Министерство образования и науки РФ
Государственное образовательное учреждение высшего профессионального образования
Уральский государственный университет им. А.М. Горького

Математико-механический факультет
Кафедра алгебры и дискретной математики

Система поддержки алгоритмов коллективного разума

Допущен к защите

_____

"__" _____ 2010

Квалификационная работа на степень бакалавра наук
по направлению «Математика. Компьютерные науки.»
студента гр. КН – 401
Кононучука Дмитрий Олеговича

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

Екатеринбург
2010

РЕФЕРАТ

Конончук Д.О. СИСТЕМА ПОДДЕРЖКИ АЛГОРИТМОВ КОЛЛЕКТИВНОГО РАЗУМА, дипломная работа: стр. 52, библ. 49 назв.

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

Рассматривается множество алгоритмов коллективного разума, выделяются их общие свойства. На их основе строится программная модель, способная эмулировать любой алгоритм коллективного разума. Рассматриваются расширения модели направленные на улучшение удобства использования. Рассматриваются основные аспекты функционирования этой модели.

Содержание

Перечень принятых в работе сокращений

ACO	Ant Colony Optimisation
GSA	Gravitational Search Algorithm
LEM	Learnable Evolution Model
PSO	Particle Swarm Optimization
АКР	Алгоритмы Коллективного Разума
АЭМ	Алгоритмы Эволюционного Моделирования
ВИ	Вычислительный Интеллект
ГА	Генетические Алгоритмы
ИИ	Искусственный Интеллект
МАА	Многоагентные Алгоритмы
ОО	Объектно Ориентированный
ПО	Программное Обеспечение

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.

Не смотря на большое количество исследований связанных с АКР, для них до сих пор не существует общеизвестной системы поддержки, которая решала бы то множество сервисных задач, которое возникает при программной эмуляции подобных алгоритмов.

В связи с этим, в каждой реализации того или иного АКР приходится заново решать проблемы распараллеливания вычислений, ввода и вывода данных из системы, ее отладки и многое другие. Это усложняет процесс разработки и делает практически невозможной интеграцию различных решений. Кроме того, из-за отсутствия единого фреймворка производительность различных алгоритмов сложнее сравнивать: время работы и потребление памяти разных решений будут в первую очередь зависеть не от принципиальных различий алгоритмов, а от особенностей их реализации.

В некоторых случаях в качестве фреймворка могут быть использованы т.н. системы агент-ориентированного моделирования. Но, к сожалению, эти системы не поддерживают многих свойств, являющиеся существенными для АКР. Кроме того, среди них нет открытых решений на языке C#.

Таким образом, для облегчения реализации, интеграции и сравнения различных типов и вариаций АКР требуется создание универсальной системы поддержки алгоритмов данного типа. Эта система должна подходить для создания на ее базе эмуляций всех общеизвестных АКР и, кроме того, быть удобной для использования, изучения и расширения (поскольку возможно применение

ее также и для поддержки огромного количества систем схожих с АКР).

В ходе моей бакалаврской работы я занимался разработкой такой системы.

Для реализации этой задачи было необходимо, во-первых, провести изучение существующих алгоритмов, обобщить и систематизировать их свойства, а, во-вторых, разработать удовлетворяющую всем вышеприведенным требованиям программную модель, их поддерживающую.

В первой главе данной работы будут рассмотрены теоретические основы АКР, а также будет приведено решение первой из вышеуказанных задач. Вторая глава будет посвящена подробному описанию разработанной модели АКР, ее возможным расширениям направленным на увеличение удобства использования, требованиям предъявляемым к ее программной реализации, особенностям ее архитектуры и механизмов ее работы.

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 т.ч. 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.

    в. Для каждой пары некоторым случайным способом создаются и затем добавляются в популяцию некоторое количество особей являющихся производными от данной пары.

    г. 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.

Генетические алгоритмы являются крайне общими методами, обладающими огромным числом параметров, существенно влияющими на их пове-

дение. 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 представлением 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.

Алгоритм Particle Swarm Optimization [18] может служить примером абсолютно другого подхода к АЭМ. В нем, как и в предыдущих алгоритмах, работа ведется с агентами (в английской литературе: particle — частица), популяцией (в англ. литературе: particle swarm — рой частиц) и метрикой неоптимальности решения (f). Однако, требуется, чтобы агент был представлен вектором в n-мерном пространстве, причем каждый вектор этого пространства является допустимым решением и, следовательно, потенциальным агентом.

1. Для каждого агента $i \in [1, S]$ (где S — количество агентов) в момент времени $t = T$ определены векторы: $x_i(T)$ — его координата; $v_i(T)$ — его скорость; $x_i^\sharp(T)$ — координата наилучшей достигнутой позиции, т.е. $x_i^\sharp(T) = \arg\min\limits_{t \leqslant T} f(x_i(t))$.
2. Для всей популяции в момент времени $t = T$ определен вектор наилучшего достигнутого решения $x^*(T) = \arg\min\limits_{i \in [1,S]} f(x_i^*)$.
3. Некоторым способом задается начальная популяция.
4. На каждом шаге алгоритма для каждой особи i выполняются следующие действия:
   а. Вычисляется новый вектор скорости:

   $$v_{ij}(t+1) = wv_{ij}(t) + c_1 r_1 (x_{ij}^\sharp(t) - x_{ij}(t)) + c_2 r_2 (x_j^*(t) - x_{ij}(t))$$

   где w, c_1, c_2 — параметры алгоритма, а r_1, r_2 — случайные, равномерно распределенные на [0; 1] числа генерируемые независимо для каждого агента.

   б. Вычисляется новая координата агента: $x_i(t+1) = x_i(t) + v_i(t+1)$ и его метрика $f(x_i(t+1))$.

   в. Определяется $x_i^\sharp(t+1)$ и если необходимо обновляется значение $x^*(t+1)$.

5. Шаг алгоритма повторяется до тех пор пока не выполняются определенные условия.

6. Результат работы алгоритма — особь с координатой $x^*(t_{\max})$.

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

1. Состояние алгоритма представлено некоторым множеством (популяцией) некоторых объектов (особь). Каждая особь определяет решение, для которых вводится мера оптимальности. Эта мера может быть перенесена на особи (что определяет функцию приспособленности особи).

2. Алгоритм итеративный. Цель каждой итерации — увеличить максимальное значение функции приспособленности для особей из популяции. По сути — провести эволюцию популяции.

3. Начальное состояние популяции дается алгоритму извне — обычно генерируется случайно.

4. На каждой итерации алгоритм многократно (например для каждой особи, либо фиксированное число раз) выполняет некоторые действия. Причем действия эти независимые и распараллеливаемые. В некоторых алгоритмах также выделяются стадии пред- и пост-процессинга.

5. Алгоритм стохастический.

Особо отмечу, что вышеприведенные алгоритмы формируют более слабые ограничения для пунктов (1) и (4). Основываясь лишь на них, мы можем сформулировать эти пункты как: «Состояние алгоритма представлено множеством (популяцией) его допустимых решений (особь)» и «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 (1) необходимо для включения в класс АЭМ алгоритмов муравейника [19], а пункта (4) — для включения алгоритма гармонического поиска [20].

Перечисленные свойства не являются обязательными для АЭМ, поскольку этот класс сформировался не вокруг общих свойств, а вокруг общего происхождения и круга решаемых задач, 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. Однако, ввиду чрезмерной общности данных свойств и их неестественности, такие библиотеки будут недостаточно удобны в использовании, хотя их примеры существуют: EO Evolutionary Computation Framework [21], ECF [22] и другие.

