Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего профессионального образования
«Уральский федеральный университет имени первого Президента России Б.Н. Ельцина»

Институт математики и компьютерных наук
Кафедра алгебры и дискретной математики

Система поддержки алгоритмов коллективного разума

Допущен к защите

_____

"__" _____ 2012

Квалификационная работа на степень магистра наук
по направлению «Математика. Компьютерные науки.»
студента гр. МГКН – 2
Кононучука Дмитрия Олеговича

_____

Научный руководитель
Окуловский Юрий Сергеевич
заведующий лабораторией РВИИМАП, к.ф.-м.н.

_____

Екатеринбург
2012

РЕФЕРАТ

Конончук Д.О. СИСТЕМА ПОДДЕРЖКИ АЛГОРИТМОВ КОЛЛЕКТИВНОГО РАЗУМА, квалификационная работа на степень магистра наук: стр. ТУДУ, библ. ТУДУ назв.

Ключевые слова: алгоритмы коллективного разума, система поддержки, общие свойства, универсальный алгоритм, программная модель, C#.

Рассматривается множество алгоритмов коллективного разума, выделяются их общие свойства. На их основе строится универсальный алогритм, способный эмулировать все алгоритмы коллективного разума. Разрабатывается программная модель универсального алгоритма. Рассматриваются расширения модели, направленные на улучшение удобства использования. Рассматриваются основные аспекты функционирования этой модели.

Содержание

Перечень принятых в работе сокращений

CRTP	Curiously Recurring Template Pattern
DIP	Dependency Inversion Principle
GSA	Gravitational Search Algorithm
LEM	Learnable Evolution Model
PSO	Particle Swarm Optimization
SDS	Stochastic Diffusion Search
SRP	Single Responsibility Principle
АКР	Алгоритмы Коллективного Разума
АЭМ	Алгоритмы Эволюционного Моделирования
ВИ	Вычислительный Интеллект
ГА	Генетические Алгоритмы
ИИ	Искусственный Интеллект
ИИС	Искуственная Иммуная Система
ОО	Объектно Ориентированный

Введение

Алгоритмы коллективного разума являются в настоящее время одной из самых динамично развивающихся отраслей алгоритмов искусственного интеллекта. Они основаны на т.н. «интеллекте толпы» — явлении, при котором система из множества слабо интеллектуальных, равноправных и децентрализованных сущностей проявляет свойства интеллектуальности. TODO: Написать

Глава 1

Теоретические основы

1.1. Понятия искусственного и вычислительного интеллекта

Под искусственным интеллектом понимается довольно обширный класс инструментов и технологий, границы которого во многом определяются историческими либо практическими причинами, а не общепринятыми формальными критериями, которых на данный момент не существует.

Наиболее распространенным подходом к определению интеллектуальности, а, следовательно, и ИИ , является агент-ориентированный подход, изложенный в одной из основополагающих работ в области — [1]. Ключевое понятие для этого подхода — агент. Это некоторая сущность, которая способна воспринимать окружающую среду посредством датчиков и влиять на нее посредством исполнительных механизмов. Поведение агента считается интеллектуальным, если оно рационально в том числе и при меняющихся условиях среды. Эта модель требует наличия свойства адаптивности у алгоритмов ИИ.

Тем не менее, существуют другие подходы, например логический [2] или основанный на поиске [3], ярким примером которого является алгоритм A-star [4], который в настоящее время не относится к интеллектуальным.

В своей работе я исследовал лишь методы, относящиеся к области вычислительного интеллекта [5], интеллектуальность которых общепризнанна. Однако, также как и для интеллекта искусственного, для вычислительного не было построено четкого определения. Обширный обзор разработанных определений можно найти в [6]. В этой-же работе автор приводит собственное, пожалуй, лучшее из представленных в литературе, хотя и излишне общее определение ВИ: «Вычислительный интеллект — это область компьютерных наук,

изучающая решение задач, для которых не существует эффективного алгоритмического решения».

Во многих работах, например [7], ВИ определяется, как совокупность различных технологий. Подобный остенсивный [8] способ определения понятия обладает очевидными недостатками, поскольку не указывает на общие свойства этих технологий и не является расширяемым. В частности, в книге [5] авторы относят к ВИ 4 типа алгоритмов, а 2 года спустя в [7] — уже 7. Работы, использующие для определения ВИ именно такой — остенсивный — подход, обычно содержат довольно подробную классификацию существующих методов.

Согласно этим классификациям алгоритмы ВИ разбиваются на три подкласса: нейронные сети [9], нечеткое управление [10] и эволюционное моделирование [11].

1.2. Алгоритмы эволюционного моделирования

Перейдем к более подробному рассмотрению третьего класса алгоритмов — алгоритмов эволюционного моделирования. Эти алгоритмы предназначены для решения задач комбинаторной оптимизации.

Исторически, первыми представителями этого класса были генетические алгоритмы, предложенные Лоуренсом Фогелем в работе [12]. В их основу положен принцип моделирования генетической эволюции в смысле синтетической теории эволюции [13]. В самом общем виде генетический алгоритм формулируется следующим образом:

1. Алгоритм оперирует особями — некоторым представлением решений задачи. Каждая особь однозначно определяет допустимое решение. Для каждой особи определена ее функция приспособленности — характеристика оптимальности решения ею определяемого.

2. Алгоритм поддерживает пул особей, называемый популяцией, который характеризует его текущее состояние.

3. При инициализации алгоритма некоторым образом создается начальная популяция.

4. На каждом шаге алгоритма производятся три процесса:

    а. Мутация: некоторым случайным образом выбираются несколько особей и для каждой из них в популяцию добавляется особь (или несколько особей), которая является ее некоторой случайной модификацией.

    б. Скрещивание: некоторым случайным (либо нет) образом выбираются пары особей. Для каждой из них некоторым случайным способом создаются и затем добавляются в популяцию некоторое количество производных особей.

    в. Отбор: из популяции удаляются особи имеющие плохую функцию приспособления либо слишком продолжительное время жизни (возможно сочетание критериев).

5. Шаг алгоритма повторяется до тех пор пока не выполняться определенные условия. Например, пока в популяции не перестанут происходить существенные изменения.

6. Результат работы алгоритма — особь с лучшей функцией приспособленности.

Для ГА известны способы доказательства корректности и сходимости алгоритмов [14], а также определения их эффективности [15].

Другим АЭМ, зачастую противопоставляемым генетическим, является алгоритм Learnable Evolution Model [16]. Он также работает с представле-

нием решений в виде особей с определенной функцией приспособленности и представлением текущего состояния в виде популяции. Однако в его основе лежит другая теория эволюции — теория направленной эволюции (номогенез) [17]. В самом общем виде алгоритм формулируется так:

1. Некоторым способом создается начальная популяция.
2. На каждом шаге алгоритма выполняются следующие действия:
    а. Из популяции выделяются две группы особей: «хорошие» и «плохие» — особи соответственно с высоким и низким значением функции приспособленности.
    б. С помощью методов машинного обучения строятся гипотезы — описания отличительных черт «хороших» особей, которые не наблюдаются у «плохих». Возможно также построение обратных гипотез — черт присутствующих у «плохих», но не у «хороших» особей.
    в. Генерируются новые особи, которые удовлетворяют гипотезам и не удовлетворяют обратным гипотезам.
    г. Эти особи тем или иным способом добавляются в популяцию. Например заменяя всех особей не являющихся «хорошими».
3. Шаг алгоритма повторяется до тех пор пока не выполняются определенные условия.
4. Результат работы алгоритма — особь с лучшей функцией приспособленности.

В качестве примера алгоритма АЭМ не оперирующего понятием «эволюция», можно привести алгоритм Stochastic Diffusion Search [18]. Он, как и

предыдущие алгоритмы, решает задачу поиска оптимального решения (элемента множества S) с точки зрения некоторой характеристики ($f : S \to \mathbb{R}$). В нем также вводится понятие особи, однако, в SDS особь не отождествляется с решением, а является независимым объектом, которому в каждый момент времени сопостовляют решение, т.о. может расссматриваться как частично определенная функция $a_i : T \to S$, $i \in \{1 \dots n\}$. Алгоритм формулируется следующим образом:

1. Некоторым способом особям сопоставляются начальные решения $a_i(0)$.
2. На каждом шаге алгоритма выполняются следующие действия:
   а. Для всех особей вычисляется качество их текущего решения: $f(a_i(t))$.
   б. Особи разделяются на две группы: «хорошие» и «плохие» — особи соответственно с высоким и низким качеством решения.
   в. Для каждой «плохой» особи i:
      i. Случайно выбирается особь $j \in \{1 \dots n\} \setminus \{i\}$.
      ii. Если особь j «хорошая», то $a_i(t+1) \leftarrow a_j(t)$,
      iii. иначе $a_i(t+1) \leftarrow \mathrm{Rand}(S)$.
   г. Для всех «хороших» особей $a_i(t+1) \leftarrow a_i(t)$.
3. Шаг алгоритма повторяется до тех пор пока всем особям не будет сопоставленно малое число (или ровно одно) решений:
$$\left\| \bigcup_{i \in \{1 \dots n\}} \{a_i(t)\} \right\| \to 1$$
4. Результат работы алгоритма — решение, которое сопоставленно наибольшему числу особей.

