Севастопольский государственный университет
Институт информационных технологий

Методы и системы искусственного интеллекта

Бондарев Владимир Николаевич

Лекция

ЗУО — Задачи удовлетворения ограничений (CSPs — Constraint Satisfaction Problems)

В.Бондарев

Общая формулировка задачи УО

Целью решения ЗУО (CSP) является нахождение значений переменных, удовлетворяющих определенным ограничениям.

В ИИ парадигма УО признана в качестве эффективного способа решения многих прикладных задач:

- назначение ресурсов;
- календарное планирование;
- обработка (разметка) изображений;
- тестирование СБИС;
- понимание ЕЯ;
- коньюнктивные запросы к БД;
- комбинаторная оптимизация и др.

Общая формулировка задачи УО

Задача поиска в пространстве состояний:
состояние – “черный ящик” (допустима любая подходящая структура данных, которая используется функциями раскрытия вершины, вычисления эвристики, проверки достижения цели)

В задачах CSP состояния и проверка целей соответствуют стандартному, очень простому представлению. Что позволяет создать **общие процедуры поиска решений**.

CSP задача определяется совокупностью трех составляющих :
1) множеством переменных $X1, X2, \ldots, Xn$ (состояние);
2) областью определения каждой переменной $D1, D2, \ldots, Dn$;
3) множеством ограничений (отношений) $C1, C2, \ldots, Cm$, каждое из которых включает некоторое подмножество переменных и задает допустимые комбинации значений для этого подмножества.

В.Бондарев

Общая формулировка задачи УО

Состояние задачи определяется путем присваивания значений некоторым или всем переменным.

Присваивание, которое не нарушает никаких ограничений, называется ***совместимым.***

Полным называется присваивание, в котором участвует каждая переменная, а ***решением*** задачи CSP является полное присваивание, которое удовлетворяет всем ограничениям.

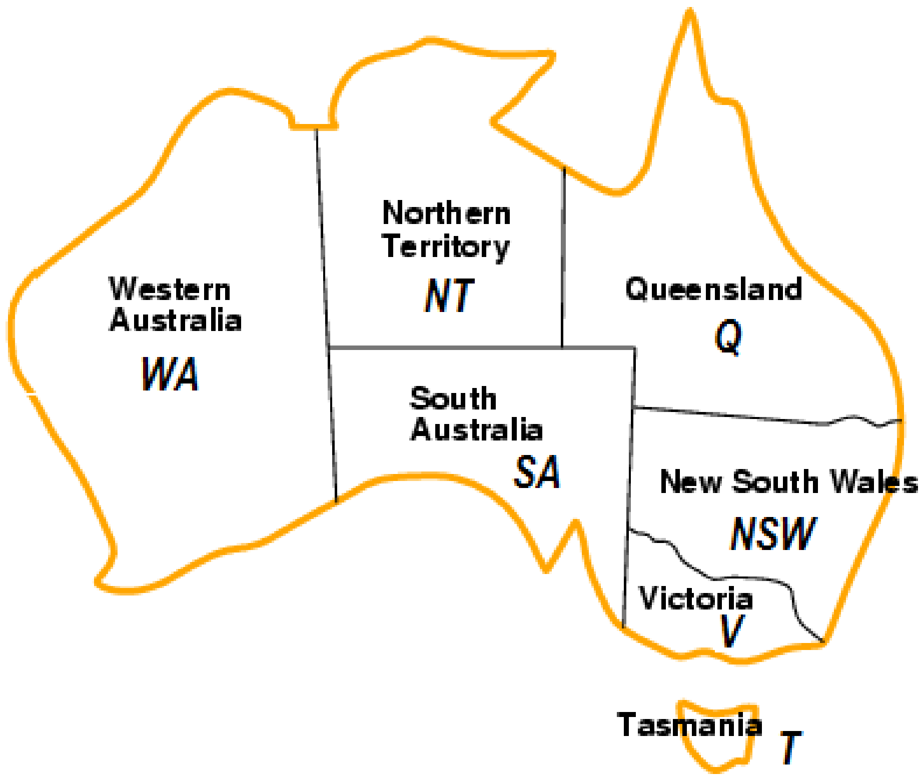
Такое описание – простой язык формального представления многих проблем. Допускает использование общих алгоритмов, позволяющих решать задачи на несколько порядков более крупные по сравнению алгоритмами поиска решений в пространстве состояний.

Особенность CSP-задач

1. Задачи поиска в пространстве состояний – в основном планирование действий

2. CSP-задачи – задачи идентификации (не важен путь, важно только целевое конечное состояние)

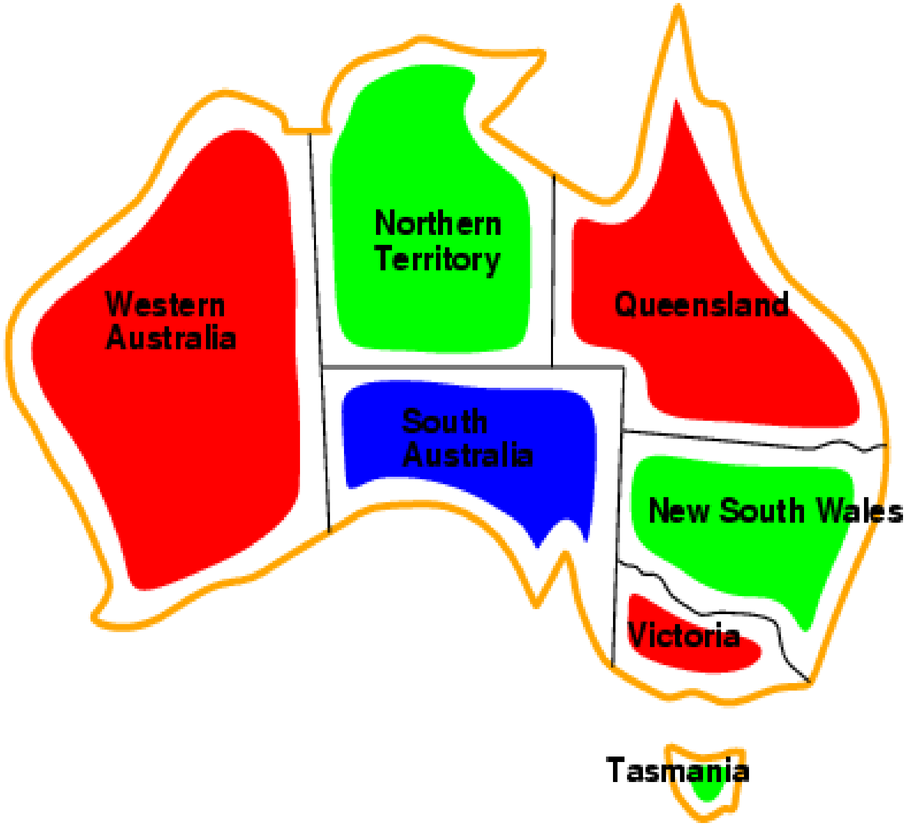
Задача раскрашивания плоской карты



- Переменные ***Xi*** : ***WA, NT, Q, NSW, V, SA, T*** .
- Области определения D_i = {red, green, blue}.
- Ограничения: соседние регионы должны иметь разные цвета:

WA ≠ NT или пары **(WA,NT)** выбирать из {(red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green)}

Задача раскрашивания плоской карты

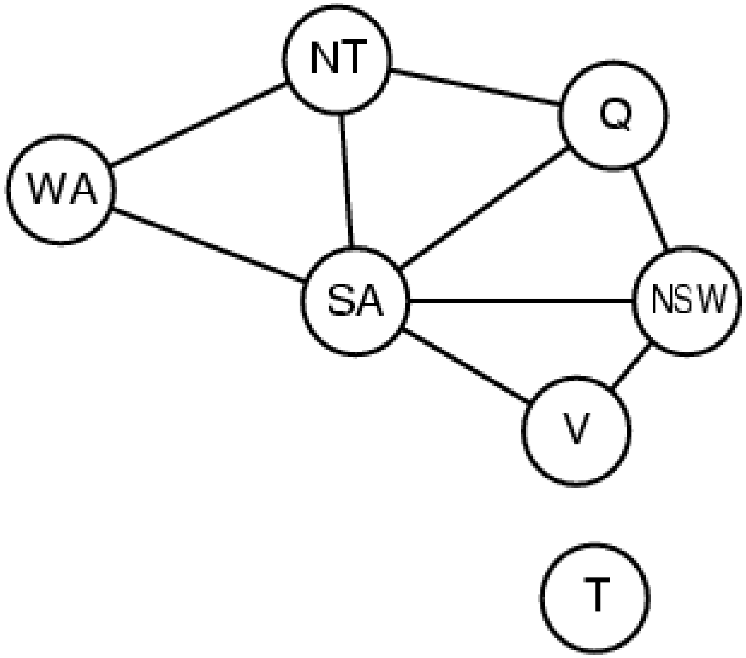


- **Решение** – полное совместимое присваивание, например:

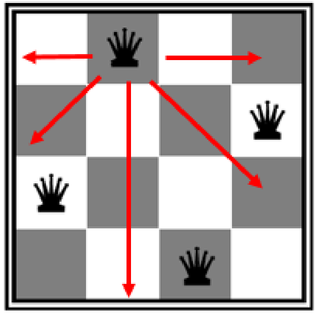
WA = red, **NT** = green, **Q** = red, **NSW** = green, **V** = red,
SA = blue, **T** = green

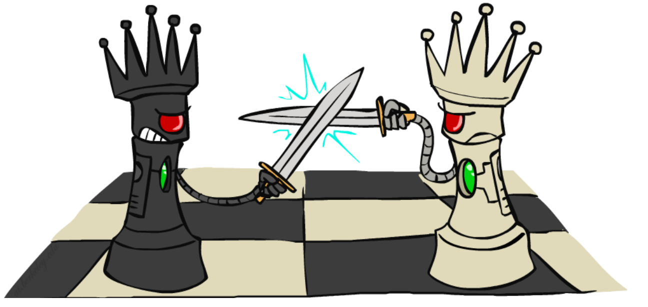
Граф ограничений

CSP задачу удобно представлять в виде ***графа ограничений***, узлы которого представляют переменные задачи, а дуги — ограничения.



Пример: N-ферзей





- **Формулировка задачи:**
  - Переменные: X_{ij}
  - Область: $\{0, 1\}$
  - Ограничения:

$$\forall i, j, k \ \ (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \ \ (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \ \ (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \ \ (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

$$\sum_{i,j} X_{ij} = N$$

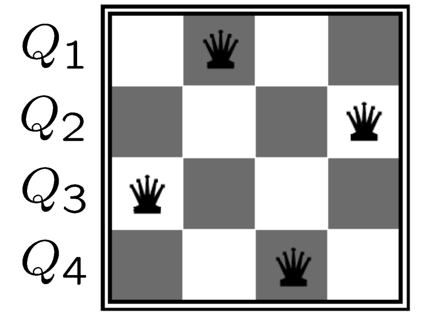
Пример: N-ферзей

- **Формулировка 2:**
  - Переменные: Q_k
  - Область: $\{1, 2, 3, \ldots N\}$
  - Ограничения:

Неявные: $\forall i, j$ не угрожают (Q_i, Q_j)

Явные: $(Q_1, Q_2) \in \{(1,3), (1,4), \ldots\}$

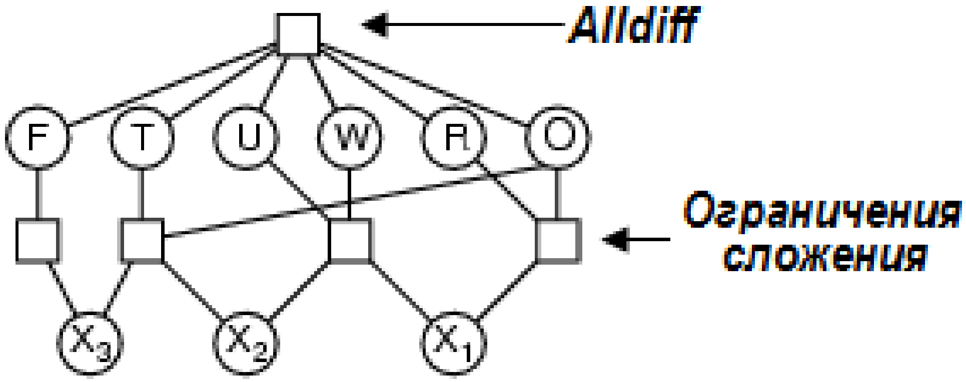
$\ldots$



В.Бондарев

Пример числового ребуса

```
   T W O
 + T W O
 -------
 F O U R
```



Каждая буква – отдельная цифра

Ведущие нули не допускаются

Переменные: ***F, T, U, W, R, O, X_1, X_2, X_3***; $\mathbf{X_i}$={0|1} вспомог.перемен.