Неестественность общих свойств АЭМ во многом объясняется тем, что данный класс состоит из двух подклассов, представители которых существенно различны. Это подклассы алгоритмов эволюционных вычислений [23] и коллективного разума [24]. 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

Родоначальником этого класса являются уже упомянутый выше алгоритм муравейника (англ. Ant Colony Optimisation) [19]. Это алгоритм является более общим, нежели другие приведенные мною ранее алгоритмы, в т.ч. алгоритм PSO, который также как и алгоритм муравейника относится к АКР. Фактически, он лишь определяет общие принципы построения аналогичных алгоритмов. Эти принципы были скопированы его разработчиками с принципов поведения колонии муравьев.

Алгоритм работает на графах. В его основе лежит использование множества независимых агентов (здесь и далее «агент» — синоним «особи» в терминологии принятой для АКР) — «муравьев», каждому из которых соответствует решение, итеративно строимое им (муравей перемещается по вершинам графа; его маршрут — решение). Ключевая особенность алгоритма муравейника в использовании «феромонов» ($\tau_{i,j}(t)$) — разновидности памяти, позволяющей ассоциировать с ребрами графа значения, указывающие на историю его использования муравьями. Кроме того, алгоритм использует эвристику $\eta_{i,j}(t)$.

На каждой итерации алгоритма муравейника для каждого муравья ant_k, $k \in \{1 \dots n\}$ расположенного в вершине i производятся следующие действия:

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

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

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

3. Фиксированным для алгоритма способом определяется функция $\Delta\tau_{i,j}^k(t)$ и $\forall l \neq j$ полагаем $\Delta\tau_{i,l}^k(t) = 0$

Кроме того, на каждой итерации, для всех ребер осуществляется пересчет соответствующих феромонов:

$$\tau_{i,j}(t+1) = \rho\tau_{i,j}(t) + \sum_{k=1}^{n} \Delta\tau_{i,j}^{k}(t), \text{ где } 0 \leqslant \rho < 1$$

Алгоритм первоначально был сформулирован для решения задачи коммивояжёра [25], однако благодаря своей обобщенности и наличию параметров был адаптирован для решения многих других задач, примеры которых можно найти в [26].

1.4. Многоагентные алгоритмы

Алгоритм муравейника является не только характерным примером АКР, но и другого, чрезвычайно похожего на первый взгляд, класса алгоритмов — многоагентных алгоритмов. Границу между этими двумя классами не проводят в большей части литературы.

Дело в том, что МАА и АКР оба базируются на агентах, однако трактуют это понятие по разному. Как уже было указано выше, для АКР агент — то же самое, что и особь для АЭМ, т.е. сущность, которой он оперирует и которая представляет решение. В то время как в МАА под агентом подразумевается агент в терминах агент-ориентированного подхода к ИИ [1], т.е. независимая, решающая задачу сущность, воспринимающая окружающую среду и взаимодействующая с ней.

Поскольку сам МАА также является агентом в этих терминах, то естественно, что необходимо вводить специальные ограничения, делающие задачи решаемые агентами разных уровней (МАА в целом $\longleftrightarrow$ агент в МАА) существенно различными. Типичным ограничением является возможность работы лишь в подмножестве пространства исходной задачи.

Из определений очевидно, что МАА являются подмножеством АКР: с

точностью до формулировок МАА являются АКР, в котором каждый агент решает задачу независимо от других (он может либо игнорировать другие агенты, либо взаимодействовать с ними лишь опосредованно — через сообщения или разделяемую память). Но это означает, что многоагентные системы, которые по сути являются МАА, естественно сводятся к терминам АКР. Этот факт с практической точки зрения представляет большой интерес.

К многоагентным системам относятся, в частности, эмуляции жизни, эволюции, социума и т.п., некоторые игры и многое другое. Все эти системы могут интерпретироваться как алгоритмы АКР и реализовываться поверх библиотеки их поддержки, что существенно расширяет область применимости такой библиотеки.

Примером АКР, которые не является МАА, может служить алгоритм гравитационного поиска (англ. Gravitational Search Algorithm) [27]. Агентами в этом алгоритме обладают координатой $X_i(t)$ в пространстве решений, скоростью перемещения в этом пространстве — $V_i(t)$ и массой (авторы предлагают использовать три вида масс: инерционную $M_{ii}(t)$, активную $M_{ai}(t)$ и пассивную $M_{pi}(t)$ гравитационные, однако способы обновления и вычисления их приводят только для ситуации, когда $M_i(t) = M_{ai}(t) = M_{pi}(t) = M_{ii}(t)$). Для агента определена его оценка — функция качества решения им определенного $f_i(t) = f(X_i(t))$.

Тогда алгоритм каждой итерации следующий:

1. Вычислить силы взаимодействия для каждой пары объектов. Для d координаты они вычисляются по формуле:

$$F_{ij}(t) = G(t)\frac{M_{pi}(t)M_{aj}(t)}{R_{ij}(t)}\left(x_i^d(t) - x_j^d(t)\right)$$

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 сте-

пень -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 уменьшить силы взаимодействия.

2. С помощью независимых случайных величин $rnd_i \in [0, 1]$, вычисляются $F_i(t) = \sum_{j=1,\, j\neq i}^{N} rnd_j F_{ij}(t)$ и $a_i(t) = F_i(t)/M_{ii}(t)$. Это, соответственно, правило сложения сил, в которое добавили элемент случайности, и второй закон Ньютона.

3. Производится перемещение объекта:

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

где $rand_i$ — независимые случайные величины равномерно распределенные на $[0, 1]$. Затем пересчитывается его оценка.

4. Обновляются массы объектов:

$$w(t) = \min_{i\in\{1\ldots N\}} f_i(t)$$
$$b(t) = \max_{i\in\{1\ldots N\}} f_i(t)$$
$$m_i(t) = \frac{f_i(t) - w(t)}{b(t) - w(t)}$$
$$M_i(t+1) = \frac{m_i(t)}{\sum_{j=1}^{N} m_j(t)}$$

На примере трех вышеприведенных алгоритмов можно рассмотреть еще одну важную характеристику АКР — тип пространства агентов. В части алгоритмов, в т.ч. в PSO и в GSA, используются агенты располагаемые (обладающие координатой) в пространстве допустимых решений, в то время как в

остальных алгоритмах (из приведенных — в ACO) — в пространстве задачи. Агенты первого типа являются агентами, оптимизирующими решения. Они аналогичны особям, используемым в ГА, LEM и других эволюционных алгоритмах [28]. Агенты второго типа являются агентами, строящими решение. Они могут быть использованы только в АКР и, более того, только в МАА. Большинство МАА (в т.ч. многоагентные системы) основаны на них. Однако, существуют исключения: упомянутый выше PSO, Stochastic diffusion search [29] и другие.

1.5. Общие свойства АКР

Сформулируем общие свойства АКР:

1. Алгоритм работает в некотором пространстве с координатами (карта).
2. Работа алгоритма сводится к оперированию множеством (рой) сущностей (агентами) в этом пространстве.
3. Состояние алгоритма определяется состоянием роя. Реже также используется память ассоциированная с точками пространства и/или небольшое количество глобальных переменных.
4. Алгоритм итеративный. Одну итерацию будем называть ходом.
5. В течении хода алгоритм производит независимые действия над каждым агентом роя (обработка). Эти действия не образуют сторонних эффектов друг на друга: т.е. результат обработки для каждого агента не зависит от того, были ли обработаны другие агенты, в каком порядке они были обработаны, как они изменились и изменили хранимые данные в ходе обработки.

6. Условие остановки алгоритмом не специфицируется.

7. Начальное состояние данных и роя дается алгоритму извне. Координаты агентов в начальном рое обычно случайны (реже при их генерации используется эвристика).

8. Алгоритм стохастический. Генерация случайных чисел (обычно многократная) необходима при обработке каждого агента.

МАА в дополнение к вышеперечисленным также обладает следующими свойствами:

1. Его агенты независимы. Результат обработки любого агента не зависит от состояния других агентов в любой момент времени. Взаимодействие агентов может быть лишь опосредованным, например, через совместный доступ к памяти.

