УДК 681.3

Л.А. Галковская, Н.Н. Глибовец, С.С. Гороховский

Гибридный алгоритм решения задачи удовлетворения ограничений

Представлен гибридный алгоритм *improved Guided Local and Systematic Search* для решения распределенной задачи удовлетворения ограничений. Алгоритм объединяет компоненты локального и конструктивного поиска. Доказаны полнота и корректность алгоритма. Приведены результаты его экспериментальной оценки на модельной задаче о ферзях и проведено сравнение его производительности с производительностью алгоритмов *Dis-GLS* и *iGL*.

The improved Guided Local and Systematic Search hybrid algorithm is presented for solving the Distributed Constraint Satisfaction Problem, which combines two local and one systematic search methods. The completeness and correctness of the algorithm are proved. The results of our experiments with queens' problem, and a comparison of productivity for our hybrid and two other algorithms Dis-GLS and iGL are given.

Представлено гібридний алгоритм *improved Guided Local and Systematic Search* розв'язання розподіленої задачі задоволення обмежень, який поєднує компоненти локального та конструктивного пошуку. Доведено повноту і коректність алгоритму. Описано результати його експериментальної оцінки на модельній задачі про ферзі. Проведено порівняння його продуктивності з продуктивністю алгоритмів класу *Dis-GLS* та *iGL*.

Введение. Парадигма программирования с ограничениями в конечных областях [1] возникла в конце 80-х годов как способ решения сложных комбинаторных задач (*NP*-полных задач). Она развилась из логического программирования с ограничениями [2], которое в свою очередь было расширением парадигмы логического программирования. Программирование с ограничениями можно рассматривать как программную инфраструктуру для комбинирования программных компонент, позволяющее получать решатели на дереве поиска для конкретных задач. Программирование с ограничениями служит гетерогенным полем исследований, начиная с математической логики и заканчивая практическими приложениями, такими как задачи календарного планирования, задачи о назначениях и др.

Многие задачи оптимизации и поиска могут быть формализованы как задачи удовлетворения ограничений (*Constraint Satisfaction Problems – CSP*) [1, 2]. Еще одним примером такой задачи может быть известная задача раскраски графа. С распространением концепции распределенных вычислений сформировался подкласс распределенных *CSP*-задач (*Distributed Constraint Satisfaction Problem – DISCSP*). Во многих случаях для их решения привлекают группу независимых исполнителей (агентов). Каждый агент оперирует частью задачи (чаще всего – одной переменной задачи или одной подзадачей) и обменивается информацией с другими исполнителями, с которыми он связан определенными условиями и ограничениями (*constraints*). Такая концепция положительно влияет на скорость нахождения решения и обеспечивает лучшую надежность.

Задачу удовлетворения ограничений можно представить в виде тройки (X, D, C), где

$X = \{x_1, \ldots, x_n\}$ – множество n переменных;

$D = \{D(x_1), \ldots, D(x_n)\}$ – множество соответствующих конечных доменов этих переменных, т.е. множеств допустимых значений;

$C = \{c_1, \ldots, c_n\}$ – множество ограничений на эти переменные.

Для уточнения постановки задачи в распределенном варианте *DISCSP* будем считать, что исполнителей представляют агенты. Под агентами подразумеваются участники вычислений, которые могут обмениваться сообщениями [3]. Агент может послать сообщение другим агентам, если он знает их имена. Задержка в доставке сообщений конечна и случайна. Агенты получают сообщения в той же последовательности, в которой они отправлены. Тогда формально *DisCSP* можно представить как пятерку (X, D, C, A, φ), где X, D определены как в *CSP*; C – множество ограничений, налагаемых на переменные из X;

$A = \{1, \ldots, p\}$ – множество агентов; φ: $X \rightarrow A$ функция, сопоставляющая каждому агенту переменные из множества X. Каждый агент может оперировать несколькими переменными, но каждая переменная должна принадлежать только одному агенту. В связи с этим множество C делится на две части: множество внутренних ограничений агента $C_{\text{intra}} = \{c_{ij} \mid \varphi(x_i) = \varphi(x_j)\},)\}$, т.е. множество ограничений, налагаемых на переменные одного агента, и множество ограничений, налагаемых на переменные различных агентов $C_{\text{inter}} = \{ c_{ij} \mid \varphi(x_i) \neq \varphi(x_j)\}$ [4].

Известно, что каждая *CSP* сводится к бинарной. Бинарной задачей *CSP* называется задача, в которой предусмотрены только бинарные ограничения. Тогда функция φ сопоставляет каждому агенту только одну переменную, и класс C_{intra} будет пустым (все ограничения бинарные, а каждый агент оперирует только одной переменной).

Введем также следующие обозначения: *belongs*(x_j, i) – переменная x_j принадлежит агенту и *known*(c_m, k) – агент k знает об ограничении c_m. Тогда *DisCSP* будем считать решенной в случае, если выполняются следующие условия [3]:

• $\forall i \ \forall x_j$ *belongs*(x_j, i) $d_j : x_j = d_j$, т.е. всем переменным всех агентов присвоено какое-то значение из их доменов. $\forall k$, $\forall c_m$ *known*(c_m, k), c_m выполняется для значения d_j, присвоенного переменной x_j.