Области определения: **{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};**

Ограничения высокого порядка: ***Alldiff (F,T,U,W,R,O)***

$$O + O = R + 10 \cdot X_1$$

$$X_1 + W + W = U + 10 \cdot X_2$$

$$X_2 + T + T = O + 10 \cdot X_3$$

$$X_3 = F,\ T \neq 0,\ F \neq 0$$

Гиперграф: ребра соединяют более 2-х узлов

Ограничения высокого порядка могут быть сведены к бинарным ограничениям путем введения вспомогательных переменных.

Разновидности CSP задач

Дискретные переменные :

1) *конечная область определения*:

n переменных, размер области определения ***d*** , количество возможных полных присваиваний оценивается величиной $\boldsymbol{O(d^{n})}$; пример, выполнимость лог. выражений – SAT проблема выполнимости конъюнктивной нормальной формы (NP-полная);

2) *бесконечная область определения*:

-целые, строки и др.; (невозможно перебрать все значения)

Непрерывные переменные : задачи линейного программирования при линейных ограничениях (пример, время начала и окончания наблюдений из непрерывной области определения для космического телескопа Хабл).

Виды ограничений

Унарные ограничения включают только одну переменную, например, **SA ≠ green** — ограничивает значение одной переменной. Унарное ограничение можно устранить, удалив соответствующее значение из области определения переменной.

Бинарные ограничения включают две переменные,например, **SA ≠ WA**

Ограничения высокого порядка включают 3 или более переменных, например, в криптоарифметических головоломках(числовых ребусах) ограничения столбцов

Предпочтения (мягкие ограничения):

- например, красный лучше чем зеленый;
- часто представляются стоимостью присваивания;
- приводят к задачам оптимизации в ограничениях.

(Будем игнорировать их, до знакомства с сетями Байеса).

Методы решения CSP задач

1. Метод «генерируй и тестируй» (generate and test).

Метод также называют методом «образуй и проверь».

В соответствии с этим методом генерируются все возможные полные присваивания переменным и каждое из них тестируется (проверяется) на совместимость.

В большинстве случае «слепая» генерация всех полных присваиваний весьма неэффективна, т.к. приводит к разрастанию дерева поиска. Можно рассматривать такой поиск как **поиск в ширину на дереве**, тогда коэфф. ветвления на каждом из уровней будут следующими:

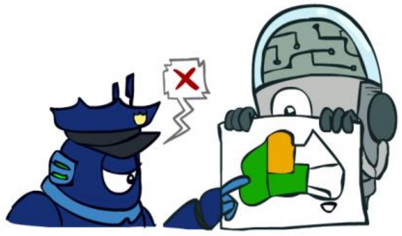
$$\left.\begin{array}{l} n\,d \\ (n-1)\,d \\ (n-2)\,d \\ \ldots \\ 1\,d \end{array}\right\}$$

- создается дерево с $n!\,d^{n}$ ветвями, несмотря на то, что имеется только d^{n} полных присваиваний

Методы решения CSP задач

2. Поиск с возвратами (backtracking search).

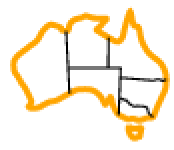
По сути, это тот же метод «генерируй и тестируй», но организованный в виде **поиска в глубину**, в котором присваивается значение очередной переменной, проверяются ограничения и выполняется возврат, если присвоение не допустимо.



Таким образом, проверка ограничений как бы погружается в процесс генерации решения, что позволяет ограничить разрастание дерева поиска.

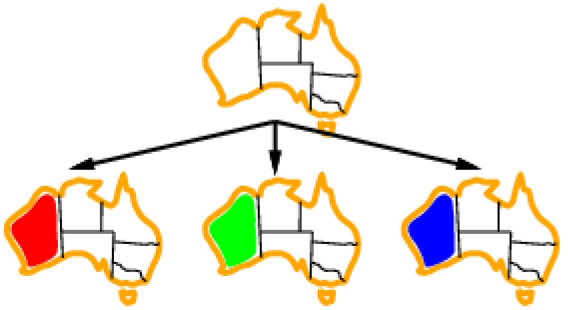
На каждом шаге коэффициент ветвления $b=d$, всего n шагов, поэтому получаем $d^{\,n}$ листьев

2.Поиск с возвратами

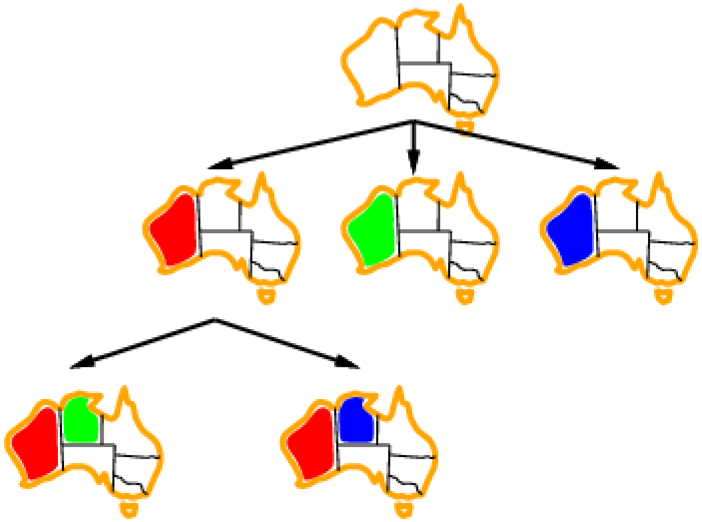


В.Бондарев

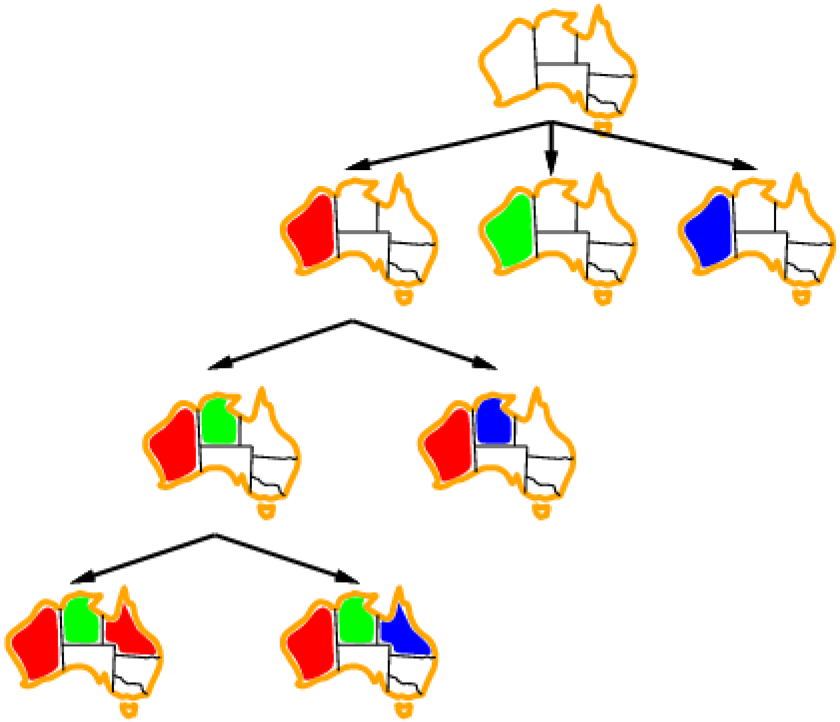
2. Поиск с возвратами



2. Поиск с возвратами



2.Поиск с возвратами



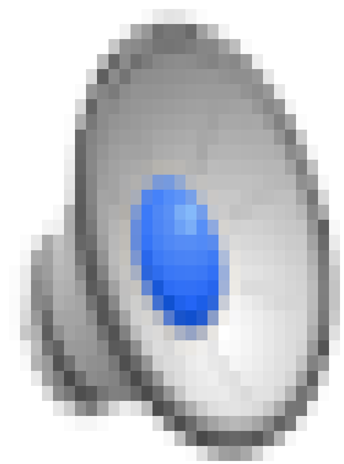
2.Поиск с возвратами

```
function BACKTRACKING-SEARCH(csp) returns решение/неудача
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns реш/неудача
   if присваивание assignment полное then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if значение value совместимо с присв . assignment
                     согласно ограничениям CONSTRAINTS[csp] then
         добавить {var = value} к присваиванию assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failure then return result
         удалить {var = value} из assignment
   return failure
```

- Поиск с возвратами = DFS + упорядочение переменных + неудача при нарушении ограничений
- Функции Select-Unassigned-Variable и Order-Domain-Values используют эвристики, рассматриваемые ниже.

Демо - Раскрашивание с возвратами



Совершенствование поиска с возвратами

Эффективность поиска зависит:

1) от порядка выбора переменных;

2) от порядка выбора возможных значений;

3) от возможности раннего обнаружения неудачи.

Упорядочение выбора переменных и значений выполняют с помощью **эвристик**:

1) Наименьшего кол-ва оставшихся значений (**MRV — M**inimum **R**emaining **V**alues);

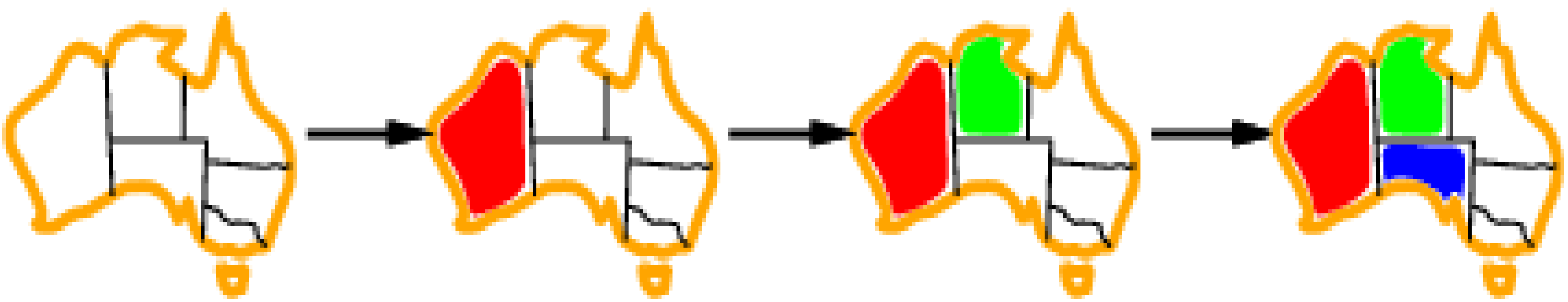
2) Степенной эвристики;

3) Наименее ограничительного значения.

