Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

И.А. Вотинов

Существо *ConquerorBeing* для проекта

Электрические джунгли

Проектная документация

Проект создан в рамках
«Движения за открытую проектную документацию»
http://is.ifmo.ru

Санкт-Петербург
2007

Оглавление

Введение

В ходе бакалаврской практики была поставлена задача – создать существо для проекта Электрические Джунгли (http://www.electricjungle.ru).

Электрические Джунгли (ЭД) – это состязание, в котором игроки программируют поведение населяющих особый мир виртуальных существ, борющихся за ограниченный ресурс – «пищу». Целью игрока является программирование поведения своих существ таким образом, чтобы его вид как целое максимально преуспел. Разработка должна вестись на языке *Java*.

Для решения данной задачи была использована SWITCH-технология, так как она в полной мере подходит для данного типа задач.

1. Правила проекта *Электрические Джунгли*

1.1. Обозначения

- `X% МЭ` — количество энергии в процентах от максимальной энергии (МЭ) существа;
- `K_someconst(5)` — постоянная, которая определяет значение одного параметра игры (все постоянные определены в исходном тексте игры). В скобках указано текущее значение. Для большинства констант эти значения во время всего конкурса сохранятся, однако возможны и переопределения.

1.2. Понятия и цели

В мире ЭД основная цель — максимальная видовая экспансия. Победившим признается тот игрок, чей вид за отведенное количество ходов добьется максимальной массы – суммарная масса всех особей которого будет наибольшей.

1.2.1. Пространство и время

В ЭД пространство устроено как замкнутое двумерное дискретное поле из клеток, закольцованное по краям (так, что получается тор). Впрочем, в движке поддерживаются и другие топологии и размерности, но победитель будет выявлен на стандартной тороидальной топологии 140x120. Игра состоит из последовательности ходов. За один ход каждому существу дается шанс совершить свое действие. Когда все существа совершат свои действия — начинается следующий ход. В общем случае порядок выполнения действий не определен, разные существа могут совершать действия одновременно.

1.2.2. Масса и скорость

Каждое существо в ЭД обладает двумя базовыми характеристиками: массой и скоростью, а также в каждый момент времени характеризуется энергетическим уровнем. Масса совпадает с максимальной энергией (МЭ), которой может обладать существо. Если энергетический уровень падает ниже `K_emin`(15)% МЭ — существо умирает и вся его оставшаяся энергия доступна для потребления другими существами. Масса не может быть больше `K_maxmass(1000)` и меньше `K_minmass`(0.1). Скорость не может быть больше `K_maxspeed`(10) и меньше `K_minspeed`(1). Общая масса вида (сумма масс всех живых существ данного вида)

определяет количество очков игрока. Масса существа определяет его энергоемкость, энергопотребление за ход и повреждение, наносимое в битве. Скорость определяет максимальное расстояние, на которое может передвигаться существо за один ход. Однако чем больше скорость, тем энергетически дороже передвижение существа. Независимо от скорости существо может получать информацию только о точке, где оно находится, а также о соседних точках.

1.2.3. Энергия

Энергия является основой существования в ЭД. Любое действие (даже просто выживание) требует потребления некоторого количества энергии. В начале игры на поле случайным образом размещаются источники энергии, из которых существа могут черпать энергию (не более 10% МЭ). Энергия в источниках постепенно восполняется. Начальное количество энергии и скорость ее роста случайны и различны для разных источников. На поле имеется `NUM_REGULAR`(130) обычных источников и `NUM_GOLDEN`(3) золотых источников, производящих гораздо больше энергии. Кроме этого, в точке, где рождается первое существо каждого игрока даётся `BORN_BONUS`(100) единиц энергии, и за каждый ход прирастает `BORN_BONUS_GROWTH`(1) единиц энергии. Хотя в различных играх энергия и прирост по-разному распределены на поле, суммарная энергия и ее прирост всегда одинаковы.

1.2.4. Информация о внешнем мире

Существо может получить информацию о внешнем мире при помощи API `BeingIntreface` и `PointInfo`. Доступна следующая информация:

- количество энергии самого существа;
- владелец и масса любого существа, находящегося достаточно близко (на той же клетке или на примыкающих клетках);
- количество энергии, скорость ее прироста и максимальная емкость в клетке;
- список всех объектов в данной точке (в частности, других существ);
- суммарная масса всех живых существ в данной точке.

1.3. Действия и события

За один ход существо может совершить одно действие, а также каждое существо получает оповещения о происходящих с ним событиях. Конкретный порядок выполнения действий не специфицирован. Доступны следующие действия:

- `ACTION_MOVE_TO` — перейти в другую клетку. Доступность клетки определяется скоростью существа. Стоит `K_movecost(1)%` скорости.
- `ACTION_EAT` — потребить энергию. Количество энергии, которую можно потребить за ход, но общая энергия не может превышать энергоемкости (массы) и потребленная энергия не может быть больше `K_bite(10)%` МЭ. Бесплатно.
- `ACTION_GIVE` — передать энергию другому существу. Бесплатно. Существа должны быть на одной и той же клетке.

- ACTION_ATTACK — Атаковать другое существо, нанося повреждение в K_fight(20)% МЭ. При этом теряется K_fightcost(1)% собственной МЭ + K_retaliate(5)% МЭ атакуемого существа.
- ACTION_BORN — породить другое существо, возможно с немного отличающимися (не более чем на K_minbornvariation(0.8)/K_maxbornvariation(1.2)) массой и скоростью (существо с массой 100 и скоростью 2 может порождать существа с массой от 80 до 120 и скоростью от 1.6 до 2.4). Энергия при этом делится пополам. Рождение возможно только тогда, когда у существа достаточно энергии – не менее K_toborn(80%) МЭ. Акт рождения стоит K_borncost(20)% МЭ. При этом существу можно передать "генокод" — произвольный Java объект.
- ACTION_MOVE_ATTACK — совмещено двигаться и атаковать, при этом наносится меньше K_fightmovepenalty(0.75) повреждений. Стоимость совпадает с суммарной стоимостью атаки и передвижения.

Также существо получает оповещения о следующих событиях:

- BEING_BORN — первое оповещение, которую существо получает после рождения. Можно инициализировать различные параметры, специфичные для данного существа.
- BEING_DEAD — последнее оповещение, передается после смерти существа.
- BEING_ATTACKED — существо атаковано кем-то, идентификатор атакующего передается как параметр.
- BEING_ENERGY_GIVEN — Кто-то передал нам энергию.

1.4. Состязание

Состязание начинается с помещения одного существа каждого игрока в случайную точку пространства, при этом один раз вызывается метод reinit(), информирующий о текущих условиях игры и позволяющий заново инициализировать статические переменные. Масса и энергия изначального существа определяется самим игроком и может быть произвольной. Параметры первого существа определяются возвращаемым значением метода Being.getParams(). Параметры всех остальных существ определяются параметрами действия ACTION_BORN. После этого у каждого существа есть возможность попытаться добиться победы в заданных условиях, как описано далее.

1.5. Виды игр

1. **Блицкриг** — SINGLE

Одному виду существ дается весь мир. Цель — добиться наивысшего рейтинга путем максимально эффективного использования ресурсов и быстрой разведки за данный временной интервал (200 ходов).

2. **Дуэль** — DUEL

Главный вид состязания в *Электрических Джунглях*. Два конкурирующих вида существ сражаются на протяжении 1000 ходов.

3. **Джунгли** — `JUNGLE`

До восьми конкурирующих видов сражаются на одном поле 2000 ходов.

1.6. Графическое представление игры

Поле игры приведено на рис. 1.