Более подробный анализ SDS с доказательством некоторых его свойств, включая временную сложность $o(n)$, приведены в [19].

На основании вышеперечисленных алгоритмов можно выделить общие черты, присущие всем АЭМ:

1. Алгоритм решает задачу оптимизации функции (называемой также функцией качества решения, функцией приспособленности) $f : S \to \mathbb{R}$, определенной на множестве значений (называемом также множеством решений) S. Чаще всего множество S велико, а функция f является трудно вычислимой.

2. Состояние алгоритма представлено некоторым множеством (популяцией) некоторых объектов (особей) и, возможно, производными данными. Состояние алгоритма однозначно определяет набор решений — выбранных значений из множества S.

3. Алгоритм итеративный. Цель каждой итерации — увеличить максимальное значение функции приспособленности для решений, определенных текущим сосотоянием. Фактически, провести эволюцию популяции.

4. Начальное состояние дается алгоритму извне — обычно генерируется случайно.

5. На каждой итерации алгоритм многократно (например для каждой особи, либо фиксированное число раз) выполняет некоторые действия. Причем действия эти независимые и замкнутые относительно особей (т.е. состояние особи не может быть изменено дважды в процессе обработки). В некоторых алгоритмах также выделяются стадии пред- и постпроцессинга.

6. Алгоритм стохастический.

Особо отметим, что вышеприведенные алгоритмы формируют более слабые ограничения для пунктов (2) и (5). Основываясь лишь на них, мы можем утчерждать, что во-первых «Особь однозначно определяет одно решение», и во-вторых, «На каждой итерации алгоритм выполняет некоторые действия для каждой особи». Однако существуют примеры АЭМ, которые не вписываются в эти, более узкие, рамки. Например, расширение пункта (2) необходимо для включения в класс АЭМ алгоритмов муравейника (англ. Ant Colony Optimisation) [20] (решение в них кодируется не особью, а производными данными), а пункта (5) — для включения алгоритма гармонического поиска [21].

Перечисленные свойства не являются определяющими для класса АЭМ, однако, они наблюдаются у всех общеизвестных алгоритмов.

Стоит заметить, что эти свойства являются весьма существенными с точки зрения программных реализаций алгоритмов. Они определяют задачи, которые должны быть решены в каждой из них. Это, в первую очередь, задачи создания удобной программной абстракции, выполнения сервисных операций с популяцией, распараллеливания и распределения по данным вычислений на каждой итерации. Их решение довольно трудоемко, что приводит к мысли о создании специализированных библиотек поддержки. Подобные библиотеки существуют (в качестве примера можно привести EO Evolutionary Computation Framework [22] и ECF [23]), однако, ввиду чрезмерной общности данных свойств и их неестественности, такие библиотеки недостаточно удобны в использовании.

Неестественность общих свойств АЭМ во многом объясняется тем, что данный класс состоит из двух подклассов, представители которых существенно различны. Это подклассы алгоритмов эволюционных вычислений [24] и коллективного разума [25]. У каждого из них имеется гораздо больше общих свойств, и эти свойства гораздо естественнее, благодаря чему создание библиотеки поддержки становится более целесообразным.

В своей работе я сфокусировался на разработке подобной библиотеки для класса алгоритмов коллективного разума.

1.3. Алгоритмы коллективного разума

Исторически первым представителем класса алгоритмов коллективного разума (англ. Swarm Intelligence Algorithms) является алгоритм Particle Swarm Optimization [26] опубликованый в 1942 г. Однако, непосредственно термин АКР а также ключевые свойства данного класса алгоритмов впервые были сформулированы только в 1989 г. в статье [27].

Основополагающей работой для всего класса класса АКР, во многом определившей его вид, считается [20]. В ней описывается уже упоминавшийся ранее алгоритм муравейника для решения задачи коммивояжёра [28]. Позже, благодаря своей обобщенности, этот алгоритм был адаптирован для решения многих других задач, примеры которых можно найти в [29]. Рассмотрим алгоритм муравейника в формулировке для решения задачи поиска кратчайшего пути на графе.

Дан большой взвешенный граф G. $\eta_{i,j}$ — величина, обратная весу ребра (i, j). Необходимо найти кратчайший путь между выделенными вершинами A и B.

Алгоритм муравейника, как и другие представители класса АЭМ, оперирует популяцией особей («агентов» в терминологии АКР; «муравьев» в терминологии данного алгоритма) и дополнительными данными: «феромонами» ($\tau_{i,j}(t)$), позволяющими ассоциировать с ребрами графа G значения, указывающие на историю его использования муравьями. Алгоритм формулируется следующим образом:

1. Изначально все муравьи находяться в вершине A. Полагаем $\forall(i, j) \in G$: $\tau_{i,j}(0) = 0$.

2. На каждом шаге алгоритма для каждого муравья ant_k, $k \in \{1 \dots n\}$ расположенного в вершине i производятся следующие действия:

   а. Вычисляется вероятность перехода муравья в каждую из допустимых вершин. Общая формула:

$$\forall j \mid (i,j) \in G : \; p_{i,j}^{k}(t) = \frac{\left(\tau_{i,j}(t)\right)^{\alpha} \left(\eta_{i,j}\right)^{\beta}}{\sum\limits_{l \in N} \left(\tau_{i,l}(t)\right)^{\alpha} \left(\eta_{i,l}\right)^{\beta}}$$

   б. Выполняется случайный переход муравья по ребру $(i,\ j)$.

   в. Ребро j записывается в конец «пройденного пути» муравья — L_k.

3. Для всех муравьев k, таких что $\text{Last}(L_k) = B$, выполняем:

   а. Обновляем уровнь ферромонов на всем пройденом муравьем пути:

$$\Delta\tau_{i,j}^{k} = \begin{cases} q / \|L_k\|\,, & \text{если } (i,j) \in L_k; \\ 0, & \text{иначе.} \end{cases}$$

   где q — константа.

   б. Переносим муравья в A и сбрасываем пройденный им путь.

4. Пересчитываем состояние феромонов:

$$\tau_{i,j}(t+1) = \rho\tau_{i,j}(t) + \sum_{k=1}^{n} \Delta\tau_{i,j}^{k}$$

   где $0 \leqslant \rho < 1$ — константа.

5. Шаг алгоритма повторяется до тех пор, пока в графе не образуется путь помеченный «высоким уровнем» феромонов: $\forall (i,j) \in \text{Path} : \tau_{i,j}(t) \geqslant \Gamma$.

6. Результат работы алгоритма — путь помеченный высоким уровнем ферромонов.

Этот алгоритм был разработан на основании анализа поведения колонии муравьев. Стоит отметить, что подобный натуралистический подход часто применяется при разработке алгоритмов АКР. Например, так были разработаны алгоритм пчелиной колонии (англ. Bee Colony Algorithm) [30] и алгоритм светлячка (англ. Firefly Algorithm) [31]. Во многом именно аналогичное происхождение объясняет схожесть свойств алгоритмов АКР.

Однако, было бы неверно утверждать, что все алгоритмы АКР базируются на принципах почерпнутых из живой природы. Вышеупомянутый PSO причисляется к классу АКР, но, очевидно, являются полностью искуственными. Так-же существует целый подкласс АКР, базирующихся на принципах неживой природы. Типичным его представителем является алгоритм гравитационного поиска (англ. Gravitational Search Algorithm) [32]. Рассмотрим его подробнее.

Задача алгоритма — в пространстве задачи $\mathcal{X}$ найти вектор X с минимальным значением функции $f : \mathcal{X} \to \mathbb{R}$. Агенты в этом алгоритме обладают координатой $X_i(t) \in \mathcal{X}$, скоростью перемещения в пространстве решений — $V_i(t)$ и массой — $M_i(t)$. Для агента определена его оценка — функция качества решения им определенного $f_i(t) = f(X_i(t))$. Алгоритм формулируется следующим образом:

1. Некоторым способом агентам присваиваются начальные координаты $X_i(0)$.

