AKADEMIA GÓRNICZO-HUTNICZA W KRAKOWIE

Wydział Informatyki, Elektroniki i Telekomunikacji

Przedmiot: Programowanie systemów agentowych i wielorobotowych

Temat projektu: Symulacja organizacji agentowych w środowisku mobilnej chmury

Autorzy: Daniel Bryła, Krzysztof Nawrot

Spis treści

1. Wstęp 3
2. Opis architektury 3
3. Zapoznanie z biblioteką Madkit 5
   3.1. Architektura 5
4. Specyfikacja realizowanych problemów 6
   4.0 Obliczanie wartości liczby PI 6
   4.1. Pomiar zatłoczenia w komunikacji miejskiej 6
   4.2. Monitorowanie aktywności radiowozów policji 10
   4.3. Poszukiwanie zaginionych 11
5. Wyniki testów 12
   5.0. Obliczanie wartości liczby PI 12
   5.1. Pomiar zatłoczenia w komunikacji miejskiej 13
   5.2. Monitorowanie aktywności radiowozów policji 14
   5.3. Poszukiwanie zaginionych 15
6. Podsumowanie 16
7. Bibliografia 16

1. Wstęp

Projekt polegał na wykorzystaniu konkretnych organizacji agentowych dla różnorakich obliczalnych problemów, dających się obliczyć w sposób rozproszony. Miało to symulować działanie wykwalifikowanych jednostek w środowisku mobilnej chmury.

Każdy realizowany przez nas problem miał przyporządkowaną organizację agentową, którą wspierała jego rozwiązywanie. Początkowo wybraliśmy 3 organizacje agentowe spośród dostarczonej nam literatury, które naszym zdaniem wydawały się rozłączne w zastosowaniach i dlatego też ciekawe. Następnie postanowiliśmy wymyślić parę przykładowych problemów, które mogłyby być dopasowane do wybranych organizacji. Ostatecznie, wykorzystaliśmy dwa projekty z tego zbioru, trzeci problem został ustalony przy pracach implementacyjnych nad ostatnią organizacją agentową.

Jako środowisko agentowe wykorzystaliśmy bibliotekę MadKit (odrzuciliśmy tym samym inne aletrnatywy; według nas rozwiązanie to przedstawiało najbardziej przejrzysty interfejs i nie miało zbyt dużych narzutów ponad to, co potrzebowaliśmy). Parametrem wejściowym tego symulatora była nazwa organizacji agentowej oraz problem, implementujący odpowiedni interfejs.

2. Opis architektury

Mimo wykorzystania zewnętrznej biblioteki do zarządzania agentami i ich cyklem życia, w naszym projekcie zaimplementowaliśmy szereg dodatkowych funkcjonalności, które wspomagały dalsze prace. Omówimy teraz najważniejsze z nich.

Każdy implementowany przez nas problem rozszerzał interfejs *Problem:*

```
public interface Problem {
   ArrayList<Step> getListOfSteps();

   void announceResult(Object finalResult);
}
```

Jak widać, wykorzystaliśmy tutaj także inny interfejs *Step*, definiujący kroki do wykonania w czasie trwania symulacji:

```
public interface Step {
   /**
    * @param if step depends on output from previous one, here it should be given in JSON
    * @return output for next step or result in JSON
    */
   String doIt(String data);
}
```

W trakcie działania aplikacji każdy z agentów otrzymuje niezbędne dla wykonania jego pracy dane w postaci ciągu napisów *data.* Na tych informacjach później operuje, celem zwrócenia odpowiedzi w formacie JSON.