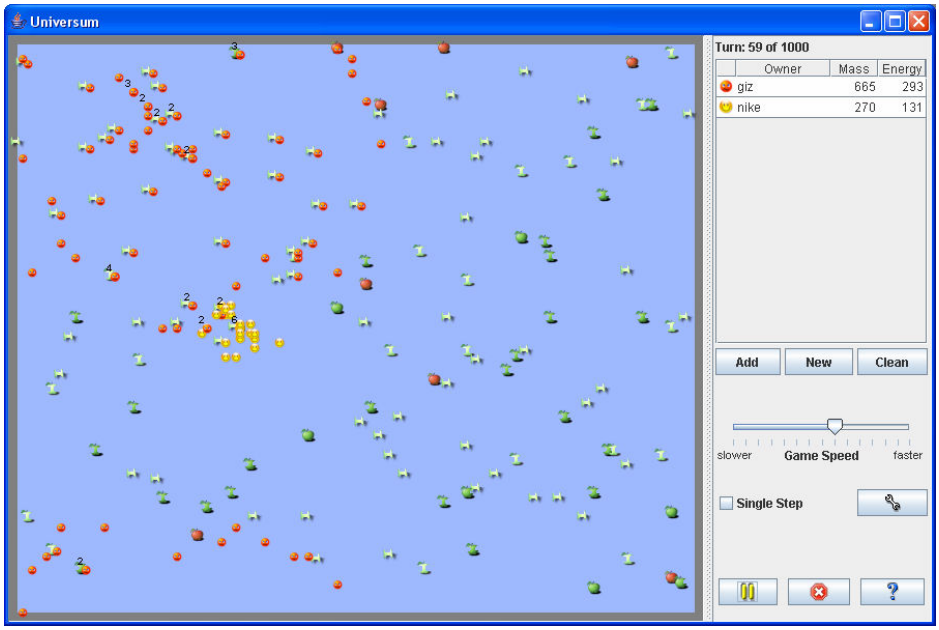


Рис. 1. Поле игры

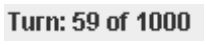
Слева на рис. 1:

 и  – существа из соответствующих команд на игровом поле;

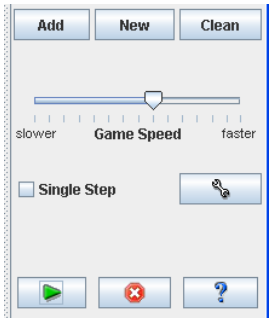
 – источники энергии;

 – богатые источники энергии.

Справа на рис. 1:

 – номер текущего хода;

 – список существ, участвующих в игре (а так же информация о суммарной массе и энергии существ);

 – элементы управления и настройки игры.

2. Описание существа

2.1. Общее описание существа

Для проекта *Электрические Джунгли* было написано существо, которое было названо *ConquerorBeing*. Основное внимание было уделено конкурсу “Блицкриг”, поэтому основные элементы стратегии относятся к оптимизации размножения существ, распределения энергии и т.п. Существо *ConquerorBeing* имеет массу четыре и скорость три.

2.2. Основные элементы стратегии

Основные элементы стратегии включают:

- Разведка – игровое поле разбивается на блоки клеток 3x3, каждое существо-разведчик выбирает ближайший неразведанный блок, помечает его как занятый и направляется на его исследование. В случае если существо не смогло добраться до выбранной точки, эта точка возвращается в список неразведанных. При этом существо начинает разведку только в том случае если ему хватит энергии для того, чтобы добраться до выбранного неразведанного блока и затем добраться до ближайшей к блоку колонии (этим предотвращается гибель существ во время разведки).

- Размножение – для уменьшения затрат на порождение существ, масса нового существа выбирается большей на 20% (`K_maxbornvariation`), чем масса родителя. При этом при тех же затратах энергии получается больший прирост массы популяции.

- Распределение энергии – когда игра близится к завершению, все существа, имеющие достаточно много энергии, собираются в группы в определенных точках для эффективного перераспределения остатков энергии. При этом существа в группе делятся на две категории: первая категория порождает новые существа, а вторая передает всю свою свободную энергию (свободная энергия – энергия существа за вычетом энергии, необходимой для выживания до конца игры) существам из первой категории.

- Защита – когда в непосредственной близости от колонии (колония – это группа существ, питающихся от одного источника энергии и распределяющих эту энергию с целью эффективного размножения) появляется существо противника, существа колонии атакуют его с целью предотвращения захвата колонии.

3. Описание автомата существа *ConquerorBeing* (*A1*)

3.1. Описание событий

События автомата *A1* приведены в табл. 1.

Таблица 1

e1	Существу следует сделать ход
e2	В области видимости существа обнаружен противник
e3	Существу следует стать разведчиком
e4	Существо погибло
e5	Существу следует идти к точке сбора для передачи энергии

3.2. Описание входных переменных

Входные переменные автомата *A1* приведены в табл. 2.

Таблица 2

x1_1	Следует ли существу породить новое существо
x1_2	Следует ли существу есть
x1_3	Следует ли существу передать часть своей энергии другому существу
x2_1	Следует ли существу создать новую колонию
x2_2	Существо совершает последний шаг поиска
x3_1	Следует ли существу атаковать противника на текущей позиции
x3_2	Следует ли существу атаковать противника на соседней позиции
x4_1	Существо совершает последний шаг к колонии
x5_1	Существо совершает последний шаг к точке сбора

3.3. Описание выходных воздействий

Выходные воздействия автомата *A1* приведены в табл. 3.

Таблица 3

z1	Породить новое существо
z2	Есть
z3	Передать энергию другому существу
z4	Создать новую колонию
z5	Совершить шаг поиска
z6	Совершить последний шаг поиска
z7	Совершить последний шаг к колонии
z8	Совершить шаг к колонии
z9	Совершить последний шаг к точке сбора
z10	Совершить шаг к точке сбора
z11	Покинуть колонию
z12	Завершить поиск
z13	Вернуться в колонию
z14	Атаковать противника на текущей позиции
z15	Атаковать противника на соседней позиции
z16	Обработка гибели защитника

3.4. Граф переходов

Граф переходов автомата *A1* изображен на рис. 2. Он построен с помощью редактора инструментального средства *UniMod* (http://unimod.sourceforge.net/).