2. На каждом шаге алгоритма для каждого агента $i \in \{1 \dots n\}$ выполняются следующие действия:

    a. Вычислить силы взаимодействия с каждым другим агентом $j : j \neq i$ по формуле:

$$F_{ij}(t) = G(t)\frac{M_i(t)M_j(t)}{R_{ij}(t)}\Big(X_i(t) - X_j(t)\Big)$$

где $R_{ij}(t) = \left\| X_i(t),\, X_j(t) \right\|_2$ — евклидово расстояние.

Это правило является видоизмененным законом всемирного тяготения. Отличия заключаются, во-первых, в том, что множитель $R_{ij}(t)$ имеет степень -1, а не -3, а во-вторых, в том что гравитационная постоянная G введена как функция от времени. Первое изменение было выведено авторами экспериментально — так алгоритм быстрее сходился. Второе является следствием поисковой природы алгоритма — для нахождения точных решений необходимо повышать «аккуратность» алгоритма со временем, для чего необходимо уменьшать силы взаимодействия.

б. С помощью независимых случайных величин $rnd_i \in [0, 1]$, вычисляется результирующая сила и ускорение:

$$F_i(t) = \sum_{j \in \mathrm{Kbest},\, j \neq i}^{N} rnd_j F_{ij}(t)$$

где Kbest $\subset \{1 \dots n\}$ — множество K лучших по оценке $f_i(t)$ агентов.

$$a_i(t) = \frac{F_i(t)}{M_i(t)}$$

Это, соответственно, правило сложения сил, в которое добавили элемент случайности, и второй закон Ньютона.

в. Производится перемещение агента:

$$V_i(t+1) = rnd_i V_i(t) + a_i(t)$$

$$X_i(t+1) = X_i(t) + V_i(t+1)$$

$$f_i(t+1) = f(X_i(t+1))$$

где rnd_i — независимые случайные величины на $[0, 1]$.

3. Пересчитываются массы объектов:

$$w(t) = \min_{i \in \{1...n\}} f_i(t)$$

$$b(t) = \max_{i \in \{1...n\}} f_i(t)$$

$$m_i(t) = \frac{f_i(t) - w(t)}{b(t) - w(t)}$$

$$M_i(t+1) = \frac{m_i(t)}{\sum_{j=1}^{n} m_j(t)}$$

4. Шаг алгоритма повторяется определенное количество раз.

5. Результат работы алгоритма — $X_i(t)$, где $i : \arg\max_{i \in \{1...n\}} f_i(t)$.

На основании вышеперечисленных алгоритмов можно составить представление об общих свойствах класса АКР. Большинство из них унаследованы от класса АЭМ, однако, есть одно специфичное: на каждом шаге алгоритма каждому его агенту однозначно сопоставляется координата некоторого пространства. Т.о. мы можем говорить, что в АКР алгоритмах «агенты располагаются в пространстве». Изменение сопоставленной агенту координате на одной итерацией алгоритма будем называть «перемещением агента».

Очевидно, что вышеописанное свойство является достаточно важным с точки зрения програмной реализации АКР, поскольку решение задач «расположить в пространстве» и «переместить по пространству» довольно трудоемко, особенно с учетом того, что пространство может быть бесконечным и обладать сложной топологией.

Существующие на данный момент системы поддержки АКР не учитывают этого свойства. Некоторые из них, например, JADE [33] или MASS [34] рассматривают агентов без привязки к определенному пространству. Большинство же рассчитаны на работу только в конечном двух- или трехмерном пространстве (таковы Repast [35], breve [36]). Кроме того, ни одна из известных систем не совместима с приложениями на платформе .NET [37].

Глава 2

Построение универсального алгоритма коллективного разума

2.1. Основные подклассы АКР

Как уже было сказано ранее АКРы достаточно часто разрабатывались по образу и подобию некоторой реально существующей и проявляющей желаемые свойства системы. Фактически, из всех АКР могут быть четко выделены несколько групп по критерию происхожднения:

1. Алгоритмы, моделирующие поведение животных (насекомых, птиц и т.п.), чаще всего стайных/роевых. К этой группе можно отнести уже упомянутые алгоритмы муравейника, улья и многих других. В том числе, алгоритм искуственных иммуных систем [38], не смотря на то, что в нем моделируется поведение элементарных и несамостоятельных биологических объектов — клеток имунной системы. Алгоритм SDS также может быть отнесен к этой группе, т.к. изначально он основывался на моделировании поведения группы людей.

2. Алгоритмы, моделирующие физические процессы. К этой группе принадлежат уже упомянутый алгоритм GSA, алгоритм Charged System Search [39] и другие.

3. Полностью искуственные алгоритмы. К этой группе относится PSO и многие его производные.

Подобная классификация, не смотря на ее однозначность и полноту, не представляет никаакого практического интереса, поскольку, как мы покажем

далее, алгоритмы, принадлежащие одной группе согласно этой классификации, могут иметь абсолютно разные свойства.

Попробуем разделить класс АКР на группы, основываясь на более практически значимых атрибутах, чем происхождение. Для этого рассмотрим некоторые общие свойства алгоритмов этого класса.

В разделе 1.3 было выделено важнейшее общее свойство АКР — наличие пространства, в котором раполагаются агенты. На основании вида этого пространства мы можем ввести следующую классификацию АКР:

1. Алгоритмы, в которых агенты располагаются в пространстве решений задачи. Например, алгоритм улья.

2. Алгоритмы, в которых агенты располагаются в пространстве задачи. Например, алгоритм муравейника.

Очевидно, что с точки зрения програмной реализации, алгоритмы этих двух типов существенно различаются. Алгориты первого типа работают в очень большом, потенциально бесконечном пространстве, которое не может целиком быть представленно в памяти приложения. Тогда как алгоритмы второго типа чаще всего (хотя и из этого правила существуют исключения) работают в ограниченом пространстве, все элементы которого могут быть загружены в память.

Следующим критерием, существенно влияющим на практическую реализацию алгоритма, является вид внешних по отношению к агентам данных, с которыми работает алгоритм. Можно выделить три их типа:

1. Общая память — небольшой (полиномиально ограниченный числом агентов) набор некоторых данных, доступных всем агентам. В простейшем случае — набор нескольких переменных (применяется, например, в PSO). Однако, возможен и более сложный сценарий: данные — это

буфер для организации широковещательных сообщений (применяется в алгоритме ИИС).

2. Пространственная память — данные ассоциированые с точками пространства. Используется в алгоритме муравейника.

3. Отсутствие дополнительных данных. Этот тип встречается чаще всего: например, в SDS, алгоритме улья и т.д.

Возможно выделить еще два формальных свойства АКР: локальность и независимость агента. Т.е агенты в алгоритмах бывают:

1. Локальными. Такие агенты могут взаимодействовать (читать/модифицировать ассоциированные данные, данные расположеных агентов, перемщаться) с точками в небольшой окрестности своего текущего положения в пространстве. Таким свойством обладают агенты алгоритма муравейника, ИИС и другие.

2. Нелокальными. Таковы, например, агенты в PSO и SDS.

1. Зависимыми. т.е. состояние агента зависит от состояния других агентов напрямую. Например, агенты в SDS, PSO, GSA.

2. Независимыми, т.е. каждый конкретный агент может взаимодействовать с окружающими лишь опосредованно — через разделяемые данные. Например, агенты в ИИС.

Стоит отметить, что в алгоритмах с независимыми агентами агенты все равно взаимодействуют друг с другом. Если бы подобного взаимодействия не было, то из АКР пропал бы синергетический эффект большого числа вычислителей. Фактически, алгоритм просто выполнял бы некоторый сценарий поведения (сценарий одного агента) много раз и выбирал лучший результат.

Значит, он был бы эквивалентен многократно повторенному стохастическому алгоритму, тело которого совпадает со сценарием агента.

Почему же, не смотря на вышесказанное, агенты называются «независимыми»? Дело в том, что обработка каждого агента такого типа не зависит от других агентов в системе: не имеет значения сколько их и в каких состояниях они находятся. В алгоритмах с подобным типом агентов, на самом деле, даже нет требования синхронного выполнения одной итерации, т.е. пока одни агенты производят вычисления i-ый раз, другие уже могут рассчитывать $(i + k)$-ый.

При програмной реализации независимых агентов может возникнуть соблазн сделать выполнение сценария каждого из агентов идеально распараллеливаемым ([40]), однако, подобный подход неизбежно приведет к некорректной работе. Дело в том, что сценарий агента формулируется с алгоритмической точки зрения, следовательно, он чувствителен к состоянию гонки ([41]). Одним из способов решить эту проблему являться использование механизма транзакций ([42]).