Эвристика наименьшего кол-ва оставшихся значений (MRV)

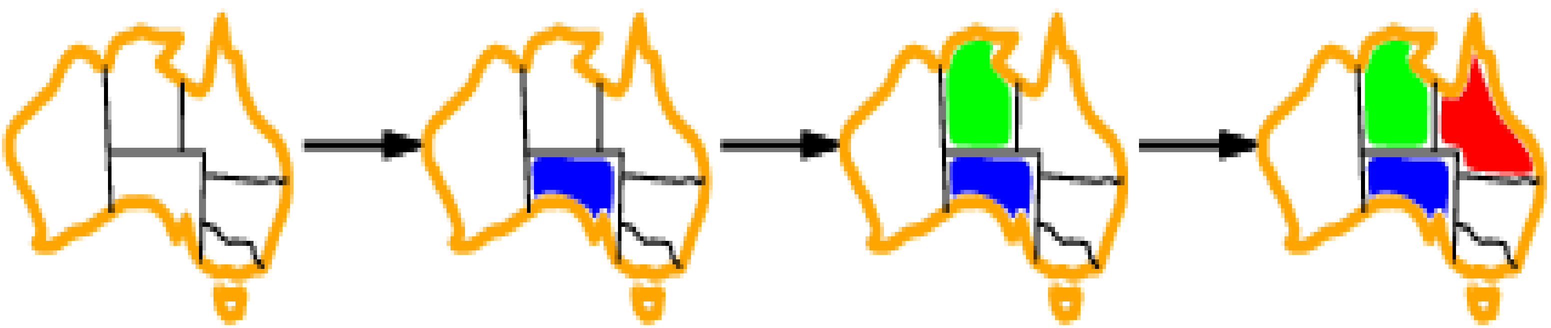
Эвристика предусматривает на каждом шаге *выбор переменной с наименьшим количеством оставшихся допустимых значений* .

Если существует переменная с нулевым количеством оставшихся значений, то именно она и будет выбрана, и неудача будет обнаруживаться быстрее.



Степенная эвристика

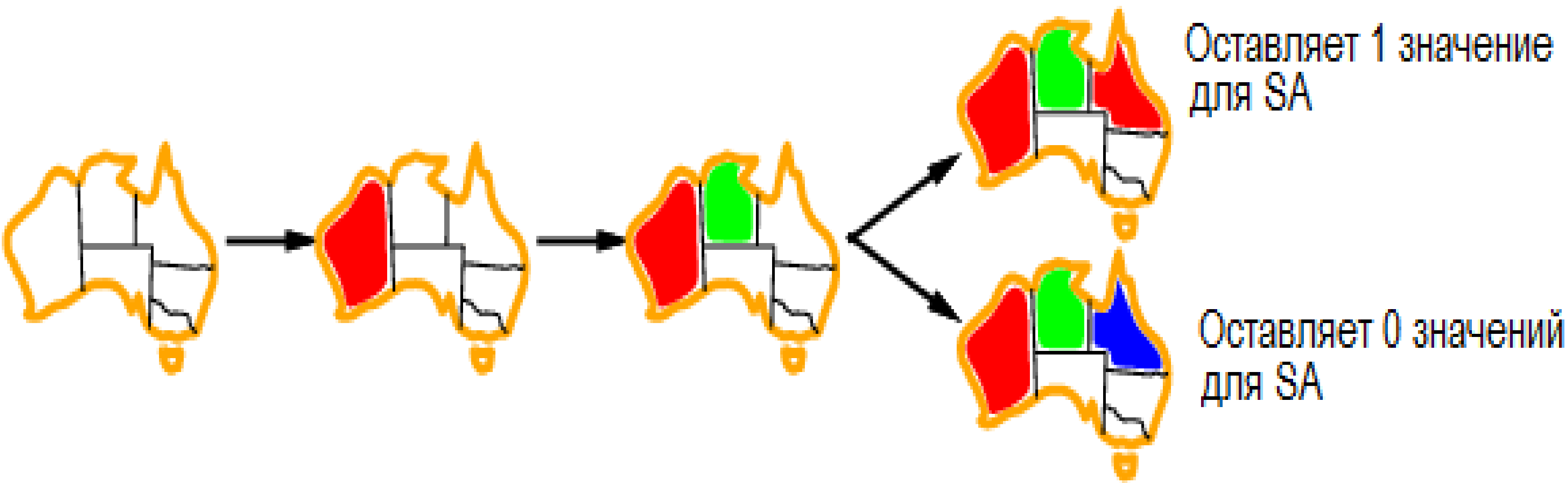
Выбор переменной, которая участвует в наибольшем количестве ограничений на другие переменные с не присвоенными значениями. Такой выбор сокращает области определения для большого количества связанных переменных, уменьшая коэффициент ветвления.



SA = 5
T = 0
Все другие 2 или 3

Эвр-ка наименее ограничительного значения

После выбора одной из переменных, необходимо принять решение, в каком порядке должны просматриваться значения. Предпочитается значение, которое исключает из рассмотрения наименьшее количество вариантов выбора значений для других связанных переменных. Т.е. эвристика пытается сохранить максимальную гибкость для последующих присваиваний.



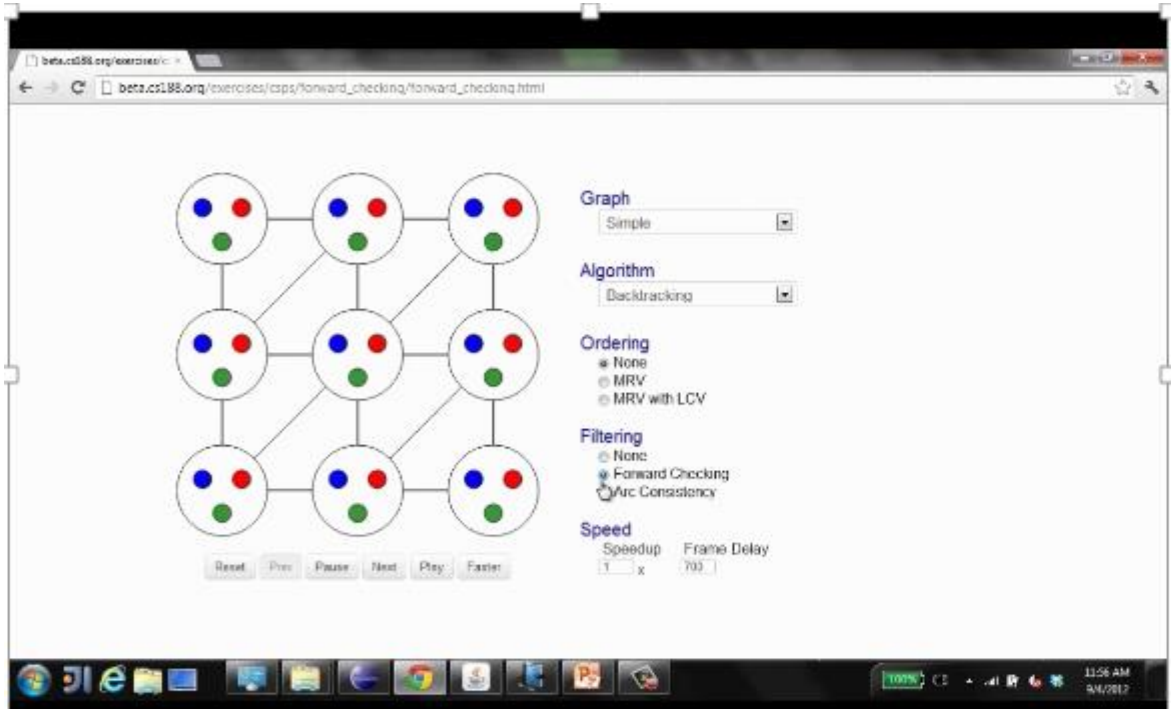
Комбинация этих эвристик повышает эффективность поиска при решении некоторых CSP задач до 1000 раз.

3. Фильтрация: Опережающая проверка

Выполнение опережающей проверки некоторых ограничений на предшествующих этапах поиска, чтобы сократить область значений переменных, которым еще не присвоены значения.

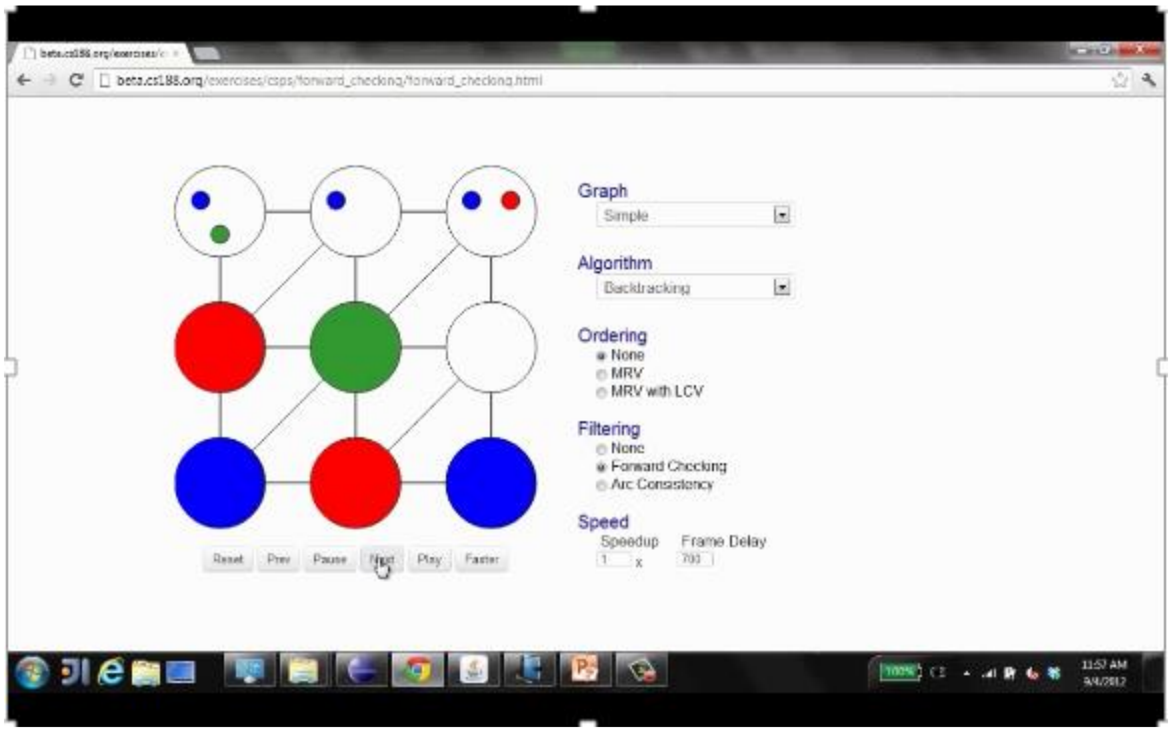
Когда присваивается значение переменной *X*, то из области определения связанной переменной Y **удаляется** любое значение, которое несовместимо со значением, присвоенным переменной *X*. Если какая-либо область определения становится пустой, то возникает возврат.

Пример: опережающая проверка

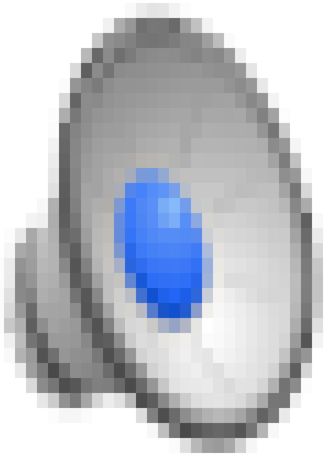


В.Бондарев

Пример: опережающая проверка



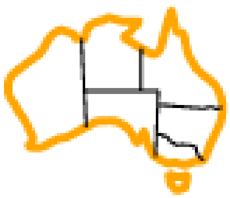
Демо – опережающая проверка

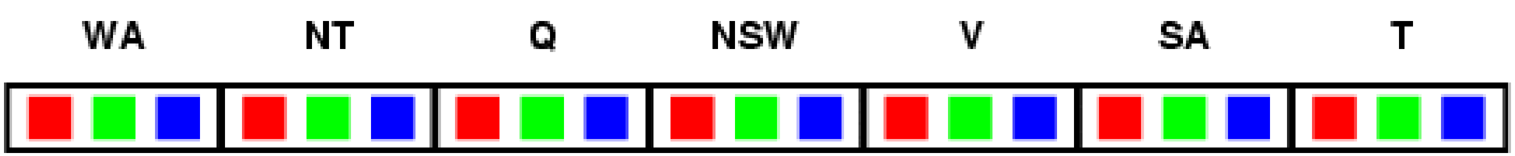


В.Бондарев

3. Опережающая проверка

Идея: выполнять опережающую проверку некоторых ограничений на предшествующих этапах поиска, чтобы сократить пространство поиска .
Когда присваивается значение переменной *X*, то из области определения связанной переменной Y удаляется любое значение, которое несовместимо со значением, присвоенным переменной *X*.
Если какая-либо область определения становится пустой, то возврат