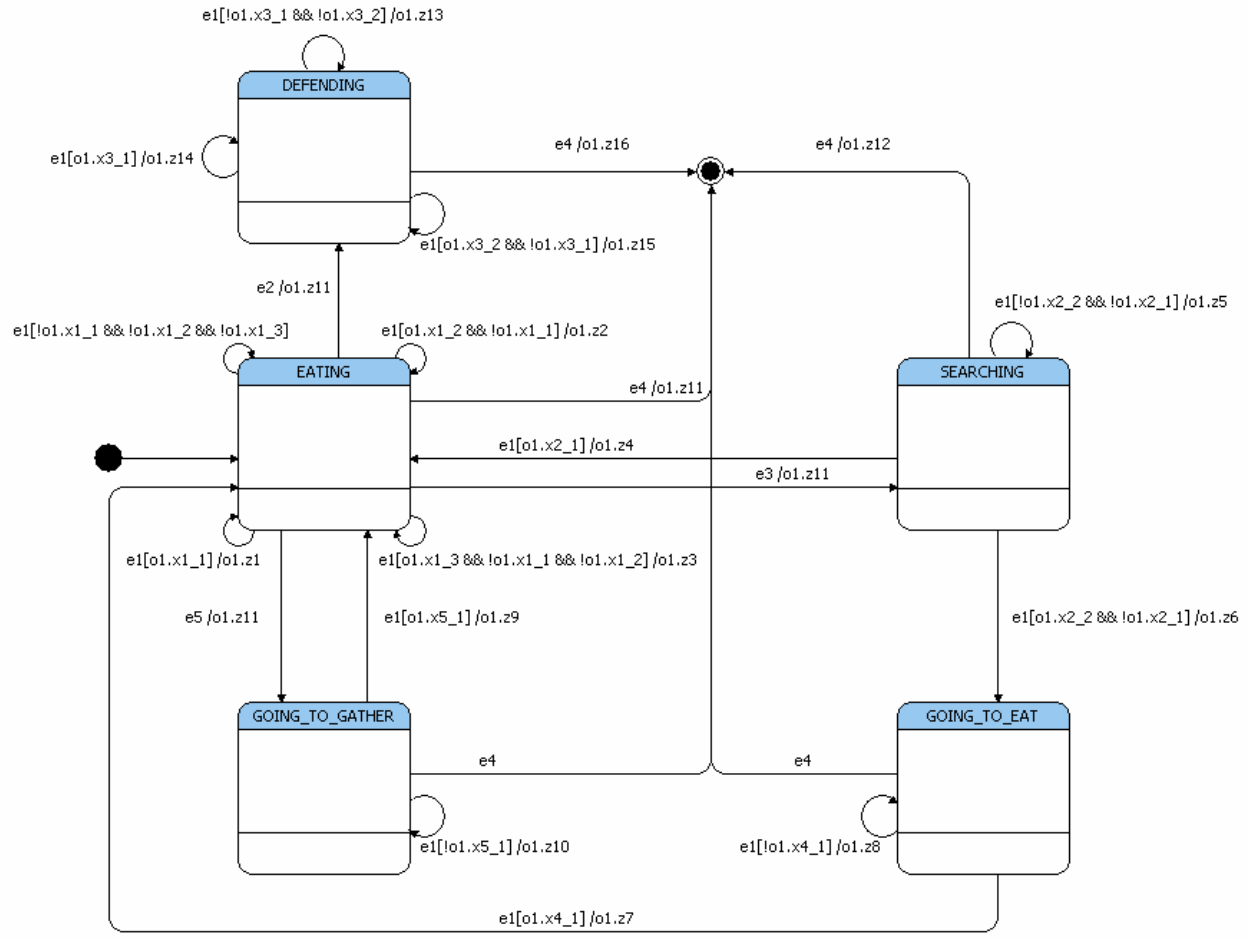


Рис. 2. Граф переходов автомата *A1*

Для реализации этого автомата *UniMod* использовать было нельзя, так как по правилам игры не разрешалось использовать внешние библиотеки. При реализации входные переменные с инверсиями в условиях переходов, которые введены для обеспечения непротиворечивости, в явном виде в коде не используются.

4. Структура программы

В файле `ConquerorBeing.java` находится главный класс существа. Именно методы этого класса вызывает игровая оболочка и именно из него игровой оболочке передаются ходы существа. В этом файле находится реализация автомата существа.

В файле `CorporateMind.java` описан “коллективный разум”, который осуществляет координацию действий отдельных существ.

Класс `EatersColony` (файл `EatersColony.java`) осуществляет взаимодействие между существами, находящимися в одной колонии.

Класс `SearchMap` (файл `SearchMap.java`) хранит неразведанную часть карты, разбитой на блоки 3x3.

Вспомогательный класс `SearchUtil` (файл `SearchUtil.java`) содержит методы, связанные с поиском, такие как расчет стоимости передвижения между двумя точками для существа и т.п.

5. Запуск программы

Для запуска скомпилированной программы необходимо:

1) С сайта *Электрические джунгли* (http://www.electricjungle.ru/) загрузить Java-архив `ejungle_distr.jar` с исполняемым кодом игры.

2) С сайта проекта скачать Java-архив `ConquerorBeing.jar`, содержащий программу, реализующую алгоритм существа.

3) Запустить игру при помощи команды[1]:

```
java -jar ejungle_distr.jar
```

4) В открывшемся главном окне (рис.3) приложения нажать клавишу «Add» для того, чтобы добавить существо в игру.

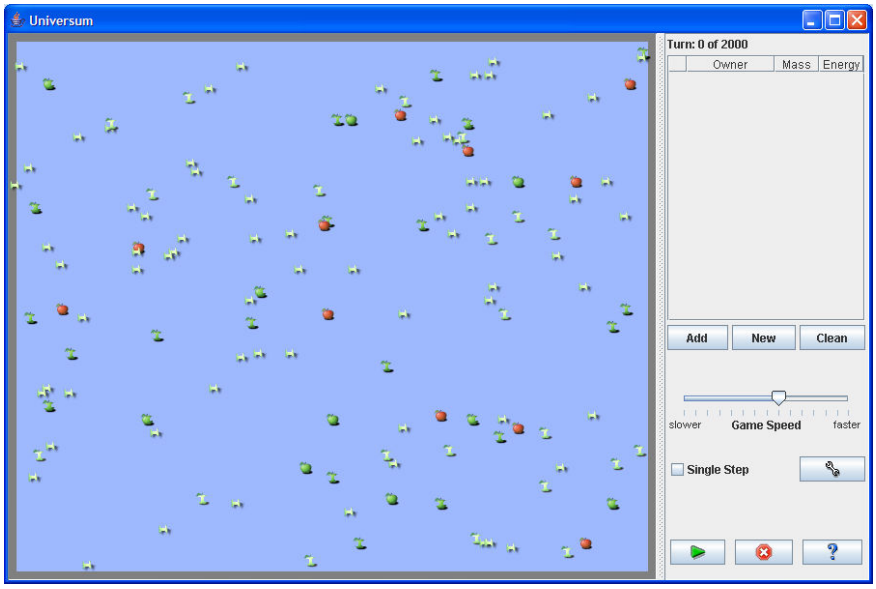


Рис. 3. Главное окно приложения

[1] Для запуска необходимо, чтобы на компьютере была установлена *Java* версии 1.5 или более поздней.

5) В появившемся диалоге (рис.4) указать путь к Java-архиву `QBeing3.jar`.

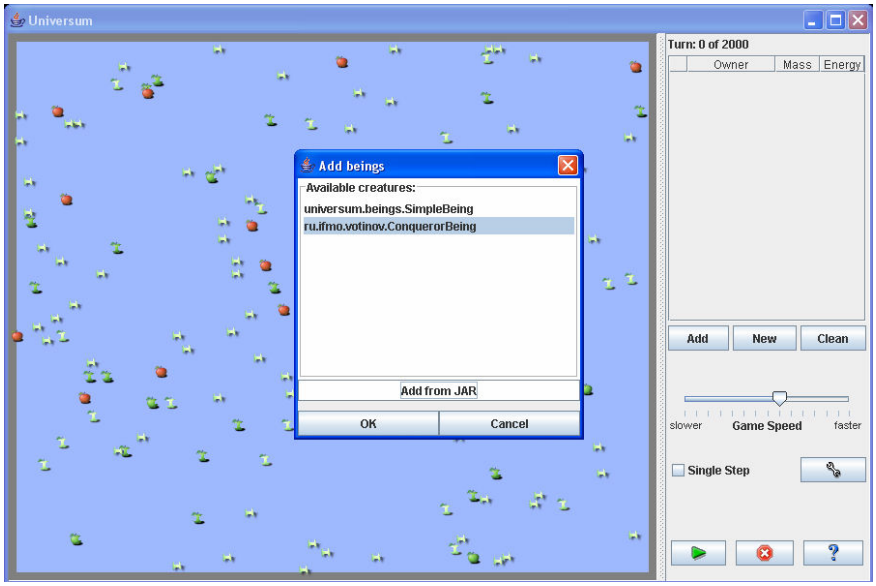


Рис. 4. Диалог выбора существ

6) Нажать кнопку  для запуска игры.

Заключение

При разработке интеллекта основное внимание уделялось одному из конкурсов – “Блицкриг”. При этом одному виду существ дается весь мир. Цель — добиться наивысшего рейтинга путем максимально эффективного использования ресурсов и быстрой разведки за данный временной интервал (200 ходов). Поэтому хороший результат был достигнут именно в этом конкурсе.