Organizacje agentów, z kolei, implementowały interfejs *Organization*:

```
public abstract class Organization {

   public static Organization getOrganizationPerType(OrganizationType type) {
       (...)
   }

   abstract public void communicate(OrganizedAgent agent);

   abstract public LinkedList<OrganizedAgent> createAndAssignProblem(Problem problem);

}
```

Ponadto, zaimplementowaliśmy naszą nakładkę na bazową funkcjonalność agenta dostarczaną przez bibliotekę MadKit:

```
public class OrganizedAgent extends Agent {
   private static final AtomicInteger agentsCounter = new AtomicInteger(0);

   public OrganizedAgent(OrganizationType organization, String group, String role) {
       (...)
       id = agentsCounter.getAndIncrement();
   }

   public OrganizedAgent(OrganizationType organization, String role) {
       this(organization, Groups.NONE.toString(), role);
   }

   @Override
```

```
    protected void activate() {
        createGroupIfAbsent(organization.toString(), group);
        requestRole(organization.toString(), group, role);
    }

    protected void live() {
        Organization.getOrganizationPerType(organization).communicate(this);
    }
}
```

Integruje ona agenta z resztą naszej dodanej funkcjonalności (role i hierarchie).

3. Zapoznanie z biblioteką Madkit

MadKit to platform multiagentowa, napisana w języku Java. Została zaprojektowana by w łatwy sposób budować rozporoszone aplikacje i symulacje, używające multiagentowego paradygmatu. Obejmuje takie funkcjonalności jak:

- Tworzenie agentów i zarządzanie ich cyklem życia
- Zoorganizowana infrastruktura dla komunikacji pomiędzy agentami
- Wysoka heterogeniczność pod względem architektury agentów – nie ma predefiniowanego modelu agenta

MadKit jest środowiskiem zbudowanym w oparciu o model AGR (Agent/Group/Role) – agenci odgrywają konrketne role w grupach i dlatego mogą tworzyć pewnego rodzaju społeczeństwa. Jest to duża zaleta w porównaniu do innych, rozważanych przez nas podejść, tym bardziej biorąc pod uwagę tematykę realizowanego przez nas projektu (wykorzystanie organizacji agentowych).

3.1. Architektura

Jedną z najistotniejszych encji z tego pakietu jest klasa *Agent*, rozszerzająca *AbstractAgent*. Była to dla nas bazowa klasa, którą później rozszerzaliśmy o dodatkowy interfejs. Klasa dostarcza funkcjonalności:

- Zarządzanie cyklem życia agenta, logowanie, nazewnictwo
- Uruchamianie agenta oraz uśmiercanie go
- Tworzenie społeczności i zarządzanie nimi (dołączenie, przydzielanie roli)
- Wysyłanie komunikatów (między agentami)

W naszym projekcie wykorzystywaliśmy także prostą klasę *Message*, rozszerzając ją o przydatne dla nas pola. W bazowej wersji ta lekka klasa jest serializowalna oraz definiuje nadawcę i odbiorcę.

Utworzenie środowisko agentowego nie byłoby możliwe dzięki bardzo istotnej klasie *MadKit*. Za pomocą konstruktora możliwe jest zestawienie środowiska agentowego z predefiniowanymi opcjami. Środowisko automatycznie kończy pracę, jeśli wszyscy agenci żyjący w tym środowisku zakończyli pracę. W naszym przypadku rozpoczynamy pracę z opcją `"--launchAgents org.agents.simulator.Application"`. Jedyny argument wywołania jest opcją pracy symulatora (spośród możliwych takich jak definiowanie plików konfiguracyjnych, katalogu logowania itd.), startującą agenta(ów) zdefiniowanych w klasach, podanych jako kolejny ciąg znaków do opcji. W naszym przypadku *Application* jest pustym agentem, który startuje całą architekturę.

4. Specyfikacja realizowanych problemów

Na początku należy wspomnieć, iż dla przetestowania poprawności konfiguracji naszego środowiska zrealizowaliśmy przykładowy problem obliczania liczby PI w popularnej architekturze master-slave. Nie był on jednak przedmiotem naszych szczególnych badań, jednak z czystej ciekawości uwzględniliśmy go także w testach, a zatem wspomnimy o nim także tutaj.