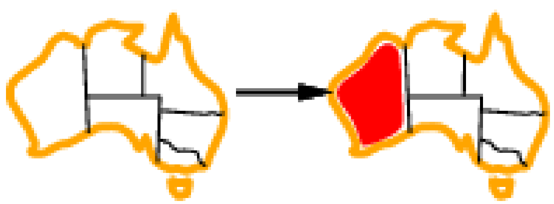


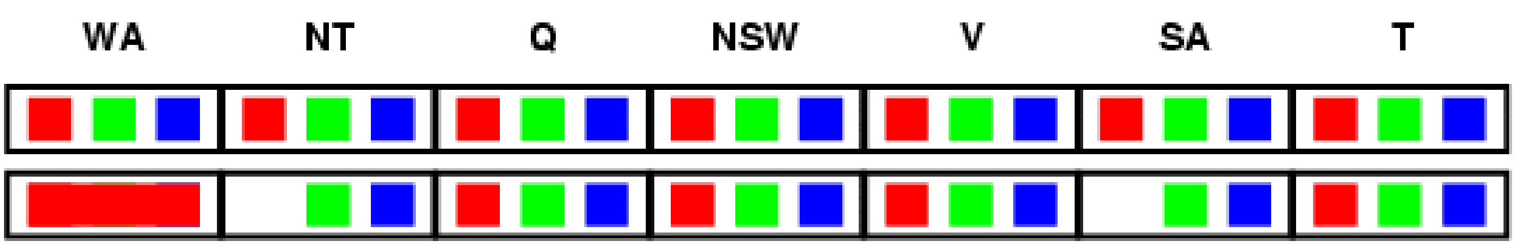
3. Опережающая проверка

Идея: выполнять опережающую проверку некоторых ограничений на предшествующих этапах поиска, чтобы сократить пространство поиска .

Когда присваивается значение переменной *X*, то из области определения связанной переменной Y удаляется любое значение, которое несовместимо со значением, присвоенным переменной *X*.

Если какая-либо область определения становится пустой, то возврат



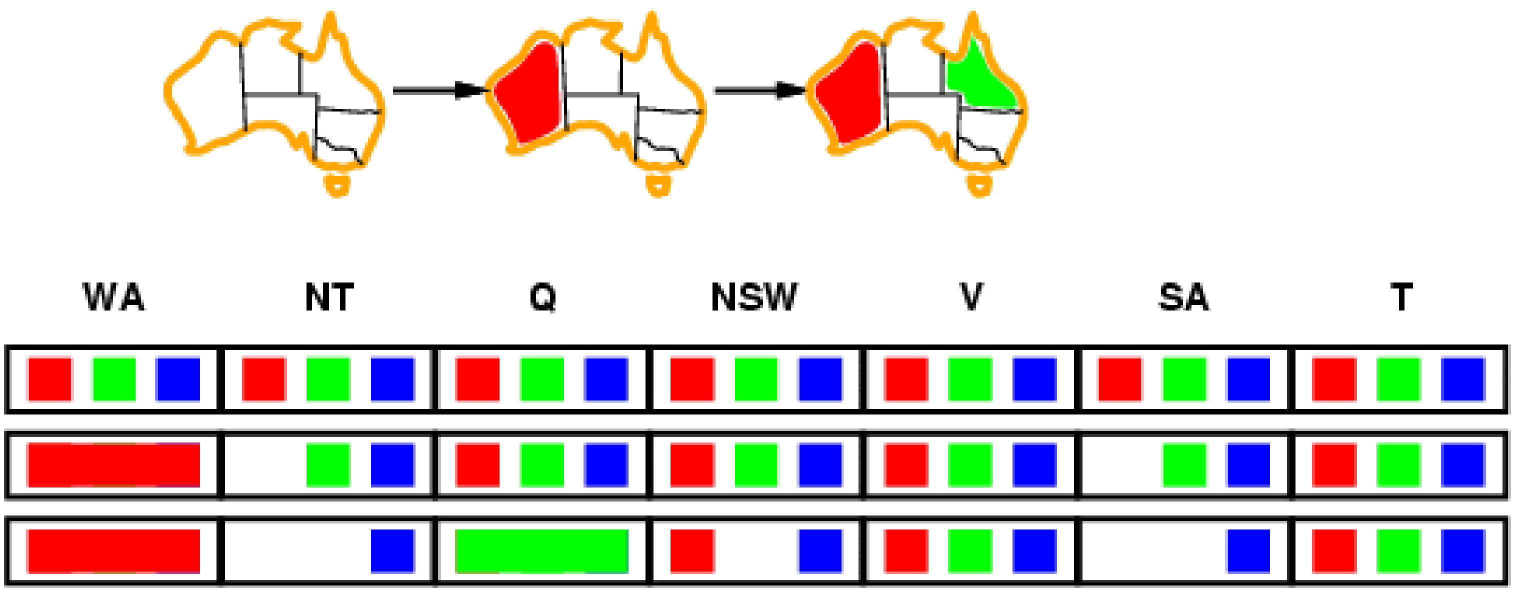


3. Опережающая проверка

Идея: выполнять опережающую проверку некоторых ограничений на предшествующих этапах поиска, чтобы сократить пространство поиска .

Когда присваивается значение переменной *X*, то из области определения связанной переменной Y удаляется любое значение, которое несовместимо со значением, присвоенным переменной *X*.

Если какая-либо область определения становится пустой, то возврат

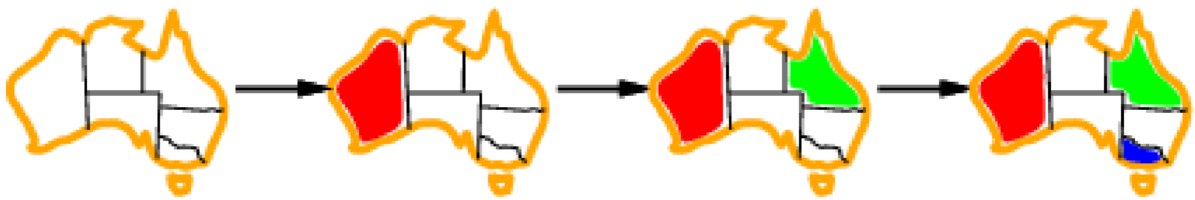


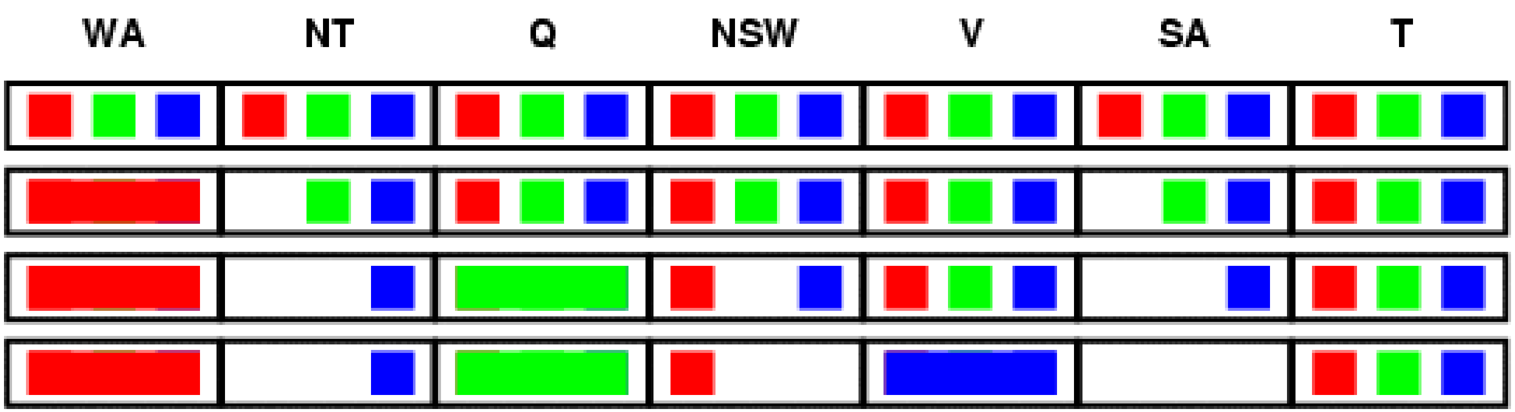
3. Опережающая проверка

Идея: выполнять опережающую проверку некоторых ограничений на предшествующих этапах поиска, чтобы сократить пространство поиска .

Когда присваивается значение переменной *X*, то из области определения связанной переменной Y удаляется любое значение, которое несовместимо со значением, присвоенным переменной *X*.

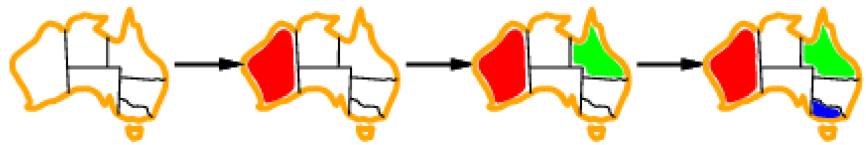
Если какая-либо область определения становится пустой, то возврат

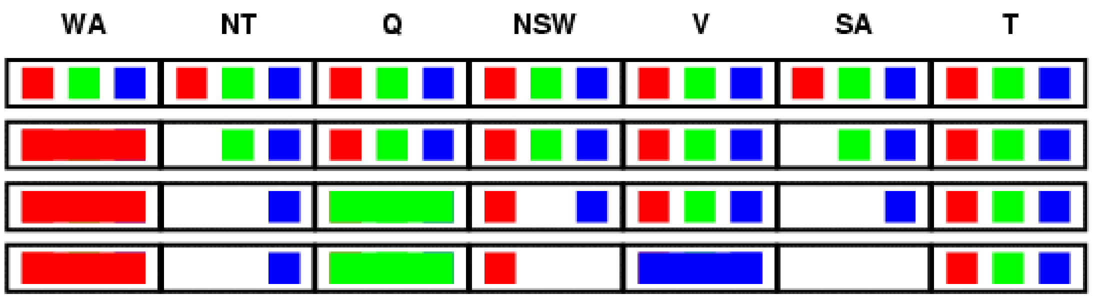




В.Бондарев

3. Опережающая проверка



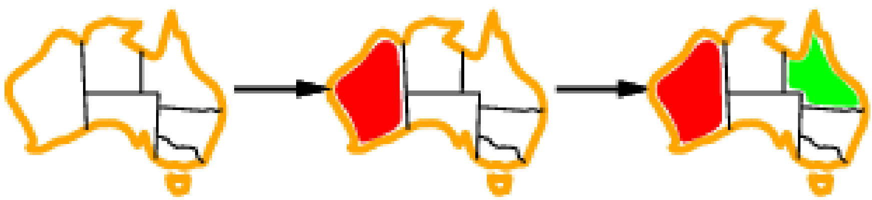


Выводы.:
1. После присваивания **WA=red** и **Q= green** области определения переменных **NT, SA** сокращаются до единственного значения; таким образом, ветвление, связанное с поиском значений для этих переменных, было полностью устранено путем распространения информации, касающейся переменных **WA и Q**. Применение эвристики MRV позволяет на следующем этапе автоматически выбрать значение для переменных **SA** и **NT**.
2. После присваивания **V=blue** область определения **SA** становится пустой. Поэтому предварительная проверка позволила обнаружить, что частичное присваивание **{WA=red,Q=green, V=blue}** является несовместимым с ограничениями этой задачи, значит, алгоритм **немедленно выполняет возврат.**

В.Бондарев

4. Фильтрация: Распространение ограничения

Предварительная проверка распространяет информацию от переменных с присвоенными значениями к переменным с не присвоенными значениями, но не обеспечивает раннее обнаружение всех несовместимостей.