2.2. Анализ иерархии подклассов АКР

С точки зрения необходимости построения универсального АКР, важным является вопрос анализа сводимости алгоритмов, относящихся к одному подклассу, к алгоритмам из другого подкласса. Действительно, допустим существует иерархия сводимости подклассов АКР:

A
↓
B

Тогда, очевдино, универсальный АКР может принадлежать только подклассу *B*. Соотвтственно, проанализировав иерархию подклассов мы сможем опреде-

лить набор свойств, которыми будет обладать универсальный АКР.

Во-первых, рассмотрим классификацию по свойству локальности агентов. Очевидно, что все АКР с локальными агентами являются также АКР с нелокальными. Покажем, что обратного включения, как и сводимости, нет.

Пусть есть АКР с пространством $\mathcal{X}$, определенном сложновычислимой монотонной и неограниченной функцией $f : \mathbb{R} \to \mathcal{X}$, при выполнении некоторых условий (напрмиер, при эвристическом признании текущей позиции безперспективной) агент алгоритма осуществляет переход в точку $f(1/\mathrm{Rand}(0,1))$. Очевидно, что в случае такого перехода, агент переместиться в точку лежащую сколь угодно далеко от его текущей позиции и, следовательно, не попадающую ни в какую, сколь угодно большую, ограниченную окрестность текущей позиции.

Во-вторых, рассмотрим классификацию по типу используемой памяти. Не требует доказательство тот факт, что алгоритмы без памяти сводимы к двум другим подклассам АКР (с общей памятью, с пространственной памятью), причем, обратного сведения не существует. Рассмотрим подробнее возможности взаимного сведения оставшихся двух подклассов.

Заметим, что в алгоритме с пространственной памятью с n агентами после m итераций потенциально были записаны данные в $\Theta(m*n)$ точек простанства, следовательно, в худшем случае все эти $\Theta(m * n)$ значений необходимо хранить (если они абсолютно случайны и, потому, несжимаемы). Попробуем построить алгоритм с общей памятью, который сможет это делать. В любом таком алгоритме в любой момент времени можно хранить не более чем $p(n)$ значений. Т.к. $\exists M : \Theta(M * n) > p(n)$ значит, начиная с некоторой итерации M, алгоритм с общей памятью не сможет хранить данные исходного. Это означает, что алгоритмы с общей памятью не сводимы к алгоритмам с пространственной памятью.

Покажем, что обратное сведение возможно. Построим по данному АКР с общей памятью эквивалентный нелокальный АКР с пространственной памятью. Пусть память исходного алгоритма ограничена $p(n)$, тогда всю ее можно представить как одну структуру с полями $1 \dots p(n)$, или, выражаясь формально, как один символ из алфавита $\Upsilon : ||\Upsilon|| = ||\Sigma||^{p(n)}$, где Σ — алфавит данных исходного алгоритма. Этот один символ можно записать в данные одной, заранее выделенной, точки пространства, к которой будут иметь доступ все агенты производного АКР, поскольку он является нелокальным. Следоавтельно, производный алгоритм сможет полностью эмулировать работу с памятью исходного.

В-третьих, рассмортим классификацию по свойству независимости агентов. Очевидно, что любой независимый агент может считаться зависимым. Докажем более интересный факт: любой исходный АКР с зависимыми агентам может быть сведен к АКР с общими данными и независмыми агентами. Схема сведения проста: изменим программу исходных агентов так, чтобы по окончанию итерации, они записывали полную информацию о своем состоянии в общие данные алгоритма (каждому агенту в них отведена ровно одна ячейка $\Rightarrow$ свойство ограниченности общего количества ячеек выполняется). Благодаря этому каждый агент вместо того, чтобы взаимодействовать с другими агентами, может просто читать информацию о них из общих данных.

На основании вышеуказанных построений можно сделать вывод, что универсальный АКР будет обладать свойством нелокальности и пространственной памятью. На свойство зависимости/независимости его агентов никаких теоретических ограничений не накладывается.

2.3. Построение универсального АКР

Построим алгоритм, основанный на общих свойствах класса АКР:

1. Алгоритм работает в некотором адресуемом пространстве (карта), которое определяется следующим образом: $M = (V, E)$ — карта, если V — множество точек карты, $E \subseteq V \times V$ — отношение «существование перехода».

2. С каждой точкой карты и с каждым переходом ассоциированы некоторые данные.

3. В каждой точке карты расположено некоторое множество (возможно пустое) агентов.

4. Агенты обладают типом, сценарием и набором данных. Агенты одного типа имеют одинаковый сценарий и одинаковый набор данных (в смысле семантики данных, а не их значений).

5. Сценарий агента — это алгоритм, допускающий следующие действия:

    а. чтение данных текущего агента;

    б. чтение данных из точек и переходов карты;

    в. запись измененных данных точек и переходов карты;

    г. перемещение текущего агента в другую точку карты;

    д. изменение данных текущего агента.

6. Однократное выполнение сценариев всех агентов на карте будем называть ходом. Работа алгоритма сводится к многократному выполнению ходов.

7. Условие остановки алгоритмом не специфицируется.

8. Начальное состояние данных дается алгоритму извне, часто все данные изначально считаются нулевыми. Точки, в которых изначально расположены агенты, обычно выбираются случайно (реже для этого используется эвристика).

9. Алгоритм стохастический.

Приведенный алгоритм, очевидно, является АКР с пространственной памятью и нелокальными и независимыми агентами, т.е. он удовлетворяет требованиям, которые мы сформулировали для универсальным АКР. Покажем, что он будет универсальным.

Для любого данного АКР с помощью построений приведенных в 2.2 можно построить эквивалентный АКР с нелокальными и независимыми агентами и пространственной памятью. После чего заметим, что в каком бы пространстве (в том числе на графах) получившийся алгоритм не был сформулирован, это пространство попадает под определение из (1). Выоплнимость (5) рассмотрим подробнее.

Поскольку АКР в целом является замкнутой системой, следовательно, в сценарии агента нет изменений некоторого внешнего состояния, а могут быть только: изменения состояни данных текущего агента, карты и дургих агентов, а также перемещение текущего и других агентов в пространстве. Но заметим, что для АКР выполняется унаследованое от АЭМ свойство замкнутости действий одного хода относительно агентов (см. свойства АЭМ на стр. 11), следовательно, изменение одним агентом состояния другого агента (в т.ч. перемещение) недопустимо, т.к. тогда его состояние будет изменено более одного раза за ход.

Таким образом, любой АКР может быть выражен, причем довольно естественно, в терминах подобного универсального АКР, что делает его идеальной моделью для использования в системе поддержки алгоритмов коллективного разума: нам достаточно реализовать поддержку только необходимых универсальному АКР механизмов и мы автоматически получим возможность реализации всех существующих АКР на нашей системе.

2.4. Расширения универсального АКР

TODO: тут про операции добавления/удаления агентов, добавление локальности и практическое применение

TODO: достижимость точек в смысле существования переходов и порожденные этим ограничения

Глава 3

Практическая реализация

3.1. Требования к системе поддержки

Целью работы была разработка системы поддержки произвольных АКР. Как уже было сказано в разделе 2.3 в качестве модели, диктующей требования для этой системы, естественно было бы использовать универсальный АКР. Также, с позиции увеличения области применимости системы, было решено при разработке учитывать расширения модели описанные в разделе 2.4.

Поскольку планируется будущее использование системы поддержки АКР в работе лаборатории РВИиМАП, в том числе в рамках спецкурса Нейронные Сети, то к ее реализации были предъявлены некоторые дополнительные требования.

- Система должна быть написана на языке C# и работать на платформе .NET, поскольку остальные разработки лаборатории выполняются с помощью этих технологий.

- Необходимо обеспечить низкий порог вхождения, посокльку система будет использоваться студентами. Следовательно, используемый способ описания АКР должен быть выразительным и простым в изучении.

- Система должна позволять решать стандартые задачи минимальными усилиями, поскольку планиеруется ее использование в учебном процессе. Это, помимо вышеперечисленного, требует наличия дополнительных средств, облегчающих решение типовых задач. В идеале небольшие задачи-примеры должны занимать всего несколько строк.

- Система должна быть легко расширяемой и, при этом, показывать неплохую производительность.

На основании вышепревиденных требований было принято решение, что все компоненты АКР будут, в рамках системы поддержки, описываться непосредственно на языке C#. Подобный подход обеспечивает низкий порог вхождения, простоту интеграции с другими решениями и легкость в расширении системы.