Алгоритмы решения распределенных задач удовлетворения ограничений

Алгоритмы решения *DISCSP* начали развиваться в конце прошлого тысячелетия и их число росло с каждым годом. Стало развиваться направление «улучшения существующих методов» разнообразными вариациями известных алгоритмов. Причем два направления решения *DISCSP* (конструктивный и локальный) развивались параллельно, но базовые идеи по улучшению алгоритмов были вскоре исчерпаны. Сейчас внимание исследователей сосредоточено на:

- разработке специальных алгоритмов для конкретных типов задач,
- объединении существующих алгоритмов в гибриды, наследующие преимущества сразу нескольких методов.

В конструктивном подходе [5–8] алгоритмы пошагово расширяют частичное допустимое решение до полного. Если частичное решение оказалось недопустимым, то происходит бектрекинг и предыдущее частичное решение расширяется в альтернативном направлении. Тот факт, что поиск происходит в пространстве частичных решений, есть отличительной чертой этих алгоритмов.

Локальный поиск – основной метод решения сложных задач из класса задач поиска *CSP*. Более того, зачастую он – единственный возможный способ решения задач, где конструктивные методы бессильны вследствие слишком большого размера пространства возможных решений.

Идея локального поиска заключается в том, чтобы начать с произвольно сгенерированного кандидата в решения проблемы и, если он окажется недопустимым, пошагово его улучшать путем уменьшения количества неудовлетворенных ограничений. Алгоритмы локального поиска отличаются методами нахождения этого улучшения [5–8].

Один из недостатков локальных алгоритмов – их неполнота, т.е. в них поиск может остановиться в локальном минимуме, который в действительности не является глобальным решением. Здесь используется терминология, сложившаяся в области *DISCSP* [9].

Состояние – это одно возможное присвоение всем переменным, число состояний равно произведению объемов множеств определения переменных, *значение оценки* – количество нарушений ограничений в *состоянии*, *сосед – состояние*, получаемое из текущего состояния изменением значения только одной переменной, *локальный минимум* – состояние, которое не есть решением и значение оценки всех его соседей больше или равно значению оценки этого состояния. Для устранения возможности неполноты часто используется рандомизация (с определенной вероятностью на каждом шагу выбирается произвольное решение). Такие методы называются стохастическими. Используют и другие подходы: *Randomized Iterative Improvement*, эволюционные алгоритмы, алгоритмы

на основе метода отжига, *Tabu Search*, динамический поиск или алгоритмы на основе штрафов, алгоритмы с использованием принципа муравьиной колонии [9–14].

Гибридные алгоритмы используют комбинацию нескольких различных подходов к решению задачи. В контексте *DisCSP* гибриды могут сочетать в себе алгоритмы из разных классов (локальный и конструктивный) и разного способа синхронизации (синхронные и асинхронные). Общепринятой классификации этих алгоритмов нет. В работах [4, 15–19] встречается классификация гибридных алгоритмов по степени интеграции локальной и конструктивной составляющих:

- низкий уровень (локальный поиск проводится до или после конструктивного);
- средний уровень (основа – локальный поиск, который пользуется средствами конструктивного алгоритма для уменьшения пространства поиска);
- высокий уровень (операторы конструктивного и локального поиска применяются поочередно).

Алгоритм 1. Общий принцип работы гибридного алгоритма

```
search (a)
{
 while a is not solution
 {
 if a is consistent
 a ← extend (a)

 else a ← repair (a)
 }

 return a
}
```

Цель гибридов – компенсация недостатков одной из компонент за счет другой. Примером может быть сочетание синхронного метода, который характеризуется последовательным выполнением (это минус) и относительно низким уровнем общения между агентами (это плюс) и асинхронного метода с противоположными характеристиками. Такой гибрид расходует минимум ресурсов на общение между агентами и пользуется преимуществами параллельного выполнения.

Гибридный алгоритм *GLoSS*

Рассмотрим предложенный вариант гибридного алгоритма. Алгоритм *GLoSS* по классификации относится к третьей категории, т.е. совмещает компоненты конструктивного и локального поиска и характеризуется высоким уровнем их интеграции, при этом операторы конструктивного и локального поиска применяются поочередно, как упоминалось. Фрагмент решения задачи расширяется средствами конструктивного алгоритма, пока это возможно (функция *extend* () в алгоритме 1). Когда же для следующей присоединенной к частичному решению переменной невозможно найти подходящее значение из ее домена, вместо стандартного бектрекинга используется оператор локального поиска (*repair* () в алгоритме 1). Он занимается улучшением частного решения, полученного конструктивным поиском, до тех пор, пока значения всех переменных этого решения не удовлетворят все ограничения. По достижении цели управление вновь передается оператору конструктивного поиска. Цикл повторяется, пока не будет найдено решение. На первый взгляд, в алгоритме все хорошо. Проблемой остается неполнота. Каждый раз, когда управление передается локальному оператору, его миссия превращения недопустимого частичного решения в допустимое с некоторой вероятностью может быть провалена. Если эта вероятность и не очень велика (для алгоритма локального поиска *DisBO-wd* на произвольной бинарной *DISCSP* она составляет примерно 3–4% [20]), это делает его крайне ненадежным, поскольку поиск решения может остановиться уже на первом переходе между ветвями.