2. Агенты могут действовать лишь в своей ограниченной окрестности.

АКР с оптимизирующими агентами обладает лишь одним специфичным свойством: карта является пространством допустимых решений, таким образом каждому агенту соответствует решение. Меру оптимальности (или отношение «быть оптимальнее») определенную на решениях можно распространить на агентов и затем использовать для их сравнения.

В АКР со строящими агентами карта является пространством задачи. Таким образом каждый агент не является полноценным решением, а значит невозможно без введения эвристики производить их сравнение.

Глава 2

Практическая реализация

2.1. Спецификация поддерживаемых АКР

Поскольку, как было сказано в разделе (1.5), существует большое количество подклассов и разновидностей АКР, то первой задачей при разработке системы их поддержки становится задача фиксации требований к ней — разработка спецификации поддерживаемого типа АКР.

Необходима была такая формулировка АКР, которая обладает следующими свойствами: наибольшей общностью, хорошей приспособленностью для использования большинства известных алгоритмов (в т.ч. наиболее популярных — в первую очередь, алгоритма муравейника [19]), высокой выразительностью для прикладных задач, близостью к формулировкам многоагентных систем. Кроме того, эта формулировка должна быть легко выразима в терминах ОО [30] архитектуры и удобна для программной реализации.

В связи с последними двумя требованиями, а также в связи с естественным для всех спецификаций ПО требованием максимальной полноты и формальности, конечная формулировка получилась иерархичной и объемной. Приведем её последовательно, начиная от общих понятий.

АКР — это изменяющаяся во времени система, обладающая следующими свойствами:

1. Система функционирует в некотором адресуемом дискретном пространстве (карте) с координатами.

2. Каждой точке этого пространства (ячейке) соответствует некоторый набор данных (фон ячейки).

3. Кроме того, в ячейках могут располагаться объекты карты (агенты). Количество агентов в ячейке не ограниченно. Каждый агент располагается в некоторой ячейке, и притом только в одной.

4. Система изменяется итеративно. Все итерации равнозначны. Одна итерация называется ходом.

На карту и координаты накладываются следующие требования:

5. Карта является, фактически, сюръективной не всюду определенной функцией, отображающей пространство координат в множество ячеек.

$$Map: Coord \to Cell$$

6. Будем называть координаты, для которых функция карты определена (т.е. те координаты, которым соответствует ячейка карты), достижимыми.

$$Reach \subseteq Coord: \forall c \in Reach\ \exists Map(c)$$

7. Для каждой пары координат определен куб, ими задаваемый — некоторая последовательность координат, про которые мы можем неформально сказать, что «они лежат в n-мерном кубе, верхний левый и нижний правый угол которого заданы исходной парой координат». Причем эта пара координат входит в свой собственный куб.

$$Cube: Coord \times Coord \to 2^{Coord};\ \forall a, b \in Coord: a, b \in Cube(a, b)$$

8. Для каждой пары координат верно, что если они достижимы, то и все координаты входящие в куб, ими заданный, также достижимы.

$$\forall a, b \in Reach: \forall c \in Cube(a, b)\ c \in Reach$$

9. Карта может быть разбита на несколько кубов, заданных некоторыми достижимыми координатами.

$$\exists a_1, \cdots, a_n, b_1, \cdots, b_n \in Reach : \ Reach = \biguplus_{i=1}^{n} Cube(a_i, b_i)$$

10. Для каждой достижимой координаты и неотрицательного числа определено непустое множество координат, называемых окрестностью данной координаты с данным радиусом. Причем каждая достижимая координата лежит в любой своей окрестности.

$$Suburb : Reach \times \mathbb{R}_+ \to 2^{Reach}; \ \forall c \in Reach, \forall r \in \mathbb{R}_+ : c \in Suburb(c, r)$$

11. В дальнейшем мы будем рассматривать только достижимые координаты, если иное не отмечено особо.

Фон ячейки можно рассматривать либо как неотъемлемое свойство самих ячеек, либо как некоторое множество (фон карты), биективно отображаемое на множество ячеек. Определения очевидно эквивалентны. При описании формулировки будет удобнее пользоваться первым, однако при описании модели программной реализации — вторым. Данные, хранимые в фоне ячеек, в общем определении АКР не специфицируются и зависят от конкретного алгоритма.

Агент — это единственная активная сущность в системе АКР. На каждом ходе он выполняет некоторый набор действий, изменяющих его и всю систему в целом. Эти действия определяются заложенным в агента алгоритмом на основе его состояния и состояния всей системы. Для агента могут быть постулированы следующие свойства:

12. Агент расположен в некоторой ячейке, т.е. для него определена координата.

13. Агент на каждом ходе обладает неизменной характеристикой — радиусом его области видимости, являющейся неотрицательным числом.

14. Агент в течении хода имеет доступ ко всем ячейкам, координаты которых входят в окрестность его координаты с радиусом, равным радиусу области его видимости. Очевидно, что ячейка, в которой расположен агент, входит в эту окрестность.

15. Под доступом к ячейке подразумевается:

    а. чтение ее координаты,

    б. доступ к данным ее фона на чтение и запись,

    в. перечисление всех агентов в ней расположенных,

    г. отправка сообщения любому из этих агентов,

    д. добавление в нее нового агента,

    е. перемещение самого себя в эту ячейку (при этой операции доступная агенту окрестность не изменяется).

16. Агент может принимать сообщения (то есть иметь доступ на чтение к их содержимому), которые отправили ему другие агенты. Сообщения являются однонаправленными, не несущими информации об отправителе, локальными во времени (сообщения, отправленные в течении хода, будут получены в начале следующего, а затем более не будут доступны). Содержимое сообщений зависит от конкретных алгоритмов и не специфицируется.

17. Агент может отправлять широковещательные сообщения. Подобные сообщения будут получены всеми агентами в начале следующего хода, включая тех агентов, которые будут созданы в течении текущего хода, а также включая отправителя.

18. Агент может удалить себя с карты: перестать быть расположенным в некоторой ее ячейке и перестать получать широковещательные сообще-

ния. Такой агент более не может изменять состояние системы и самого себя.

19. Действия агента на каждом ходе не зависят от действия других агентов в том же ходе: все сообщения, изменения фона карты, добавление/перемещение/удаление агентов — результаты всех действий других агентов не будут доступны ему. Фактически, если мы определим текущее состояние системы, как функцию от времени (роль времени играет номер текущего хода), тогда каждый агент будет по текущему значению этой функции строить ее преобразование, а значение функции в следующий момент времени будет результатом применения композиции всех таких преобразований.

20. Более того, действия всех агентов (в терминах преобразования системы) инвариантны относительно порядка применения. То есть система будет приведена в одно и тоже состояние в независимости от того, в каком порядке будут применены эти преобразования. Причем, очевидно, что не обязательно группировать все действия одного агента в течении хода в одно преобразование, а можно представлять каждое действие отдельным.

Мы закончили формулировку выбранной вариации АКР. Очевидно, что работа любого из алгоритмов, обладающих свойствами, описанными в разделе (1.5), может быть эмулирована ими.

Система поддержки АКР должна брать на себя все сервисные функции алгоритма с вышеописанными свойствами, оставляя разработчику конечного алгоритма лишь работу по реализации логики агентов, спецификации типов данных сообщений и фона карты, а также описания координат карты. Причем, очевидно, что для такого разработчика наибольший интерес будет представлять создание агента, поэтому все остальные задачи нужно максимально упростить.

2.2. Пояснения и уточнения свойств

Перед тем как переходить к следующим вопросам, уточним причины использования подобных формулировок некоторых свойств.

Свойство (10) необходимо для поддержки МАА системой. Стоит отметить особо, что система ориентирована в первую очередь именно на них, поскольку МАА алгоритмов известно больше, чем просто АКР, а также поскольку все неалгоритмические приложения системы используют многоагентный подход.