NT и SA не могут одновременно иметь синий цвет!
Почему бы не обнаруживать это раньше?
Распространить ограничения: от ограничения к ограничению

4. Распространение ограничения

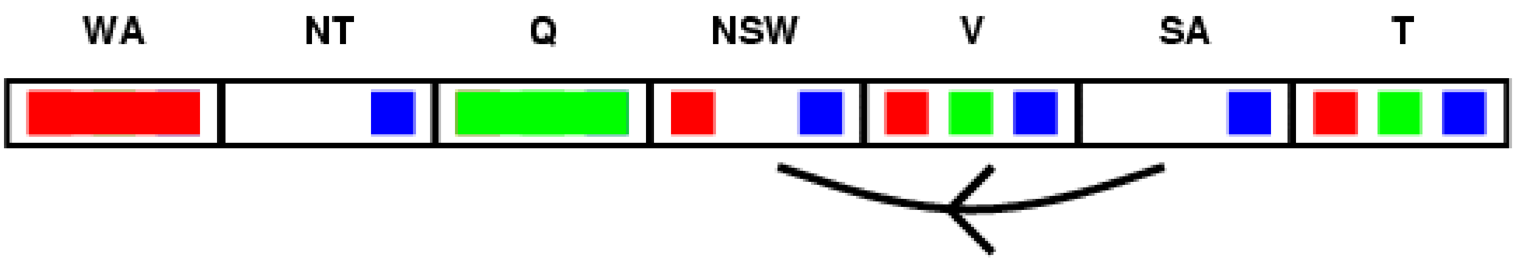
Распространение ограничения — это общее название методов обнаружения потенциальных несовместимостей на ранних этапах решения задачи за счет распространения последствий применения некоторого ограничения к одной переменной.

Для быстрого распространения ограничения выполняется проверка ***совместимости дуг***. Дуга (X, Y) называется ***совместимой***, если для каждого значения x из области определения переменной X существует некоторое значение y из области определения переменной Y, которое удовлетворяет бинарному ограничению между переменными X и Y.
Понятие совместимости дуг является **направленным**, т.е. если дуга (X, Y) совместима, то это не означает, что обратная дуга (Y, X) также совместима.

Совместимость дуг

Дуга $X \rightarrow Y$ совместима, е.е.

для каждого значения x из X существует допустимое y.

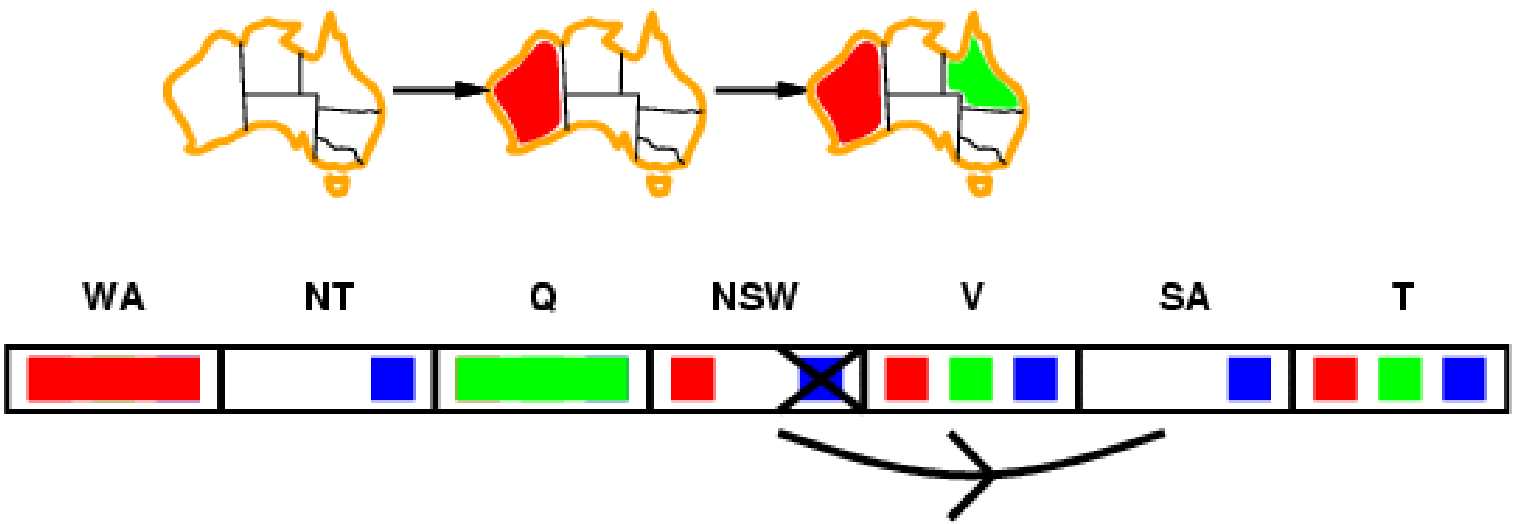


В.Бондарев

Совместимость дуг

Дуга $X \rightarrow Y$ совместима, е.е.

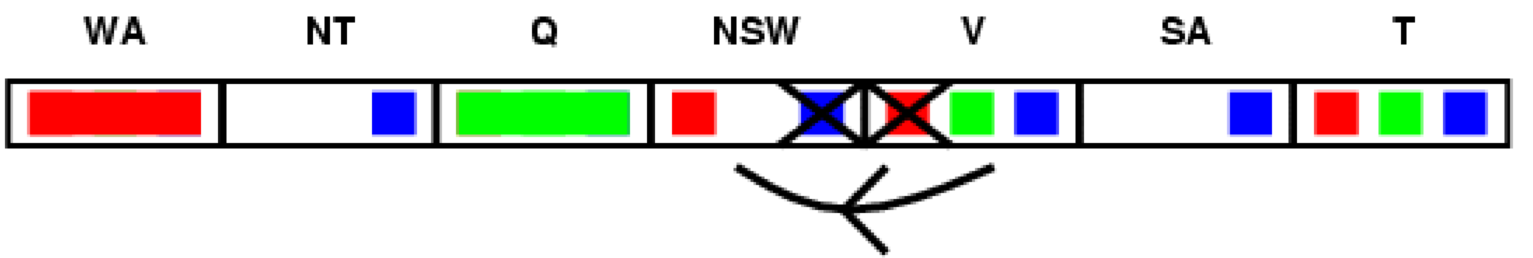
для каждого значения x из X существует допустимое y



Обратную дугу можно сделать совместимой, удалив значение **blue** из области определения ***NSW***.

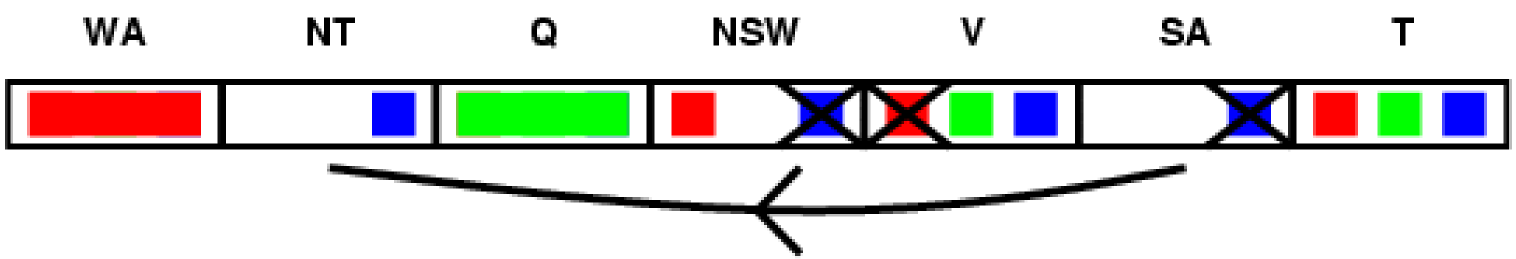
Совместимость дуг

Однако, если из области *X* удалено значение, то необходимо перепроверить совместимость всех «соседей» *X*.



Совместимость дуг

Проверка совместимости дуг позволяет обнаруживать неудачу раньше, чем предварительная проверка. Проверка совместимости дуги ***SA* → *NT*** на этапе присвоения ***Q*=green** показывает, что **blue** должно быть удалено из области определения ***SA*.** И эта область становится пустой (раннее обнаружение несовместимости).



В.Бондарев

Алгоритм проверки совместимости дуг

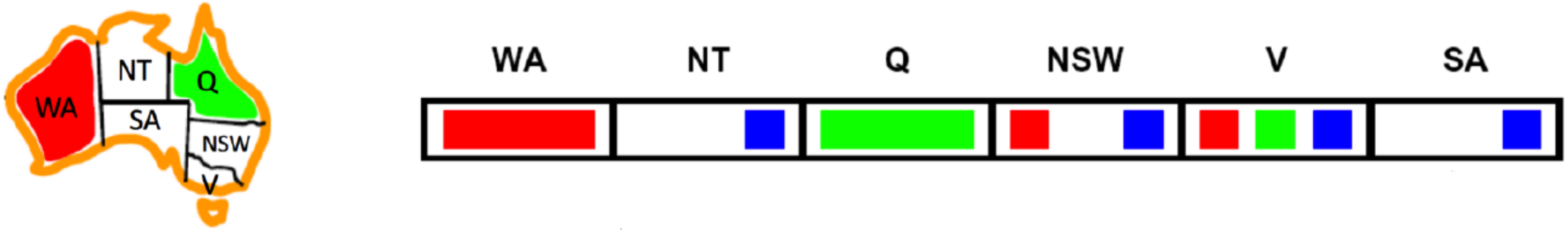
1. Сохранить все дуги графа ограничений CSP в очереди Q.

2. Итеративно удалять дуги из Q и проверять выполнение условия, чтобы для удаленной дуги $Xi \rightarrow Xj$ для каждого оставшегося значения v переменной Xi, имелось по крайней мере одно оставшееся значение w для переменной Xj такое, чтобы $Xi = v$ и $Xj = w$ не нарушало никаких ограничений. Если какое-то значение v для Xi не согласуется с любым из оставшихся значений для Xj, то удалить v из множества возможных значений для Xi.

3. Если хотя бы одно значение удалено для Xi при проверке согласованности дуги $Xi \rightarrow Xj$, добавьте дугу в форме $Xk \rightarrow Xi$ в Q для всех переменных Xk с не присвоенными значениями. Если дуга $Xk \rightarrow Xi$ уже находится в Q, то её не нужно добавлять.

4. Повторяйте до тех пор, пока очередь Q не станет пустой или домен некоторой переменной не станет пустым и не вызовет возврат.

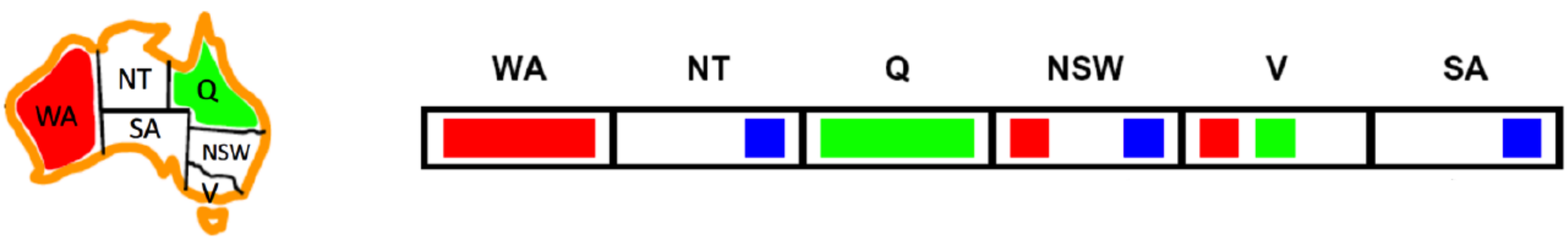
Совместимость дуг



Мы начинаем с добавления всех дуг между переменными **с неприсвоенными значениями**, участвующими в ограничениях, в очередь Q:

Q = [SA→V, V →SA, SA→NSW, NSW →SA, SA→NT, NT →SA, V →NSW, NSW→V]

Для первой дуги SA→V для каждого значения из области SA {blue}, есть по крайней мере одно значение в области V {red; green; blue}, которое не нарушает ограничений, и поэтому никакие значения не нужно удалять из SA. Однако для V→SA, если мы установим V = blue, то увидим, что для SA не будет согласованных значений и поэтому мы **удаляем синий цвет из области V**.