Описанный недостаток можно преодолеть простым способом – ограничив количество шагов алгоритма локального поиска. В этом случае он гарантированно остановится, даже не найдя решения. В случае преждевременной остановки (т.е. вынужденной остановки по достижении лимита шагов) работу по нахождению решения продолжит конструктивный алгоритм.

Воспользуемся такой схемой работы гибридного алгоритма. Предлагаемый гибрид *GLoSS* базируется на трех алгоритмах: *Synchronous*

Backtracking (*SBT*), *improved Guided Local Search* (*iGL*) и *Asynchronous Weak-Commitment Search* (*AWCS*).

Описание компонент алгоритма *GloSS.* Основная компонента гибрида – *SBT*, которая может периодически обращаться к алгоритмам *iGL* и *AWCS*.

Алгоритм Synchronous Backtracking (*SBT*). На этапе инициализации этого алгоритма для всех агентов устанавливается приоритет, чтобы можно было отсортировать агентов по этим приоритетам в нисходящем порядке. Каждый следующий агент (движение происходит от агента с большим приоритетом к агенту с меньшим приоритетом) получает от предыдущего агента значения, присвоенные переменным пройденных агентов. Получив информацию, он пытается установить значение для собственной переменной, анализируя известные ему ограничения, связанные с этой переменной. Если допустимое значение будет найдено – решение передастся следующему по приоритету агенту; если нет – произойдет возврат к предыдущему агенту.

Данный алгоритм не использует возможностей распараллеливания, поскольку в каждый момент времени активен только один агент. Преимущество алгоритма – значительная разгрузка каналов общения между агентами. Количество отправленных сообщений для этого алгоритма минимально, поскольку решение передается от одного агента к другому, как эстафета. Эта разгрузка обеспечивает эффективную реализацию алгоритма.

Алгоритм improved Guided Local Search (*iGL*) базируется на алгоритме локального поиска *Distributed Guided Local Search* [10]. Он относится к классу *penalty-based* алгоритмов локального поиска, практикующих наложения штрафов. Каждый агент имеет определенный неизменный приоритет. Во время инициализации каждый агент выбирает себе значение только после того, как все агенты с более высоким приоритетом выберут себе значение, которое должно минимизировать функцию:

$f(x)$ = количество конфликтов с агентами с высшим приоритетом + сумма инкрементных штрафов за данное значение + сумма временных штрафов за данное значение, где x – определенное значение из домена переменной агента.

Во время инициализации существенно только первое слагаемое, поскольку все штрафы равны нулю. Наложение штрафов на значение из домена переменной агента происходит в случае попадания агента в квазилокальный минимум[1]. Штрафы бывают двух типов: временные и инкрементные. Процедуру наложения штрафов можно описать так.

- Попав в квазилокальный минимум, агент проверяет состояние текущей ситуации (т.е. своего состояния и состояния соседей) в списке сохраненных ранее «плохих» ситуаций *nogood_store*. Если такая ситуация ранее не возникала, то агент налагает временный штраф на текущее значение и рассылает сообщение всем соседям меньшего приоритета с требованием так же наложить временный штраф на свои значения. Этот штраф отменяется сразу после выбора агентом для себя нового значения [10]. Временный штраф, обычно очень большое число, гарантирует избрание агентом нового значения.
- Если ситуация уже встречалась ранее, то агент налагает на текущее значение инкрементный штраф (увеличивая штраф на единицу) и посылает запрос к соседям с низким приоритетом на выполнение этого действия для их значений.

Во избежание бесконечного увеличения штрафов каждому агенту выставляется верхняя граница. Когда штраф превышает этот предел, агенту присваивается его наихудшее значение (с наибольшим штрафом). Если агент находит для себя допустимое значение, все инкрементные штрафы аннулируются [10].

Проанализируем главный цикл алгоритма. Агент активируется только после получения сообщения. Последние бывают двух типов – сообщения о штрафах и сообщения, содержащие

[1] Ситуация квазилокального минимума возникает у агента, когда он нарушает некоторые, связанные с ним ограничения и при этом не может изменить свое состояние к лучшему. Поскольку в такой ситуации участвуют только данный агент и его соседи, она не будет локальным минимумом, а только квазилокальным, потому что агенты, не связанные с данным, все еще могут улучшить ситуацию.

значение одного из соседей агента. В случае, если агент получает значение одного из своих соседей (алг. 2, стр. 9–22), то проверяет, не конфликтует ли это значение с его собственным. Если конфликты отсутствуют, агент обнуляет все свои инкрементные штрафы (алг. 2, стр. 10, 11). Кстати, алгоритм *Dis-GLS*, на котором базируется *iGL*, кроме обнуления инкрементных штрафов, рассылает всем агентам свое значение (алг. 2, стр. 12). По мнению авторов, это действие лишнее, оно забивает и без того перегруженный сообщениями канал общения агентов. Особых оснований для его использования не существует, ибо значение агента с прошлого раза не менялось и конфликтов нет. Кроме того, каждое из отправленных сообщений активирует всех соседей агента, вынуждая их проводить лишние вычисления. Алгоритм *iGL* лишен такой возможности и в этом его преимущество.