4.0 Obliczanie wartości liczby PI

Przybliżenie wartości liczby Pi można uzyskać generując stochastycznie punkty wewnątrz kwadratu o boku równym 1, sprawdzając jednocześnie, ile z nich znajduje się wewnątrz koła o promieniu 1/2. Wystarczy zatem wygenerować równomiernie rozłożony zestaw punktów wewnątrz kwadratu, policzyć ile z nich leży wewnątrz koła i ich iloraz pomnożeć przez 4. Otrzymana wartość będzie przybliżeniem liczby Pi, tym lepszym im więcej punktów zostanie wygenerowanych. Metoda ta nosi nazwę Monte-Carlo i posłużyła nam do rozwiązania problemu.

4.1. Pomiar zatłoczenia w komunikacji miejskiej

Problem wiąże się z pomiarem zatłoczenia w poszczególnych autobusach na różnych liniach. Efektem końcowym jest wygenerowanie przykładowego raportu o zatłoczeniu w danej chwili.

Organizacja agentów

W tym przypadku wykorzystano hierarchiczną organizację agentów. W naszym przypadku mamy głównego agenta MAIN_AGENT, który odpowiedzialny jest za generowanie raportu, ADMINISTRATOR_AGENT, odpowiedzialnego za oczekiwanie na informacje o zatłoczeniu na konkretnej linii oraz BASIC_AGENT, czyli najniżej pod względem hierarchii położonego agenta, działającego w autobusie. Agenci rozszerzają bazową klasę *OrganizedAgent* o klasy *LineDelegateAgent* i *BusDelegateAgent*, odpowiednio dla ADMINISTRATOR_AGENT i BASIC_AGENT. Agent główny wykorzystuje podstawową funkcjonalność klasy *OrganizedAgent*.

```
public class LineDelegateAgent extends OrganizedAgent {
   private final int lineNum;
   private final String workersGroup;
   private final String role;

   public LineDelegateAgent(OrganizationType organization, String workersGroup, String role,
int lineNum);
   public String getWorkersGroup();
   public int getLineNum();
   @Override
   public void activate() {
       super.activate();
       if (!createGroupIfAbsent(organization.toString(), workersGroup)) {
           requestRole(organization.toString(), workersGroup, role);
       }
   }
}

public class BusDelegateAgent extends OrganizedAgent {
   private final int myBusId;
   private final String group;
   private final String role;

   public BusDelegateAgent(OrganizationType organization, String group, String role, int
```

```
myBusId);
    public int getMyBusId();
    @Override
    public void activate() {
        super.activate();
        if (!createGroupIfAbsent(organization.toString(), group)) {
            requestRole(organization.toString(), group, role);
        }
    }
}
```

Kontrola życia agentów oraz wykonywane przez nich kroki (operacje) znajdują się w klasie *HierarchicalOrganization*. Działa ona w następujący sposób:

```
public class HierarchicalOrganization extends Organization {

    @Override
    public void communicate(OrganizedAgent agent) {
        if (agent.getRole().equals(MAIN_AGENT)) {
            // send requests to agents, collect responses and print results
        }
        else if (agent.getRole().contains(ADMINISTRATOR_AGENT)) {
            while (true) {
                // get message...
                switch (message.type) {
                    case WORK:
                        // send request to BASIC_AGENT's and gather result
                        // perform step and send result
                    case DIE:
                        // finish working
                }
            }
        }
```

```
        else {
            while (true) {
                // get message...
                switch (message.type) {
                    case WORK:
                        // perform step and send result
                    case DIE:
                        // finish working
                }
            }
        }
    }
}
```

Definicja problemu

Problem zdefiniowany jest w postaci kroków, które należy wykonać, aby uzyskać końcowy rezultat. Oczywiście, biorąc pod uwagę organizację agentów, każdy agent wykonuje inny krok. Poniżej kod klasy ***PublicTransport*** odpowiedzialnej za realizację naszego zagadnienia:

```
public class PublicTransport implements Problem {
    //...

    public PublicTransport() {
        STEPS.add(data -> {
            //count number of passengers in a bus and return it
        });
        STEPS.add(data -> {
            //gather info about buses on line and return it
        });
        STEPS.add(data -> {
            //gather all info from bus lines and prepare overall report
        });
    }

    //...
}
```

4.2. Monitorowanie aktywności radiowozów policji

Problem wiąże się z pomiarem aktywności radiowozów policji w danej dzielnicy Krakowa. Efektem końcowym jest możliwość odpytania agenta o fakt, czy na jego ulicy znajduje się radiowóz.