3.2. Представление карты в програмной модели

Для разработки системы необходимо, в перевую очередь, четко определить, чем будет являться модель АКР в тех же терминах, в которых будет описана и сама система, т.е. в терминах объектно-ориентированной ([43]) архитектуры ([44]). Сформулировать из каких компонентов она будет сосотоять, и как они будут взаимодействовать.

В первую очередь необходимо определить самое базовое понятие любого АКР — карту. Напомним, что карта $M = (V, E)$, где V — множество точек карты, $E \subseteq V \times V$ — отношение «существование перехода» (может рассматриваться как множество переходов — пар $(v_1, v_2) \in E$). В АКР карта выполняет одновременно четыре роли: во-первых, точки карты являются способом адресации пространства, используемым для описания алгоритмов агентов и АКР в целом; во-вторых, карта обеспечаивает топологию пространства, ограниченвая возможности для перемещения/взаимдействия агентов; в-третьих, в точках карты располагаются агенты; и, в-четвертых, точки и переходы карты являются носителями данных.

В нашей реализации все функции карты будут разнесены в разные сущности, как того требует Single Responsibility Principle [45]. Кроме того, подобное разделение эффективно в связи с абсолютно разными сценариями использо-

вания разных ролей карты. Например, точка карты, как координата, должна быть строго неизменяемой ([46]), легковестной и клонируеммой сущностью, поскольку активно используется в вычислениях и адресациях. В тоже время очевидно, что точка как вместилище агентов или данных таковыми свойствами обладать не должна.

Первой роли карты — адресации пространства, мы сопоставим сущность называемую координатой. Координата — это адрес одной точки карты. В системе она будет отражена следующим интерфейсом:

```
public interface ICoordinate<TCoordinate>: IEquatable<TCoordinate>, ICloneable<TCoordinate>
    where TCoordinate: ICoordinate<TCoordinate>
{
}
```

Интерфейс поддерживает основные операции, допустимые над координатами: сравнение двух координат на равенство и копирование. Другие возможные свойства координат, такие как: сравнение, вычисление расстояния, алгебраические операции — не включены в интерфейс, поскольку не являются достаточно общими (не во всех картах эти операции имеют смысл) либо не используются в системе.

По приведенному фрагменту кода видно, что в базовой архитектуре приложения заложены принципы активного использования обобщенных типов [37] и техники параметрического наследования (англ. curiously recurring template pattern) [47], на них основаной. Подробнее причины использования этих подходов в сиситеме будут описаны в разделе 3.3.

Продолжим рассмотрение програмных сущностей, соответствующих теоретическому понятию «карта». Следующая роль — обеспечение топологии пространства. В системе она будет отражена следующим базовым классом:

```
public abstract class Topology<TCoordinate>
    where TCoordinate: ICoordinate<TCoordinate>
```

```
{
    // реализация методов опущена для краткости
    public abstract bool Lays(TCoordinate coord);
    public abstract IEnumerable<TCoordinate> GetSuccessors(TCoordinate coord);
    public abstract IEnumerable<TCoordinate> GetPredecessors(TCoordinate coord);
    public virtual bool Lays(Edge<TCoordinate> edge) {}
    public virtual IEnumerable<TCoordinate> GetAdjacent(TCoordinate coord) {}
    public IEnumerable<Edge<TCoordinate>> GetOutgoing(TCoordinate coord) {}
    public IEnumerable<Edge<TCoordinate>> GetIncoming(TCoordinate coord) {}
    public IEnumerable<Edge<TCoordinate>> GetAdjacentEdges(TCoordinate coord) {}
}
```

Топология обладает парными методами для `TCoordinate` и `Edge<TCoordinate>`, который является предсталением перехода. Топология позволяет проверять лежит ли данная точка/переход на карте; получить по точке список последующих точек (таких, в которые из нее можно перейти) и наоборот — предшествующих.

Третья роль карты — размещение агентов в ее точках. Это роль существенно более комплексная, поэтому будет описана отдельно в разделе 3.6.

Четвертая роль — размещение данных в точках и переходах карты. Она реализуется посредством двух интерфейсов: слоя данных точек и переходов соответственно.

```
public interface IEdgesDataLayer<TCoordinate, TEdgeData>:
        ICompleteMapping<Edge<TCoordinate>, TEdgeData>
    where TCoordinate: ICoordinate<TCoordinate>
{
    Topology<TCoordinate> Topology { get; }
}

public interface INodesDataLayer<TCoordinate, TNodeData>:
        ICompleteMapping<TCoordinate, TNodeData>
  where TCoordinate: ICoordinate<TCoordinate>
{
    Topology<TCoordinate> Topology { get; }
}
```

Общий предок этих интерфейсов `ICompleteMapping<TKey,TValue>` — это интерфейс, описывающий всюду определенное отображение из ключей типа

TKey в значения типа TValue. Интерфейс имеет следующий вид:

```
public interface ICompleteMapping<in TKey, TValue>
{
    TValue Get(TKey key);
    void Set(TKey key, TValue value);
}
```

То есть слой данных: во-первых, привязан к топологии карты (это нужно для обеспечения корректных внешних контрактов класса, см. раздел 3.4) и, во-вторых, умеет по произвольной точке/переходу карты возвращать либо устанавливать ассоциированные с ним данные. Заметим, что реализовать такой интерфейс просто как поверх поставщиков статических данных, напрмер таковым будет слой данных, читающий данные из двумерного изображения и выбрасывающий исключение NotSupportedException при попытке записи в него, так и поверх обычных хранилищ данных, снабженных операцией генерации начального значения данных в точке.

3.3. Использование обобщенных типов в системе

По выдержкам кода, приведенным в разделе 3.2, можно заметить, что в системе активно исполльзуются обобщенные типы. Фактически, почти вся система параметризована тремя тип-параметрами: TCoordinate, TNodeData, TEdgeData. Эти параметры определяют, соответственно, тип координаты, используемой в системе, тип данных ассоциированых с точками пространства и тип данных ассоциированый с переходами.

Использование обощенных типов дает несколько существенных преимуществ. Покажем их на примере.

Пусть у нас есть интерфейс описывающий некий стандартный компонент системы, реализация которой будет задаче-специфична (таковым будет, в

частности, компонент представляющий агента). Публичный интерфейс этого компонента почти наверняка будет использовать передачу координат. Пользуясь стандартным техникой dependency inversion [48] мы могли бы получить следующий интерфейс компонента:

```
public interface IComponent
{
    void Method(ICoordinate coord);
}
```

Теперь попробуем представить себе его реализацию. Т.к. компонент задаче-специфичный, то, очевидно, что он будет разрабатываться с учетом условий конкретной задачи, в том числе, с рассчетом на работу только в пространстве фиксированого вида и, следовательно, с фиксированными типом координат. Значит, его код будет иметь следующий вид:

```
public class Component: IComponent
{
    public void Method(ICoordinate coord)
    {
        var point = (Coordinate2D) coord;
        // некоторый код
    }
}
```

Такой код обладает сразу несколькими недостатками. Он слаботипизированый ([49]), следовательно ошибки связанные с неправильным использованием компонента не будут выявлены компилятором во время проверки типов, а обнаружатся только во время исполнения. Он основывается на неявных контрактах, т.е. формально на уровне языка никак не отражено, что этот компонент может работать только в системах с координатой типа `Coordinate2D`. Более того, требование к типу координат невозможно отразить даже с помощью средств контрактного программирования ([50]), поскольку эти требования в терминах контрактов являются предусловиями, а предусловия могут

быть определены только в объясвлении высшего уровня (в нашем случае — в интерфейсе `IComponent`) дабы не противоречить принципу Лисков [51].

В дополнение ко всему вышеперечисленному, приведенный код имеет проблемы с производительностью. Дело в том, что операция приведения типов на платформе .NET имеет достаточно высокие накладные расходы [52], а в приведенном примере она будет проводиться по крайней мере дважды. Но, что более существенно, поскольку чаще всего координаты — это преддставляемые структурами примитивные значения, то операция приведения типов будет приводить к успаковке/распаковке значений [53], накладные расходы которых огромны.