Алгоритм 2. Главный цикл алгоритма *iGL*

```
procedure iGL_main                                  1
{                                                   2
  do                                                3
    when active                                     4
      evaluate state                                5
      if penalty message received                   6
        responde_to_message()                       7
                                                    8
      else                                          9
        if curr val is consistent                   10
          reset inc penalties                       11
                                                    12
        else                                        13
          if sender priority <                      14
             current agent's prior                  15
            resolve_conflict()                      16
          else                                      17
            send(id_val,null) back                  18
          end if                                    19
        end if                                      20
                                                    21
      end if                                        22
    return to inactive state                        23
  until terminate                                   24
}                                                   25
```

Алгоритм 3. Главный цикл алгоритма *DisGLS*

```
procedure DisGLS_main
{
  do
    when active
      evaluate state
      if penalty message received
        responde_to_message()

      else
        if curr val is consistent
          reset inc penalties

          send(id,val,null) to all
        else

          resolve_conflict()

        end if

      end if
    return to inactive state
  until terminate
}
```

Рассмотрим случай, когда значение агента имеет конфликты с полученным значением соседа. Агент берется решать конфликт, только если он имеет приоритет выше конфликтного агента. Иначе агент возвращает текущее значение адресату (и только ему), вынуждая того начать процедуру разрешения конфликта. Как видно из псевдокода алгоритмов 1 и 2, поведение *iGL* и *Dis-GLS* в этой ситуации отличается. *Dis-GLS* вынуждает агента немедленно начать процедуру разрешения конфликта. Рассмотрим процедуру решения конфликтов (алг. 4 и 5).

Процедура разрешения конфликта для *iGL* начинает проверку наличия текущей конфликтной ситуации в *nogood_store* (алг. 4, стр. 3, 4). В *Dis-GLS* она тоже имеется, но происходит несколько позже (алг. 5, стр. 12, 13). Проверка гарантирует попадание конфликтной ситуации сразу в «черный список» с первого раза. В *Dis-GLS* для этого потребуется как минимум два шага, и как минимум одно лишнее сообщение всем агентам.

Алгоритм 4. Процедура разрешения конфликта в алгоритме *iGL*

```
procedure iGL_resolve_conflict                      1
{                                                   2
  if state is not in nogood_store                   3
    add state to nogood_store                       4
                                                    5
    if sender state has changed                     6
      select new value                              7
      send(id,val,null) to all                      8
      return                                        9
                                                    10
                                                    11
                                                    12
    else                                            13
      impose temp penalty                           14
      select new value                              15
      send(id,val,TempPen) to                       16
     agents with lower priority                     17
    end if                                          18
                                                    19
  else                                              20
    if inc penalty on cur val <                     21
       upper bound                                  22
     increase inc penalty by 1                      23
     select new value                               24
                                                    25
    else                                            26
     select worst val in domain                     27
    end if                                          28
                                                    29
      send(id,val,IncPen) to                        30
      agents with lower priority                    31
  end if                                            32
}                                                   33
```

Алгоритм 5. Процедура разрешения конфликта в алгоритме *DisGLS*

```
procedure DisGLS_resolve_conflict
{

      if sender state has changed
        select new value
        send(id,val,null) to all
        return
      end if

      if state is not in nogood_store
        add state to nogood_store
        impose temp penalty
        select new value
        send(id,val,TempPen) to
        agents with lower priority
      end if

      else
        if inc penalty on cur val <
           upper bound
          increase inc penalty by 1
          select new value

        else
          select worst val in domain
        end if

        send(id,val,IncPen) to
        agents with lower priority
      end if
}
```

Далее в процедуре алгоритма *iGL* происходит проверка состояния адресата. Если он не изменился, необходимо принять меры для выхода из квазилокального минимума, поэтому на текущее значение налагается временный штраф, и аналогичные действия применяются для всех конфликтных агентов с низким приоритетом. Бесконфликтным агентам посылаются сообщения с новым значением агента (алг. 4, стр. 13–18). Потом происходит наложение инкрементного штрафа.

Видно, что основным преимуществом алгоритма *iGL* над *Dis-GLS* есть способ выхода из квазилокального минимума. Интересным примером попадания в квазилокальный минимум стал случай попадания в конфликт агента с наименьшим приоритетом, и для разрешения конфликта должны быть задействованы все другие агенты. Алгоритм *iGL* выходит из такой

критической ситуацией в среднем на 30 процентов быстрее *Dis-GLS*, одного из лучших среди алгоритмов локального поиска. Это обусловило выбор алгоритма *iGL* на роль компоненты локального поиска в гибридном алгоритме.

Алгоритм Asynchronous Weak-Commitment Search (*AWCS*). Этот алгоритм относится к классу асинхронных алгоритмов конструктивного поиска. Поэтому все агенты всегда активны и координируют свои действия с помощью сообщений двух типов: *ok*? – для пересылки соседям своего текущего значения, и *nogood* – для передачи информации о нарушении определенных ограничений. Коротко работу алгоритма можно описать как в [7].

