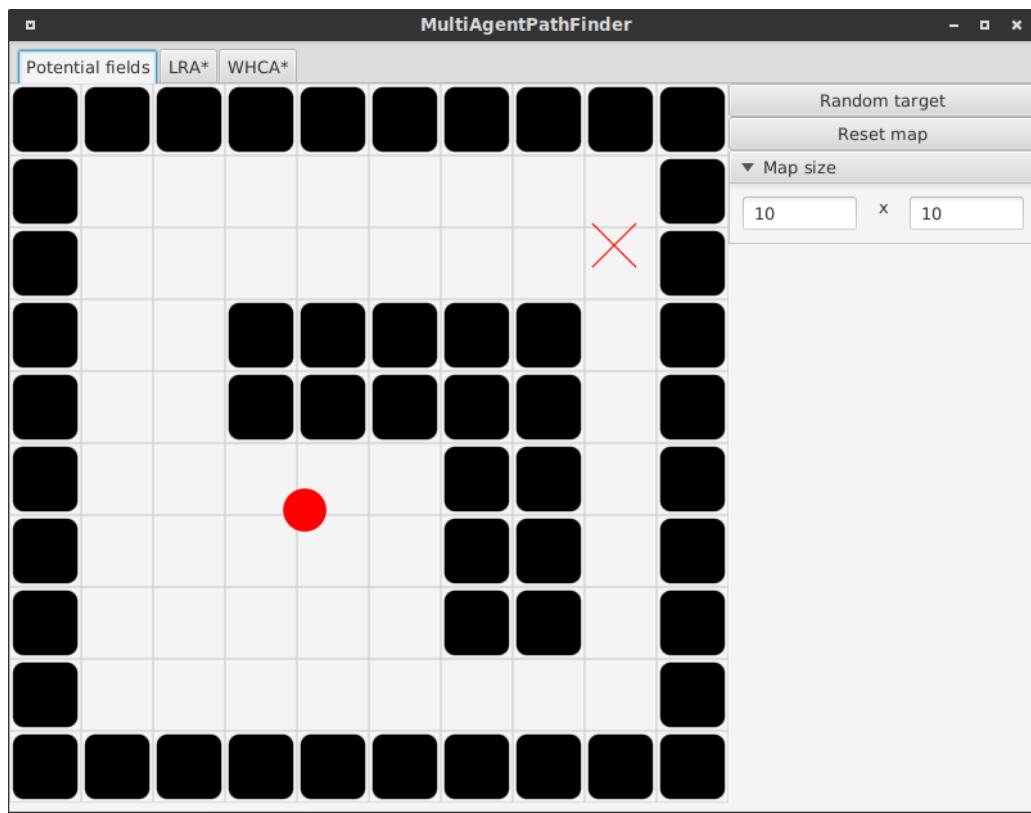
TODO screeny ciekawych przypadków

6.3 Środowiska

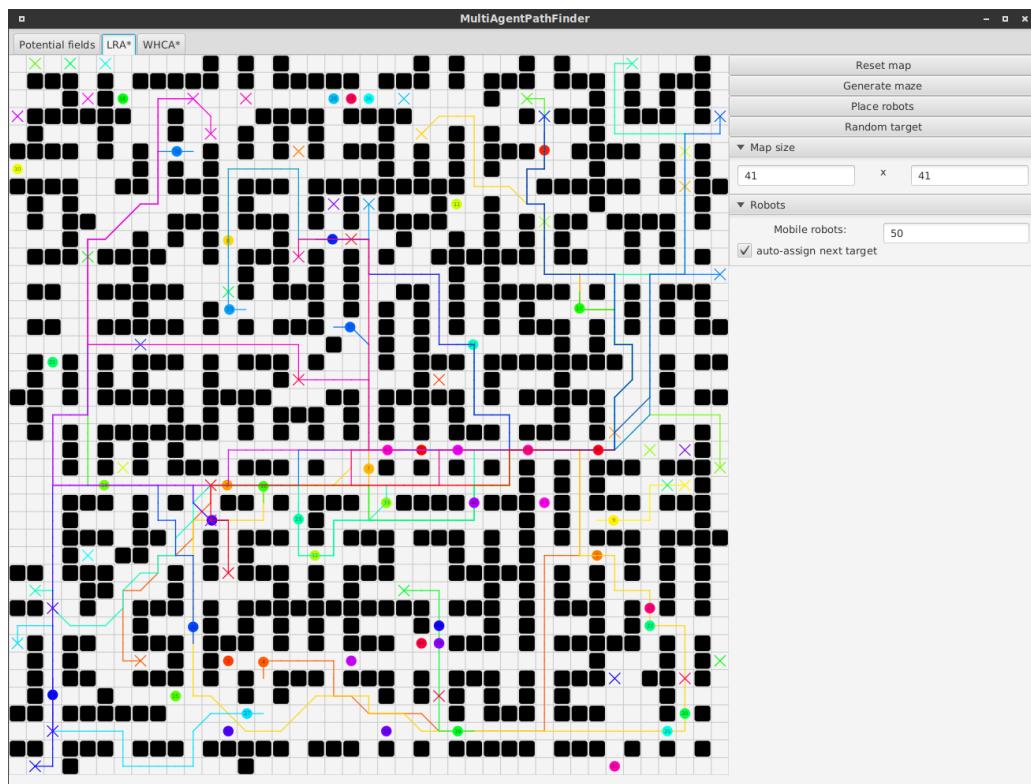
labirynt, otwarте, puzzle 15