Вышеперечисленных проблем можно избежать, если представить интерфейс и реализацию компонента в следующем виде:

```
public interface IComponent<TCoordinate>
    where TCoordinate: ICoordinate
{
    void Method(TCoordinate coord);
}

public class Component: IComponent<Coordinate2D>
{
    public void Method(Coordinate2D coord)
    {
        // некоторый код
    }
}

public class AbstractComponent<TCoordinate>: IComponent<TCoordinate>
    where TCoordinate: ICoordinate
{
    public void Method(TCoordinate coord)
    {
        // некоторый код
    }
}

public class OtherComponent
{
    public void Method2<TCoordinate>(TCoordinate coord)
        where TCoordinate: ICoordinate
```

```
    {
        // некоторый код
    }
}
```

При подобном подходе возможно, во-первых, явно указывать, что все методы компонента работают с координтатой одного типа, что актуально для большинства компонентов; во-вторых, возможно указывать, что конкретный компонент может работать только с конкретным типом координат; в-третьих, не возникает проблем при написании абстрактных компонентов, которые могут работать с любыми типами координат (`AbsctractComponent<TCoordinate>` в примере), и компонентов, методы которых могут работать с разными типами координат (`OtherComponent` в примере). Кроме того, благодаря механизму реализации обощенных типов в .NET, во всех вышеприведенных случаях генерируется оптимальный для каждой ситуации код без приведения типов и успаковки/распаковки.

Недостатками вышеописанного подхода можно считать необходимость написания дополнительного кода в декларации каждого типа и накладные расходы на первый вызов обобщенных типов и методов, связанные с механизмом конкретизации обобщенных типов в .NET.

При проектировании системы, после анализа вышеприведенных недостатков и преимуществ использования тип-параметров, было решено использовать их только для базовых компонентов, которые могут быть представлены структурами, т.е. для координат и данных ассоциированых с точками и с переходами пространства.

Еще одной техникой используемой при проектрировании системы была CRTP [47]. Ее использование приводит к появлению следующих контструкций в описании интерфейсов:

```
public interface ICoordinate<T>
```

```
    where T: ICoordinate<T>
{
    // некоторые объявления
}
```

Единственный способ корректно реализовать подобный интерфейс следующий:

```
public struct GraphCoordinate: ICoordinate<GraphCoordinate>
{
    // реализация
}
```

В чем преимущество использования этой техники? Во время реализации системы оказалось, что достаточно много компонентов могут быть реализованы (полностью или частично) без привязки к конкретному типу координат. В коде подобных компонентов часто было нужно проводить простейшие операции над координатами, а именно — сравнение на равенство. Стандартный способ сделать сравнение, это вызвать метод bool Equals(object other), определенный для каждого объекта. Однако, можно заметить, что при таком подходе мы получаем те-же самые недостатки, как и при описании компонентов без использования обобщенных типов: невозможно указать, что координаты можно сравнивать только с координатами такого-же типа; невозможно указать, что в реализации `ICoordinate` обязательно нужно переопределять операцию сравнение; необходимость писать лишний код в реализации сравнения; существенные проблемы с производительностью.

Техника CRTP позволяет избежать этих проблем. Мы можем определить интерфейс следующим образом:

```
public interface ICoordinate<T>
    where T: ICoordinate<T>
{
    bool Equals(T other);
}
```

Тогда в каждом из его реализаций необходимо будет реализовать метод `Equals`, принимающий в качестве аргумента объект с типом равным типу самой реализации:

```
public struct GraphCoordinate: ICoordinate<GraphCoordinate>
{
    public bool Equals(GraphCoordinate other)
    {
        // некоторый код
    }
}
```

Очевидно, подобный подход решает все вышеописанные проблемы.

3.4. Использование контрактов в системе

По мере реализации системы стало понятно, что в ее компонентах возникает очень большое число ограний на входные параметры методов. В качестве самого часто встречающегося примера можно привести проверку допустимости координаты: т.е. проверку того, что переданая координата вообще лежит в текущей карте.

В результате система начала содержать слишком много проверочного кода, котрый во-первых затруднял чтение исходников и, во-вторых, оказывал негативное влияние на производительность системы. Действительно, таже операция проверки принадлежности точки карте может быть достаточно нетривиальной.

В качестве решения этой проблемы было решено использовать библиотеку контрактного программирования для платформы .NET — CodeContracts [54] и специализированный инструментарий для нее. С помощью этой библиотеки стало возможным вынести все проверки входных данных в блоки вида

Contract.Requres(arg != null), либо в отдельные классы, воспользовшись техникой разделения контракта и ограничеваемого типа.

В результате код контрактов стал отделен от кода логики (в случае использования отдельных классов-контрактов), либо стал отличим визуально (инструментарий библиотеки CodeContracts интегрируется в среду разработки и выделяет особым образом контракты в коде). Также существенным преимуществом является то, что включение/отключение проверок контрактов делается исключительно через настройки сборки проекта. Кроме того использование контрактов уменьшило дублируемость проверок: если раньше проверки входных данных необходимо было делать в каждой реализации интерфейса или перегрузке метода, то теперь проверки должны быть написаны ровно в одном месте — при первом объявлении метода/свойства. Также контракты полезны с точки зрения документирования кода, т.к. существуют автоматические средства включающие информацию о контрактах в сгенерированную документацию и кроме того, существует явное требования формализовать все контракты.

В дальнейшей работе помимо описания предусловий методов: контрактов на аргументы и публичное состояние компонента — стали описывается еще и постусловия. Их использование оказалось полезным с точки зрения автоматической валидации кода на этапе компиляции: CodeContracts пытается статически доказать все ограничения, ктоторые были описаны в коде и, если какие-то из них нарушаются, то это приводит к ошибке этапа компиляции. Подобное раннее обнаружение ошибок оказалось достаточно полезной функциональностью.

Необходимость описывать контракты для методов во всех компонентах системы также позволяет яснее представлять зависимости и связности компонентов. В частности, уже отмеченный ранее факт, что многие компоненты системы содержат в себе поле `Topology` (реже `Map`), объясняется именно

необходимостью проверять внутри предусловий методово корректность переданых в качестве аргументов значений координат `TCoordinate` и переходов `Edge<TCoordinate>`. Также можно отметить, что описаный ранее метод ICoordinate<T>.Equals(T other) чаще всего используется в пост- и предусловиях.

3.5. Представление агента в програмной модели

Вернемся к рассмотрению програмной модели системы. Рассмотрим ее важнейшую компоненту — агента. Агент — это единственная активная сущность в АКР, одновременно с этим, сущность, которую будут програмное реализовывать чаще всего. Более того, фактически, агент — это единственный компонент, который надо реализовывать при решении каждой задачи, остальные чаще всего будут переиспользоваться. По этому, к удобству реализации интерфейса агента предъявляются особые требования. Приведем этот интерфейс:

```
public interface IAnt<TCoordinate, TNodeData, TEdgeData>
    where TCoordinate: ICoordinate<TCoordinate>
{
    TCoordinate Coordinate { get; }
    ICell<TCoordinate, TNodeData, TEdgeData> Cell { get; }
    void ProcessTurn();
}
```

Кажется очевидным, что компонент, есдинственная функция которого — выполнять одно действие, должен иметь ровно один метод и ничего более. Тем не менее в вышеприведенном интерфейсе это не так. Рассмотрим подробнее причины подобного отступления от традиционных принципов проектирования.

Чрезвычайно важным нюансом является то, что агент должен обладать некоторой информацией о своем текущем состоянии, в том числе и о своем

текущем положении в пространстве. Будем достаточно неестественно, если информацию о координате должен будет отслеживать разработчик конечного агента, т.к. во всех остальных ситуациях система берет на себя всю работу по манипуляциям с пространством. Фактически, если мы переложим ответственность за отслеживание положения агента на пользователя библиотеки поддержки, то мы нарушим инкапсуляцию ([55]) системы и создадим почву для возникновения ошибок.

Следовательно, информацию о текущей координате агента должна отслеживать и передавать агенту система поддержки. Способов пододбной передачи несколько: через аргументы метода и через поля/свойства. Передача через аргументы кажется более естественной с точки зрения архитектуры приложения, однако, после экспериментов над прототипом системы стало понятно, что подобный подход достаточно неудобен. Дело в том, что чаще всего сосотояние агента хранится в его полях и используется при вычислениях из различных private методов. В результате все переданные из системы в агента данные о его положении либо немедленно складываются в поля, либо начинают обрабатывается совершенно иным способом, нежели остальная информаци о состоянии. Чтобы избегать подобных неестетсвенностей было принято решение передавать информацию о координате через поля агента.

3.6. Размещения агентов на карте

Размещение агентов — это одна из ролей, которую выполняет карта в терминах модели АКР. Как уже было сказано в разделе 3.2, каждой такой роли в реализации системы отводится отдельная сущность. Для карты, как контейнера агентов, таковой програмной сущностью является карта (в дальнейшем, чтобы различать «карту» ы терминах модели АКР, от «карты» в програмной модели, будем называть первую «пространством»). Карта хранит ячейки, ко-

торые являются програмными моделями точек пространства в смысле контейнера для размещения агентов.