● Для каждого агента устанавливается приоритет – неотрицательное целое число. Чем больше число – тем выше приоритет. Если у нескольких агентов числа оказались одинаковыми, больший приоритет получит тот, имя которого по алфавиту идет первым. Начальные значения приоритета для всех агентов равно нулю.

● Получив сообщение *ok*?, агент записывает полученное значение в свой *agent_view* – структуру данных, характеризующую представление агента о «внешнем мире».

● Если текущее значение агента удовлетворяет всем ограничениям, связанным со значением переменных высшего приоритета по его *agent_view*, оно считается допустимым, иначе агент должен выбрать новое допустимое значение (алг. 7, стр. 9, 10).

● Если для переменной не существует допустимого значения, агент отсылает сообщения типа *nogood* конфликтным агентам с большим приоритетом и меняет свой приоритет на max +1, где max – наибольший приоритет из приоритетов его соседей на данный момент (алг. 8, стр. 10–14). Если при отправке сообщения *nogood* агент имел самый высокий приоритет среди соседей, то сформированный *nogood* будет пустым, и это послужит сигналом завершения алгоритма в связи с отсутствием решения задачи [3].

На роль третьей компоненты был выбран именно *AWCS*, поскольку он – один из наиболее эффективных алгоритмов конструктивного поиска. Он уступает разве что алгоритму *Maintaining Asynchronously Consistencies* (*MAC*). Последний, как и целый ряд его алгоритмов–потомков, теперь успешно развивают другое направление алгоритмов решения задач *DisCOP* (*Distributed Constraint Optimisation Problem*). Алгоритм же *AWCS* и все его многочисленные вариации утвердили себя как эффективный метод решения *DisCSP*. По оценке его авторов [3], *AWCS* в 3–10 раз эффективнее *Asynchronous Backtracking*.

Алгоритм 6. Процедура *responde to message*

```
procedure responde_to_message                          1
{                                                      2
  if message is inc penalty                            3
                                                       4
    if inc penalty on cur val <                        5
       upper bound                                     6
      increase inc penalty by 1                        7
    end if                                             8
                                                       9
  else                                                 10
     impose temp penalty                               11
  end if                                               12
                                                       13
  select new value                                     14
  send(id,val,null)                                    15
}                                                      16
```

Алгоритм 7. Псевдокод процедур *AWCS*

```
procedure receive_ok (value)
{
add value to agent_view;
check_agent_view()
}

procedure check_agent_view ( )
{
if curr val is not consistent
select new value

if can't find consistent val
backtrack()
else
send ok message to all
end if

end if
}
```

Алгоритм 8. Псевдокод процедур *AWCS*

```
1   procedure receive_nogood( )
2   {
3    add state to nogood_store
4    check_agent_view();
5   }
6
7   procedure backtrack ( )
8   {
9    add state to nogood_store
10   for all agents with higher prior
11   if not consistent with cur agent
12   send nogood message
13
14   prior ← max(all agents' prior) + 1
15   select new value with min number of
16 constraint violations
17
18   send ok message to all
19  }
```

Гибридный алгоритм *improve Guided Local and Systematic Search* (*GLoSS*). Алгоритм *GLoSS* по классификации гибридных алгоритмов относится к третьей категории, т.е. совмещает в себе компоненты конструктивного и локального поиска и характеризуется высоким уровнем их интеграции, при этом операторы конструктивного и локального поиска применяются поочередно.

База алгоритма – синхронный бектрекинг (*Synchronous Backtracking*). Он осуществляет инициализацию переменных. Учитывая синхронность компонента, агенты действуют не парал-

лельно, но им уже не требуется сообщать о присвоении значения каждому соседу отдельно. Они передают свой *agent view* следующему агенту со всеми известными им значениями соседей и собственным значением. Следующий агент, выбирающий себе значение, стремится выбрать его так, чтобы избежать возможных конфликтов с уже известными ему значениями соседей с высшим приоритетом. Этот процесс описывается процедурой *extend* (алг. 10).

Алгоритм 9. Главный цикл

```
procedure main
{
   a is an empty partial solution
   while a is not solution
   {
     do
     {
       extend( a )

       if a is solution
         terminate
       end if

     }until a is not consistent

     iGL( a )

     if a is not consistent
       emergency( a )
     end if

     if a is solution
       terminate
     end if
   }
}
```

Алгоритм 10. Процедура конструктивного поиска

```
1   procedure extend(a)
2   {
3      select value for current agent
4      add new value to a
5
6      if new value is consistent
7        if agent is not the last
8          send a to next agent
9
10       else
11         return a
12       end if
13
14     else
15       send a to all with higher prior
16       send ok mess to conflict agents
17     end if
18
19  }
20
21
22
23
24
25
26
27
```

В случае, когда очередной агент не может расширить частичное решение (не может найти себе значение), не попав в конфликтную ситуацию, происходит переход к локальному поиску (алг. 9, стр. 8–10):

- агент выбирает себе значение с наименьшим количеством конфликтов;
- агент рассылает всем соседям с высшим приоритетом свой *agent_view* для того, чтобы каждый агент имел собственный экземпляр частичного решения, и был в курсе всех присвоений;
- агент рассылает конфликтным агентам сообщение со своим значением, инициируя процедуру локального поиска**.**

Переход к локальному поиску происходит, когда в *SBT* должен начаться бектрекинг, но эта процедура в алгоритме *GLoSS* никогда не применяется. Следует заметить, что в локальном поиске будут участвовать только те агенты, которые уже прошли этап инициализации. Такой подход существенно сокращает количество сообщений, которыми агенты обмениваются в процессе решения.