Свойства (7), (8) и (9) чрезвычайно удобны для введения возможности разбиения карты на некоторое количество непересекающихся частей, которая необходима для оптимальной организации распределения вычислений в системе поддержки АКР. Поскольку эти свойства довольно естественны и легко реализуемы, то их было решено ввести в общую модель.

Свойства (19) и (20) агента являются одними из наиболее важных. Фактически, они представляют собой другую формулировку свойства (5) АКР (стр. 16) и являются существенными с точки зрения реализации, поскольку создают идеальные условия для распараллеливания и распределения вычислений действий агентов и их применения на каждом ходе.

Из-за наличия этих свойств становится невозможным введение операций удаления или перемещения одного агента другим (эти операции были бы довольно полезны в некоторых многоагентных системах). А также невозможно непосредственное взаимодействие агентов, то есть, если возвращаться к терминам программной модели, вызов методов одного агента другим, что и привело к необходимости введения понятия сообщений и операций их отправки и получения.

Теоретически возможно организовать работу АКР и без обмена сообщениями. В этом случае, взаимодействие агентов будет основано на передаче информации через фон карты. То есть, для того, чтобы агент *A* передал неко-

торую информацию агенту *B* необходимо, чтобы агент *A* записал ее в фон ячейки, в которой находится агент *B*. Однако, очевидно, что такой подход содержит «подводные камни»: если агент *B* переместился в другую ячейку на том же ходе, когда агент *A* передал ему информацию, то он ее не получит; если агенты *A* и *C* передали информацию агенту *B* таким способом одновременно, то одна информация перекроет другую и возникнет ситуация «потерянной записи» [31].

Вышеописанная техника применялась в прототипе существующей системы [32, 33], в котором не требовалось выполнение свойства (20). Она показала себя как чрезвычайно неудобная для реализации и использования. В том же прототипе вместо широковещательных сообщений (свойство (17)) использовались глобальные переменные, которые не смотря на то, что нарушают свойство (20), являются более удобными для использования.

Свойство (16) требует, чтобы отправленные в текущем ходе сообщения были приняты лишь в следующем. Это требование является следствием свойства (19), однако оно крайне неудобно с практической точки зрения. Как показала практика работы с прототипом системы и анализ возможных сценариев ее применения, чаще всего сообщения отправленные в текущем ходе содержат данные, актуальные именно в нем.

Поскольку, чтобы на момент получения сообщения было возможно восстановить контекст его отправки и правильно его интерпретировать, приходится сохранять довольно много информации в состоянии агента, то почти постоянно при разработке агента тратится множество времени на подобные вспомогательные операции, которые, теоретически, должна целиком брать на себя система поддержки. В худших, но нередких, случаях приходится даже разделять логически цельный ход на два хода системы: в первом производится отправка сообщений, а во втором — некоторые действия, зависящие от сообщений. Очевидно, что необходимо обеих ситуаций избегать.

В связи с вышеуказанными проблемами, в системе использовался несколько видоизмененный ход, разбитый на две стадии: начальная и конечная. В начальной стадии хода агент может посылать сообщения и читать сообщения, принятые в предыдущем ходе. В конечной стадии агент не может отсылать сообщения, но может принимать сообщения, посланные на начальной стадии. В остальном стадии равноправны. Подобное разделение позволяет агентам сделать последовательные ходы максимально независимыми друг от друга и существенно сократить количество информации хранимой в их состоянии (в лучшем случае — избавиться от хранимого состояния полностью). Агенты, для которых «актуальность» сообщений не критична, могут реализовывать только начальную стадию хода.

Для улучшения удобства работы с широковещательными сообщениями были введены несколько средств: именованные сообщения, постоянные сообщения и обработчики сообщений.

Именование сообщений означает, что каждому широковещательному сообщению будет присвоено имя (по умолчанию — пустое), и агенты смогут фильтровать сообщения по этому имени.

Постоянные сообщения — это широковещательные сообщения, которые будут доступны для чтения во всех последующих ходах, а не только в одном.

Обработчики сообщений — это некоторые функции, которые могут модифицировать список широковещательных сообщений с определенным именем, например оставлять в нем только сообщение с самыми большими значениями или заменять все сообщения одним, содержащим сумму их данных.

Сочетание двух последних техник позволяет эмулировать глобальные переменные, не нарушающие свойства (20).

Свойство (14) требует, чтобы агент имел доступ только к ячейкам своей окрестности. Практика использования прототипа системы и анализ возможных сценариев применения ее самой показали, что практически во всех слу-

чаях агенты в каждый конкретный момент времени работают в окрестностях, имеющих одну и ту же топологию. Причем, автор агента еще на этапе разработки может легко ее спрогнозировать. Однако использовать этот прогноз непросто, поскольку каждый раз одинаковая топология выражается через различный набор координат, т.к. агент движется и следовательно движется центр его окрестности.

Чтобы повысить прогнозируемость окружения агента на каждом ходе и, следовательно, простоту формулирования алгоритма для него, вводится понятие локальных координат. Локальные координаты для агента — это координаты, аналогичные глобальным, полученные из них некоторым обратимым преобразованием, переводящим точку в которой расположен агент в начало координат. Цель этого преобразования — представить окрестности с одинаковыми топологиями одинаковыми координатами. Причем с практической точки зрения само преобразование может быть скрыто в объектах, реализующих получение окрестности агента, но обратное преобразование обязательно должно быть представлено во внешних интерфейсах.

2.3. Требования к реализации

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. Во-первых, система должна быть написана на языке C# и работать на платформе .NET, поскольку остальные разработки лаборатории выполняются с помощью этих технологий. Во-вторых, было необходимо достичь максимальной простоты решения задач, следовательно, система поддержки должна иметь средства, облегчающие решения типовых задач. В идеале небольшие задачи-примеры должны занимать всего несколько строк, причем легко чи-

таемых и понятных. В-третьих, система должна иметь документацию, а ее код должен быть прокомментирован. В-четвертых, система должна быть легко расширяемой и модифицируемой и, при этом, показывать неплохую производительность.

Систему было решено писать на C# 4.0 под фреймворк .NET 4.0 [34]. Выбор столь новых инструментов обусловлен тем, что в фреймворке 4.0 существует большое количество механизмов для удобной работы с распараллеливанием, а также с асинхронными вычислениями [35], которые неизбежно возникнут при реализации распределения вычислений. Также .NET 4.0 содержит набор библиотек WCF [36], с помощью которых можно легко организовать обмен сообщениями между вычислительными узлами, необходимый для распределения вычислений (подобное решение является временным и будет использоваться лишь на начальном этапе).

Также среди инструментов стоит отметить особо библиотеку Code Contracts [37] для .NET 4.0, реализующую идеологию контрактного программирования [38]. Одним из преимуществ этой библиотеки можно назвать то, что операции проверки контрактов, ею проводимые, могут быть исключены из итогового кода. Т.е. все проверки, замедляющие работу конечного кода, но гарантирующие его корректность, могут проводиться только в режиме отладки, а в рабочем режиме быть отключены.

При реализации необходимо учитывать следующие моменты:

1. Размер карты может быть очень большим. Поэтому хранение каких-либо сервисных структур данных для каждой ячейки невозможно.

2. Имеет смысл хранить сервисные структуры данных лишь для ячеек, в которых расположен хотя-бы один агент.

3. Организация данных фона карты существенно зависит от специфики

этих данных. Поэтому фон карты воспринимается как отдельный объект, используемый для получения данных каждой конкретной ячейки.

4. При перемещении агентов из одних ячеек в другие возникает необходимость удалять ячейки, в которых нет больше агентов, и добавлять ячейки, в которые агенты переместились. Поскольку вышеописанные операции происходят многократно в течении каждого хода, а операции с выделением/освобождением памяти в .NET довольно медленные, то необходимо минимизировать их количество, что возможно сделать с помощью реализации пула сервисных объектов ячеек [39].