Существо участвует в конкурсе под названием *ConquerorBeing* (автор *giz*) и занимает в конкурсе:

- Достижения в Одиночных играх

1	Vovka	universum.beings.vovka.BaseBeing	8822
2	kammerer	ru.kammerer.being.snake.Snake	8618
3	Enotus	universum.beings.Enot1	8613
4	kammerer	ru.kammerer.being.snake.Snake2	8410
5	Griff	universum.beings.griff.BeingA	8284
6	Executer	universum.beings.HomoSapiensPack.HomoSapiens	8270
7	sergonaft	universum.beings.SerZhenCreature	7705
8	giz	ru.ifmo.votinov.ConquerorBeing	7646
9	sergonaft	universum.beings.SerZhenSimplest	7477
10	forlik	universum.beings.forlik.Wasp	7363

- Сильнейшие популяции по результатам дуэлей

3 лига			
123	Firefox	universum.beings.Firebat	139
124	sergey	ejungle.Beast01	138
125	giz	ru.ifmo.votinov.ConquerorBeing	134
126	root	universum.beings.TrioBeing	124
127	Firefox	universum.beings.Firefox	122

- Самые живучие существа Джунглей

7	Acinonyx	universum.beings.Canis	7
8	Enotus	universum.beings.Enot1	7
9	Anatol	universum.beings.XTol	6
10	wp	universum.beings.ThirdBeing	6
81	happy	universum.beings.BugagaZverushkaPro	0
82	demon	demon.beings.ant.Ant	0
83	giz	ru.ifmo.votinov.ConquerorBeing	0
84	Volna80	com.volna80.jungle.Ant	0
85	Lotor	universum.beings.Asmodey	0

Источники

1. *Sun Microsystems.* Конкурс алгоритмов «Электрические Джунгли» http://www.electricjungle.ru
2. *Паньгин А.А.* Электроджунгли. Новый конкурс по программированию на Java //Компьютерные инструменты в образовании. 2006. № 2, с.85–87. http://is.ifmo.ru/elejungle/_CIE-EJ.pdf
3. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
4. *Электрические джунгли*. http://is.ifmo.ru/elejungle/