filmiki można se obejrzeć na YT, link

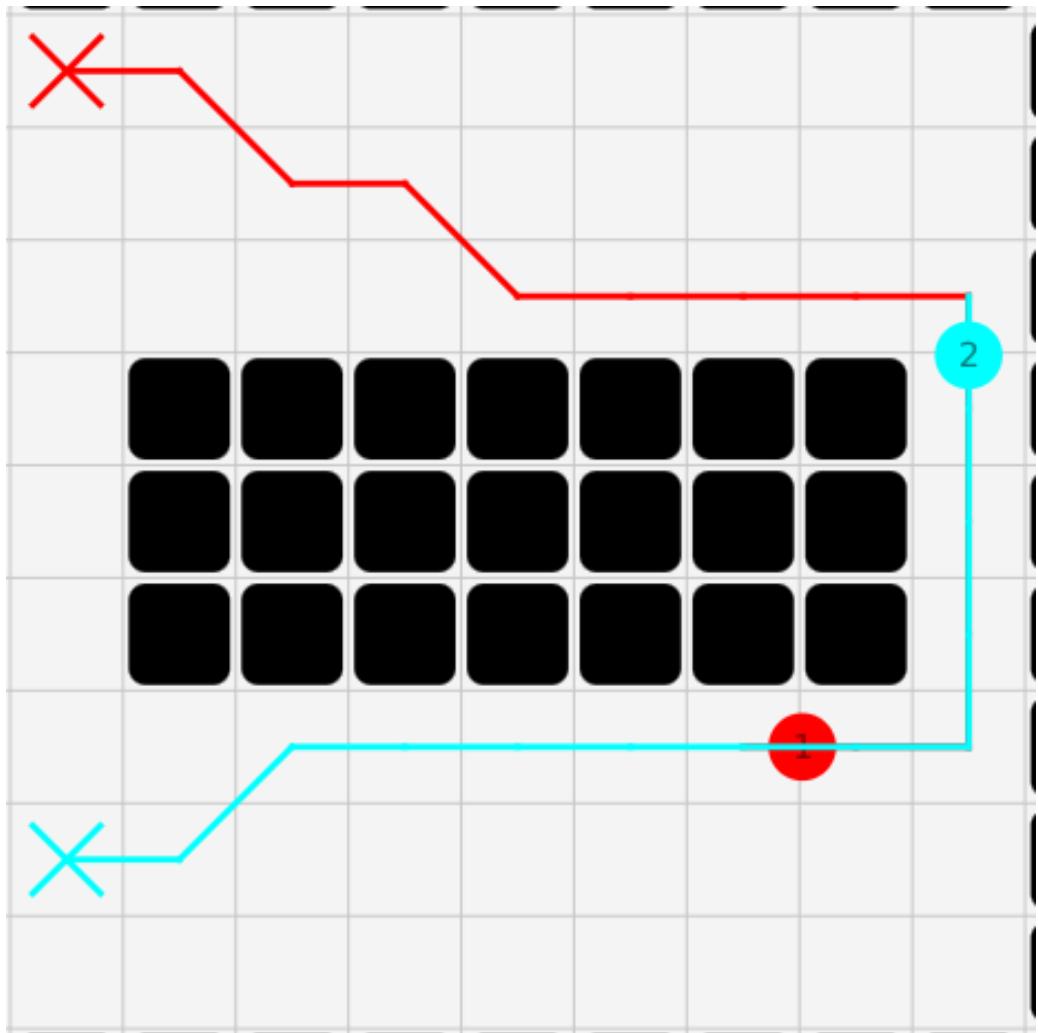
LRA: wyszło całkiem nieźle, testy ograniczać trzeba liczbą kroków symulacji, bo LRA może trwać wiecznie