Organizacja agentów

W tym przypadku wykorzystano organizację agentów - koalicję. Wedle definicji, każdy agent w koalicji wykonuje tę samą czynność. Agenci zrzeszeni są w grupy (koalicje). Nie ma tu żadnej struktury, każdy agent jest równy innemu i każdy generuje końcową odpowiedź w ramach swojej pracy. Agenci wykorzystują jedynie podstawową funkcjonalność klasy *OrganizedAgent*. Kontrola życia agentów oraz wykonywane przez nich krok znajdują się w klasie *CoalitionOrganization*. Jej struktura jest analogiczna do hierarchii, więc nie będziemy jej przedstawiać, jako, że dostępna jest w repozytorium.

Definicja problemu

Problem zdefiniowany jest w postaci jednego kroku, który wykonuje każdy agent, aby uzyskać końcowy rezultat. Poniżej kod klasy *PatrolledArea* odpowiedzialnej za realizację naszego zagadnienia:

```
public class PatrolledArea implements Problem {
   //...

   public PatrolledArea() {
       STEPS.add(data -> {
           //get unhandled street...
       });
       STEPS.add(data -> {
           //... and check for patrol
       });
       //...
   }
}
```

4.3. Poszukiwanie zaginionych

Problem związany jest z poszukiwaniem zaginionych (w górach/mieście/po katastrofie). Poszukiwanie odbywa się w 3 drużynach poszukiwawczych o określonych rozmiarach. Drużyny komunikują się ze sobą, jeśli którejś udało się znaleźć jakiegoś zaginionego.

Organizacja agentów

Wykorzystana organizacja to federacja. Drużyny mają swoich dwódców, którzy komunikują się z innymi drużynami, natomiast zwykli ratownicy nie mają takich możliwości. Jej struktura jest analogiczna do poprzednich, więc nie będziemy jej przedstawiać, jako, że dostępna jest w repozytorium.

Definicja problemu

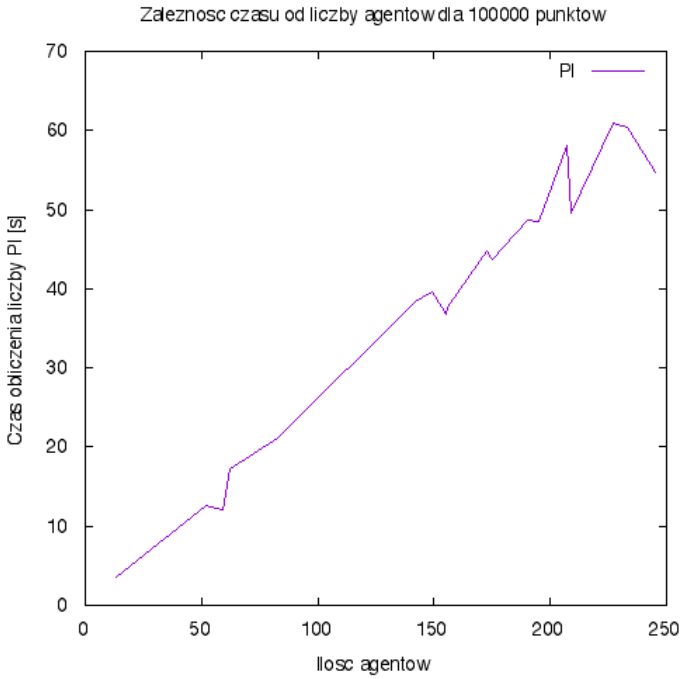
Problem zdefiniowany jest w postaci dwóch kroków. Pierwszy z nich który wykonuje każdy agent ratownik, drugi jest końcową akcją wykonywaną przez dowódcę drużyny. Poniżej kod klasy *RescueExpedition* odpowiedzialnej za realizację naszego zagadnienia:

```
public class RescueExpedition implements Problem {
   //...

   public RescueExpedition() {
       STEPS.add(data -> {
           //try to find lost people in my area
       });
       STEPS.add(data -> {
           //prepare info for team report
       });
       //...
   }

   //...
}
```