Приведем интерфейс ячейки карты:

```
public interface ICell<TCoordinate, TNodeData, TEdgeData>:
        IEnumerable<IAnt<TCoordinate, TNodeData, TEdgeData>>
    where TCoordinate: ICoordinate<TCoordinate>
{
    IMap<TCoordinate, TNodeData, TEdgeData> Map { get; }
    TCoordinate Coordinate { get; }
}
```

Он довлольно естественный: ячейка умеет перечислять находящихся в ней агентов и знает то, в какой точке какой карты она расположена.

Интерфейс карты представляет больший интерес:

```
public interface IMap<TCoordinate, TNodeData, TEdgeData>:
        ISparsedMapping<TCoordinate, ICell<TCoordinate, TNodeData, TEdgeData>>
    where TCoordinate: ICoordinate<TCoordinate>
{
    Topology<TCoordinate> Topology { get; }
}

public interface ISparsedMapping<TKey, TValue>:
        IEnumerable<KeyValuePair<TKey, TValue>>
{
    bool TryGet(TKey key, out TValue value);
}
```

Легко заметить, что одно из основных свойств интерфейса — обеспечение разреженного хранения ячеек. Почему именно разреженного? Очевидно, что не имеет смысла держать в памяти представления тех ячеек, в которых на данный момент отсутствуют агенты. Тем не менее, в интерфейсе `IDataLayer`, который, как кажется естественным, должен быть аналогичен `IMap`, используется прямо противоположный подход — в нем (с точки зрения потребителя) одновременно храняться данные *всех* точек пространства.

Изначально поведения `IMap` и `IDataLayer` были аналогичными, однако

после проведения экспериментов над прототипом системы стало понятно, что не смотря на общую схожетсть функциональности этих двух компонентов, сценарии их использования существенно отличаются.

Для `IDataLayer` справедливо утверждение, что если в определенную точку пространства данные ни разу не записывались, то при чтении из нее мы получим некоторое значение по умолчанию, которое естественным образом будет использовтся в наших алгоритмах. Например, в расчетах нового значения данных, нам нужно знать старое, тогда удобно считать, что если старого значения не было, то оно равно значению по умолчанию — 0. Соответственно, для потребителя интерфейса было бы удобно никогда не думать про случай «отсутствующего значения», а работать с `IDataLayer`, как со всюду-определенным отображением.

Более того, подобный интерфейс можно легко реализовать без дополнительных накладных расходов. Поскольку данные ячеек на практике почти всегда представляются неизменяемыми структурами (существенно реже — неизменяемыми объектами; использование изменяемых типов данных в качестве представления данных ячеек имеет очень ограниченную область применимости), следовательно нет необходимости хранить значение по умолчанию для каждой точки (более того, на картах с бесконечным колличеством точек пришлось бы хранить бесконечное множество экзепляров этого значения), можно просто сконструировать его один раз и при запросах возвращать эакешированое значение [56].

Для `IMap` подобная техника не эффективна, поскольку содержимое карты — это ячейки, которые представляют ценность только как контейнер для агентов. Соответственно, код исполльзующий карту, обычно посроен следующим образом:

for all $i \in \{ v \mid v \in V, \exists(\text{current}, v) \in E \}$ **do**

```
    for all agent ∈ Agents(i) do
        do something with agent
    end for
end for
```

Очевидно, что в рамках такого кода возвращение «ячейки по умолчанию» бессмысленно и неэффективно. В связи с этим, в системе для `IMap` был выбран противоположный к `IDataLayer` подход: возвращать только ячейки реально содержащие данные, причем даже если в карте хранится пустая ячейка, то она никогда не будет возвращена на запрос map.TryGet(ccordinate, out cell) и на map.GetEnumerator().

3.7. Событийная модель

3.8. Особенности реализации системы

Ранее были рассмотрены базовые интерфейсы, позволяющие описать произвольный АКР и его пространство вместе с ассоциированными данными. Эти интерфейсы являются основополагающими в системе, однако, очевидно, сами по себе никаких прикладных задач они не решают. Фактически, они представляют из себя абстрактную часть системы — сборку `SwarmIntelligence.Core`.

Непосредственно код, выполняющий функции по поддержке карты, ячеек, слоев данных и логгированию — все абстрагируемые от конкретной задачи функции, находится в другой сборке: `SwarmIntelligence.Engine`. Этот код достаточно сложен и учитывет множество нюансов, таких как распараллеливание операций с помощью семейства технологий TPL [57] и PLINQ [58], конкурентый доступ к данным с помощью Concurrent Collections [59], сборка мусора и повторное использование объектов [60] (поскольку выделение

памяти под новые объекты — дорогостоящая операция в среде .NET, то переиспользование объектов дает существенный прирост производительности).

Типы из `SwarmIntelligence.Engine` являются реализациями интерфейсов из `SwarmIntelligence.Core` и предполагают, что все сущности в системе будут их наследниками. Причина этого в том, что интерфейсы слоев данных, карты и ячейки являются сильно связанными, в следствии этого, их реализации так-же сильно связны. Например, агент умеет выполнять действие «перейти в точку». Оно реализуется следующим образом:

```
ICell<TCoordinate, TNodeData, TEdgeData> fromCell;
if(!map.TryGet(ant.Coordinate, out fromCell))
    throw new AssertionException();

var targetCell = map.GetOrCreate(to);

fromCell.Remove(ant);
targetCell.Add(ant);
ant.Coordinate = target;
```

Понятно, что присвоение координаты агенту, удаление и добавлене агента в ячейки невозможно через описанные ранее интерфейсы. Решение построить систему именно таким образом было принято после нескольких экспериментов с различными архитектурами. В результате решение, максимально скрывающее аспекты своей реализации, а значит и менее предрасполагающее конечных разработчиков к ошибкам, было признанно наиболее успешным. По этому, все нюансы, связанные с прямым добавлением/удалением агентов из ячеек, ячеек в карту и т.п. были исключены из публичных интерфейсов.

Стоит отметить, что следствием вышеприведенного решения является то, что все публичные интерфейсы `SwarmIntelligence.Core` не позволяют менять состояние агентов на карте. Причина в том, что набор модифицирующих операций сильно зависит от реализации интерфейсов. В частности, реализация максимально использующая распараллеливание, не может позволить себе

операцию удаления агента, распределенная — операции выполняемые в произвольной точке пространства и т.д.

Существующий сейчас `SwarmIntelligence.Engine` позволяет перемещать текущего агента, создавать нового в произвольной точке и самоуничтожать агента.

3.9. Библиотека решений

Помимо непосредственно системы поддержки `SwarmIntelligence.Engine`, была также реализована библиотека стандартных решений — `SwarmIntelligence.Library`, включающая в себя реализации множества переиспользуемых компонентов. В том числе: карты и координаты для трехмерного мерного 6- и 24-связного пространства, для двумерного 4- и 8-связного пространства, для графов. Были реализованы агенты поддерживающие удаленное уничтожение, и операция над агентами «уничтожить данного агента».

Одна из интерестных функций вошедших в `SwarmIntelligence.Library` — это реализация сборки системы на основе конфигурации. Фактически, это узко–специализированый Dependency Injection Container [61], учитывающий нюансы приложения. К сожалению, стандартные утилиты для решения этой задачи (например [62]) были признаны неподходящими, в связи с активным использованием в системе обобщенных типов и некоторых других нюансов, приводивших к чрезвычайно сложному конфигурационному коду для сторонних утилит.

Заключение

В рамках работы над системой поддержки АКР:

TODO: написать

Литература

1. Russell S. J., Norvig P. Artificial Intelligence: A Modern Approach. — 2nd edition. — Englewood Cliffs, NJ, USA : Prentice-Hall, 2003.

2. McCarthy J. Artificial intelligence, logic and formalizing common sense // Philosophical Logic and Artificial Intelligence. — Dordrecht, Netherlands : Kluwer Academic, 1989. — P. 161–190.

3. Nilsson N. J. Principles of Artificial Intelligence. — Palo Alto, CA, USA : Tioga Publishing Company, 1979.

4. Hart P. E., Nilsson N. J., Raphael B. Correction to «a formal basis for the heuristic determination of minimum cost paths» // SIGART Bulletin. — 1972. — no. 37. — P. 28–29.

5. Engelbrecht A. P. Computational Intelligence: An Introduction. — 2nd edition. — Hoboken, NJ, USA : John Wiley and Sons, 2007.