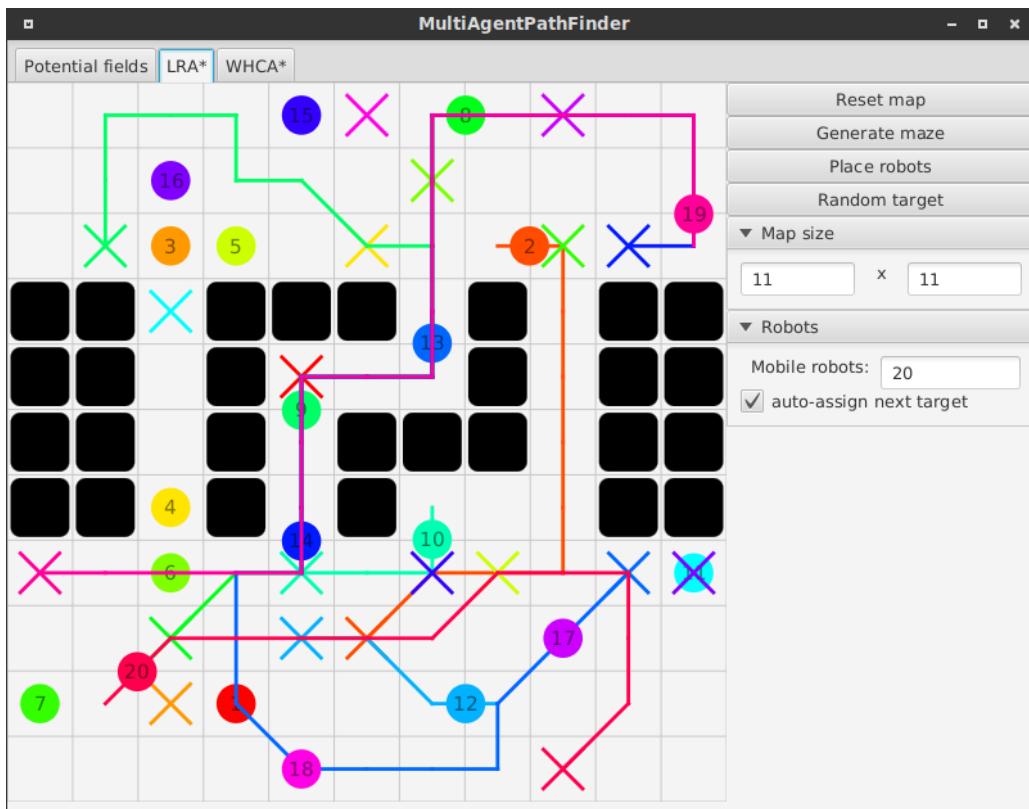
Rysunek 6.1: Robot uwięziony w studni potencjału. Zerowa siła wypadkowa nie pozwala mu dotrzeć do celu.