Приложение. Исходные коды

ConquerorBeing.java

```
package ru.ifmo.votinov;

import universum.bi.*;

import java.util.List;

public class ConquerorBeing implements Being {
    private static final String OWNER = "giz";
    private static final String NAME = "Conqueror";

    private static final float M0 = 4f;
    private static final float S0 = 3f;

    // states
    private static final int EATING = 1;
    private static final int SEARCHING = 2;
    private static final int DEFENDING = 3;
    private static final int GOING_TO_EAT = 4;
    private static final int GOING_TO_GATHER = 5;

    // events
    /**
     * make new turn
     */
    static final int E1 = 1;

    /**
     * enemy is near
     */
    static final int E2 = 2;

    /**
     * should become searcher
     */
    static final int E3 = 3;

    /**
     * being died
     */
    static final int E4 = 4;

    /**
     * should go to gather
     */
    static final int E5 = 5;

    private int state;
```

```
private float mass;
private float speed;
private float energy;
private Location colonyLocation;

private Event actionForCurrentTurn;
private BeingInterface beingInterface;

public void reinit(UserGameInfo info) {
    CorporateMind.createInstance(info);
}

public String getName() {
    return NAME;
}

public String getOwnerName() {
    return OWNER;
}

public BeingParams getParams() {
    return new BeingParams(M0, S0);
}

public void processEvent(BeingInterface bi, Event e) {
    beingInterface = bi;
    switch (e.kind()) {
        case BEING_BORN:
            state = EATING;
            mass = ((BeingParams)e.param()).M;
            speed = ((BeingParams)e.param()).S;
            fireBeingIsBorn();
            break;
        case BEING_DEAD:
            // processing being died event
            processEvent(E4);
            fireBeingIsDead();
            break;
        case BEING_ATTACKED:
            break;
        case BEING_ENERGY_GIVEN:
            break;
    }
}

public Event makeTurn(BeingInterface bi) {
    updateInfo(bi);
    // processing make turn event
    processEvent(E1);
    return actionForCurrentTurn;
}

void processEvent(int event) {
```

```
switch (state) {
    case EATING:
        switch (event) {
            // make new turn
            case E1:
                // should born
                if (x1_1()) {
                    // born new being
                    z1();
                }
                // should eat
                else if (x1_2()) {
                    // eat
                    z2();
                }
                // should give energy
                else if (x1_3()) {
                    // give energy
                    z3();
                }
                break;
            // enemy is near
            case E2:
                // leave eaters colony
                z11();
                state = DEFENDING;
                break;
            // should become searcher
            case E3:
                // leave eaters colony
                z11();
                state = SEARCHING;
                break;
            // being died
            case E4:
                // leave eaters colony
                z11();
                break;
            // should go to gather
            case E5:
                // leave eaters colony
                z11();
                state = GOING_TO_GATHER;
                break;
        }
        break;
    case SEARCHING:
        switch (event) {
            // make new turn
            case E1:
                // being should start new eaters colony.
                if (x2_1()) {
```

```
                    // start new eaters colony.
                    z4();
                    state = EATING;
                }
                // being is making it's last possible search step.
                else if (x2_2()) {
                    // make last search step.
                    z6();
                    state = GOING_TO_EAT;
                } else {
                    // make search step.
                    z5();
                }
                break;
            // being died
            case E4:
                // stop searching
                z12();
                break;
        }
        break;
    case DEFENDING:
        switch (event) {
            // make new turn
            case E1:
                // being should attack enemy at the same location
                if (x3_1()) {
                    // attack enemy at the same location
                    z14();
                }
                // being should attack enemy at another location
                else if (x3_2()) {
                    // attack enemy at another location
                    z15();
                } else {
                    // return to eaters colony
                    z13();
                    state = EATING;
                }
                break;
            // being died
            case E4:
                // handle defender died.
                z16();
                break;
        }
        break;
    case GOING_TO_EAT:
        switch (event) {
            // make new turn
            case E1:
            // being is making it's last step to feeding location.
                if (x4_1()) {
```

```
                        // make last step to feeding location.
                        z7();
                        state = EATING;
                    } else {
                        // make step to feeding location.
                        z8();
                    }
                    break;
            }
            break;
        case GOING_TO_GATHER:
            switch (event) {
                // make new turn
                case E1:
                 // being is making it's last step to gathering location.
                    if (x5_1()) {
                        // make last step to gathering location.
                        z9();
                        state = EATING;
                    } else {
                        // make step to gathering location.
                        z10();
                    }
                    break;
            }
            break;
    }
}

private void fireBeingIsBorn() {
    getCorporateMind().addBeing(this);
    joinTheColony(getLocation());
}

private void fireBeingIsDead() {
    getCorporateMind().removeBeing(this);
}

private Location getLocation() {
    return beingInterface.getLocation(this);
}

private Location getColonyLocation() {
    return colonyLocation;
}

private CorporateMind getCorporateMind() {
    return CorporateMind.getInstance();
}

private void joinTheColony(Location location) {
    colonyLocation = location;
    getCorporateMind().joinTheColony(this, location);
```

```
    }

    private void leaveOfColony() {
        getCorporateMind().leaveColony(this, colonyLocation);
    }

    private BeingParams getParamsForNewBeing() {
        return new BeingParams(Constants.K_maxbornvariation * mass, speed);
    }

    private void updateInfo(BeingInterface bi) {
        beingInterface = bi;
        actionForCurrentTurn = null;
        getCorporateMind().updateEnemyMasses(bi, this);
        getCorporateMind().checkIfNewTurnStarted(bi, this);
    }

    float getMass() {
        return mass;
    }

    float getSpeed() {
        return speed;
    }

    float getEnergy() {
        return energy;
    }

    float getEnergyPercent() {
        return energy / mass;
    }

    void updateEnergy() {
        energy = beingInterface.getEnergy(this);
    }

    /**
     * Returns whether being should born new being.
     * @return whether being should born new being.
     */
    private boolean x1_1() {
        return getEnergyPercent() > Constants.K_toborn;
    }

    /**
     * Returns whether or not being should eat.
     * @return whether or not being should eat.
     */
    private boolean x1_2() {
        return getCorporateMind().shouldEat(beingInterface, this,
colonyLocation);
    }
```

```
    /**
     * Returns whether or not being should give energy.
     * @return whether or not being should give energy.
     */
    private boolean x1_3() {
        return getCorporateMind().shouldGiveEnergy(beingInterface, this,
colonyLocation);
    }

    /**
     * Returns whether or not being should start new eaters colony.
     * @return whether or not being should start new eaters colony.
     */
    private boolean x2_1() {
        return getCorporateMind().getUnoccupiedEatingLocation(beingInterface,
this) != null;
    }

    /**
     * Returns whether or not being is making it's last possible search step.
     * @return whether or not being is making it's last possible search step.
     */
    private boolean x2_2() {
        List<Location> locations =
getCorporateMind().getSearchLocations(beingInterface, this);
        if (locations.size() == 0) {
            return !getCorporateMind().canSearchFromLocation(beingInterface,
this, getLocation());
        }
        if (locations.size() == 1) {
            Location location = locations.get(0);
            return
beingInterface.getReachableLocations(this).contains(location) &&
                    !getCorporateMind().canSearchFromLocation(beingInterface,
this, location);
        }
        return false;
    }

    /**
     * Returns whether or not being should attack enemy at the same location.
     * @return whether or not being should attack enemy at the same location.
     */
    private boolean x3_1() {
        Location location =
getCorporateMind().getAttackLocationForDefender(beingInterface, this,
colonyLocation);
        return location != null &&
location.equals(beingInterface.getLocation(this));
    }
```

```
    /**
     * Returns whether or not being should attack enemy at another location
     * location.
     * @return whether or not being should attack enemy at another location
     * location.
     */
    private boolean x3_2() {
        Location location =
getCorporateMind().getAttackLocationForDefender(beingInterface, this,
colonyLocation);
        return location != null &&
!location.equals(beingInterface.getLocation(this));
    }

    /**
     * Returns whether or not being is making it's last step to feeding
     * location.
     * @return whether or not being is making it's last step to feeding
     * location.
     */
    private boolean x4_1() {
        Location location =
getCorporateMind().getClosestLocationWithEnergy(beingInterface, this);
        return beingInterface.getReachableLocations(this).contains(location)
||
                location.equals(beingInterface.getLocation(this));
    }

    /**
     * Returns whether or not being is making it's last step to gathering
     * location.
     * @return whether or not being is making it's last step to gathering
     * location.
     */
    private boolean x5_1() {
        Location location =
getCorporateMind().getGatheringLocation(beingInterface, this);
        return beingInterface.getReachableLocations(this).contains(location);
    }

    /**
     * Born new being.
     */
    private void z1() {
        actionForCurrentTurn = new Event(EventKind.ACTION_BORN,
getParamsForNewBeing());
    }

    /**
     * Eat.
     */
    private void z2() {
        actionForCurrentTurn = new Event(EventKind.ACTION_EAT, mass);
```

```
    }

    /**
     * Give energy.
     */
    private void z3() {
        Integer id =
beingInterface.getId(getCorporateMind().getEnergyTaker(this,
colonyLocation));
        if (id != null) {
            actionForCurrentTurn = new Event(EventKind.ACTION_GIVE, id,
                    getCorporateMind().getAmountToGive(this,
colonyLocation));
        }
    }

    /**
     * Start new eaters colony.
     */
    private void z4() {
        getCorporateMind().stopSearching(this);
        Location location =
getCorporateMind().getUnoccupiedEatingLocation(beingInterface, this);
        joinTheColony(location);
        actionForCurrentTurn = new Event(EventKind.ACTION_MOVE_TO, location);
    }

    /**
     * Make search step.
     */
    private void z5() {
        makeSearchStep();
    }

    /**
     * Make last search step.
     */
    private void z6() {
        makeSearchStep();
        getCorporateMind().stopSearching(this);
    }

    /**
     * Make last step to feeding location.
     */
    private void z7() {
        Location location =
getCorporateMind().getClosestLocationWithEnergy(beingInterface, this);
        joinTheColony(location);
        actionForCurrentTurn = new Event(EventKind.ACTION_MOVE_TO, location);
    }
```

```
    /**
     * Make step to feeding location.
     */
    private void z8() {
        Location location =
getCorporateMind().getClosestLocationWithEnergy(beingInterface, this);
        location = SearchUtil.stepToward(beingInterface, this, location);
        actionForCurrentTurn = new Event(EventKind.ACTION_MOVE_TO, location);
    }

    /**
     * Make last step to gathering location.
     */
    private void z9() {
        Location location =
getCorporateMind().getGatheringLocation(beingInterface, this);
        joinTheColony(location);
        actionForCurrentTurn = new Event(EventKind.ACTION_MOVE_TO, location);
    }

    /**
     * Make step to gathering location.
     */
    private void z10() {
        Location location =
getCorporateMind().getGatheringLocation(beingInterface, this);
        location = SearchUtil.stepToward(beingInterface, this, location);
        actionForCurrentTurn = new Event(EventKind.ACTION_MOVE_TO, location);
    }

    /**
     * Leave eaters colony
     */
    private void z11() {
        leaveOfColony();
    }

    /**
     * Stop searching
     */
    private void z12() {
        getCorporateMind().stopSearching(this);
    }

    /**
     * Return to eaters colony
     */
    private void z13() {
        joinTheColony(colonyLocation);
        actionForCurrentTurn = new Event(EventKind.ACTION_MOVE_TO,
colonyLocation);
    }
```

```
    /**
     * Attack enemy at the same location.
     */
    private void z14() {
        actionForCurrentTurn = new Event(EventKind.ACTION_ATTACK,
                getCorporateMind().getTargetEnemyId(this,
getColonyLocation()));
    }

    /**
     * Attack enemy at another location.
     */
    private void z15() {
        actionForCurrentTurn = new Event(EventKind.ACTION_MOVE_ATTACK,
                getCorporateMind().getTargetEnemyId(this,
getColonyLocation()));
    }

    /**
     * Handle defender died.
     */
    private void z16() {
        getCorporateMind().removeDefender(this, getColonyLocation());
    }

    private void makeSearchStep() {
        List<Location> locations =
getCorporateMind().getSearchLocations(beingInterface, this);
        if (locations.size() == 0) {
            return;
        }
        Location location = locations.get(0);
        if (beingInterface.getReachableLocations(this).contains(location)) {
            locations.remove(location);
        } else {
            location = SearchUtil.stepToward(beingInterface, this, location);
        }
        actionForCurrentTurn = new Event(EventKind.ACTION_MOVE_TO, location);
    }
}
```