Локальный поиск представлен в алгоритме 10 как *repair*-процедура. Алгоритм применяется к проинициализированным агентам максимум *n* раз. Ограничение необходимо для устранения проблемы неполноты локальных алгоритмов.

Закончив локальный поиск, активный агент с наивысшим приоритетом отправляет дежурному по приоритету агенту (еще неактивному) частичное решение, и тем самым инициирует новый вызов процедуры *Synchronous Backtracking*. Расширение частного решения продолжается до тех пор, пока очередной агент не сможет его расширить.

Алгоритм 11. Процедура локального поиска

```
procedure iGL(a)
{
   for n steps do
   {
      do iGL on a
      if a is consistent
         terminate
   }
}
```

Алгоритм 12. Процедура вспомогательного конструктивного поиска

```
1   procedure emergency(a)
2   {
3      AWCS(a)
4   }
5
6
7
8
9
```

В цепочке вызовов *SBT* и *iGL* могут возникнуть вызовы компоненты *AWCS* (алг. 12). Она становится нужна в случае, когда *iGL* не смог найти частичное решение за отведенные ему *n* шагов. Тогда *AWCS* завершает поиск, причем гарантированно (он, в отличие от *iGL*, – полный).

Остановимся на нескольких важных особенностях алгоритма. Все компоненты пользуются одним централизованным *nogood_store*, обеспечивая сохранение и передачу «неудачного» опыта, приобретенного агентами на каком-то из этапов работы гибридного алгоритма, всеми компонентами. Переход к следующему компоненту не происходит, если решение найдено. Для этого в алгоритме выполняются соответствующие проверки во всех трех компонентах (алг. 10–12).

Теорема 1. Гибридный алгоритм *GLoSS* – полный и корректный**.**

Доказательство. На каждой итерации главного цикла алгоритма происходит поиск допустимого частичного решения. Поиск организуется средствами как локального, так и конструктивного операторов. Конструктивный оператор выполняется после локального и гарантирует

нахождение допустимого частичного решения, если такое существует, или завершение работы алгоритма в связи с отсутствием решения.

На последней итерации главного цикла алгоритма частичное решение рассмотрения становится полным при привлечении к частичному решению последней незадействованной переменной. Поэтому последовательное выполнение над ним операторов локального поиска, а затем, возможно, *AWCS*, гарантированно дает решение всей задачи, если такое существует.

Теорема 2. Для решения *DISCSP* задачи с N переменными, в которой мощность каждого из доменов переменных не превышает M, временная сложность алгоритма *GLoSS* есть $O(M^N)$. Средняя временная сложность есть $O(M * N)$.

Доказательство. Оценка временной сложности гибрида требует оценки сложности каждой из его компонент.

Начнем с *SBT*. На каждой итерации алгоритма имеется выбор из M значений, которые можно присвоить переменной: $M + M + M + \ldots + M \sim M * N$. В этой оценке не учтен бектрекинг, поскольку он в данной реализации никогда не происходит.

Сложность работы алгоритма *iGL* можно оценить константой C. Вследствие неполноты алгоритма количество выполненных им итераций ограничивается некоторой константой C, которая есть оценкой алгоритма в худшем случае.

Алгоритм *AWCS* имеет экспоненциальную сложность в худшем случае. Поэтому сложность гибрида можно оценить как $M * N + C + M \wedge N$. Иначе говоря, просто $M \wedge N$. Это наихудшая оценка. Средней же оценкой алгоритма можно считать $M * N + C$ или просто $M * N$, поскольку алгоритм *AWCS*, по теоретическим оценкам, участвует в решении задачи не более чем в трех процентах случаев.

Отметим, что после появления метода *Asynchronous Weak-Commitment Search*, алгоритмы решения *DisCSP* стали оценивать еще и по объему памяти, в частности, необходимой для запоминания «неразрешимых ветвей» в процессе решения. Ведь выигрыш *AWCS* в скорости сравнительно с его предшественниками, различными модификациями бектрекинга, компенсируется непомерно большим пространством «плохих» состояний. Эту характеристику *AWCS* наследует и гибрид *GLoSS*. Для оптимизации памяти введена общая для всех трех компонент гибрида база неудовлетворительных состояний агентов, которая в указанных процедурах называется *nogood_store*. Для каждого из операторов поиска опыт других должен сократить его собственное пространство поиска.

Экспериментальная оценка эффективности алгоритма

Алгоритм *GLoSS* предназначен для решения бинарных *DisCSP*. Известно, что каждая *CSP* сводится к бинарной. Классическим примером задачи из этого класса служит задача о ферзи. Попытаемся оценить скорость работы алгоритма *GLoSS* на ней, варьируя количество ферзей.