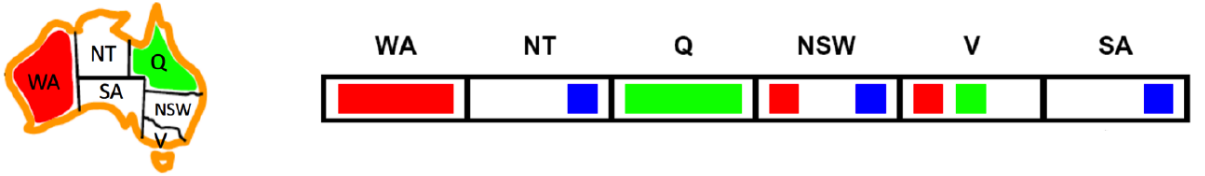


В.Бондарев

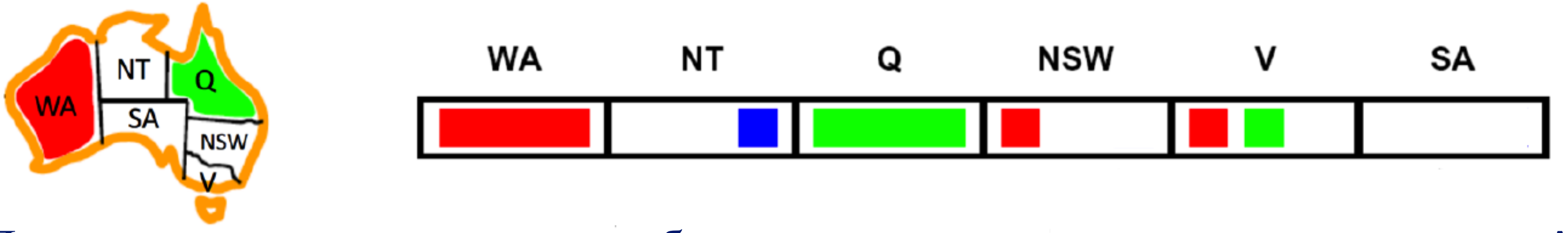
Совместимость дуг

Т.к. удалено значение из области V, то **нужно поставить в очередь все дуги с головой V**, т.е. SA→V, NSW→ V. Т.к. NSW→ V уже находится в Q, то нужно только добавить SA→V:

Q = [SA→NSW, NSW →SA, SA→NT, NT →SA, V →NSW, NSW→V, SA→V]



Продолжаем процесс, пока в конечном итоге **не удалим дугу SA→NT** из Q. Проверка согласованности этой дуги удаляет синий цвет из области SA, оставляя её пустой и вызывая возврат. Обратите внимание, что дуга NSW→SA находится перед SA→ NT в Q и что проверка согласованности этой дуги удаляет синий цвет из области NSW.



Проверка согласованности дуг обычно реализуется с помощью алгоритма АС-3 (Arc Consistency #3), псевдокод которого имеет следующий вид:

Алгоритм проверки совместимости дуг AC3

function AC-3(*csp*) **returns** CSP-задачу с сокращенными обл. определения
  inputs: *csp*, бинарная CSP с переменными $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, очередь из дуг, первоначально включающая все дуги

  while очередь *queue* не пуста **do**
    $(X_i, X_j) \leftarrow$ Remove-First(*queue*)
    if Remove-Inconsistent-Values(X_i, X_j) **then**
      for each X_k **in** Neighbors[X_i] **do**
        добавить (X_k, X_i) к очереди *queue*

---

function Remove-Inconsistent-Values(X_i, X_j) **returns** true е.е. успех
  removed $\leftarrow$ *false*
  for each x **in** Domain[X_i] **do**
    if ни одно значение y из области Domain$[X_j]$ не позволяет использовать (x, y)
      для удовлетворения ограничения между $X_i \leftrightarrow X_j$
      then удалить x из области Domain[X_i]; *removed* $\leftarrow$ *true*
  return *removed*

После применения алгоритма AC-3 либо каждая дуга является совместимой, либо некоторая переменная имеет пустую область определения, указывая на то, что эту задачу CSP невозможно сделать совместимой по дугам (и поэтому данная задача CSP не может быть решена).

В.Бондарев

Сложность проверки совместимости дуг

Проверку совместимости дуг можно использовать либо в качестве этапа предварительной обработки перед началом процесса поиска, либо в качестве этапа распространения ограничения (аналогично опережающей проверке) после каждого присваивания во время поиска. (Последний алгоритм иногда называют MAC – Maintaining Arc Consistency)

Сложность проверки совместимости дуг можно проанализировать следующим образом:

- любая бинарная задача CSP имеет самое большее $O(n^2)$ дуг;
- каждая дуга $Xk \rightarrow Xi$ может быть "внесена в повестку дня" только d раз, поскольку область определения *Xi*, имеет самое большее d значений, доступных для удаления;
- проверка совместимости любой дуги может быть выполнена за время $O(d^2)$, поэтому **в наихудшем случае затраты времени составляют $O(n^2 d^3)$**.

Хотя такой метод является значительно более дорогостоящим по сравнению с опережающей проверкой, все эти дополнительные затраты обычно окупаются.

k-совместимость

Более строгие формы распространения ограничения можно определить с помощью понятия **k-совместимости.**

Задача CSP является k-совместимой, если для любого множества из k переменных совместимое присваивание для любого подмножества из k-1 переменной гарантирует, что любой k-й переменной всегда можно присвоить некоторое совместимое значение.

Например, 1-совместимость означает, что совместимой является каждая отдельная переменная сама по себе; это понятие называют также **совместимостью узла.**
Далее, **2-совместимость** — то же, что и совместимость дуги,
а **3-совместимость** означает, что любая пара смежных переменных всегда может быть дополнена третьей соседней переменной; это понятие именуется также **совместимостью пути.**

Строгая k-совместимость

Любой граф называется **строго k-совместимым**, если он является k-совместимым, а также (k-1)-совместимым, (k-2)-совместимым, ... и т.д. вплоть до 1-совместимого.

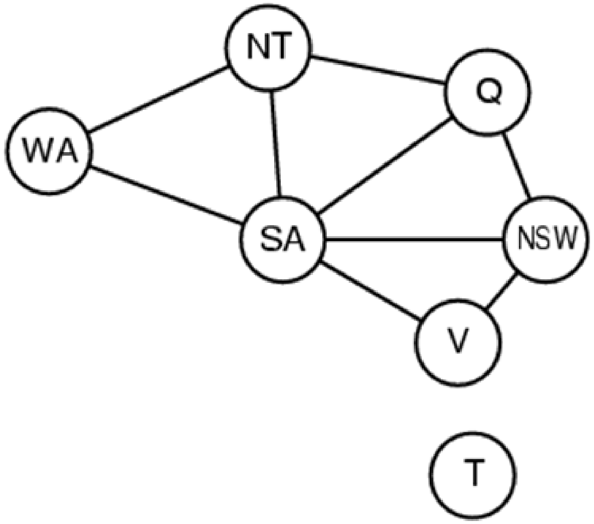
Теперь предположим, что имеется некоторая задача CSP с k узлами, которая ***сделана*** *строго k-совместимой . Тогда эту задачу можно решить без возвратов.*

Для этого вначале можно выбрать совместимое значение для X_1. В таком случае существует гарантия, что удастся выбрать значение для X_2, поскольку граф является 2-совместимым, для X_3, поскольку он — 3-совместимый, и т.д. Для каждой переменной X_i необходимо выполнить поиск только среди d значений в ее области определения, чтобы найти значение, совместимое с X_i ,..., X_{i-1}. Это означает, что **гарантируется нахождение решения за время O(nd).**

Безусловно, за такую возможность также приходится платить: *любой алгоритм обеспечения k-совместимости в наихудшем случае должен требовать времени, экспоненциально зависящего от k.*

Структура задач

Для быстрого поиска решений CSP–задачи можно использовать информацию о структуре задачи, представленной в виде графа ограничений. Идея состоит в том, чтобы **разложить задачу на множество подзадач**. Например, для задачи раскраски карты раскраска материка и Тасмании – это 2 независимые подзадачи.



Пусть каждая подзадача имеет c переменных из общего количества n переменных. Тогда n/c – число подзадач, для решения каждой из них требуется d^c операций. Отсюда сложность равна **O(d^c n/c) и она линейно зависит от n**; без декомпозиции сложность пропорциональна O(d^n) и она экспоненциально зависит от n.

Пример: разделение булевой CSP с n=80 на 4 подзадачи с c=20:

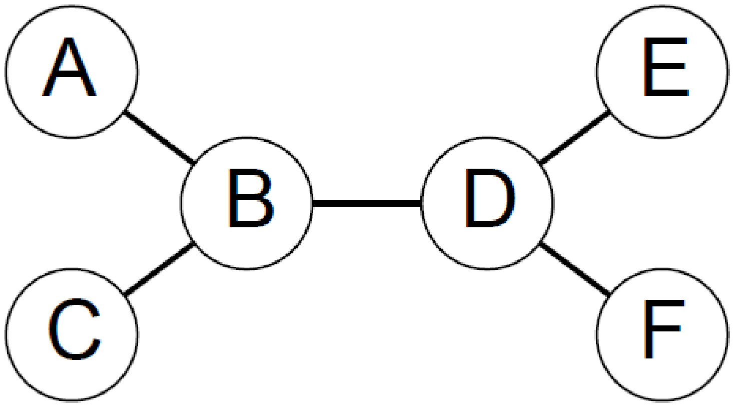
n=80, d=2, c=20

2^{80} = 4 млрд. лет при 10 млн. узлов/сек

4 2^{20} = 0.4 сек. при 10 млн. узлов/сек

Древовидные структуры CSP

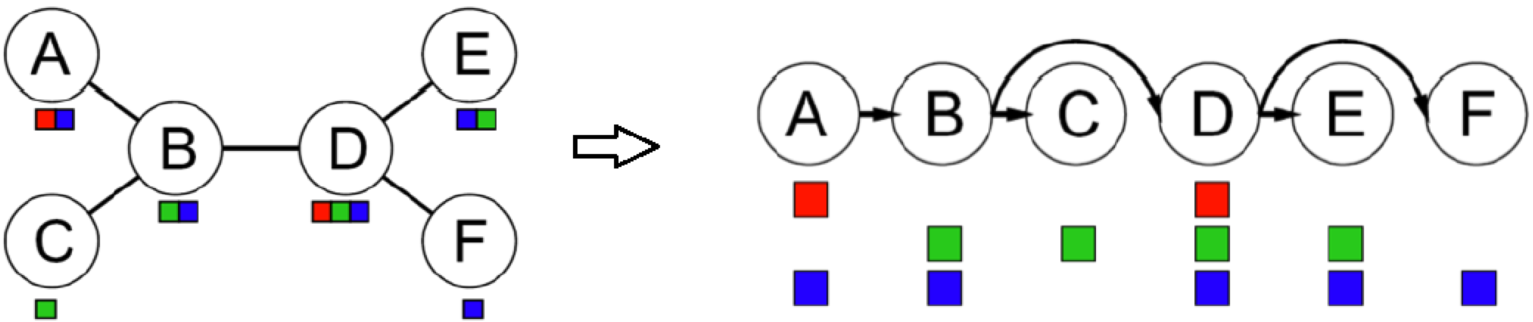
Полностью независимые подзадачи являются привлекательными, но встречаются редко. В большинстве случаев подзадачи любой задачи CSP связаны друг с другом. Простейшим случаем является тот, в котором **граф ограничений образует дерево:** любые две переменные связаны не больше чем одним путем.