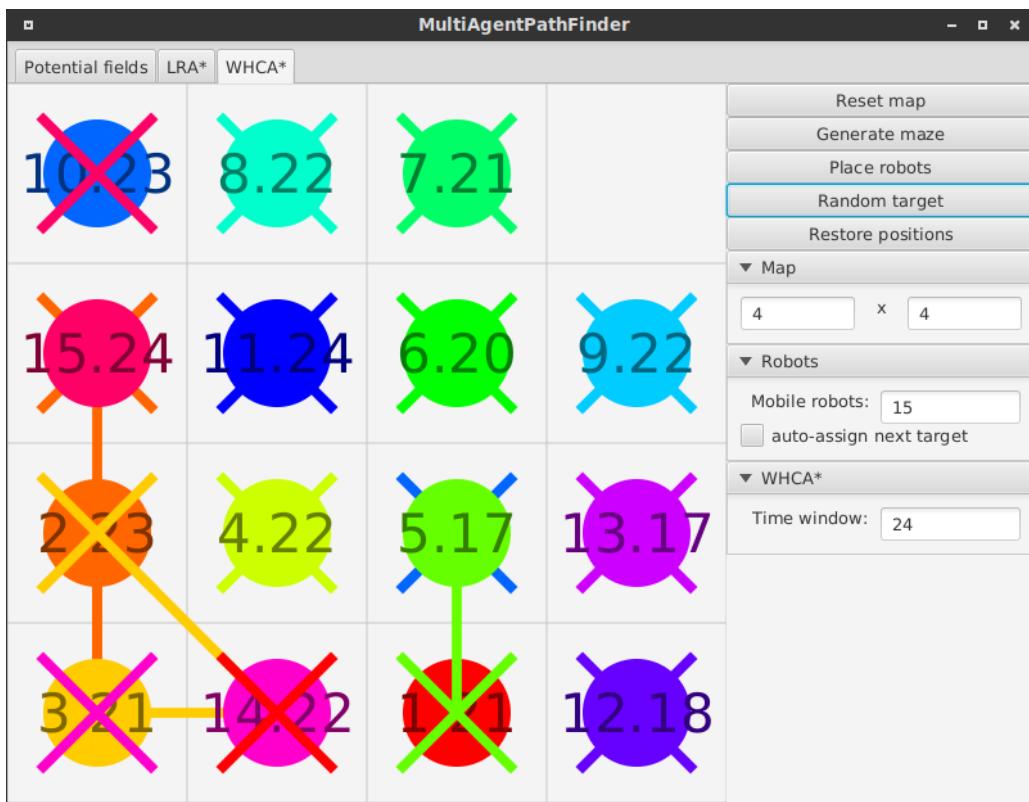
Rysunek 6.2: Metoda LRA*: duża mapa z dużą liczbą robotów



Rysunek 6.3: Metoda LRA*: 2 roboty w cyklu akcji



Rysunek 6.4: Metoda LRA*: dużo robotów, mała mapa



Rysunek 6.5: Metoda WHCA*: puzzle 15

6.4 Porównanie wyników

jak często zwykły A* to za mało - ile razy pojawia się chociaż jedna kolizja? porównanie WHCA* przy różnych oknach czasowych porównanie metod przydziału i zmiany priorytetów - jak zmienia się skuteczność po wprowadzeniu zmiany priorytetów porównanie LRA* z WHCA* porównanie CA* z WHCA* porównanie z potential fields porównanie WHCA z promocją priorytetów (własną, autorską) i bez

Rozdział 7

Podsumowanie

TODO Większość popularnych algorytmów wykorzystywanych do planowania tras dla wielu robotów mobilnych (agentów) opiera się o A*.