Остановимся на основных моментах. Сначала поясним применение компонента *Synchronous Backtracking* – наименее эффективного алгоритма. В нашем алгоритме *GLoSS* компонента *SBT* используется только для расширения частичного решения, пока не возникнет первый конфликт. Разрешение конфликтных ситуаций происходит не путем бектрекинга, а средствами двух других компонентов алгоритма.

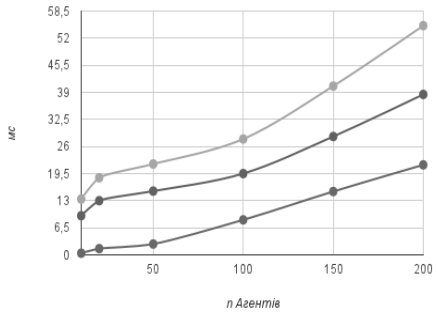
Согласно проведенным экспериментам, *SBT* способен за первый же проход присвоить значение в среднем 60–70 процентам агентов, на что расходуется примерно 20 процентов от общего времени решения. После этого управление процессом поиска передается компоненту *iGL*. Он примерно в 96–97 процентах случаев находит решение задачи, не превышая отведенный ему лимит шагов.

Возникает вопрос: как определить факт нахождения локальным поиском частичного решения и момент окончания работы этой компоненты? Самый простой способ – перерыв в работе в течение некоторого длительного промежутка времени, существенно зависящего как от самого алгоритма и способа его реализации, так и от аппаратного обеспечения.

Еще один вопрос требует комментария. Все агенты завершают выполнение процедуры локального поиска в разное время. Время оста-

новки какого из агентов следует считать временем окончания процедуры *iGL*? Поскольку для продолжения работы алгоритма требуется активный агент с самым высоким приоритетом, который бы направил следующему по приоритету (еще не активному) агенту частичное решение, инициировав начало работы процедуры *SBT*, то временем окончания процедуры локального поиска следует считать время окончания работы этого агента. Временем начала работы процедуры *SBT* следует считать момент отправки агентом (с наибольшим приоритетом среди всех агентов) сообщения с частичным решением следующему соседу. Конец работы процедуры *SBT* инициирует начало работы процедуры *iGL*, поэтому простоя между ними нет.

На рисунке изображены графики производительности трех алгоритмов: вертикальная ось отражает время в миллисекундах, горизонтальная – количество агентов, т.е. количество ферзей в задаче (поскольку каждому агенту соответствует один ферзь, т.е. одна переменная). Верхний график соответствует производительности работы *Dis-GLS*-алгоритма, средний – *iGL*-алгоритма, и нижний – алгоритма *GLoSS*. Итак, видим, что сочетание *SBT* с *iGL* дало выигрыш и положительный результат. Для количества ферзей менее 50 гибрид *GLoSS* тратит на решение в пять–восемь раз меньше времени, чем его эффективный компонент *iGL*.



Для большего количества ферзей (эксперименты проводились для количества ферзей до 200) производительность гибрида в полтора–два раза больше *iGL*.

Очевидно, что расширение частичного решения на одно значение не дает значительного прироста производительности. Во-первых, сам вызов процедуры *SBT* – затратный для алгоритма. Во-вторых, добавление к частичному решению только одной переменной говорит о том, что ее значение – конфликтно, поэтому *SBT* не может продолжать поиск, и процедура только «портит» допустимое частичное решение предыдущего этапа. Поэтому – это проблема на будущее.

Временную оценку гибрида *GLoSS* желательно было бы сравнить с оценками других гибридов. К сожалению, реализация даже одного гибрида требует значительных усилий, поэтому вся работа по оценке производительности алгоритма была сведена к его сопоставлению с производительностью работы его быстрой (эффективной) компоненты. Таким образом, удалось показать, что гибрид, по крайней мере, не хуже каждого из своих компонент.

Заключение. Сделан краткий обзор методов решения класса распределенных задач удовлетворения ограничений (*Disctributed Constraint Satisfaction Problem*). На основе анализа этих методов разработано два алгоритма: *iGL* (*improved Guided Local Search*), основанный на алгоритме локального поиска *Dis-GLS*, и гибридный алгоритм *GLoSS* (*Guided Local & Systematic Search*). Гибрид сочетает в себе алгоритмы *Synchronous Backtracking*, *improved Guided Local Search* и *Asynchronous Weak-Commitment Search*. Компоненты гибридного алгоритма отличаются не только методами поиска решения задачи (локальный и конструктивный), но и способом синхронизации. Быстрый, но неполный алгоритм локального поиска *iGL* подстрахован конструктивным алгоритмом *AWCS* для обеспечения нахождения решения, если оно существует. Синхронный главный алгоритм существенно уменьшает поток сообщений между агентами, обеспечивая последовательную обработку переменных. Недостаток синхронного бектрекинга – отсутствие распараллеливания, компенсируется наличием параллельности вспомогательных алгоритмов *iGL* и *AWCS*.

В план дальнейших исследований входит усовершенствование алгоритма, связанное с поиском решения проблемы эффективности сотрудничества компонента *SBT* с другими компонентами гибрида, а также проблема рационализации использования памяти, необходимой для хранения пространства ветвей решения, не приводящих к решению.