Теорема: *если граф ограничений не имеет циклов, то CSP задачи задача может быть решена за время* ***$O(nd^2)$ -*** *линейно зависящее от количества переменных.*

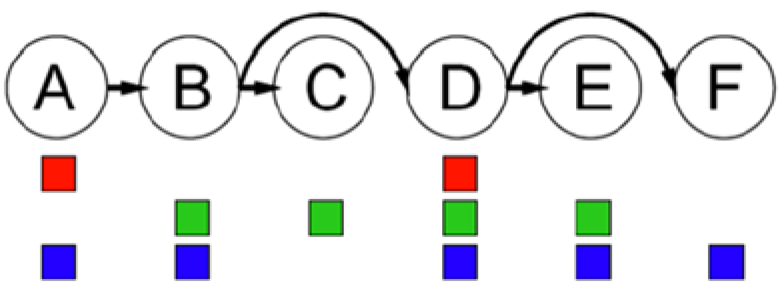
Алгоритм решения древовидных CSP

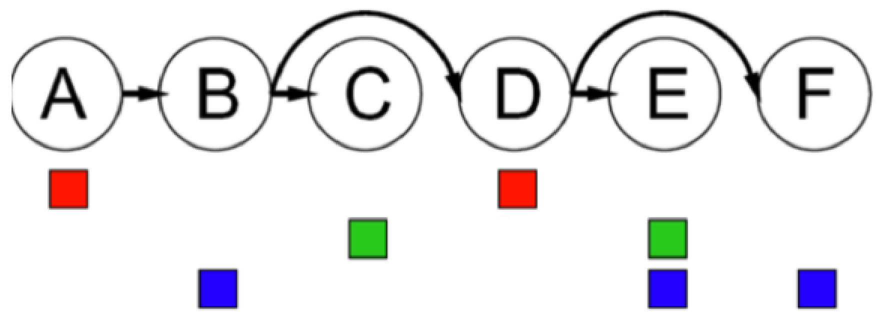
1. **Выбрать в качестве корня дерева любую переменную и упорядочить переменные от корня до листьев** таким образом, чтобы родительский узел каждого узла в дереве предшествовал этому узлу в таком упорядочении. Обозначить эти переменные по порядку как X1, ...,Xn. Теперь каждая переменная, кроме корня, имеет только одну родительскую переменную.



Алгоритм решения древовидных CSP

2. **Выполнить обратный обход и проверить совместимость дуг.**
В цикле по j от n до 2 применять проверку совместимости к дугам (Xi →Xj), где Xi — родительский узел узла Xj, удаляя значения из области определения Domain [Xi] по мере необходимости.



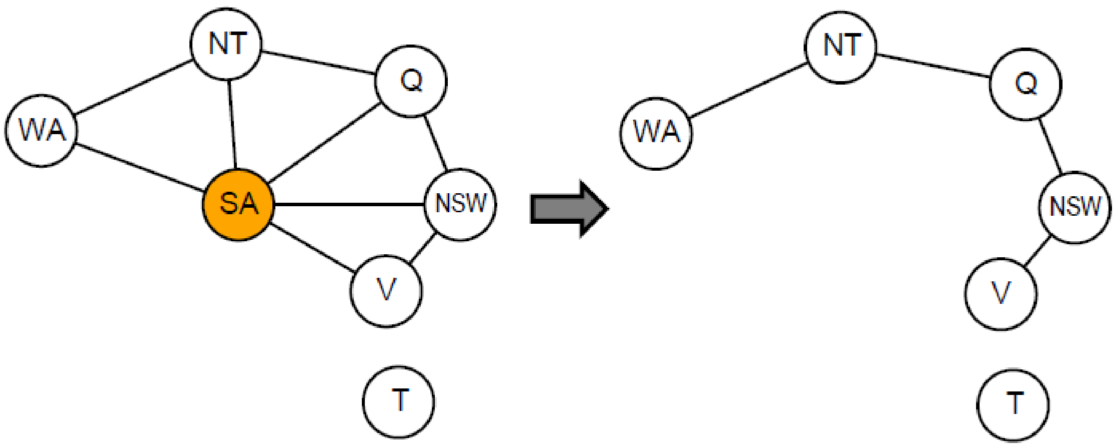


3. **Выполнить прямое присваивание.**
В цикле по j от 1 до n присваивать Xj любое значение, совместимое со значением, присвоенным Xi, где Xi — родительский узел узла Xj.

Близкие к древовидным структуры CSP

Каким-образом приводить к древовидным структурам более общие графы ограничений? Один из способов выполнения этой задачи основан на **удалении узлов.**

Способ предусматривает присваивание значений некоторым переменным так, чтобы оставшиеся переменные образовывали дерево. Например, после удаления узла SA и связанных с ним ограничений, любое решение данной задачи CSP будет совместимым со значением, выбранным для SA:



Теперь появляется возможность решить задачу, представленную оставшимся деревом, с помощью приведенного выше алгоритма и таким образом решить всю проблему.

Близкие к древовидным структуры CSP

Общий алгоритм преобразования графов ограничений к древовидным структурам (алгоритм определения условий выбора множества разрыва цикла (cutset conditioning)):

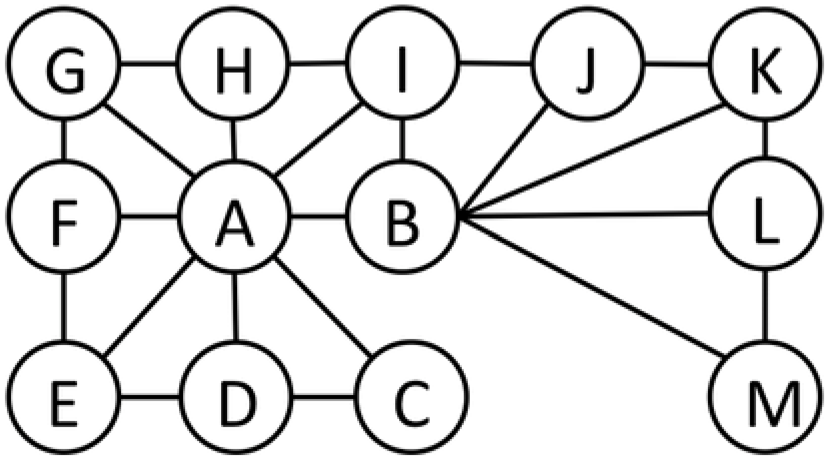
1. Выбрать подмножество S из множества Variables[csp], такое, что граф ограничений после удаления S становится деревом. Подмножество S называется **множеством разрыва цикла** (cycle cutset).

2. Для каждого возможного присваивания переменным в S, которое удовлетворяет всем ограничениям в S, выполнить следующее:
   - удалить из областей определения оставшихся переменных любые значения, несовместимые с данным присваиванием для S;
   - если оставшаяся задача CSP имеет решение, вернуть это решение вместе с присваиванием для S.

Пусть множество разрыва цикла имеет размер c. При этом возможны $\boldsymbol{d^c}$ возвратов. После удаление этого множества получаем CSP с древовидной структурой с ($\boldsymbol{n}$-$\boldsymbol{c}$) переменными. В этом случае решение задачи будет иметь сложность $\mathbf{O}((\boldsymbol{n}-\boldsymbol{c})\boldsymbol{d}^2)$. Следовательно, общее время выполнения $\mathbf{O}(\boldsymbol{d}^c(\boldsymbol{n}-\boldsymbol{c})\boldsymbol{d}^2)$, что очень хорошо для малых $\boldsymbol{c}$.

Задача

- Найдите наименьшее подмножество разрыва цикла для графа.



Древовидная декомпозиция

Иной подход основан на построении **древовидной декомпозиции графа ограничений в набор связанных подзадач**. **Каждая подзадача решается независимо, а полученные решения затем объединяются.**

Как и большинство алгоритмов «разделяй и властвуй», это работает хорошо, если подзадачи не слишком велики.

Древовидное разложение должно удовлетворять следующим трем **требованиям**:

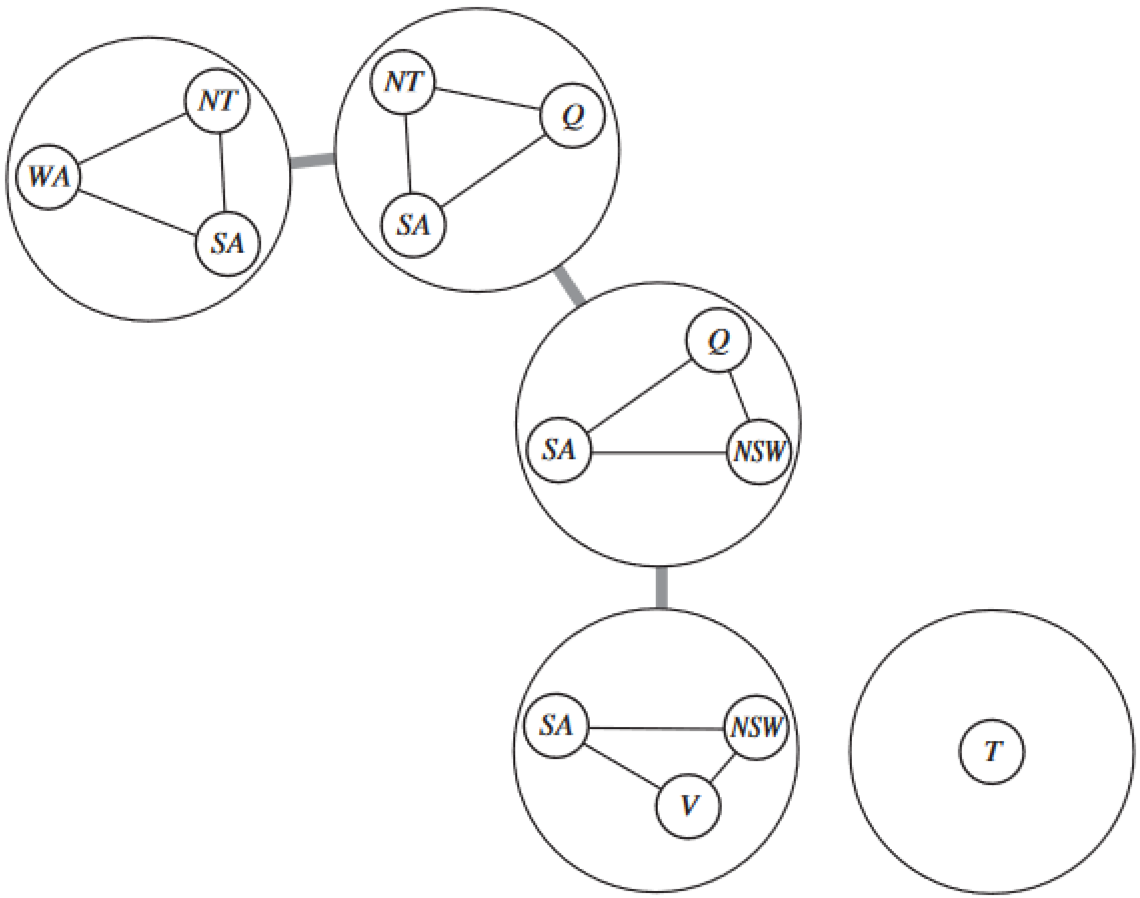
• Каждая переменная исходной задачи появляется по крайней мере в одной из подзадач.

• Если две переменные связаны ограничением в исходной задаче, они должны появляться вместе (вместе с ограничением) по крайней мере в одной из подзадач.

• Если переменная появляется в двух подзадачах в дереве, она должна появляться в каждой подзадаче по пути, соединяющему эти подзадачи.

Древовидная декомпозиция

Древовидная декомпозиция графа ограничений задачи раскраски карты



Древовидная декомпозиция

Мы **решаем каждую подзадачу независимо**; если какая-либо из них не имеет решения, то вся задача не имеет решения.

Глобальное решение строится следующим образом:

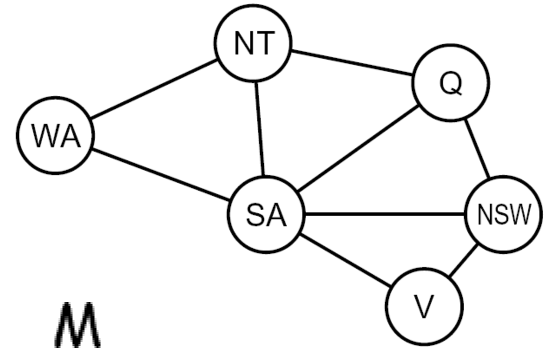
1. Рассматриваем каждую подзадачу как **«мегапеременную»**, областью определения которой является множество всех решений для подзадачи. Например, самая левая подзадача имеет шесть решений — одно из них {WA = красный, SA = синий, NT = зеленый}.