Kooperacyjne planowanie tras jest ogólną techniką koordynacji dróg wielu jednostek. Znajduje zastosowanie, gdzie wiele jednostek może komunikować się ze sobą, przekazując informację o ich ścieżkach. Poprzez planowanie wprzód w czasie, jak również i w przestrzeni, jednostki potrafią schodzić sobie z drogi nawzajem w celu uniknięcia kolizji. Metody kooperacyjnego planowania są bardziej skuteczne i znajdują trasy wyższej jakości niż te uzyskane przez A* z metodą *Local Repair*.

Wiele z udoskonaleń przestrzennego algorytmu A* może być również zaadaptowane do czasoprzestrzennego A*. Ponadto, wprowadzenie wymiaru czasu otwiera nowe możliwości do rozwoju algorytmów znajdowania dróg.

Najbardziej obiecującym pod względem skuteczności algorytmem wydaje się być metoda WHCA*.

Aby wydajnie prowadzić obliczenia, zakłada się, że każdy ruch robota trwa tyle samo. Wprowadza to upraszczające, błędne założenie, że ruch robota na pole w kierunku poziomym lub pionowym trwa tyle samo, co na ukos.

W wielu przypadkach metody do planowania bezkolizyjnych tras w systemach wieloagentowych mogą być wykorzystywane zamiennie zarówno do wyznaczania trajektorii robotów mobilnych, jak i w grach komputerowych, np. strategiach czasu rzeczywistego do planowania tras wielu jednostek.

Zaprezentowane algorytmy mogą znaleźć zastosowanie również w środowiskach z ciągłą przestrzenią oraz w dynamicznych środowiskach, w których to ścieżki muszą być przeliczane po wykryciu zmiany na mapie.

Zaproponowano własną metodę dynamicznego przydziału priorytetów jako rozszerzenie metody WHCA*, która znacząco poprawia skuteczność, co wykazały obszerne testy. Należy zazna-

czyć, że opisywane metody aplikują się do środowisk specyficznego typu, w innych wypadkach możliwe, że sprawdzą się dużo prostsze metody. Spopularyzowane metody, żeby mogła być używana w robotyce i grach rts.

7.1 Dyskusja wyników

wolno działa WHCA* przy dużych mapach / oknach czasu / robotach. Dałoby się zoptymalizować (RRA) mój WHCA* zajebiście działa nawet przy rozwiązywaniu dużych deadlocków, problem z puzzle 15

Bibliografia

- [1] Bennewitz M.; Burgard W.; Thrun S. *Optimizing Schedules for Prioritized Path Planning of Multi-Robot Systems*. 2001.
- [2] Cap M.; Novak P.; Vokrinek J.; Pechoucek M. *Asynchronous Decentralized Algorithm for Space-Time Cooperative Pathfinding*. Workshop Proceedings of the European Conference on Artificial Intelligence (ECAI 2012), 2012.
- [3] Duc L. M.; Sidhu A. S.; Chaudhari N. S. *Hierarchical Pathfinding and AI-Based Learning Approach in Strategy Game Design*. International Journal of Computer Games Technology, 2008.
- [4] Geramifard A.; Chubak P. *Efficient Cooperative Path-Planning*. Computing Science Department, University of Alberta, 2005.
- [5] Hart P. E.; Nilsson N. J.; Raphael B. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Transactions on Systems Science and Cybernetics SSC4, 1968.
- [6] Koenig S.; Likhachev M. *D* Lite*. Proceedings of the AAAI Conference of Artificial Intelligence, 2002.
- [7] Latombe J. *Robot Motion Planning*. Boston, MA: Kluwer Academic, 1991.
- [8] Mówinski K.; Roszkowska E. *Sterowanie hybrydowe ruchem robotów mobilnych w systemach wielorobotycznych*. Postępy Robotyki, 2016.
- [9] Przybylski M.; Putz B. *D* Extra Lite: A Dynamic A* With Search-Tree Cutting and Frontier-Gap Repairing*. International Journal of Applied Mathematics and Computer Science, 2017.
- [10] Siemiątkowska B. *Uniwersalna metoda modelowania zachowań robota mobilnego wykorzystująca architekturę uogólnionych sieci komórkowych*. Oficyna Wydawnicza Politechniki Warszawskiej, 2009.