5. В ходе работы системы будет чрезвычайно активно вестись работа с координатами. Поэтому координаты должны быть структурами (т.к. операции с ними несколько быстрее), реализующие некоторый интерфейс. Кроме того, использовать координаты надо не как этот интерфейс, поскольку подобное использование приведет к боксингу структур в объекты с существенной потерей производительности, а как генрик-тип реализующий интерфейс.

6. Вызов виртуального метода занимает больше времени, чем вызов обычного. Таким образом, надо максимально отказаться от их использования. Также стоит избегать использования ссылок на интерфесы, вызов методов через них существенно дольше чем через ссылки на объекты.

7. Работа со свойствами медленнее, чем с полями. Наличие блоков try-catch также существенно снижает производительность. Вызов делегата работает быстрее вызова виртуального метода. Стандартные field-like [40] события неэффективно реализуют операции добавления и удаления делегата — эффективнее использовать вместо них List<T>. Все методы синхронизации работают очень медленно.

8. Существуют способы избегания проблем производительности, описанных в предыдущем пункте, однако эти способы приводят к нетривиальным решениям, которыми сложно пользоваться и сопровождать. Поэтому нужно сохранять баланс между производительностью кода и его качеством.

При проектировании системы, для достижения ее расширяемости и модифицируемости, было решено следовать набору принципов SOLID [41]. И в первую очередь — принципу SRP (Single Responsibility Principle — англ., принцип единственной обязанности), согласно которому каждый объект должен выполнять только одну задачу [42]. Благодаря использованию SRP удается создать систему, каждый из аспектов поведения которой можно расширять или модифицировать, заменяя одни ее компоненты аналогами. К сожалению, принципы SRP и ISP [43] из набора SOLID не применимы к некоторым объектам в системе, поскольку эти объекты исполняют роль тех или иных сущностей в АКР, и потому их интерфейсы логически неделимы.

Использование SOLID приводит к созданию многокомпонентных программных модулей, причем «сборка» из этих компонент работоспособной системы зачастую является трудоемкой задачей [44]. Часто для облегчения этой задачи используются т.н. DI-контейнеры [45], однако на начальном этапе от их использования в проекте решено отказаться — добавление зависимости от большой библиотеки при том, что выгода от ее использования не столь очевидна, было признано нецелесообразным. Но ничто не мешает конечным пользователям библиотеки использовать их.

2.4. Описание представления координат в программной модели

Начнем описание программной модели с описания наиболее базовой ее части — интерфейсов, определяющих координаты. Как уже было сказано выше, реализациями этих интерфейсов должны быть структуры, причем используемые как генрик-параметры.

Сущность координаты была разбита на две: собственно координаты и «измеритель» – вспомогательная сущность, через которую реализуются получение для данной координаты окрестности с локальными координатами и преобразование локальных координат к глобальным. Подобное разбиение необходимо, поскольку результат операции «взятия окрестности» зависит от топологии текущей карты, а значит и сущность, реализующая эту операцию, должна использовать данные о карте и, в некоторых случаях, реализовывать нетривиальную логику. Добавление ссылки на карту в координату нарушает логическую иерархию зависимостей объектов, а добавление логики — принцип SRP.

Таким образом интерфейс координат будет следующим:

```
public interface ICoordinate<C>: ICloneable, IEquatable<C>
    where C: struct, ICoordinate<C>
{
    // Вспомогательный метод:
    C Cast { get; }
    // Строит куб координат, где this — нижняя граница:
    IEnumerable<C> Range(C upperBound);
    // Проверяет, лежит ли текущая координата в кубе заданном аргументами:
    bool IsInRange(C lowerBound, C upperBound);
}
```

Особый интерес в нем вызывает описание и использование генерик-параметра C. Этот параметр необходим для избежания боксинга при выполнении операций, описанных в интерфейсе. Если обойтись без него, тогда каждый из

методов будет принимать и возвращать аргументы типа ICoordinate, т.е. принимать и возвращать объекты, тогда как реальная работа всегда ведется со структурами.

Очевидно, что C подразумевается использовать в реализациях интерфейса единственным образом: реализация в качестве генерик-параметра будет использовать сама себя. Подобное использование трактует необходимость задания именно такого ограничения на параметр C, как и было использовано. Теоретически это ограничение должно быть еще более жестким, указывающим на то, что использовать необходимо самого себя, но система типов языка C# не обладает достаточно мощными инструментами для введения такого ограничения. Однако это не представляет проблемы, поскольку все классы, использующие ICoordinate, вводят для него следующее требование:

```
public class Something<C, B>
    where C: struct, ICoordinate<C>
{ ... }
```

Очевидно, что этот контракт является формализацией вышеуказанного неформального контракта для наследников ICoordinate. То есть координаты, определенные следующим образом, не смогут быть использованы в системе, не смотря на то, что подобное определение само по себе допустимо:

```
public struct Useless: ICoordinate<Useful> { ... }
```

Подобное использование генерик-парметров не является стандартным, а является, по сути, попыткой эмулирования системы классов типов данных из языков с более мощной системой типов. Поэтому оно может вызывать некоторое недоумение. Тем не менее, это единственный способ избежать проблемы боксинга и сохранить возможность выделить общий интерфейс.

Интерфейс т.н. «измерителя» описывается следующим образом:

```
public interface IGauge<C, B>
    where C: struct, ICoordinate<C>
{
    // Получить карту в которой работает «измеритель»:
    Map<C, B> Map { get; }
    // Получить окрестность точки с заданным радиусом:
    IEnumerable<C> Suburb(C center, int radius);
    // Перевести координаты в окрестности с данным центром и радиусом в глобальные:
    C ToGlobal(C point, C center, int radius);
}
```

Отметим особо, что генерик-параметр B (о природе которого будет сказано далее) является, в отличие от C, обыкновенным генерик-параметром, т.е. все реализации IGauge будут также иметь генерик-параметр B (но не будут иметь параметра C).

Интерфейсы ICoordinate и IGauge представляют собой одну из двух точек вариативности системы, т.е. мы можем существенно менять систему, меняя именно их. Второй точкой является набор классов, связанных с фоном. Все прочие классы системы (за исключением, конечно же, классов, описывающих агентов) либо полностью реализованы и неизменяемы, либо имеют реализацию «по умолчанию», которая является подходящей практически для всех задач.

2.5. Взаимодействие агента с системой

Необходимо так выстроить взаимодействие агента с системой, чтобы все свойства и ограничения АКР выполнялись, причем были интуитивно понятны. Однако, при этом существовала возможность обойти эти ограничения, причем «обходной путь» не должен быть чрезмерно трудным или чрезмерно легким: в первом случае им будет нереально пользоваться, во втором случае будет сложно провести границу между тем, что правилами системы разрешено, и тем, что является «обходным путем». И, в первую очередь, необходимо, чтобы

интерфейсы этого взаимодействия были удобны для использования программистами агентов.