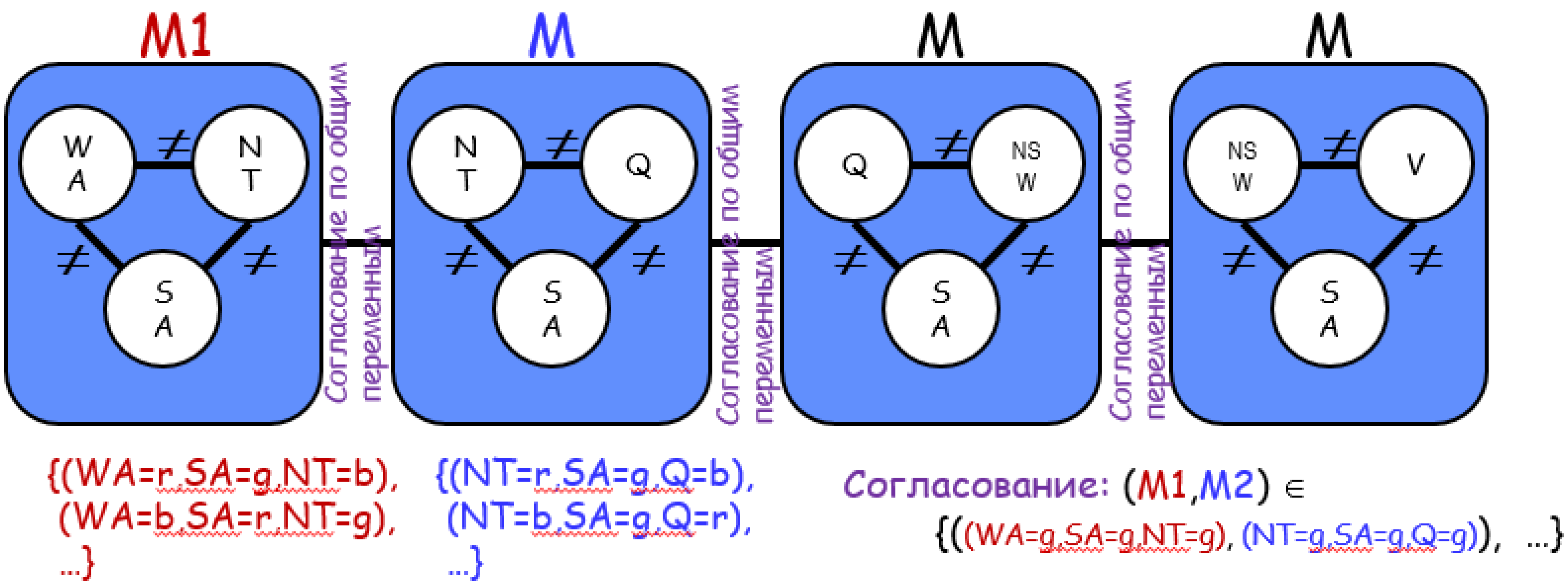
2. Решаем задачу с ограничениями, связывающими подзадачи, используя эффективный алгоритм для деревьев, приведенный ранее.

Ограничения, связывающие подзадачи, указывают на то, что решения подзадач должны быть согласованы по их общим переменным. Например, если {WA = красный, SA = синий, NT = зеленый} для первой подзадачи, то единственным решением для следующей подзадачи будет {SA = синий, NT = зеленый, Q = красный}.

Древовидная декомпозиция

- Идея: создать древовидный граф мегапеременных (M)
- Каждая мегапеременная кодирует часть исходной CSP
- Подзадачи перекрываются для обеспечения согласованных решений





Если граф имеет ширину дерева w, то задача может быть решена за время $O(nd^{w+1})$. ***Следовательно, CSP с графами ограничений конечной ширины дерева разрешимы за полиномиальное время***. К сожалению, нахождение декомпозиции с минимальной шириной дерева является NP-трудной задачей, но существуют эвристические методы, которые хорошо работают на практике.

В.Бондарев

Локальный поиск для задач CSP

Локальный поиск работает путем **итеративного улучшения** - начинают со случайного полного присваивания. Затем итеративно выбирают случайную конфликтующую переменную и присваивают ей значение, которое нарушает наименьшее количество ограничений, пока не исчезнут нарушения ограничений (политика, известная как эвристика минимальных конфликтов- MRV).

Особенности применительно к CSP задачам:

– допускает состояния, неудовлетворяющие ограничениям;

– операторы переназначают значения переменным;

– выбор переменных: случайно выбирается любая конфликтующая переменная;

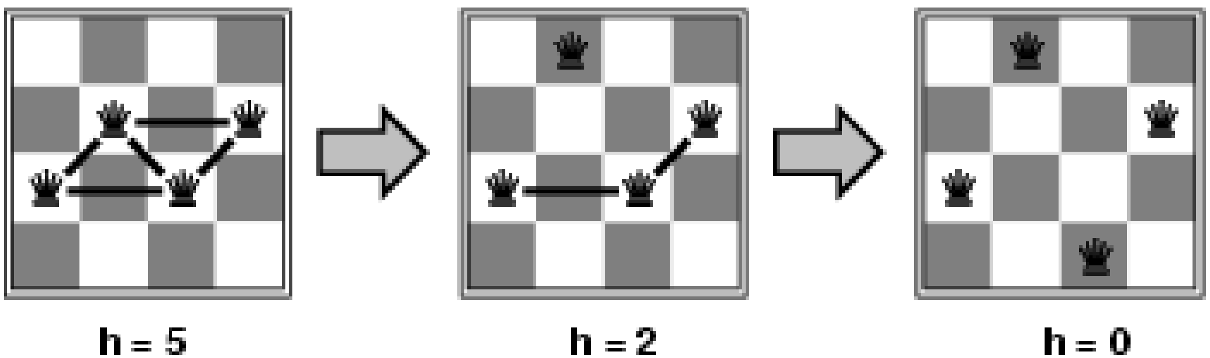
– выбор значений на основе **эвристики минимальных конфликтов:**

выбирается значение, которое нарушает минимум ограничений, например, $h(n)$ = общее кол-во нарушенных ограничений.

– допускают оперативную корректировку полученных решений при изменении условий задачи (например, старт от текущего состояния при внесении изменений в расписание)

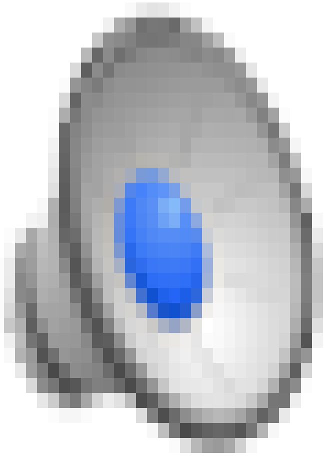
Пример: Задача 4-ферзя

- Состояния: 4 ферзя в 4 столбцах ($4^4 = 256$ состояний)
- Действие: перемещать ферзь по столбцу
- Целевой тест: отсутствие атак
- Эвристика: $h(n)$ = число атак



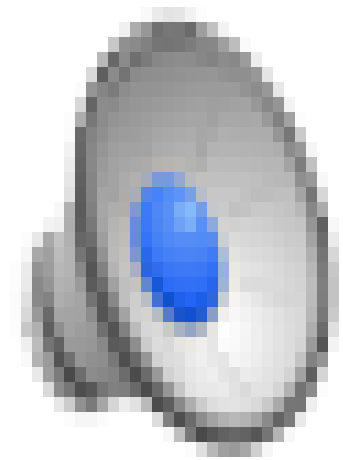
- Задав случайное начальное состояние, можно решать задачу n-ферзей практически за постоянное время для произвольных n (например, n = 1 000 000)

Итеративное улучшение – n Ферзей



В.Бондарев

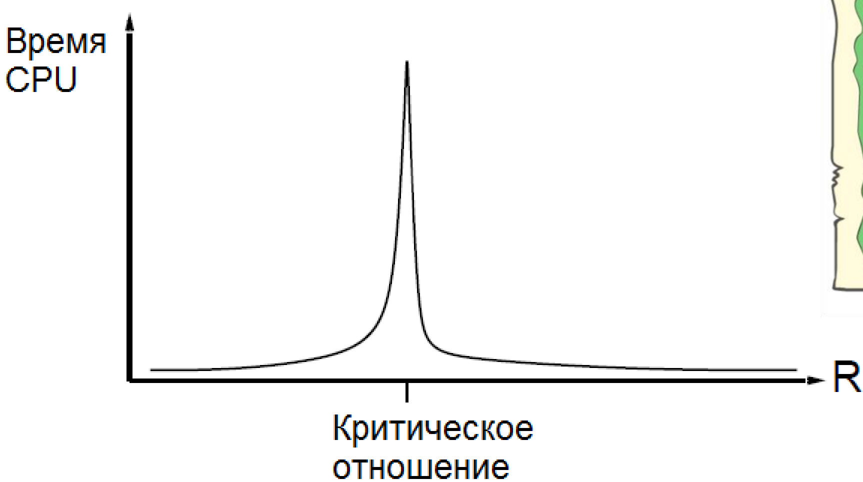
Итеративное улучшение - раскрашивание

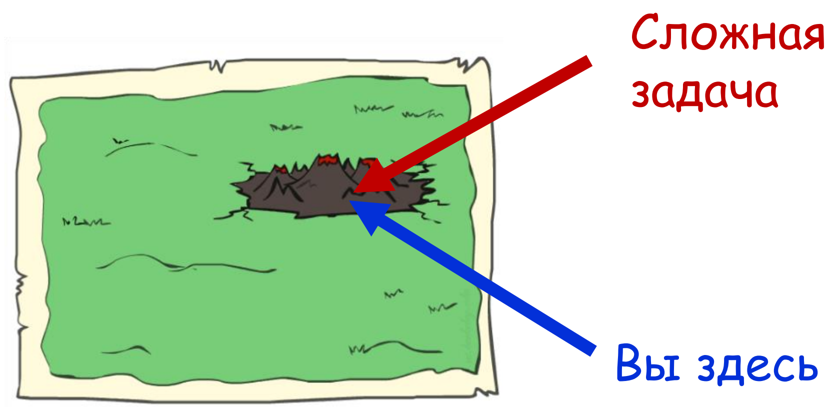


Локальный поиск для задач CSP

Фактически, локальный поиск, **выполняется почти за постоянное время** и имеет высокую вероятность успеха не только для N-ферзей со сколь угодно большим N, но и для любой случайно сгенерированной CSP! За исключением узкой области определяемой отношением:

$$R = \frac{\text{число ограничений}}{\text{число переменных}}$$





Однако, несмотря на эти преимущества, **локальный поиск является неполным и неоптимальным,** поэтому он не обязательно приведет к оптимальному решению.

Выводы

Важно помнить, что задачи удовлетворения ограничений, как правило, не имеют эффективного алгоритма, который решал бы их за полиномиальное время по отношению к количеству задействованных переменных. Однако, используя различные эвристики, мы часто можем найти решения за приемлемое время:

- **Фильтрация –** позволяет выполнять упреждающее сокращение областей определения переменных с неприсвоенными значениями, чтобы предотвратить ненужный поиск с возвратом. Мы рассмотрели два важных метода фильтрации: опережающую проверку и согласованность дуг.
- **Упорядочивание** - управляет выбором, какую переменную или значение использовать для присваивания, чтобы сделать обратный поиск как можно маловероятным. При выборе переменных используется политика MRV (минимального кол-ва оставшихся значений), а при выборе значений - политика LCV (наименее ограничительного значения).
- **Структура задач.** Если CSP имеет древовидную структуру или близка к древовидной структуре, то можно использовать древовидный алгоритм CSP, чтобы получить решение за линейное время. Если CSP близка к древовидной структуре, можно использовать удаление для преобразования CSP-задачи в одну или несколько независимых CSP-задач с древовидной структурой и решать каждую из них отдельно.
- **Стратегия локального поиска, минимизирующая конфликты**, обычно весьма эффективна на практике.