- [11] Silver D. *Cooperative Pathfinding*. Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference, 2005.
- [12] Standley T.; Korf R. *Complete Algorithms for Cooperative Pathfinding Problems*. Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, 2011.
- [13] Toresson A. *Real-time Cooperative Pathfinding*. 2010.
- [14] Zhu Q.; Yan Y.; Xing Z. *Robot Path Planning Based on Artificial Potential Field Approach with Simulated Annealing*. Intelligent Systems Design and Applications, 2006.
- [15] A* pathfinding for beginners. <http://homepages.abdn.ac.uk/f.guerin/pages/teaching/CS1013/practicals/aStarTutorial.htm>. Dostęp: 2018-01-02.
- [16] Amazon warehouse demand devours robots and workers. https://www.roboticsbusinessreview.com/supply-chain/amazon_warehouse_demand_devours_robots_and_workers. Dostęp: 2018-01-05.
- [17] Dokumentacja API Java Platform, Standard Edition 8. <https://docs.oracle.com/javase/8/docs/api/>. Dostęp: 2018-01-30.
- [18] Dokumentacja API JavaFX 8. <https://docs.oracle.com/javase/8/javafx/api/toc.htm>. Dostęp: 2018-01-30.
- [19] Dokumentacja biblioteki SpringBoot JavaFx 8.0 Support. <https://springboot-javafx-support.readthedocs.io/en/latest/>. Dostęp: 2018-01-30.
- [20] Dokumentacja Spring Boot. <https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/>. Dostęp: 2018-01-30.
- [21] Porównanie wzorców architektonicznych MVC i MVP. <https://www.techyourchance.com/mvp-mvc-android-1/>. Dostęp: 2018-01-05.
- [22] Choset H. Robotic motion planning: Potential functions. https://www.cs.cmu.edu/~motionplanning/lecture/Chap4-Potential-Field_howie.pdf. Dostęp: 2018-01-02.
- [23] Roboty TUG i HOMER firmy Aethon. <http://www.aethon.com/tug/tughealthcare/>. Dostęp: 2018-01-02.
- [24] Searching using A*. <http://web.mit.edu/eranki/www/tutorials/search/>. Dostęp: 2018-01-02.

- [25] Wsparcie dla JavaFX od Oracle. <http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6>. Dostęp: 2018-01-02.

Wykaz skrótów

| | |
|-------|--------------------------------------|
| API | Application Programming Interface |
| CA* | Cooperative A* |
| HCA* | Hierarchical Cooperative A* |
| IoC | Inversion of Control |
| JVM | Java Virtual Machine |
| LRA* | Local Repair A* |
| MAS | Multi-Agent System |
| MVP | Model-View-Presenter |
| RRA* | Reverse Resumable A* |
| RTS | Real Time Strategy |
| TDD | Test-driven Development |
| WHCA* | Windowed Hierarchical Cooperative A* |