Окончание на стр. 88

1. *Schulte Ch.*, *Carlsson M.* Finite Domain Constraint Programming Systems // Handbook of Constraint Programming / F. Rossi, P. van Beek, T. Walsh (Eds.). – Elsevier, 2006. – P. 493–524.
2. *Letichevsky A.*, *Gilbert D.* A Model for Interaction of Agents and Environments // Lecture Notes in Comp. Sci. N 1827. Recent Trends in Algebraic Development Techniques. – 2000. – P. 119–131.
3. *Yokoo M.* Asynchronous Weak-commitment Search for solving Distributed Constraint Satisfaction Problems // Proc. of the First Int. Conf. on Principles and Practice of Constraint Programming, 1995. – P. 88–102.
4. *Brito I.* Synchronous, Asynchronous and Hybrid Algorithms for DisCSP // Proc. of the Tenth Int. Conf. on Principles and Practice of Constraint Programming (CP–2004). Lecture Notes in Comp. Sci., Jan. 2004. – **3258**. – 791 p.
5. *Silaghi M.-C.*, *Faltings B.* Asynchronous aggregation and consistency in distributed constraint satisfaction, Artificial Intelligence, Jan. 2005. – **161**, Issues 1–2. – P. 25–53.
6. *Distributed* Constraint Satisfaction for Formalizing Distributed Problem Solving / M. Yokoo, E. Durfee, T. Ishida et al. // Proc. of the Twelfth IEEE Int. Conf. on Distributed Computing Systems, 1992. – P. 614–621.
7. *Yokoo M.*, *Hirayama K.* Distributed Constraint Satisfaction Algorithm for Complex Local Problems // Third Int. Conf. on Multiagent Systems (ICMAS–98), 1998. – P. 372–379.
8. *Zivan R.*, *Meisels A.* Synchronous and Asynchronous Search on DisCSPs // Proc. of EUMAS–2003, Oxford, UK, 2003. – P. 103–107.
9. *Salido M.F.*, *Barber F.* Stochastic Local Search for Distributed Constraint Satisfaction Problems // Proc. of IJCAI Workshop on Stochastic Search Algorithms, Acapulco, México, 2003. – P. 49–54.
10. *Basharu M.*, *Arana I.*, *Ahriz H.* Distributed Guided Local Search for Solving Binary DisCSPs // Proc. of the 18th Int. FLAIRS (The Florida AI Research Society) Conf. 16–18 May 2005. Florida, 2005. – P. 660–665.
11. *Hirayama K.*, *Yokoo M.* Distributed Breakout Algorithm for Solving Distributed Constraint Satisfaction Problems // Proc. of the Second Int. Conf. on Multi-Agent Systems, MIT Press, 1996. – P. 401–408.
12. *Hoos H.H.*, *Tsang E.* Local search methods. Foundations of Artificial Intelligence. – 2006. – **2**. – P. 135–167.
13. *Morris P.* The Breakout method for escaping from the local minima // Proc. of the Eleventh Nat. Conf. on Artificial Intelligence, 1993. – P. 40–45.
14. *Basharu M.*, *Arana I.*, *Ahriz H.* Solving DisCSPs with Penalty Driven Search // Proc. of the 20th national conf. on Artificial intelligence, 2005. – **1**. – P. 47–42.
15. *Chatzikokolakis K.*, *Boukeas G.*, *Stamatopoulos P.* Construction and Repair: A Hybrid Approach to Search in CSPs. G.A. Vouros, T. Panayiotopoulos (Eds.). – SETN 2004, LNAI 3025, 2004. – P. 342–351.
16. *Eisenberg C.*, *Faltings B.* A Hybrid Solving Scheme for Distributed Constraint Satisfaction Problems // The Fourth Int. Workshop on Distributed Constraint Reasoning at IJCAI, 2003. – P. 19–35.
17. *Jussien N.*, *Lhomme O.* Local search with constraint propagation and conflict-based heuristics. Artificial Intelligence 139, 2002. – P. 21–45.
18. *A Hybrid* Approach to Distributed Constraint Satisfaction / D. Lee, I. Arana, H. Ahriz et al. // Proc. of the 13th int. conf. on Artificial Intelligence, 2008. – P. 375–379.
19. *Multi-hyb*: a hybrid algorithm for solving DisCSPs with complex local problems / D. Lee, I. Arana, H. Ahriz et al. / P. Boldi, G. Vizzari, G. Pasi, R. Baeza-Yates (Eds.) // Web Intelligence and Intelligent Agent Technologies, 2009. – P. 379–382.
20. *Basharu M.*, *Arana I.*, *Ahriz H.* DisBO-wd: a distributed constraint satisfaction algorithm for coarse-grained distributed problems / M. Bramer, F. Coenen, M. Petridis (Eds.). Research and Development in Intelligent Systems XXIV // Proc. of the 27th SGAI Int. Conf. on Artificial Intelligence, AI–07. 10–12 Dec. 2007, Cambridge, 2007. – P. 23–36.

Тел. для справок: +38 044 463-6985, +38 050 720-9361
E-mail: *gloria.jjl@gmail.com*, *glib@ukma.kiev.ua*, *gor@ukma.kiev.ua*