Встраивание агента в систему осуществляется следующим образом: выделяется абстрактный класс агента, реализующий все функции взаимодействия с системой, и имеющий несколько абстрактных методов, определяющих логику поведения конкретного агента. Экспортируемый интерфейс (по отношению к системе, т.е. без internal и private сущностей) этого класса следующий:

```
public abstract class Agent<C, B>: ILocatable<C, B>, ICommunicative<C, B>, IVisualizable
    where C: struct, ICoordinate<C>
{
    // Конструктор:
    protected Agent(District<C, B> district) { ... }

    #region ILocatable
    // Область (см. ниже) в которой агент находится:
    public District<C, B> District { get; }
    // Координата в которой он расположен:
    public C Coordinate { get; }
    #endregion

    #region ICommunicative
    // Послать сообщение этому агенту:
    public void SendMessage(dynamic msg) { ... }
    // Список принятых сообщений:
    protected IList<dynamic> Messages { get; }
    #endregion

    // Радиус окрестности агента, к которой он имеет доступ:
    public int Radius { get; protected set; }
    // Объект представляющий доступ к этой окрестности:
    protected Zone<C, B> { get; }
    // Удалить агента с карты:
    protected void Die() { ... }


    // Этот метод выполняется на начальной стадии хода:
    protected abstract void OnTurnBegin();
    // Этот — на конечной (он может не переопределяться):
    protected virtual void OnTurnEnd() {}

    // метод визуализации (см. ниже):
    public abstract void Draw(FastBitmap bmp);
}
```

Этот абстрактный класс обладает, как и все остальные классы системы, двумя генерик-параметрами B и C (C — координата в текущем пространстве карты, о природе B будет рассказано ниже). Разработчик агента реализует наследника этого класса, причем этот наследник уже не будет обладать генерик-параметрами, в силу того, что алгоритм поведения, очевидно, зависит от пространства, в котором агент работает, и от типа данных (см. ниже), которыми может оперировать. Т.е. сигнатура класса имеет вид:

---

public class Ant: Agent<GraphCoordinates, SomeData> { ... }

Формальных контрактов, гарантирующих именно такой подход в наследниках, нет, что позволяет создавать цепочки наследования, в которых промежуточные звенья обладают генерик-параметрами. Однако, в силу самой природы АКР, последний класс в этой цепочке генерик-параметров иметь не будет.

Стоит отметить, что для работы с сообщениями используется тип dynamic. Альтернативным решением является выделение специального интерфейса для объектов сообщений. Этот интерфейс будет предоставлять доступ по типу словаря, причем значения этого словаря будут иметь разный тип. Опыт использования прототипа системы показал, что эта техника весьма неудобна и гораздо менее понятна, нежели использование dynamic. Кроме того, операции с dynamic более быстры.

Перейдем к описанию тех интерфейсов, которые класс Agent предоставляет для работы агентам. Это, в первую очередь, объект Zone, предоставляющий доступ к окрестности агента (ее радиус равен значению поля Radius на момент начала стадии хода) через локальные координаты. «Обходным путем» является доступ к своей области (поле District) и знание своей координаты.

Основная функциональность как Zone так и District выражается в следующем методе:

```
    public Cell<C, B> At(C cord) { ... }
```

Этот метод позволяет получить объект, описывающий ячейку карты по ее координате (в Zone — локальной для области; в District — нет). Этот объект позволяет «получить доступ» (свойство (15) АКР) к ячейке. Его экспортируемый интерфейс:

```
public struct Cell<C, B>: ILocatable<C, B>, IEnumerable<Stone<C, B>>
    where C: struct, ICoordinate<C>
{
    #region ILocatable
    // Область (см. ниже) в которой находится ячейка:
    public District<C, B> District { get; }
    // Координата ячейки в области (глобальная):
    public C Coordinate { get; }
    #endregion

    // Фасад фона ячейки (см. ниже):
    public B Backround { get; }
    // Количество агентов в ячейке:
    public int Count { get; }
    // Первый агент в ячейке (удобно для систем, где в одной ячейке может быть только один агент):
    public Agent<C, B> First { get; }
    // Перечислить всех агентов в ячейке:
    public IEnumerator<Stone<C, B>> GetEnumerator() { ... }
    public IEnumerator IEnumerator.GetEnumerator() { ... }
    // Добавить агента в ячейку:
    public void Add(Agent<C, B> agent);
    // Переместится в ячейку:
    public void MoveInto();
}
```

Как видно, предоставляемые интерфейсы весьма интуитивны и совпадают с описанием свойств АКР, что делает невозможным их разночтение. Их использование выглядит следующим образом:

```
    protected override void OnTurnBegin() {
        this.Zone.At(new Coordinate2D(1, 2)).MoveInto();
    }
```

2.6. Принцип реализации ходов

Как уже было неоднократно указанно выше, свойства АКР (19) и (20) накладывают большое количество ограничений на систему и являются, несомненно, наиболее проблематичными в реализации. Фактически, для того, чтобы их реализовать, необходимо разработать систему транзакций [46], в которой ход каждого из агентов будет изолированной транзакцией.

При полном анализе возможностей системы можно заметить, что агент имеет лишь минимальный доступ к изменениям, которые он внес на текущем ходе. Фактически, он может лишь отсылать сообщения агентам, созданным им, и перечислять их, а также работать с фоном, модифицированным им. Практика использования прототипа системы показала, что даже этими возможностями никогда не приходиться пользоваться.

Таким образом, возможно отказаться, не потеряв при этом никакой функциональности, от доступа к изменениям в карте и фоне, которые создал агент на текущем ходе. В тех редчайших случаях, когда такой доступ нужен, его возможно эмулировать силами самого агента, поскольку он, очевидно, имеет ссылки на все агенты, добавленные им на карту, и все данные, добавленные им в фон.

Единственная функциональность, доступная агенту, имеющая сторонний эффект на него самого — это удаление его с карты. После удаления агент не может более выполнять никаких действий. Таким образом, эта операция должна обрабатываться особо: вызов любой операции после нее должен приводить к возникновению исключения.

С учетом отсутствия доступа к собственным изменениям, система транзакций может быть реализована с помощью шаблона проектирования «команда» [47]. Его суть заключается в представлении действий в виде объектов, которые можно хранить, комбинировать и отложено выполнять.

Для реализации этого шаблона можно предложить два принципиально различных метода: действие в команде может кодироваться на некотором языке (обычно — состояние объекта определенного типа), либо может быть представлено делегатом. Второй подход обладает очевидными недостатком: для разработчика агента добавление (каким либо способом) команды-делегата выглядит абсолютно одинаково вне зависимости от тех действий, которые этот делегат выполняет, соответственно, невозможно четко ограничить (либо сделать ограничения ясными для разработчика) набор допустимых действий.

Поэтому в прототипе системы использовался первый подход. Однако практика показала, что он приводит к созданию крайне сложного кода, пользоваться которым практически невозможно. Более того, фактически, именно использование этого подхода привело к тому, что от использования прототипа системы было решено отказаться и стало стимулом для развития ее текущей версии.

В текущей версии был выбран первый подход, однако, создание и использование делегатов было скрыто от разработчика агента. Все объекты предоставляют внешние интерфейсы, которые используются как обычные методы, однако эти методы не выполняют действий сами по себе, а добавляют делегат, их реализующий, в текущую команду. Команда содержит список делегатов, добавленный всеми агентами, которые отправляются на исполнение после начальной и после конечной стадий хода.

Контроль того, что каждый из методов вызывается из неудаленного агента, производится с помощью контрактов и возможен благодаря тому, что все доступные агенту интерфейсы взаимодействия с системой являются лишь фасадами [47], созданными специально под нужды данного агента и, следовательно, имеющие на него ссылку и имеющие возможность проверить его статус.

Команда, как хранилище и исполнитель делегатов без параметров, являет-

ся отдельной независимой абстрактной сущностью с несколькими неабстрактными методами, которые обеспечивают приведение делегатов с параметрами посредством замыканий к непараметрической форме. Стандартная реализация хранит эти делегаты в списке типа List<Action> и исполняет их параллельно с помощью Parallel.ForEach.

В определенных случаях, например при эмуляции некоторых игр, имеет смысл отказаться от свойств (19) и (20) системы. Чтобы достичь этого, необходимо сделать исполнение делегатов немедленным, а не отложенным. Этого можно легко достичь, заменив логику метода добавления делегата в команду: вместо добавления делегат немедленно исполняется.