Spis rysunków

| | | |
|-----|---|----|
| 1.1 | Przykładowe środowisko z dużą liczbą przeszkode (czarne kwadraty) i rozmieszczonymi robotami (kolorowe koła). Źródło: własna implementacja oprogramowania symulacyjnego | 16 |
| 1.2 | Roboty Kiva pracujące w magazynie firmy Amazon. Źródło: [16] | 17 |
| 1.3 | Popularny problem zakleszczania się jednostek w wąskich gardłach występujący w grach typu RTS. Źródło: gra komputerowa Age of Empires II Forgotten Empires | 18 |
| 2.1 | Zasada działania metody pól potencjałowych. Dodatni ładunek q_{start} reprezentuje robota. Przyciągany jest w stronę ujemnego ładunku celu q_{goal} , zaś odpychany jest od dodatnio naładowanej przeszkode. Źródło: [22] | 21 |
| 2.2 | Ciągła przestrzeń mapy zdyskretyzowana do siatki pól. Źródło: edytor map z gry Warcraft III. | 21 |
| 2.3 | Sytuacja, w której żadne rozwiązanie nie zostanie znalezione, stosując planowanie uwzględniające priorytety, jeśli robot 1 ma wyższy priorytet niż robot 2. Źródło: [1] | 22 |
| 2.4 | a) Niezależne planowanie optymalnych tras dla 2 robotów; b) suboptymalne rozwiązanie, gdy robot 1 ma wyższy priorytet; c) rozwiązanie, gdy robot 2 ma wyższy priorytet. Źródło: [1] | 23 |
| 3.1 | Ilustracja wyznaczania działania przez A*. Każdy odwiedzony węzeł wskazuje na swojego rodzica, co umożliwia późniejszą rekonstrukcję drogi. Źródło: [15] . . | 27 |
| 3.2 | Dwie jednostki kooperacyjnie poszukujące tras. (A) Pierwsza jednostka znajduje ścieżkę i zaznacza ją w tablicy rezerwacji. (B) Druga jednostka znajduje ścieżkę, uwzględniając istniejące rezerwacje pól, również zaznaczając ją w tablicy rezerwacji. Źródło: [11] | 31 |
| 3.3 | Tablica rezerwacji jest współdzielona między wszystkimi agentami. Jej rozmiar powinien być odpowiednio dopasowany do agentów o różnych prędkościach. Źródło: [11] | 31 |

| | | |
|-----|---|----|
| 4.1 | Kolejne etapy generowania labiryntu: (a) Zaznaczenie co drugiego pola jako wolne i wybór ziarna rozrostu labiryntu. (b) Wylosowanie i łączenie kolejnego wierzchołka poprzez "wyburzanie" przeszkód na drodze (c) Wynikowa mapa pochodząca z generatora | 37 |
| 4.2 | Przykładowy labirynt rozmiaru 75×75 pochodzący z zaprojektowanego generatora map | 39 |
| 4.3 | Przykładowa ścieżka wyznaczona przez zaimplementowany w aplikacji algorytm A*. Kolorowe koło reprezentuje robota, kolorowe koło - robota, linia łamana - wyznaczoną ścieżkę, czarne kwadraty - przeszkody. | 42 |
| 4.4 | Przykład powtarzających się cykli planowanych akcji uzyskanych przez metodę LRA*. Dwa roboty "zderzą się" w wąskim przejściu po prawej stronie i wyznaczają trajektorie przechodzące przez przejście po lewej stronie, gdzie ponownie dojdzie do kolizji. | 43 |
| 5.1 | Zrzut ekranu aplikacji w trakcie wizualizacji metody pól potencjałowych. | 47 |
| 5.2 | Zrzut ekranu aplikacji w trakcie wizualizacji metody Local-Repair A*. | 48 |
| 5.3 | Zrzut ekranu aplikacji w trakcie wizualizacji metody Windowed Hierarchical Cooperative A*. | 49 |
| 5.4 | Zrzut ekranu środowiska deweloperskiego IntelliJ IDEA Ultimate | 53 |
| 5.5 | Relacja pomiędzy poszczególnymi elementami wzorca architektonicznego Model-View-Presenter. Źródło: [21] | 55 |
| 6.1 | Robot uwięziony w studni potencjału. Zerowa siła wypadkowa nie pozwala mu dotrzeć do celu. | 58 |
| 6.2 | Metoda LRA*: duża mapa z dużą liczbą robotów | 58 |
| 6.3 | Metoda LRA*: 2 roboty w cyklu akcji | 59 |
| 6.4 | Metoda LRA*: dużo robotów, mała mapa | 60 |
| 6.5 | Metoda WHCA*: puzzle 15 | 60 |

Spis tabel

Spis załączników

Na załączonej do pracy płycie CD znajdują się następujące treści:

- Niniejsza praca w formacie PDF – plik
Praca/Praca_Magisterska_Ireneusz_Szulc.pdf.
- Archiwum zawierające kody źródłowe programu symulacyjnego – plik
Kody_źródłowe/aplikacja-symulacja.zip