CorporateMind.java

```
package ru.ifmo.votinov;

import universum.bi.*;

import java.util.*;

class CorporateMind {
    private static final int STEPS_TO_EAT = 2;
    private static final int GATHERING_LOCATIONS_COUNT = 10;

    private static CorporateMind instance;

    private Map<Integer, Float> enemyMasses;
    private Map<ConquerorBeing, List<Location>> searchLocations;
    private Map<Location, EatersColony> colonies;
    private SearchMap searchMap;
    private Map<Location, Float> energyMap;
    private Map<Location, Location> closestEnergyLocation;
    private List<Location> allSeekLocations;
    private List<ConquerorBeing> beings;
    private List<Location> gatheringLocations;
    private int turn;
    private int maxTurn;

    static void createInstance(UserGameInfo info) {
        instance = new CorporateMind(info);
    }

    static CorporateMind getInstance() {
        return instance;
    }

    private EatersColony getOrCreateColony(Location location) {
        if (colonies.get(location) == null) {
            colonies.put(location, new EatersColony());
        }
        return colonies.get(location);
    }

    private CorporateMind(UserGameInfo info) {
        turn = -1;
        searchMap = new SearchMap(3);
        energyMap = new HashMap<Location, Float>();
        closestEnergyLocation = new HashMap<Location, Location>();
        allSeekLocations = new ArrayList<Location>();
        colonies = new HashMap<Location, EatersColony>();
        beings = new ArrayList<ConquerorBeing>();
        searchLocations = new HashMap<ConquerorBeing, List<Location>>();
        enemyMasses = new HashMap<Integer, Float>();
        maxTurn = info.maxTurns;
    }
```

```
    int getTurn() {
        return turn;
    }

    int getMaxTurn() {
        return maxTurn;
    }

    synchronized void checkIfNewTurnStarted(BeingInterface bi, ConquerorBeing
being) {
        if (bi.getTurnsCount() > turn) {
            turn = bi.getTurnsCount();
            handleNewTurnStarted(bi, being);
        }
    }

    private void init(Location startLocation) {
        if (!searchMap.isInitialized()) {
            searchMap.init(startLocation);
            allSeekLocations.addAll(searchMap);
            searchMap.remove(startLocation);
        }
    }

    void addBeing(ConquerorBeing being) {
        beings.add(being);
    }

    void removeBeing(ConquerorBeing being) {
        beings.remove(being);
    }

    private void handleNewTurnStarted(BeingInterface bi, ConquerorBeing
being) {
        init(bi.getLocation(being));
        updateEnergies(bi);
        Collection<EatersColony> colonies = new
ArrayList<EatersColony>(this.colonies.values());
        for (EatersColony colony : colonies) {
            colony.handleNewTurnStarted();
        }
    }

    private SearchMap getSearchMap() {
        return searchMap;
    }

    private float getEnergy(Location location) {
        Float energy = energyMap.get(location);
        return energy != null ? energy : 0;
    }
```

```
    private Location getClosestLocationWithEnergy(BeingInterface bi,
ConquerorBeing being, Location location) {
        Location closestLocation = closestEnergyLocation.get(location);
        if (closestLocation == null) {
            for (PointInfo pi : bi.getNeighbourInfo(being)) {
                closestLocation =
closestEnergyLocation.get(pi.getLocation());
                if (closestLocation != null) {
                    break;
                }
            }
        }
        return closestLocation;
    }

    Location getClosestLocationWithEnergy(BeingInterface bi, ConquerorBeing
being) {
        return getClosestLocationWithEnergy(bi, being,
bi.getPointInfo(being).getLocation());
    }

    Location getGatheringLocation(BeingInterface bi, ConquerorBeing being) {
        if (gatheringLocations == null) {
            gatheringLocations = new ArrayList<Location>(colonies.keySet());
            Collections.sort(gatheringLocations, new Comparator<Location>() {
                public int compare(Location o1, Location o2) {
                    return
((Integer)colonies.get(o2).size()).compareTo(colonies.get(o1).size());
                }
            });
            gatheringLocations = gatheringLocations.subList(0,
                    Math.min(GATHERING_LOCATIONS_COUNT,
gatheringLocations.size()));
        }
        return SearchUtil.getClosestLocation(bi, bi.getLocation(being),
gatheringLocations);
    }

    private void updateEnergies(BeingInterface bi) {
        for (ConquerorBeing being : beings) {
            being.updateEnergy();
            List<PointInfo> ni = bi.getNeighbourInfo(being);
            for (PointInfo pi : ni) {
                updateEnergies(bi, being, pi);
            }
            updateEnergies(bi, being, bi.getPointInfo(being));
        }
    }

    private void updateEnergies(BeingInterface bi, ConquerorBeing being,
PointInfo pi) {
        float count = pi.getCount(being);
        energyMap.put(pi.getLocation(), count);
```

```
        if (count > 0) {
            updateClosestEnergyLocations(bi, pi.getLocation());
        }
    }

    private void updateClosestEnergyLocations(BeingInterface bi, Location
location) {
        for (Location seekedLocation : allSeekLocations) {
            updateClosestEnergyLocations(bi, seekedLocation, location);
        }
    }

    private void updateClosestEnergyLocations(BeingInterface bi, Location
seekLocation, Location location) {
        Location closestLocation = closestEnergyLocation.get(seekLocation);
        if (closestLocation == null || bi.distance(seekLocation, location) <
bi.distance(seekLocation, closestLocation)) {
            closestEnergyLocation.put(seekLocation, location);
        }
    }

    synchronized void updateEnemyMasses(BeingInterface bi, ConquerorBeing
being) {
        for (PointInfo pi : bi.getNeighbourInfo(being)) {
            Integer[] ids = pi.getEntities(being, null);
            for (Integer id : ids) {
                if (!bi.getOwner(being, id).equals(bi.getOwner(being,
bi.getId(being)))) {
                    float mass = bi.getMass(being, id);
                    if (mass != 0) {
                        enemyMasses.put(id, mass);
                    }
                }
            }
        }
    }

    synchronized void stopSearching(ConquerorBeing being) {
        if (searchLocations.get(being) != null) {
            getSearchMap().addAll(searchLocations.get(being));
            searchLocations.put(being, null);
        }
    }

    Location getUnoccupiedEatingLocation(BeingInterface bi, ConquerorBeing
being) {
        Location eatingLocation = null;
        float maxCount = 0f;
        List<Location> rll = bi.getReachableLocations(being);
        for (Location rl : rll) {
            float count = getEnergy(rl);
            if (count > maxCount && !hasEaters(rl)) {
                maxCount = count;
```

```
                eatingLocation = rl;
            }
        }
        return eatingLocation;
    }

    synchronized List<Location> getSearchLocations(BeingInterface bi,
ConquerorBeing being) {
        if (searchLocations.get(being) == null ||
searchLocations.get(being).size() == 0) {
            List<Location> locations = getSearchLocations(bi, being,
bi.getLocation(being));
            searchLocations.put(being, locations);
            getSearchMap().removeAll(locations);
        }
        return searchLocations.get(being);
    }

    synchronized boolean canSearchFromLocation(BeingInterface bi,
ConquerorBeing being, Location location) {
        return getSearchLocations(bi, being, location).size() != 0;
    }

    private List<Location> getSearchLocations(BeingInterface bi,
ConquerorBeing being, Location fromLocation) {
        int stepsCanMake = SearchUtil.stepsCanMake(being);
        List<Location> result = new ArrayList<Location>();
        List<Location> closestSeekLocations =
SearchUtil.getClosestSeekLocations(bi, fromLocation, getSearchMap());
        closestSeekLocations.removeAll(result);
        int minSteps = Integer.MAX_VALUE;
        Location closestLocation = null;
        for (Location location : closestSeekLocations) {
            int steps = SearchUtil.movesCount(fromLocation, location, being)
+ SearchUtil.movesCount(
                    location, getClosestLocationWithEnergy(bi, being,
location), being) + STEPS_TO_EAT;
            if (steps < minSteps) {
                minSteps = steps;
                closestLocation = location;
            }
        }
        if (minSteps <= stepsCanMake) {
            result.add(closestLocation);
            stepsCanMake -= SearchUtil.movesCount(fromLocation,
closestLocation, being);
        }
        return result;
    }

    private boolean hasEaters(Location location) {
        return colonies.get(location) != null &&
colonies.get(location).size() > 0;
```

```
    }

    float getEnemyMass(Integer id) {
        Float enemyMass = enemyMasses.get(id);
        return enemyMass != null ? enemyMass : 0f;
    }

    void removeDefender(ConquerorBeing being, Location location) {
        getOrCreateColony(location).removeDefender(being);
    }

    Integer getTargetEnemyId(ConquerorBeing being, Location location) {
        return getOrCreateColony(location).getTargetEnemyId(being);
    }

    float getAmountToGive(ConquerorBeing being, Location location) {
        return getOrCreateColony(location).getAmountToGive(being);
    }

    ConquerorBeing getEnergyTaker(ConquerorBeing being, Location location) {
        return getOrCreateColony(location).getEnergyTaker(being);
    }

    Location getAttackLocationForDefender(BeingInterface bi, ConquerorBeing
being, Location location) {
        return getOrCreateColony(location).getAttackLocationForDefender(bi,
being);
    }

    void joinTheColony(ConquerorBeing being, Location location) {
        getOrCreateColony(location).addBeing(being);
    }

    void leaveColony(ConquerorBeing being, Location location) {
        getOrCreateColony(location).removeBeing(being);
    }

    synchronized boolean shouldEat(BeingInterface bi, ConquerorBeing being,
Location location) {
        return getOrCreateColony(location).shouldEat(bi, being);
    }

    boolean shouldGiveEnergy(BeingInterface bi, ConquerorBeing being,
Location location) {
        return getOrCreateColony(location).shouldGiveEnergy(bi, being);
    }
}
```