2.7. Описание интерфейсов фона

Как было сказано в разделе (2.3), фон карты необходимо воспринимать как объект, используемый лишь для получения фонов отдельных ячеек. Однако, на сам фон ячейки накладываются более сложные ограничения, и он не может рассматриваться как обычный объект/структура, содержащая данные. Дело в том, что такой примитивный подход в случае, если данные структуры являются изменяемыми, приводит к нарушению свойств (19) и (20).

О сохранении этих свойств должен позаботиться разработчик фона ячейки, причем никаких контрактов (как языковых, так и предоставляемых библиотекой Code Contracts) на него не может быть наложено, поскольку о структуре и поведении фона, в отличие от координаты, вообще ничего не известно. Таким образом я, как разработчик системы, могу лишь сформулировать рекомендации и предоставить некоторые утилиты, результат работы которых будет этим рекомендациям удовлетворять.

Для сохранения вышеуказанных свойств необходимо прибегнуть к уже использовавшемуся ранее методу — к шаблону «команда» (подход на основе

сообщений, который также является эффективным средством обхода данной ситуации, основывается на тех же принципах, что и «команда»). Необходимо разбить сущность фона ячейки на две: фасад и данные фона.

Фасад доступен для работы из остальной системы. Именно фасад возвращает объект фона карты. Фасад является легковесной, неизменяемой структурой (использование структур для легковесных и неизменяемых сущностей оптимально с точки зрения производительности). Он обладает ссылкой на агента, его запросившего, на текущую команду и на данные фона.

Любое действие над фасадом никак не должно менять результаты любых действий над любыми фасадами на текущем ходе. В большинстве случаев это означает, что все изменения данных фона должны проводиться отложено с помощью добавления соответствующего делегата в команду. Валидация аргументов методов должна проводится в фасаде и вызывать исключения до добавления делегата в команду. Методы чтения обычно делегируются фасадом непосредственно в методы данных фона. Все методы записи обязательно должны проверять (желательно с помощью Code Contracts), что вызвавший их агент не удален с карты. Для методов чтения такая проверка является рекомендуемой.

Данные фона являются просто некоторой сущностью, для которой даже не специфицируется, существует ли она. Например, в случае если система использует в качестве данных изображение, то отдельных сущностей данных фона не существует, просто фасад использует ссылку на соответствующий пиксель картинки. Единственное условие, налагаемое на данные фона — возможность возникновения исключения в тех методах, которые вызываются отложено, должна быть минимизирована. Дело в том, что такие исключения, с точки зрения программиста агента, возникают «на пустом месте» — далеко от контекста, их спровоцировавшего. Их отладка является крайне нетривиальной задачей.

Практика использования прототипа системы и анализ возможных сценариев применения ее самой показали, что доступ к различным данным фона ячейки является одной из наиболее частых операций, проводимых в агенте. Для упрощения (и повышения производительности) этого доступа, тип фона был вынесен в генерик-параметр. Генерик-параметр B, встречающийся в большинстве приведенных выше определений — это тип фасада фона ячейки.

В системе представлен простейший вариант фона карты, который реализует требуемый интерфейс делегируя процедуру создания фасада фона внешней функции, либо конструктору фиксированной сигнатуры структуры, чей тип передан ему как генерик-аргумент (второй вариант реализуется с помощью Reflection [48]).

2.8. Организация распределений вычислений

Как уже было неоднократно сказано, система АКР обладает хорошим потенциалом распараллеливания и распределения вычислений. Однако, его реализация весьма нетривиальна.

Распределение вычислений базируется на разбиении всей карты на несколько непересекающихся частей (областей) с помощью операции построения куба для пары координат. Каждая область определяется двумя ее граничными точками, задающими ее куб. Существует некая операция, не экспортируемая во внешние интерфейсы, которая по координате позволяет определить к какой из областей она принадлежит (в большинстве случаев, это просто сравнение координаты с координатами граничных точек).

Каждая область обрабатывается на отдельном узле, за счет чего достигается высокий уровень распределенности. Вся работа с системой у агентов и прочих сущностей должна вестись через области. Для организации системы в распределенной среде выделяются еще две сущности: карта и координатор.

Экземпляр объекта карты присутствует на каждом узле и отвечает за взаимодействие с другими узлами. Объект координатор существует в единственном узле, его задача — централизованное управление системой.

Очевидно, что распределение вызывает несколько проблем: взаимодействие агента с ячейками извне области, в которой он находится; широковещательные сообщения; визуализация; синхронизация стадий хода и ходов в целом.

Синхронизация производится силами координатора: все карты отчитываются ему о завершении в своей области текущей стадии хода, и, когда текущая стадия завершена у всех, он рассылает команду о начале следующей стадии/следующего хода.

Визуализация реализуется по отдельности в каждой из областей. Команду на ее начало дает координатор. Подробнее будет рассказано в следующем разделе.

Синхронизация широковещательных сообщений не вызывает больших проблем: каждая карта в каждой области имеет идентичный объект для работы с широковещательными сообщениями. Каждое добавление нового сообщения в любой из этих объектов вызывает, помимо самого добавления, еще и пересылку данного сообщения всем другим картам.

Наибольшую проблему представляет взаимодействие агента с ячейками вне области. Можно рассмотреть несколько аспектов этой проблемы: доступ к координате ячейки; перечисление агентов в ней; доступ на чтение к данным ее фона (это представляет проблему, только если фон изменяемый); отсылка сообщения агентам и внесение изменений в фон; перемещение агента.

Для решения вышеуказанных проблемы используется подход, основанный на частичном зеркалировании областей друг другом. Дело в том, что ячейки, находящиеся в области, могут получить доступ лишь к ограниченному количеству ячеек вне ее. Эти ячейки (по первому требованию или предвари-

тельно) создаются в специальном сервисном объекте, называемом зеркалом. Однако, создаваемые зеркалом ячейки, их фон и объекты, содержащиеся в них, (называемыми зеркальными ячейками, фоном и объектами соответственно) не являются полноценными: они не хранят данные и не исполняют команды, а являются лишь фасадом для реальных сущностей расположенных на удаленных узлах.

Очевидно, что каждая зеркальная ячейка знает свою координату, поэтому первый аспект не представляет проблем. Последних два аспекта также разрешаются легко: каждый зеркальный объект вместо того, чтобы выполнять какие либо действия пересылает команду на их выполнение в реальный объект; перемещение агента в ячейку можно трактовать как выполнения действия над ней и обрабатывать также.

Проблема заключается в двух оставшихся аспектах: перечислению объектов и чтению фона. Дело в том, что они оба подразумевают получение результата, который нельзя предсказать, не зная данных реальных объектов.

Возможны два подхода: поддерживать в зеркалах данные, актуальные данным в реальных объектах, и обрабатывать все запросы к зеркальным объектам локально; либо при запросе к зеркальному объекту выполнять удаленный запрос к реальному (с кэшированием ответа в течении одной стадии хода). Первый, очевидно, гораздо более сложный и практически не совместим с некоторыми нетривиальными типам фона ячейки (например изменяющих себя при чтении из них). Поэтому система ориентирована на использование второго типа.

Для его реализации выделяются следующие сущности: вышеуказанное зеркало (отвечает за зеркалирование ячеек и агентов; нарушения SRP это не вызывает, поскольку эти зеркальные объекты используют одинаковую логику), зеркало фона отвечает за зеркалирование фона карты и порождает зеркальный фон ячеек), система удаленного вызова (является оберткой над той или иной

технологией RPC [49] и также используется объектом карты).

В системе реализованы абстрактный класс карты и частные реализации RPC (на основе WCF) и зеркала вместе с зеркальной ячейкой и агентом.

2.9. Организация визуализации системы

Очевидно, что система поддержки АКР будет чрезвычайно неудобна для использования, если она не будет предоставлять средств для обзора/анализа ее состояния. Двумя подобными стандартными средствами являются текстовые логи и графическое представление. С учетом сложности и комплексности системы АКР второй подход предпочтительней.