6. Wlodzislaw D. What is computational intelligence and where is it going? // Challenges for Computational Intelligence. — 2007.

7. Konar A. Computational Intelligence: Principles, Techniques and Applications. — New York, NY, USA : Springer, 2005.

8. Гастев Ю. А. Определение // БСЭ. — 3-е изд. — Москва : Советская энциклопедия, 1988.

9. Хайкин С. Нейронные сети: полный курс. — 2е изд. — Москва : Вильямс, 2006.

10. Passino K. M., Yurkovich S. Fuzzy Control. — Menlo Park, CA, USA : Addison–Wesley–Longman, 1998. — P. 475.

11. Емельянов В. В., Курейчик В. В., Курейчик В. М. Теория и практика эволюционного моделирования. — Москва : Физматлит, 2003. — С. 432.

12. Fogel L. J., Owens A. J., Walsh M. J. Artificial intelligence through simulated evolution. — Hoboken, NJ, USA : John Wiley and Sons, 1966.

13. Галал . М. Эволюционное учение // Энциклопедия Кирилла и Мефодия. — Москва : Кирилл и Мефодий, 2003.

14. Рутковская Д., Пилиньский М., Рутковский Л. Нейронные сети, генетические алгоритмы и нечеткие системы. — Москва : Горячая Линия–Телеком, 2007. — С. 452.

15. Jong K. A. D. An analysis of the behavior of a class of genetic adaptive systems : Ph. D. thesis / K. A. De Jong ; University of Michigan. — Ann Arbor, MI, USA, 1975.

16. Michalski R. S. Learnable evolution model: Evolutionary processes guided by machine learning // Machine Learning. — 2000. — Vol. 38, no. 1-2. — P. 9–40.

17. Берг Л. С. Номогенез или Эволюция на основе закономерности. — Петергоф : Государственное издательство, 1922. — С. 306.

18. Bishop J. M. Stochastic Searching Networks // First IEE Conference on Artifical Neural Networks. — 1989. — P. 329–331.

19. Nasuto S. J., Bishop J. M., Lauria S. Time Complexity of Stochastic Diffusion Search // Neural Computation ’98. — 1998.

20. Dorigo M., Maniezzo V., Colorni A. The Ant System: Optimization by a colony of cooperating agents // IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics. — 1996. — Vol. 26, no. 1. — P. 29–41.

21. Geem Z. W., Kim J.-H., Loganathan G. V. A new heuristic optimization algorithm: Harmony search. // Simulation. — 2001. — Vol. 76, no. 2. — P. 60–68.

22. Eo evolutionary computation framework. — URL: http://eodev.sourceforge.net/.

23. Evolutionary computation framework. — URL: http://gp.zemris.fer.hr/ecf/.

24. DeJong K. A. Evolutionary Computation: A Unified Approach. — Cambridge, MA, USA : The MIT Press, 2006.

25. Bonabeau E., Dorigo M., Theraulaz G. Swarm Intelligence: From Natural to Artificial Systems. — New York, NY, USA : Oxford University Press, 1999.

26. Kennedy J., Eberhart R. C. Swarm intelligence. — San Francisco, CA, USA : Morgan Kaufmann, 2001.

27. Beni G., Wang J. Swarm intelligence in cellular robotic systems // NATO Advanced Workshop on Robotics and Biological Systems. — 1989.

28. Асанов М. О., Баранский В. А., Расин В. В. Дискретная математика: графы, матроиды, алгоритмы. — Ижевск : НИЦ «РХД», 2001. — С. 288.

29. Swarm Intelligence in Data Mining / Ed. by Ajith Abraham, Crina Grosan, Vitorino Ramos. — New York, NY, USA : Springer, 2006. — Vol. 34 of Studies in Computational Intelligence.

30. Zhang C., Ouyang D., Ning J. An artificial bee colony approach for clustering // Expert Systems with Applications. — 2010. — Vol. 37, no. 7. — P. 4761–4767. — URL: http://linkinghub.elsevier.com/retrieve/pii/S0957417409009452.

31. Yang X.-S. Nature–Inspired Metaheuristic Algorithms. — Luniver Press, 2008. — ISBN: 1905986106, 9781905986101.

32. Rashedi E., Nezamabadi-pour H., Saryazdi S. Gsa: A gravitational search algorithm. // Information Science. — 2009. — Vol. 179, no. 13. — P. 2232–2248.

33. Jade library. — URL: http://jade.tilab.com/.

34. Multi-agent simulation suite. — URL: http://mass.aitia.ai/.

35. The repast suite. — URL: http://repast.sourceforge.net/.

36. The breve simulation environment. — URL: http://www.spiderland.org/.

37. Troelsen A. Pro C# 2010 and the .NET 4.0 Platform. — 5th edition. — New York, NY, USA : Apress, 2009. — P. 1350.

38. Kephart J. O. A biologically inspired immune system for computers // Artificial Life IV Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems. — 1994. — P. 130–139. — URL: citeseer.ist.psu.edu/kephart94biologically.html.

39. Kaveh A., Talatahari S. Charged system search for optimal design of frame structures // Appl. Soft Comput. — 2012. — Vol. 12, no. 1. — P. 382–393.

40. Almasi G. S., Gottlieb A. Highly parallel computing. — Redwood City, CA, USA : Benjamin–Cummings Publishing Co., Inc., 1989. — ISBN: 0-8053-0177-1.

41. Unger S. H. Hazards, critical races, and metastability // IEEE Transactions on Computers. — 1995. — Vol. 44. — P. 754–768.

42. Bernstein P. A., Newcomer E. Principles of Transaction Processing. — 2nd edition. — San Fransisco, CA, USA : Morgan Kaufmann, 2009.

43. Abadi M., Cardelli L. A theory of objects. Monographs in computer science. — New York, NY, USA : Springer, 1996.

44. Bass L., Clements P., Kazman R. Software Architecture in Practice (2nd Edition). — 2 edition. — Reading, MA, USA : Addison–Wesley Professional, 2003. — ISBN: 0321154959.

45. McLaughlin B. D., Pollice G., West D. Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D. — Sebastopol, CA, USA : O'Reilly Media, 2006.

46. Goetz B. Java theory and practice: To mutate or not to mutate? — URL: `http://www.ibm.com/developerworks/java/library/j-jtp02183/index.html`.

47. Abrahams D., Gurtovoy A. C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. — Reading, MA, USA : Addison–Wesley Professional, 2004. — ISBN: 0321227255.

48. Martin R. C. The Dependency Inversion Principle // C++ Report. — 1996.

49. Pierce B. C. Types and programming languages. — Cambridge, MA, USA : MIT Press, 2002. — ISBN: 0-262-16209-1.

50. Meyer B. Object-Oriented Software Construction. — 2nd edition. — Englewood Cliffs, NJ, USA : Prentice-Hall, 1997. — P. 1254.

51. Liskov B. Data Abstraction and Hierarchy // Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Addendum to the Proceedings. — Vol. 23. — 1987. — P. 17–34.

52. Guijarro E. Type casting impact over execution performance in

C#. — URL: http://www.codeproject.com/Articles/8052/Type-casting-impact-over-execution-performance-in.

53. Jagger J., Perry N., Sestoft P. C# annotated standard. — San Fransisco, CA, USA : Morgan Kaufmann, 2007. — P. 825.

54. Code contracts. — URL: http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx.

55. Design patterns: elements of reusable object-oriented software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. — Addison-Wesley Professional, 1995.

56. Norvig P. Techniques for automatic memoization with applications to context-free parsing // Comput. Linguist. — 1991. — Vol. 17. — P. 91–98.

57. Freeman A. Pro .Net 4.0 Parallel Programming in C#. — New York, NY, USA : Apress, 2010. — P. 350.

58. Ostrovsky I. More Powerful Aggregations in PLINQ. — URL: http://blogs.msdn.com/b/pfxteam/archive/2008/06/05/8576194.aspx.

59. Wagner B. Concurrent Collections in the .NET Framework 4. — URL: http://msdn.microsoft.com/en-us/vstudio/gg274329.aspx.

60. Paveza R. Speedy C#, Part 2: Optimizing Memory Allocations - Pooling and Reusing Objects. — URL: http://geekswithblogs.net/robp/archive/2008/08/07/speedy-c-part-2-optimizing-memory-allocations---pooling-and.aspx.

61. Weiskotten J. Dependency injection & testable objects: Designing loosely coupled and testable objects // Dr. Dobb's Journal. — 2006. — May.

62. Ninject project. — URL: http://www.ninject.org/.