5. Wyniki testów

Dla wszystkich powyższych problemów przeprowadziliśmy różne testy, starając się zmieniać najistotniejsze dla symulacji w danym przypadku parametry. Mierzyliśmy czas wykonania oraz różne wydajności oraz szukalismy regularności. Prezentujemy nasze obserwacje.

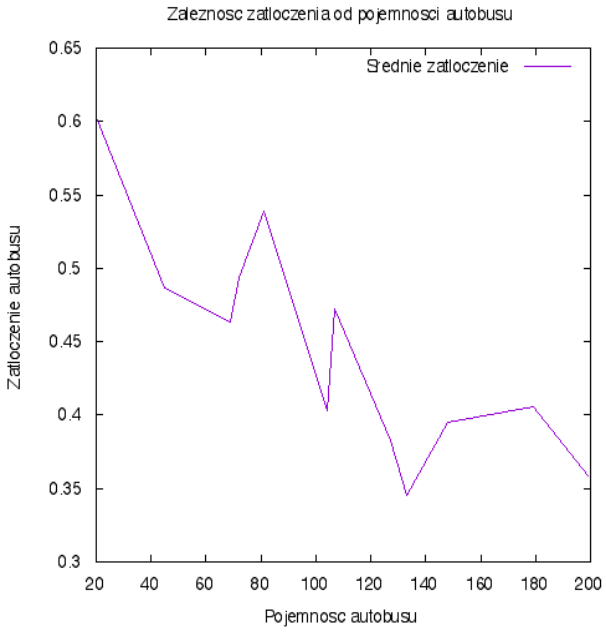
5.0. Obliczanie wartości liczby PI

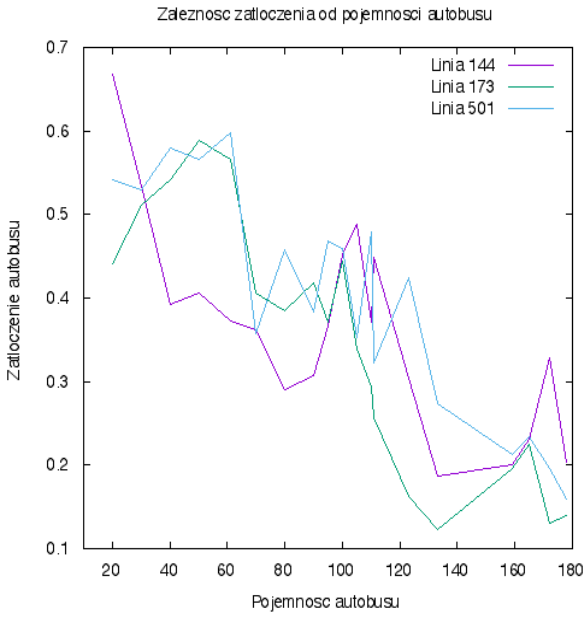


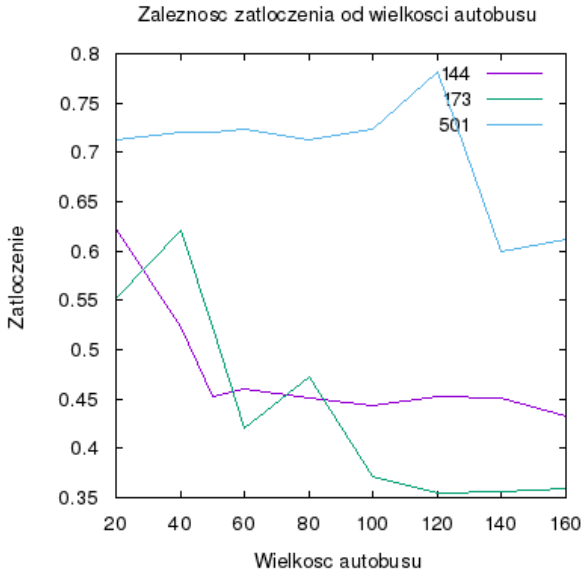
Wnioski:

- proporcjonalny wzrost czasu przy zwiększeniu wielkości problemu
- charakterystyczne wyniki dla problemu równoległego obliczania wartości liczby PI

5.1. Pomiar zatłoczenia w komunikacji miejskiej



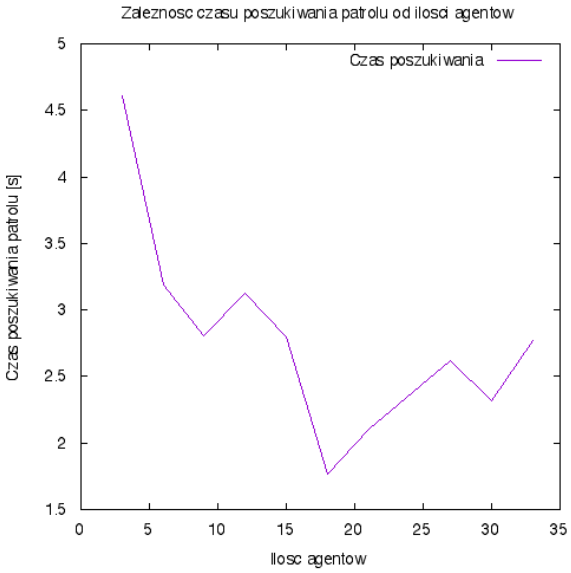


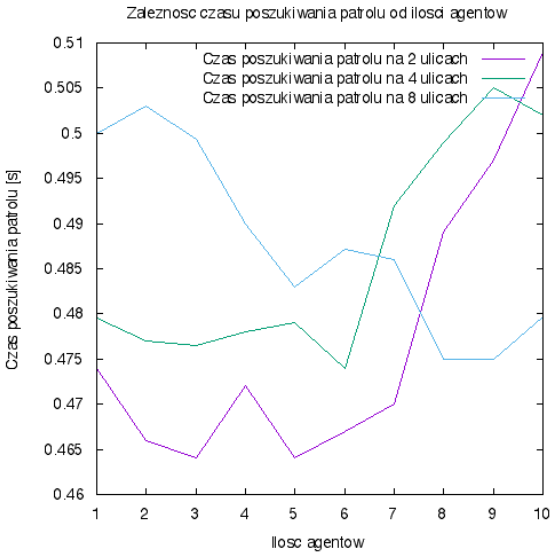


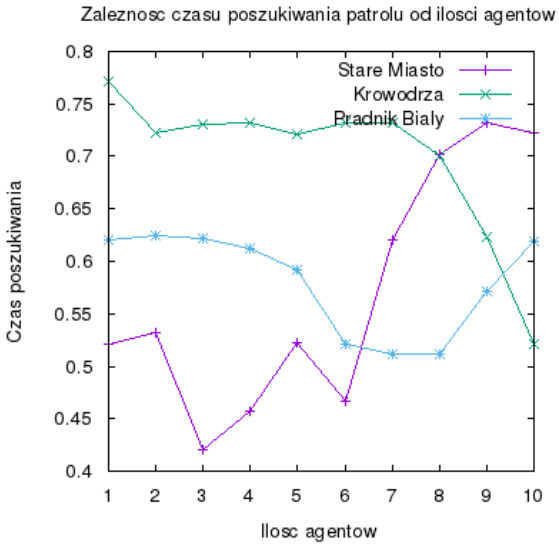
Wnioski:

- stałe monitorowanie zatłoczenia w autobusie przy pomocy urządzeń mobilnych może doprowadzić do wybrania optymalnej wielkości autobusu i zwiększenia ogólnego zadowolenia pasażerów
- organizacja pozwalająca pobieranie danych dla osobnych zbiorów

5.2. Monitorowanie aktywności radiowozów policji



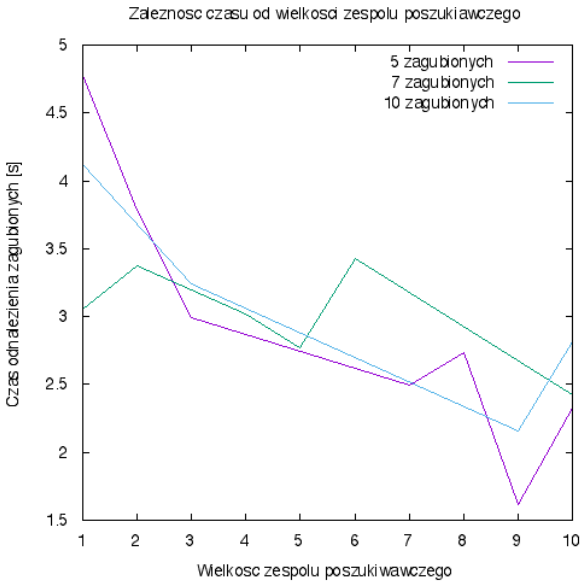


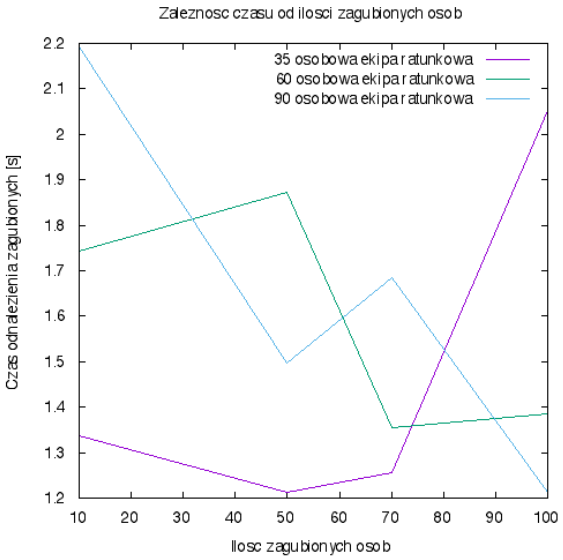


Wnioski:

- problem dający się zrównoleglić
- większa ilość agentów pozwala odnaleźć rozwiązanie szybciej
- optymalny dobór ilości agentów ma znaczenie
- ilość agentów powinna być proporcjonalna do ilości przeszukiwanych ulic

5.3. Poszukiwanie zaginionych





Wnioski:

- tendencja pokazująca, że czas poszukiwania maleje wraz ze zwiększeniem ekipy poszukiwawczej
- architektura mobilnej chmury obliczeniowej pokazuje, że poszukiwania osób przy stałej komunikacji wpływa korzystnie na szybkość znalezienia zagubionych
- czas jest najoptymalniejszy przy podobnej wielkości ekipy ratunkowej jak ilości zagubionych osób

6. Podsumowanie

Z pewnością zrealizowane przez nas problem można było by bardziej “urealnić”, wprowadzając nieco bardziej zaawansowną logikę niż czystę tak/nie występowania pewnych zjawisk, z pewnością można by także wzbogacić funkcjonalności agenta w każdym problemie. Tego nie udało nam się zrealizować. Uważamy jednak, że ze sporym powodzeniem udało nam się trafnie zastosować organizacje agentowe do zadanych problemów, co zresztą pokazują wyniki ekspreymentów.

7. Bibliografia

- Bryan Horling and Victor Lesser, *A Survey of Multi-Agent Organizational Paradigms*, University of Massachusetts, Amherst, MA, May, 2005
- http://www.madkit.net/madkit/documents.php