Более того, в некоторых задачах результатом работы системы должно быть именно изображение ее конечного состояния, а в некоторых — последовательность изображений, описывающих эволюцию системы.

Следовательно система поддержки должна предоставлять некоторые средства поддержки визуализации текущего состояния системы — ее отрисовки. Причем средства эти должны быть адаптированны к условиям системы, т.е. хорошо распараллеливаться и распределяться.

Поскольку в нормальных условиях система функционирует с довольно большой производительностью, то проведение отрисовки на каждом ход становится нецелесообразным: чрезмерно частая смена кадра будет мешать восприятию информации, кроме того операция отрисовки, которая является одной из самых медленных операций, будет существенно замедлять работу системы. В связи с вышесказанным вводится некоторый интервал отрисовки: изменяемое значение, общее для всех областей карты. Отрисовка системы производится только один раз за количество ходов, указанное в интервале.

Следуя принципу SRP мы выносим функциональность отрисовки в отдельную сущность, причем ссылка на эту сущность в системе не хранится,

а значит и ее интерфейс не специфицируются. Для взаимодействия с ней используется механизм синхронных событий (единственный параметр события — текущая область).

В связи с вышесказанным очевидно, что никаких контрактов или требований к реализации отрисовки не предъявляется. Тем не менее система предоставляет вспомогательные средства и рекомендации для ее организации. К ним относится специализированный метод агента Draw, отрисовывающий его на изображении (FastBitmap), переданном ему в качестве параметра.

Опыт использования прототипа системы и анализ сценариев ее применения показал, что, во-первых, в абсолютном большинстве случаев агенты отрисовываются в виде элементарных фигур (чаще всего точкой или квадратом), во-вторых, изображения агентов, расположенные в разных ячейках, не накладываются друг на друга, т.е. каждой ячейке карты соответствует некоторая область, в итоговом изображении не пересекающаяся с областями соответствующими другим ячейкам.

Благодаря первому замечанию становится возможно использовать FastBitmap вместо Bitmap. Подобная замена дает огромный выигрыш: FastBitmap возможно использовать из нескольких потоков, более того, для некоторых операций (чтения, преобразования цвета пикселя) он является потокобезопасным; операции с ним быстрее в разы.

Второе замечание приводит к еще более интересным следствиям: каждую ячейку области можно отрисовывать параллельно с другими, и при этом не возникнет конкуренции при доступе к одним и тем же точкам изображения. Благодаря использованию FastBitmap подобная параллельная отрисовка возможна. Если, при выполнении второго замечания, отрисовку агентов находящихся в одной ячейке сделать последовательной, то в FastBitmap не возникнет конкуренции при изменениях пикселей, благодаря чему из его реализации может быть исключены атомарный операции, что существенно улучшит как

простоту кода, так и быстродействие (атомарные операции в FastBitmap весьма нетривиальны).

Представленные в системе отрисовщики реализуют следующую логику: отдельная сущность, не входящая в состав отрисовщика, создает фон изображения на холсте GDI+ (она при этом может использовать данные фона и любые другие), параллельно создается прозрачный FastBitmap и начинается параллельная обработка ячеек карты, в каждой из которых агенты последовательно отрисовывают себя. На конечном этапе FastBitmap преобразуется к Bitmap, который отрисовывается на холсте GDI+.

При распределении вычислений процесс визуализации проводится для каждой области отдельно, затем готовые изображения преобразуются к формату bmp и отправляются координатору.

Заключение

В рамках работы над системой поддержки АКР:

1. была проведена систематизация и обобщение существующих на данный момент алгоритмов относимых к классу АКР;
2. был проведен анализ свойств этого обобщения, и на их основе была построена программная модель;
3. на основании программной модели был разработан прототип системы;
4. были тщательно изучены возможные области применимости модели, а также опыт использования ее прототипа;
5. на основе полученных данных были спроектированы расширения разработанной ранее модели, делающие ее максимально удобной для решения большого класса задач связанных с АКР;
6. были разработаны все аспекты функционирования системы, отвечающей требованиям модели, сформирована ее четкая спецификация;
7. спецификация была формализована в виде контрактов на код.

Система поддержки АКР в будущем будет использоваться в нескольких запланированных научных проектов лаборатории РВИиМАП, в рамках спецкурса Нейронные Сети и как база для научных и практических исследований студентов.

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. Pp. 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. Pp. 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. Chichester, WS, UK: John Wiley and Sons, 1966.

13. Галал Я. 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 / 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. Pp. 9–40.

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

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

19. 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. Pp. 29–41.

20. Geem Z. W., Kim J.-H., Loganathan G. V. A New Heuristic Optimization Algorithm: Harmony Search. // Simulation. 2001. Vol. 76, no. 2. Pp. 60–68.

21. EO Evolutionary Computation Framework. URL: `http://eodev.sourceforge.net/`.

22. Evolutionary Computation Framework. URL: http://gp.zemris.fer.hr/ecf/.

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

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

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

26. Swarm Intelligence in Data Mining, Ed. by A. Abraham, C. Grosan, V. Ramos. New York, NY, USA: Springer, 2006. Vol. 34 of Studies in Computational Intelligence.

27. Rashedi E., Nezamabadi-pour H., Saryazdi S. GSA: A Gravitational Search Algorithm. // Information Science. 2009. Vol. 179, no. 13. Pp. 2232–2248.

28. Baykasoglu A. Evolutionary Computation for Modeling and Optimization. // The Computer Journal. 2008. Vol. 51, no. 6. P. 743.

29. Hurley S., Whitaker R. M. An agent based approach to site selection for wireless networks // SAC '02: Proceedings of the 2002 ACM symposium on Applied computing. New York, NY, USA: ACM, 2002. Pp. 574–577.

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

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

32. Конончук Д. О., Окуловский Ю. С. Модель и реализация конструктора генетических алгоритмов и алгоритмов коллективного разума // Секция «Интеллектуальные системы и технологии» в рамках «Научной сессии МИФИ-2009». Москва: НИЯУ МИФИ, 2009.

33. Конончук Д. О., Окуловский Ю. С. Универсальный пакет поддержки интеллектуальных вычислений GANS // 6-ая Всероссийская межвузовская конференция молодых ученых. Труды конференции. Выпуск 4. Математическое моделирование и программное обеспечение. Санкт-Петербург: 2009. С. 151–157.

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

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

36. Löwy J. Programming WCF Services. 2nd edition. Sebastopol, CA, USA: O'Reilly Media, 2008. P. 750.

37. Code Contracts. URL: `http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx`.

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

39. Kircher M., Jain P. Pooling Pattern // The EuroPLoP conference. Irsee, Germany: 2002.

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

41. Metz S. SOLID Object-Oriented Design // Gothan Ruby Conference. 2009.

42. 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.

43. Martin R. C. The Interface Segregation Principle // C++ Report. 1996. — August. Vol. 8.

44. Fowler M. Inversion of Control Containers and the Dependency Injection pattern. 2004. — January. URL: http://www.itu.dk/courses/VOP/E2005/VOP2005E/8_injection.pdf.

45. Weiskotten J. Dependency Injection & Testable Objects: Designing loosely coupled and testable objects // Dr. Dobb's Journal. 2006. — May.

46. Weikum G., Vossen G. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control. San Fransisco, CA, USA: Morgan Kaufmann, 2001.

47. Freeman E., Freeman E., Bates B., Sierra K. Head First Design Patterns. Sebastopol, CA, USA: O'Reilly Media, 2004.

48. Liberty J., Xie D. Programming C# 3.0. Sebastopol, CA, USA: O'Reilly Media, 2007. P. 587.

49. Bloomer J. Power programming with RPC. Sebastopol, CA, USA: O'Reilly Media, 1992. P. 486.