EatersColony.java

```
package ru.ifmo.votinov;

import universum.bi.*;

import java.util.*;

class CorporateMind {
    private static final int STEPS_TO_EAT = 2;
    private static final int GATHERING_LOCATIONS_COUNT = 10;

    private static CorporateMind instance;

    private Map<Integer, Float> enemyMasses;
    private Map<ConquerorBeing, List<Location>> searchLocations;
    private Map<Location, EatersColony> colonies;
    private SearchMap searchMap;
    private Map<Location, Float> energyMap;
    private Map<Location, Location> closestEnergyLocation;
    private List<Location> allSeekLocations;
    private List<ConquerorBeing> beings;
    private List<Location> gatheringLocations;
    private int turn;
    private int maxTurn;

    static void createInstance(UserGameInfo info) {
        instance = new CorporateMind(info);
    }

    static CorporateMind getInstance() {
        return instance;
    }

    private EatersColony getOrCreateColony(Location location) {
        if (colonies.get(location) == null) {
            colonies.put(location, new EatersColony());
        }
        return colonies.get(location);
    }

    private CorporateMind(UserGameInfo info) {
        turn = -1;
        searchMap = new SearchMap(3);
        energyMap = new HashMap<Location, Float>();
        closestEnergyLocation = new HashMap<Location, Location>();
        allSeekLocations = new ArrayList<Location>();
        colonies = new HashMap<Location, EatersColony>();
        beings = new ArrayList<ConquerorBeing>();
        searchLocations = new HashMap<ConquerorBeing, List<Location>>();
        enemyMasses = new HashMap<Integer, Float>();
        maxTurn = info.maxTurns;
    }
```

```
    int getTurn() {
        return turn;
    }

    int getMaxTurn() {
        return maxTurn;
    }

    synchronized void checkIfNewTurnStarted(BeingInterface bi, ConquerorBeing
being) {
        if (bi.getTurnsCount() > turn) {
            turn = bi.getTurnsCount();
            handleNewTurnStarted(bi, being);
        }
    }

    private void init(Location startLocation) {
        if (!searchMap.isInitialized()) {
            searchMap.init(startLocation);
            allSeekLocations.addAll(searchMap);
            searchMap.remove(startLocation);
        }
    }

    void addBeing(ConquerorBeing being) {
        beings.add(being);
    }

    void removeBeing(ConquerorBeing being) {
        beings.remove(being);
    }

    private void handleNewTurnStarted(BeingInterface bi, ConquerorBeing
being) {
        init(bi.getLocation(being));
        updateEnergies(bi);
        Collection<EatersColony> colonies = new
ArrayList<EatersColony>(this.colonies.values());
        for (EatersColony colony : colonies) {
            colony.handleNewTurnStarted();
        }
    }

    private SearchMap getSearchMap() {
        return searchMap;
    }

    private float getEnergy(Location location) {
        Float energy = energyMap.get(location);
        return energy != null ? energy : 0;
    }
```

```
    private Location getClosestLocationWithEnergy(BeingInterface bi,
ConquerorBeing being, Location location) {
        Location closestLocation = closestEnergyLocation.get(location);
        if (closestLocation == null) {
            for (PointInfo pi : bi.getNeighbourInfo(being)) {
                closestLocation =
closestEnergyLocation.get(pi.getLocation());
                if (closestLocation != null) {
                    break;
                }
            }
        }
        return closestLocation;
    }

    Location getClosestLocationWithEnergy(BeingInterface bi, ConquerorBeing
being) {
        return getClosestLocationWithEnergy(bi, being,
bi.getPointInfo(being).getLocation());
    }

    Location getGatheringLocation(BeingInterface bi, ConquerorBeing being) {
        if (gatheringLocations == null) {
            gatheringLocations = new ArrayList<Location>(colonies.keySet());
            Collections.sort(gatheringLocations, new Comparator<Location>() {
                public int compare(Location o1, Location o2) {
                    return
((Integer)colonies.get(o2).size()).compareTo(colonies.get(o1).size());
                }
            });
            gatheringLocations = gatheringLocations.subList(0,
                    Math.min(GATHERING_LOCATIONS_COUNT,
gatheringLocations.size()));
        }
        return SearchUtil.getClosestLocation(bi, bi.getLocation(being),
gatheringLocations);
    }

    private void updateEnergies(BeingInterface bi) {
        for (ConquerorBeing being : beings) {
            being.updateEnergy();
            List<PointInfo> ni = bi.getNeighbourInfo(being);
            for (PointInfo pi : ni) {
                updateEnergies(bi, being, pi);
            }
            updateEnergies(bi, being, bi.getPointInfo(being));
        }
    }

    private void updateEnergies(BeingInterface bi, ConquerorBeing being,
PointInfo pi) {
        float count = pi.getCount(being);
        energyMap.put(pi.getLocation(), count);
```

```
            if (count > 0) {
                updateClosestEnergyLocations(bi, pi.getLocation());
            }
        }

        private void updateClosestEnergyLocations(BeingInterface bi, Location
location) {
            for (Location seekedLocation : allSeekLocations) {
                updateClosestEnergyLocations(bi, seekedLocation, location);
            }
        }

        private void updateClosestEnergyLocations(BeingInterface bi, Location
seekLocation, Location location) {
            Location closestLocation = closestEnergyLocation.get(seekLocation);
            if (closestLocation == null || bi.distance(seekLocation, location) <
bi.distance(seekLocation, closestLocation)) {
                closestEnergyLocation.put(seekLocation, location);
            }
        }

        synchronized void updateEnemyMasses(BeingInterface bi, ConquerorBeing
being) {
            for (PointInfo pi : bi.getNeighbourInfo(being)) {
                Integer[] ids = pi.getEntities(being, null);
                for (Integer id : ids) {
                    if (!bi.getOwner(being, id).equals(bi.getOwner(being,
bi.getId(being)))) {
                        float mass = bi.getMass(being, id);
                        if (mass != 0) {
                            enemyMasses.put(id, mass);
                        }
                    }
                }
            }
        }

        synchronized void stopSearching(ConquerorBeing being) {
            if (searchLocations.get(being) != null) {
                getSearchMap().addAll(searchLocations.get(being));
                searchLocations.put(being, null);
            }
        }

        Location getUnoccupiedEatingLocation(BeingInterface bi, ConquerorBeing
being) {
            Location eatingLocation = null;
            float maxCount = 0f;
            List<Location> rll = bi.getReachableLocations(being);
            for (Location rl : rll) {
                float count = getEnergy(rl);
                if (count > maxCount && !hasEaters(rl)) {
                    maxCount = count;
```

```
                eatingLocation = rl;
            }
        }
        return eatingLocation;
    }

    synchronized List<Location> getSearchLocations(BeingInterface bi,
ConquerorBeing being) {
        if (searchLocations.get(being) == null ||
searchLocations.get(being).size() == 0) {
            List<Location> locations = getSearchLocations(bi, being,
bi.getLocation(being));
            searchLocations.put(being, locations);
            getSearchMap().removeAll(locations);
        }
        return searchLocations.get(being);
    }

    synchronized boolean canSearchFromLocation(BeingInterface bi,
ConquerorBeing being, Location location) {
        return getSearchLocations(bi, being, location).size() != 0;
    }

    private List<Location> getSearchLocations(BeingInterface bi,
ConquerorBeing being, Location fromLocation) {
        int stepsCanMake = SearchUtil.stepsCanMake(being);
        List<Location> result = new ArrayList<Location>();
        List<Location> closestSeekLocations =
SearchUtil.getClosestSeekLocations(bi, fromLocation, getSearchMap());
        closestSeekLocations.removeAll(result);
        int minSteps = Integer.MAX_VALUE;
        Location closestLocation = null;
        for (Location location : closestSeekLocations) {
            int steps = SearchUtil.movesCount(fromLocation, location, being)
+ SearchUtil.movesCount(
                    location, getClosestLocationWithEnergy(bi, being,
location), being) + STEPS_TO_EAT;
            if (steps < minSteps) {
                minSteps = steps;
                closestLocation = location;
            }
        }
        if (minSteps <= stepsCanMake) {
            result.add(closestLocation);
            stepsCanMake -= SearchUtil.movesCount(fromLocation,
closestLocation, being);
        }
        return result;
    }

    private boolean hasEaters(Location location) {
        return colonies.get(location) != null &&
colonies.get(location).size() > 0;
```

```
    }

    float getEnemyMass(Integer id) {
        Float enemyMass = enemyMasses.get(id);
        return enemyMass != null ? enemyMass : 0f;
    }

    void removeDefender(ConquerorBeing being, Location location) {
        getOrCreateColony(location).removeDefender(being);
    }

    Integer getTargetEnemyId(ConquerorBeing being, Location location) {
        return getOrCreateColony(location).getTargetEnemyId(being);
    }

    float getAmountToGive(ConquerorBeing being, Location location) {
        return getOrCreateColony(location).getAmountToGive(being);
    }

    ConquerorBeing getEnergyTaker(ConquerorBeing being, Location location) {
        return getOrCreateColony(location).getEnergyTaker(being);
    }

    Location getAttackLocationForDefender(BeingInterface bi, ConquerorBeing
being, Location location) {
        return getOrCreateColony(location).getAttackLocationForDefender(bi,
being);
    }

    void joinTheColony(ConquerorBeing being, Location location) {
        getOrCreateColony(location).addBeing(being);
    }

    void leaveColony(ConquerorBeing being, Location location) {
        getOrCreateColony(location).removeBeing(being);
    }

    synchronized boolean shouldEat(BeingInterface bi, ConquerorBeing being,
Location location) {
        return getOrCreateColony(location).shouldEat(bi, being);
    }

    boolean shouldGiveEnergy(BeingInterface bi, ConquerorBeing being,
Location location) {
        return getOrCreateColony(location).shouldGiveEnergy(bi, being);
    }
}
```

SearchMap.java

```
package ru.ifmo.votinov;

import universum.bi.Location;
import universum.bi.Constants;

import java.util.ArrayList;
import java.util.Collections;

class SearchMap extends ArrayList<Location> {
    protected boolean initialized;
    private int networkStep;

    SearchMap(int networkStep) {
        initialized = false;
        this.networkStep = networkStep;
    }

    void init(Location location) {
        init(location, true);
    }

    void init(Location startLocation, boolean includeBorders) {
        int startX = startLocation.getX();
        int startY = startLocation.getY();
        int x = startX;
        while (x > 0 || (includeBorders && x == 0)) {
            init(x, startY, includeBorders);
            if (x > 0) {
                x = Math.max(0, x - networkStep);
            } else {
                x = -1;
            }
        }
        x = startX + networkStep;
        while (x < Constants.getWidth()) {
            init(x, startY, includeBorders);
            x += networkStep;
        }
        initialized = true;
    }

    private void init(int x, int startY, boolean includeBorders) {
        int y = startY;
        while (y > 0 || (includeBorders && y == 0)) {
            add(new Location(x, y));
            if (y > 0) {
                y = Math.max(0, y - networkStep);
            } else {
                y = -1;
            }
        }
```

```
            y = startY + networkStep;
            while (y < Constants.getHeight()) {
                add(new Location(x, y));
                y += networkStep;
            }
        }

        protected boolean isInitialized() {
            return initialized;
        }
}
```

SearchUtil.java

```
package ru.ifmo.votinov;

import universum.bi.Location;
import universum.bi.BeingInterface;
import universum.bi.Constants;

import java.util.List;
import java.util.ArrayList;

class SearchUtil {
    static float moveCost(ConquerorBeing being) {
        return Constants.K_movecost * being.getParams().S +
Constants.K_masscost * being.getMass();
    }

    static int movesCount(Location from, Location to, ConquerorBeing being) {
        int speed = (int)being.getSpeed();
        int dx = Math.abs(from.getX() - to.getX());
        int dy = Math.abs(from.getY() - to.getY());
        dx = Math.min(dx, Constants.getWidth() - dx);
        dy = Math.min(dy, Constants.getHeight() - dy);

        return (int)Math.ceil(((float)Math.max(dx, dy)) / speed);
    }

    static float moveCost(Location from, Location to, ConquerorBeing being) {
        return moveCost(being) * movesCount(from, to, being);
    }

    static Location stepToward(BeingInterface bi, ConquerorBeing being,
Location to) {
        Location from = bi.getPointInfo(being).getLocation();
        float speed = being.getParams().S;

        float d = bi.distance(from, to);

        if (d <= speed) {
            return to;
        }
```

```
        int x1 = from.getX(), y1 = from.getY();
        int x2 = to.getX(),   y2 = to.getY();

        // X distance
        int dx1 = Math.abs(x2 - x1);
        int dx = Math.min(dx1, Constants.getWidth() - dx1);
        // Y distance
        int dy1 = Math.abs(y2 - y1);
        int dy = Math.min(dy1, Constants.getHeight() - dy1);

        int signx = (x1 == x2) ? 0 : (dx1 == x2 - x1) ? 1 : -1;
        int signy = (y1 == y2) ? 0 : (dy1 == y2 - y1) ? 1 : -1;

        if (Constants.getWidth() - dx1 < dx1) {
            signx = -signx;
        }
        if (Constants.getHeight() - dy1 < dy1) {
            signy = -signy;
        }
        int s = (int)speed;

        int xd = Math.min(dx, s) * signx;
        int yd = Math.min(dy, s) * signy;

        return new Location(normal(x1+xd,Constants.getWidth()), normal(y1+yd,
Constants.getHeight()));
    }

    private static int normal(int x, int max) {
        return (x + max) % max;
    }

    static int stepsCanMake(ConquerorBeing being) {
        return (int)Math.floor((being.getEnergy() - Constants.K_emin *
being.getMass()) / moveCost(being));
    }

    static Location getClosestLocation(BeingInterface bi, Location location,
List<Location> variants) {
        float minDistance = Float.MAX_VALUE;
        Location closestLocation = null;
        for (Location variant : variants) {
            float distance = bi.distance(location, variant);
            if (distance < minDistance) {
                minDistance = distance;
                closestLocation = variant;
            }
        }
        return closestLocation;
    }

    static List<Location> getClosestSeekLocations(BeingInterface bi, Location
curLocation, List<Location> locations) {
```

```
        List<Location> closestLocations = new ArrayList<Location>();
        float minDistance = Float.MAX_VALUE;
        for (Location seekLocation : locations) {
            if (seekLocation == curLocation) {
                continue;
            }
            float distance = bi.distance(curLocation, seekLocation);
            if (distance < minDistance) {
                minDistance = distance;
                closestLocations.clear();
                closestLocations.add(seekLocation);
            } else if (distance == minDistance) {
                closestLocations.add(seekLocation);
            }
        }
        return closestLocations;
    }
}
```